Lecture Note

OpenIDConnect

Using Keycloak for authentication

Lektor Karsten Jeppesen
kaje@ucn.dk

Revision 3.3

Indhold

1. Abstract

3rd semester AP is taught the basics of password handling being hashing, salting, and peppering. However more secure and modern means are available for this purpose.
This paper describes the OpenID Connect Identity provider Keycloak and the OAUTH2 flows needed for Visual Studio projects. The use of API keys is outside scope for this paper.

2. Revision

This document is valid for the version 3.x of the demo software.

• 2024.02.05:	Updated due to Server burn. Table inserted to accommodate future similar changes.
• 2024.10.31:	Microsoft enhanced the support for OpenID making things so much easier.
• 2024.12.05:	Added sample code for accessing the access_token using JWT decoding.
• 2025.01.23:	UML Sequence diagrams added
• 2025.04.22:	Update for .Net 8

3. Method

This paper will first identify the actors within the client scope, then move to the realm scope. This is done to facilitate the understanding of the underlying mechanics.

4. Actors

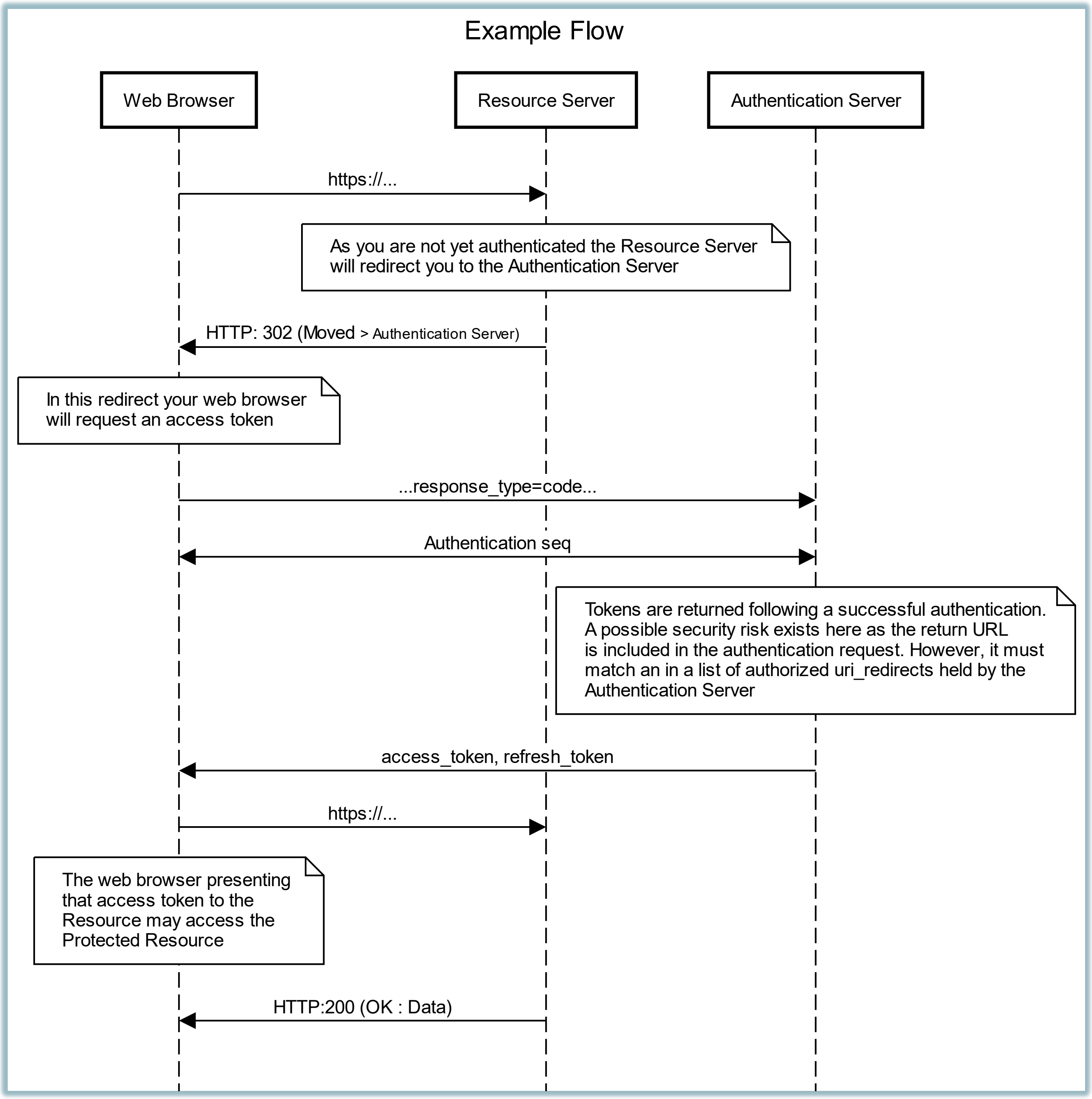
As with any theater play, we must set the scene, describing who is playing a part here. We will start of by defining the players involved in the first scene:

- A Protected Resource is anything that is protected by the authorization scheme.
- Anyone in possession of credentials to ensure access to a Protected Resource is a Resource Owner. The Resource Owner is thus not who or whatever owns the resource but a question of permissions.
- The Client is the application requesting access to a Protected Resource on behalf of the Resource Owner.
- The Resource Server is hosting the Protected Resources. This is the API you want to access.
- The Authorization Server authenticates the Resource Owner and issues Access Tokens after getting proper authorization. In this case, Keycloak.
- The User Agent is used by the Resource Owner to interact with the Client (for example, a browser or a native application).

5. A real example

You want to access a simple static webpage (https://test.a.ucnit.eu). The page is the Protected Resource, and the webserver is the Resource server. If you are in possession of credentials (username, password) granting access, then you are a Resource Owner. The web browser you are using for that is both User Agent and Client. In this case all Resource Owners are authorized to access the Protected Resource.

You (the Resource Owner) type in the URL (https://test.aucnit.eu) and your web browser (the Client) attempts to secure the protected resource:

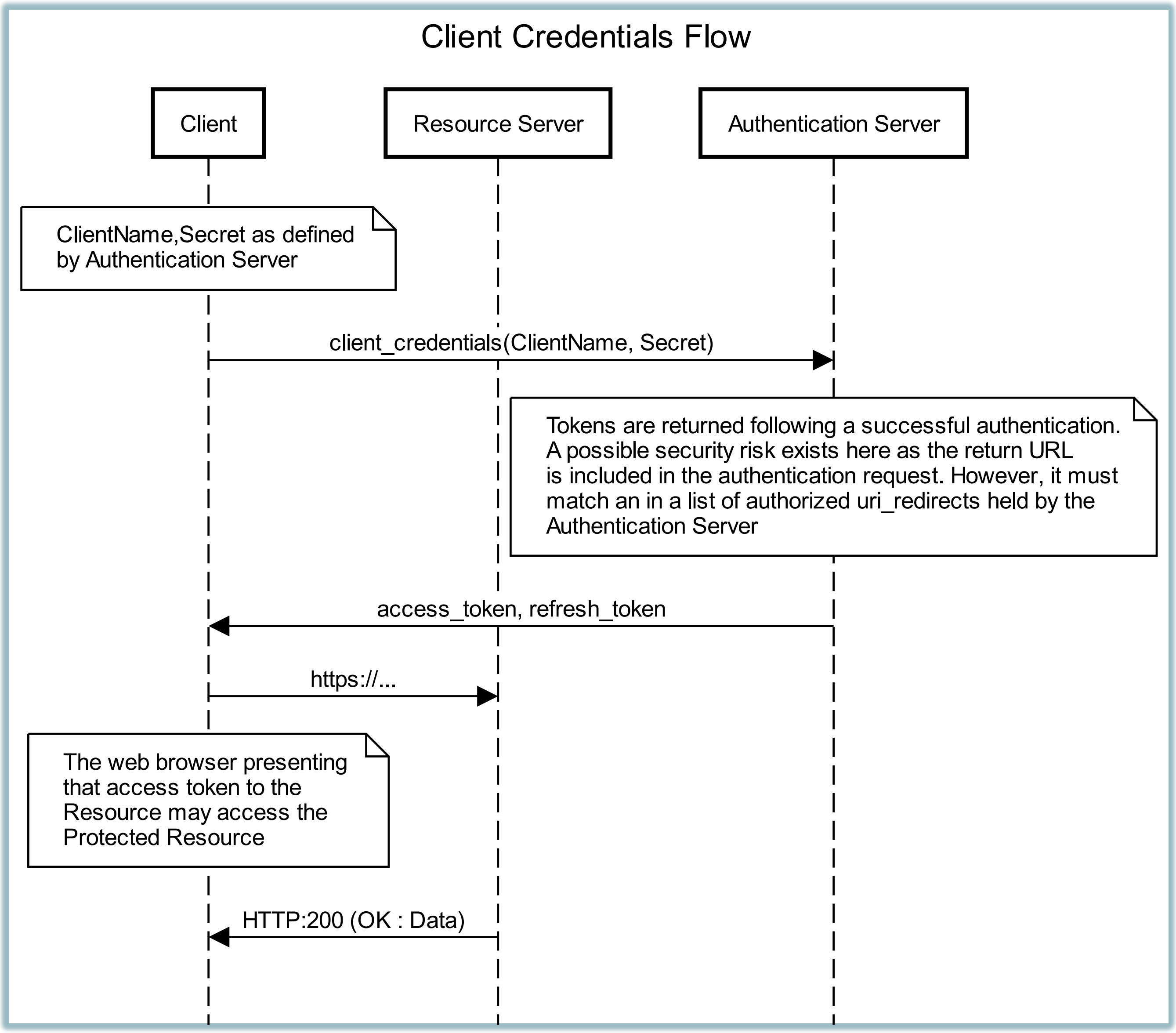

Example Flow
Web Browser
Resource Server
Authentication Server
https://...
As you are not yet authenticated the Resource Server will redirect you to the Authentication Server
HTTP: 302 (Moved > Authentication Server)
In this redirect your web browser will request an access token
...response_type=code...
Authentication seq
Tokens are returned following a successful authentication. A possible security risk exists here as the return URL is included in the authentication request. However, it must match an in a list of authorized uri_redirects held by the Authentication Server
access_token, refresh_token
https://...
The web browser presenting that access token to the Resource may access the Protected Resource
HTTP:200 (OK : Data)

6. Flows

The OpenIDConnect must deal with a variety of situations, some of which are not trivial. The above illustrated sequence or flow is so simplified that it doesn't take place in real life. It, however, very precisely points out how information is flowing between actors. In the following some, but not all, flows will be discussed. It is worth mentioning that those who applies to the semester project are included.

6.1 Client Credentials Flow

This flow is used if the Client is a Resource Owner. This could be an application which is granted access to the Protected Resource despite who is connected to the application. An example would be a backend application which needs to connect to a protected API for database access:


Client Credentials Flow
Client
Resource Server
Authentication Server
ClientName,Secret as defined by Authentication Server
client_credentials(ClientName, Secret)
Tokens are returned following a successful authentication. A possible security risk exists here as the return URL is included in the authentication request. However, it must match an in a list of authorized uri_redirects held by the Authentication Server
access_token, refresh_token
https://...
The web browser presenting that access token to the Resource may access the Protected Resource
HTTP:200 (OK : Data)

- The Web Browser makes a request to the Client (application).
- The Client itself holds the credentials (ClientID, Secret) which it then sends to the Authentication Server
- The Authentication Server returns an Access Token (JWT)
- The Client can now make an authenticated request to the Resource Server
- The Resource Server returns the Protected Resource

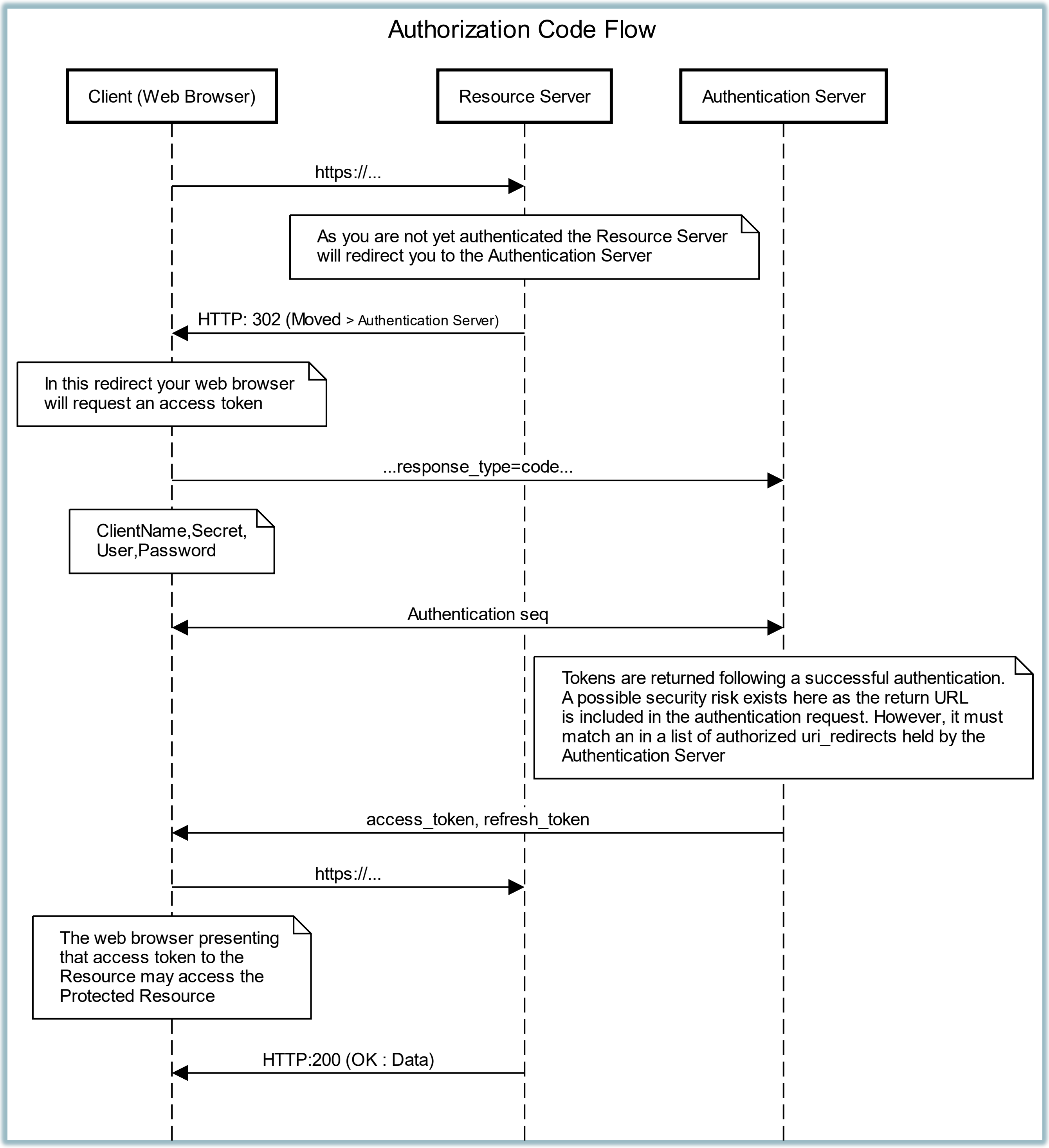
Please note that as no user was presented to the Authentication Server, then the JWT access token will not contain any useful “role” information.

6.2 Resource Owner Password Flow

This flow is used if the application is provided with username and password which will serve as credentials. The flow is identical to the Client Credentials flow except for that the returned JWT access token now also holds additional user information including role information.

6.3 Authorization Code Flow

This flow is used if you want the Authentication Server to manage the login sequence. As you may invoke a browser component or the system browser from your application, this flow can be used in a multitude of situations. The benefit of this workflow is that your application at no point in time handles neither username nor password and additionally you gain access to the OTP (One Time Password) which is one of the components in MFA (Multi Factor Authentication).


Authorization Code Flow
Client (Web Browser)
Resource Server
Authentication Server
https://...
As you are not yet authenticated the Resource Server will redirect you to the Authentication Server
HTTP: 302 (Moved > Authentication Server)
In this redirect your web browser will request an access token
...response_type=code...
ClientName,Secret, User,Password
Authentication seq
Tokens are returned following a successful authentication. A possible security risk exists here as the return URL is included in the authentication request. However, it must match an in a list of authorized uri_redirects held by the Authentication Server
access_token, refresh_token
https://...
The web browser presenting that access token to the Resource may access the Protected Resource
HTTP:200 (OK : Data)

- The application starts a web browser (Client) which sends a request to the Authentication Server.
- The Authentication Server returns a login screen to capture the credentials.
- Upon successful authentication the Authentication Server returns JWT tokens (access + refresh)
- The access token contains user info including role information.
- The application retrieves the JWT tokens from the web browser.

- The application can now make an authenticated request to the Resource Server
- The Resource Server returns the Protected Resource

7. Keycloak

Keycloak (https://www.keycloak.org/) is an OpenID Connect Identity Provider. This means that Keycloak is fully capable of all flows and can handle OTP/MFA (One Time Password/Multi Factor Authentication) as well. Keycloak is a Redhat product and is both free and open source. It is also available as a docker container.

- The version used in this setting is: 23.0.3
- It is within the scope of this paper to describe the setup needed for the flows.

7.1 Realm

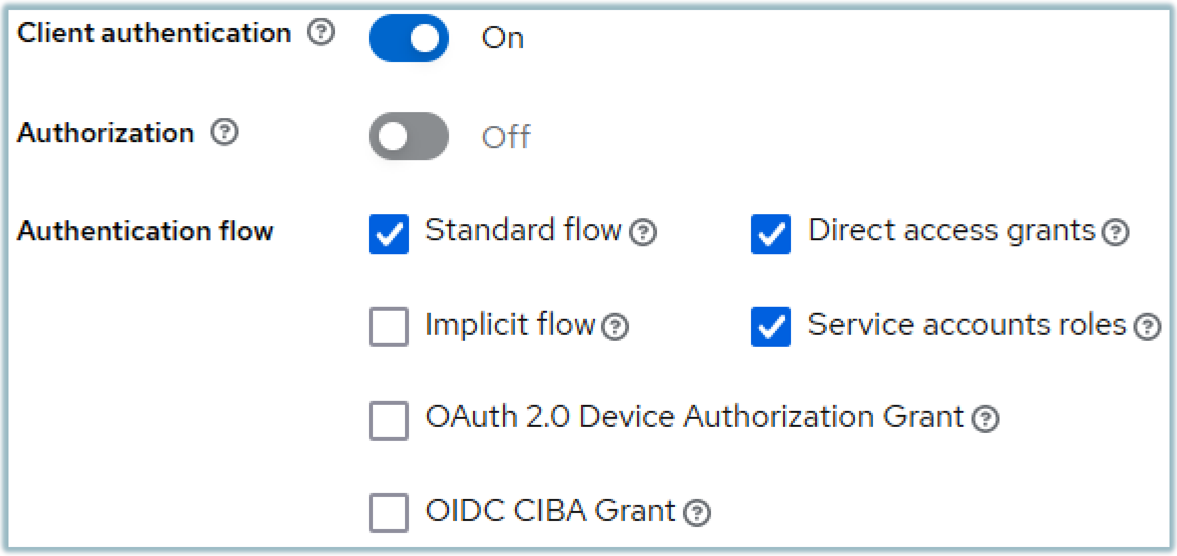
A realm is like a namespace. I can have one for fun and one for work, you may have any number of realms for whatever purposes. It is just a workspace. To this paper, I set up a realm with the incredible original name of "xOIDCx". The name was chosen to be short and distinct as you will later meet environment variables with names like OIDC_ not to be mistaken by.

7.2 Clients

Within a realm you will find clients and users. Clients are authentication entities. A client like "General" could be used to authenticate any number of web sites and applications. But all clients in "General" will share the same attributes and are dependent on the continued existence of "General".

Clients are entities of a realm, so the client "General" may exist in any realm while never sharing any information just because of the name.

To this paper a client named "Alice" was created:

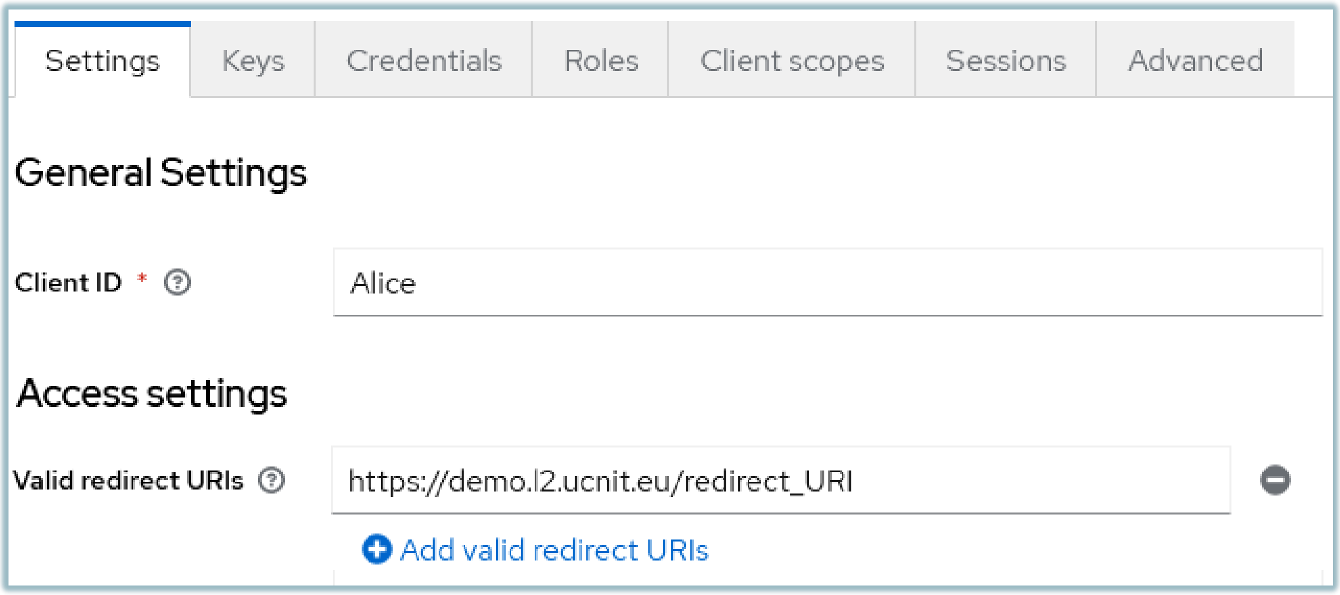


Client authentication: On

Authorization: Off

Authentication flow: [✓ Standard flow] [✓ Direct access grants] Implicit flow [✓ Service accounts roles] OAuth 2.0 Device Authorization Grant, OIDC CIBA Grant

In the settings the following Redirect URI was set up:



General Settings

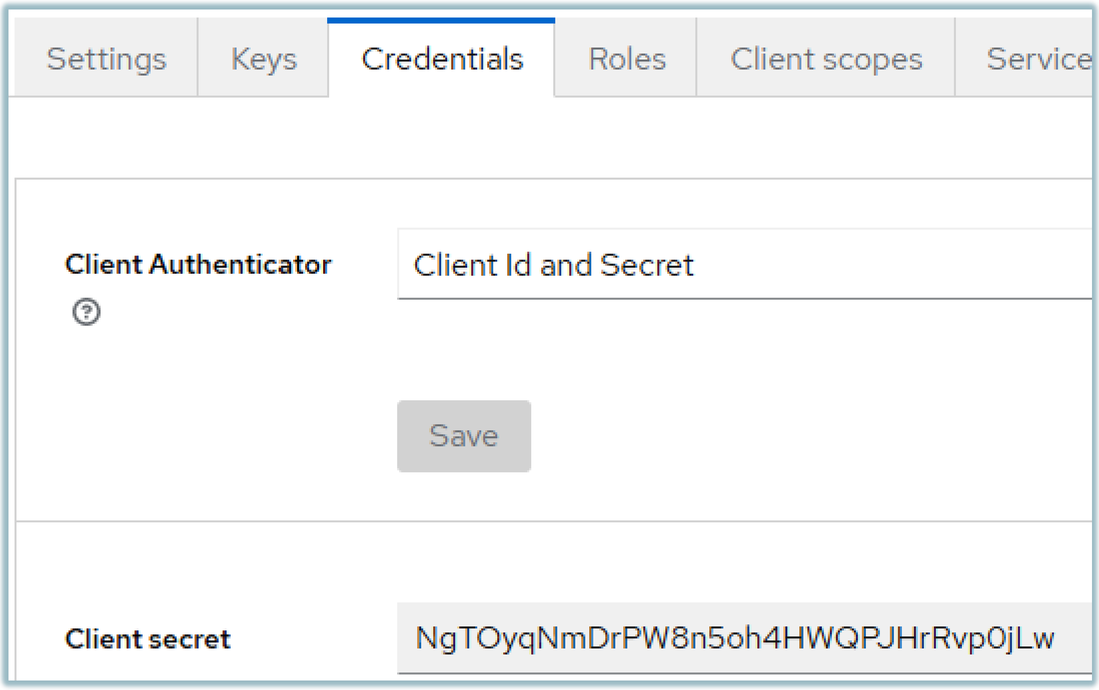
Client ID * Alice

Access settings

Valid redirect URIs https://demo.l2.ucnit.eu/redirect_URI

The Redirect URIs are a list of allowed returns from the authentication process. The Client will provide such one when it requests the access token in the authentication process. This mechanism guards against returning to a hostile URI.

In the Credentials tab the important information is the “Client secret”:



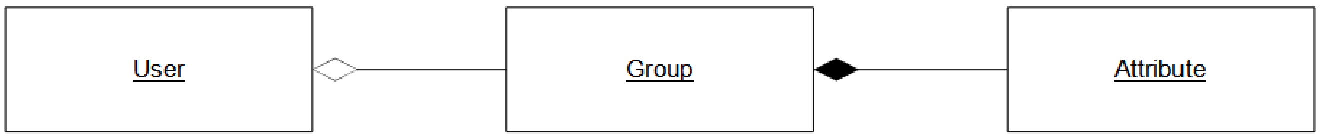
Client Authenticator Client Id and Secret

Client secret NgTOyqNmDrPW8n5oh4HWQPJHrRvp0jLw

Obviously, this information should be shared with nobody, but it is exposed in this case as this realm with its clients and users exists for the purpose of this paper. Please see chapter 13.

7.2.1 Users, Groups and Attributes

Users may be members of groups and groups may define any number of attributes. In UML this could be visualized this way:



The xOIDCx realm defines the the following groups:

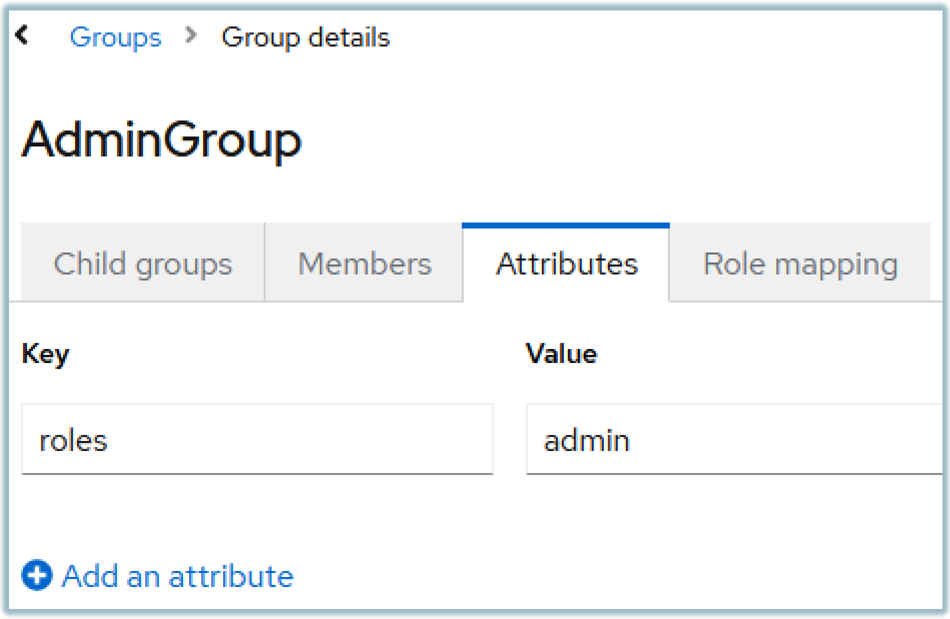
- AdminGroup: [roles: admin]
- UserGroup: [roles: user]
- WarlockGroup: [roles: warlock]

All these names are case sensitive.

The following Users are defined:

- admin (password: admin), member of AdminGroup
- user (password: user), member of UserGroup
- warlock (password warlock), member of WarlockGroup
- nobody (password nobody), member of no groups

Setting up a group looks like this (AdminGroup used as example):



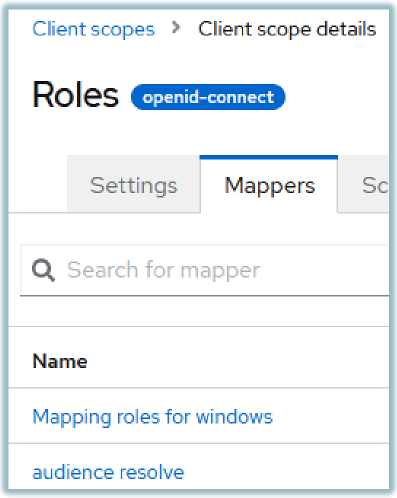
Key	Value
roles	admin

7.3 Mappers

Microsoft require the “roles” attribute to be at root level in the data structure. To achieve that it is necessary to “move” the role attribute defined in the groups (AdminGroup, UserGroup and WarlockGroup) into place.

The client (Alice) has several predefined “Scopes”. A “Scope” is a collection of attributes which can be requested by the Client (the browser or App). One of these “Scopes” is the “roles” scope.

For this scope a mapper (Mapping for windows) was defined



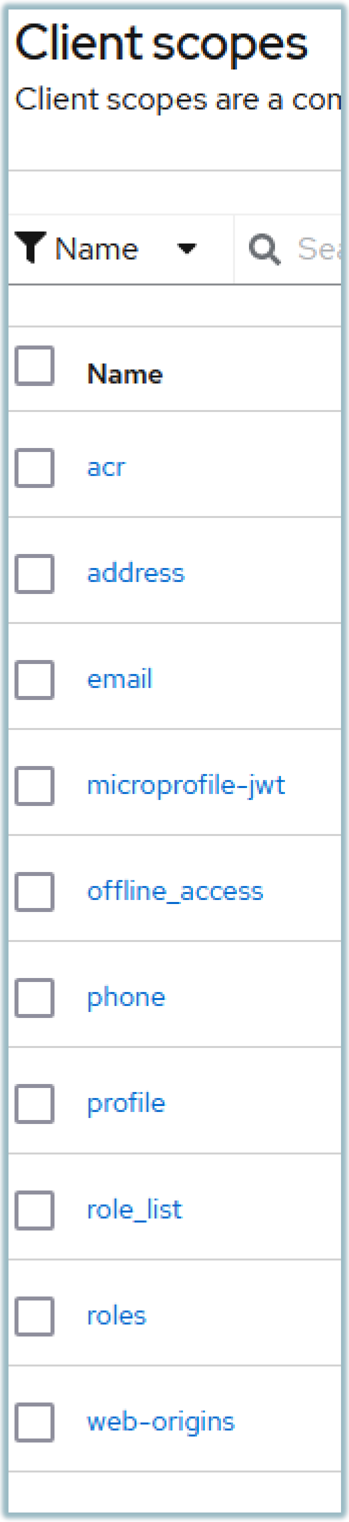
Client scopes > Client scope details

Roles openid-connect

Settings | Mappers | Sc

Search for mapper

Name
Mapping roles for windows
audience resolve



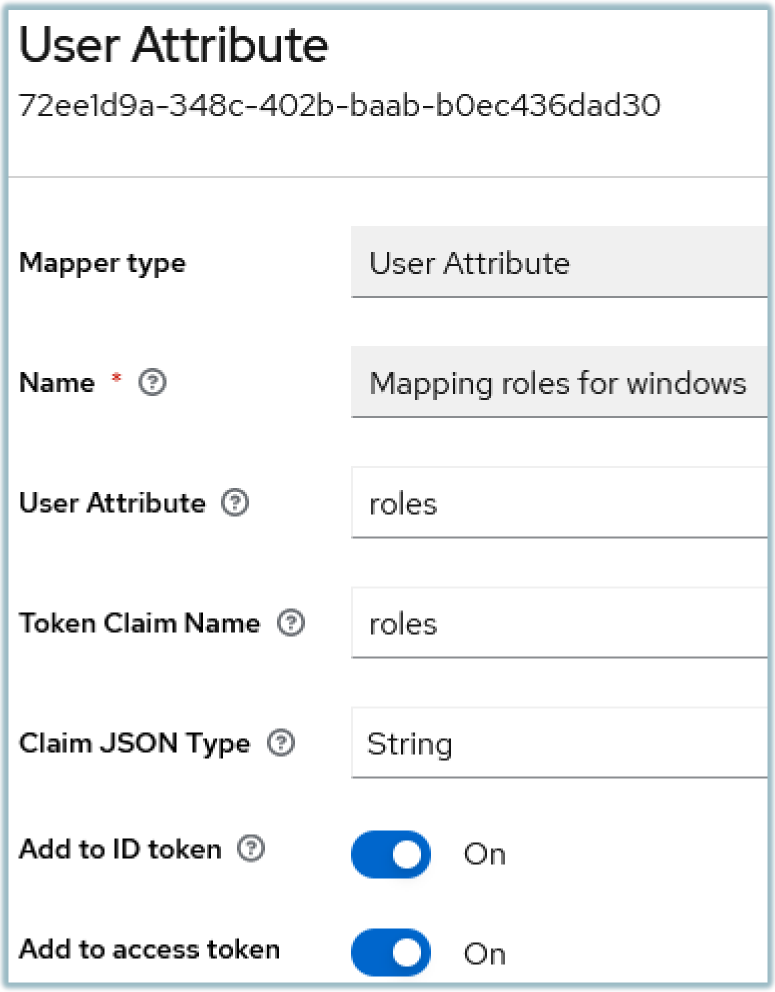
Client scopes

Client scopes are a con

Name | Sea

Name
acr
address
email
microprofile-jwt
offline_access
phone
profile
role_list
roles
web-origins

This mapper maps the user attribute “roles” into the “roles” attribute in the “roles” scope to be included in the access token:



User Attribute

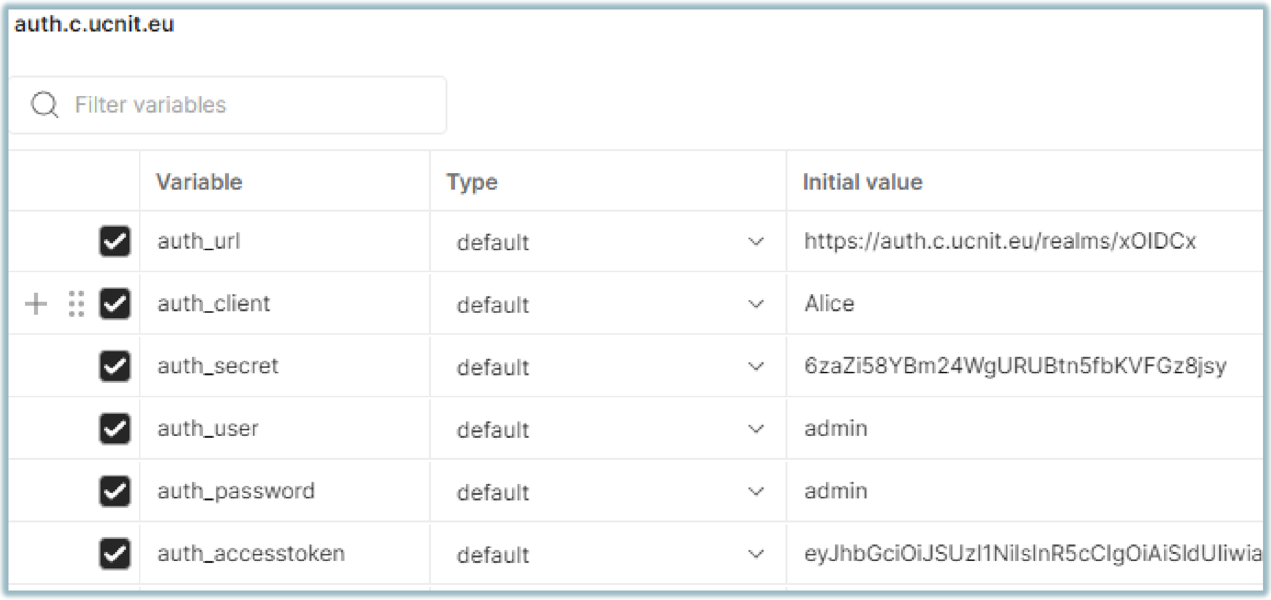
72ee1d9a-348c-402b-baab-b0ec436dad30

Mapper type	User Attribute
Name *	Mapping roles for windows
User Attribute	roles
Token Claim Name	roles
Claim JSON Type	String
Add to ID token	On
Add to access token	On

8. Flows in detail.

In the following chapters the flows will be described in more details. Depending on the flow this will include examples using either Postman or debugger views from a web browser. Some familiarity with those functionalities is thus prerequisites to these chapters.

The environment for the following operations is:



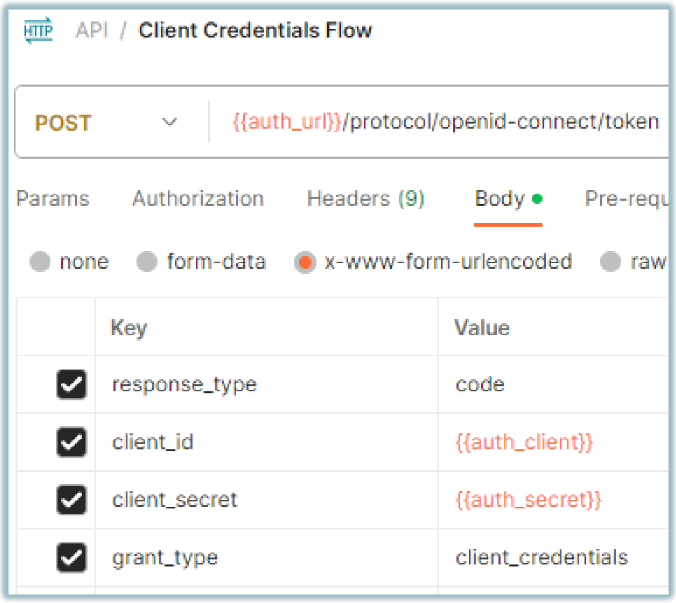
auth.c.ucnit.eu

	Variable	Type	Initial value
✓	auth_url	default	https://auth.c.ucnit.eu/realms/xOIDCx
✓	auth_client	default	Alice
✓	auth_secret	default	6zaZi58YBm24WgURUBtn5fbKVFGz8jsy
✓	auth_user	default	admin
✓	auth_password	default	admin
✓	auth_accesstoken	default	eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia

8.1 Client Credentials Flow

This flow does not use role-based authorization and does not utilize the User part.

This is how a Postman request would look like:



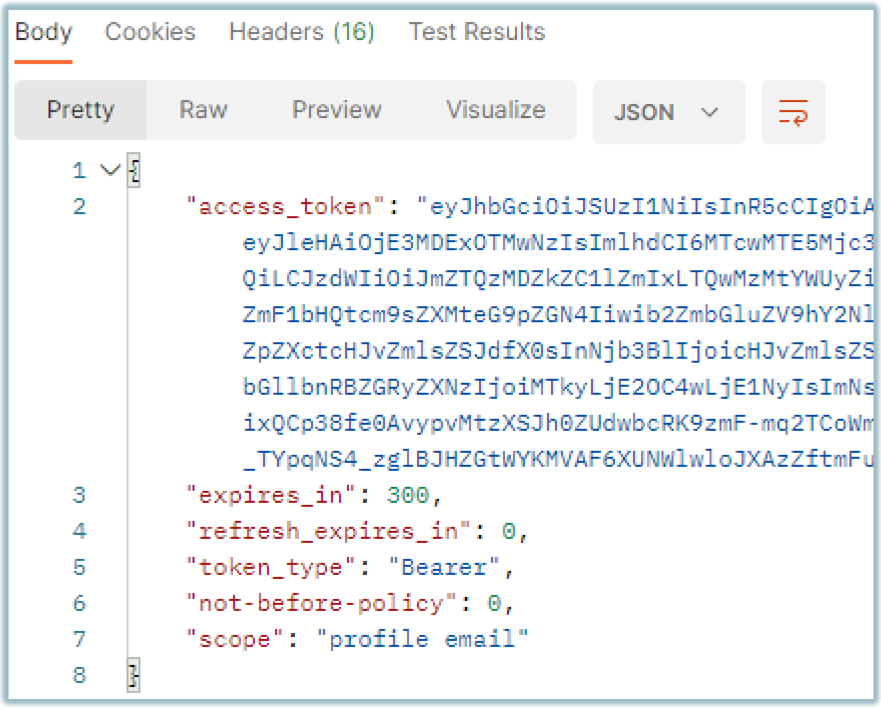
API / Client Credentials Flow

POST {{auth_url}}/protocol/openid-connect/token

Body: x-www-form-urlencoded

	Key	Value
✓	response_type	code
✓	client_id	{{auth_client}}
✓	client_secret	{{auth_secret}}
✓	grant_type	client_credentials

And the response will be:



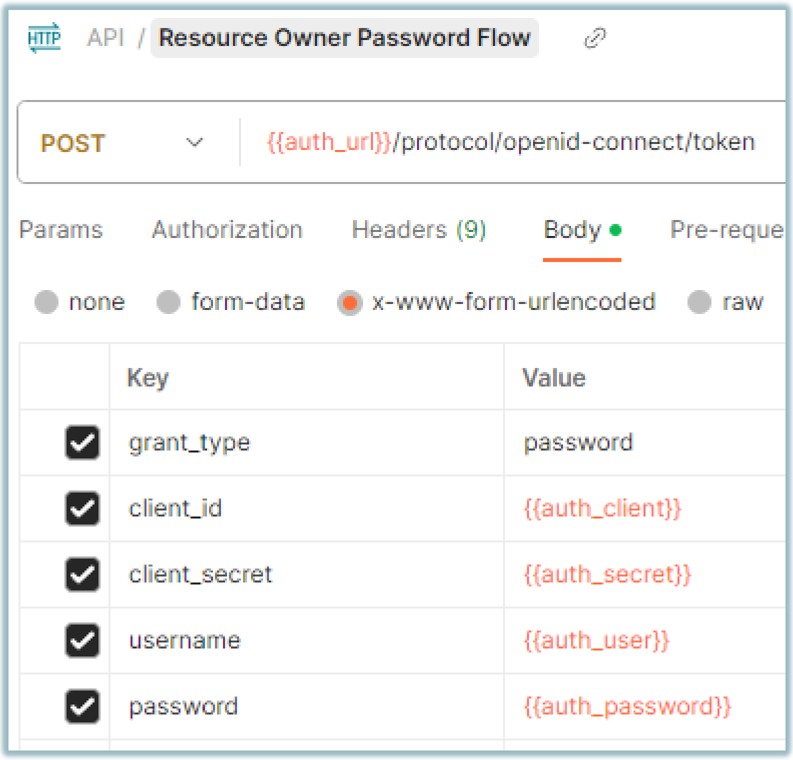
```
{
    "access_token": "eyJhbGciOiJSUzI1NiIsInR5cCIgOiA
        eyJleHAiOjE3MDExOTMwNzIsImlhdCI6MTcwMTE5Mjc3
        QiLCJzdWIiOiJmZTQzMDZkZC1lZmIxLTQwMzMtYWUyZi
        ZmF1bHQtcm9sZXMteG9pZGN4Iiwib2ZmbGluZV9hY2Nl
        ZpZXctcHJvZmlsZSJdfX0sInNjb3BlIjoicHJvZmlsZS
        bGllbnRBZGRyZXNzIjoiMTkyLjE2OC4wLjE1NyIsImNs
        ixQCp38fe0AvypvMtzXSJh0ZUdwbcRK9zmF-mq2TCoWm
        _TYpqNS4_zglBJHZGtWYKMVAF6XUNWlwloJXAzZftmFu
    "expires_in": 300,
    "refresh_expires_in": 0,
    "token_type": "Bearer",
    "not-before-policy": 0,
    "scope": "profile email"
}
```

If we decode the JWT access token, we get:

```
{
  "exp": 1701193072,
  "iat": 1701192772,
  "jti": "818cea31-7081-41e1-8ba5-506c954e399c",
  "iss": "https://auth.c.ucnit.eu/realms/xOIDCx",
  "aud": "account",
  "sub": "fe4306dd-efb1-4033-ae2f-c4d10bd10233",
  "typ": "Bearer",
  "azp": "Alice",
  "acr": "1",
  "allowed-origins": [
    "/*"
  ],
  "realm_access": {
    "roles": [
      "default-roles-xoidcx",
      "offline_access",
      "uma_authorization"
    ]
  },
  "resource_access": {
    "account": {
      "roles": [
        "manage-account",
        "manage-account-links",
        "view-profile"
      ]
    }
  },
  "scope": "profile email",
  "email_verified": false,
  "clientHost": "192.168.0.157",
  "preferred_username": "service-account-alice",
  "clientAddress": "192.168.0.157",
  "client_id": "Alice"
}
```

8.2 Resource Owner Password Flow

This flow additionally adds user/password information which allows for the return reply to add user-based info like roles. The Postman Request looks like this:


API / Resource Owner Password Flow

POST {{auth_url}}/protocol/openid-connect/token

Params | Authorization | Headers (9) | Body ● | Pre-reque

○ none ○ form-data ● x-www-form-urlencoded ○ raw

	Key	Value
✓	grant_type	password
✓	client_id	{{auth_client}}
✓	client_secret	{{auth_secret}}
✓	username	{{auth_user}}
✓	password	{{auth_password}}

Decoding the JWT access token, we get the additional info:

```
"scope": "profile email",
"sid": "fe540098-ab85-4b11-9922-bcf78b74e7a6",
"email_verified": true,
"roles": "admin",
"name": "Alice Wonderland",
"preferred_username": "admin",
"given_name": "Alice",
"family_name": "Wonderland",
"email": "alice@wonderland.nowhere"
```

8.3 Authorization Code Flow

For this we use a web browser:

https://auth.a.ucnit.eu/auth/realms/xOIDCx/protocol/openid-connect/auth?client_id=Alice&redirect_uri=https://demo.l2.ucnit.eu/redirect_URI

9. Repositories

In this version 3.0 of the documentation all sample code is secured in the repository OpenID-8.
https://github.com/goldfingyr/OpenID-8

10. Securing a Forms Application

This is still an open question, since the amount of software needed to symbiotically open the system browser to handle the login process is substantial.
The easier way to do this is to simply request the access token directly. This task is trivial. Study the Postman actions and implement similar behavior.

11. Securing an API

For this, we will use a Visual Studio 2022 scaffolding: ASP.NET Core Web API using .NET 8.0 as it is LTS.
There is too much code to show here so please refer to repository:
https://github.com/goldfingyr/OpenID-8/tree/main/WebAPI/WebAPI

To make the API and the MVC work, it must be ensured that the https ports are locked to port 8888 (API) and port 7777 (MVC). This is due to the required correspondence with OpenID. OpenID uses the RedirectURI

11.1 Required packages

- Microsoft.AspNet.WebApi.Core
- Microsoft.AspNetCore.Authentication.OpenIdConnect
- Microsoft.AspNetCore.Authentication.JwtBearer
- Swashbuckle.AspNetCore

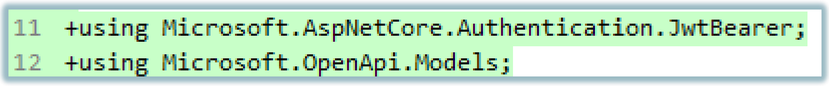
11.2 Environment

Several environment variables have been defined:

- OpenIDRealmURI: Base URI for authentication

11.3 Program.cs

11.3.1 Namespaces, line [11-12]



```
11  +using Microsoft.AspNetCore.Authentication.JwtBearer;
12  +using Microsoft.OpenApi.Models;
```

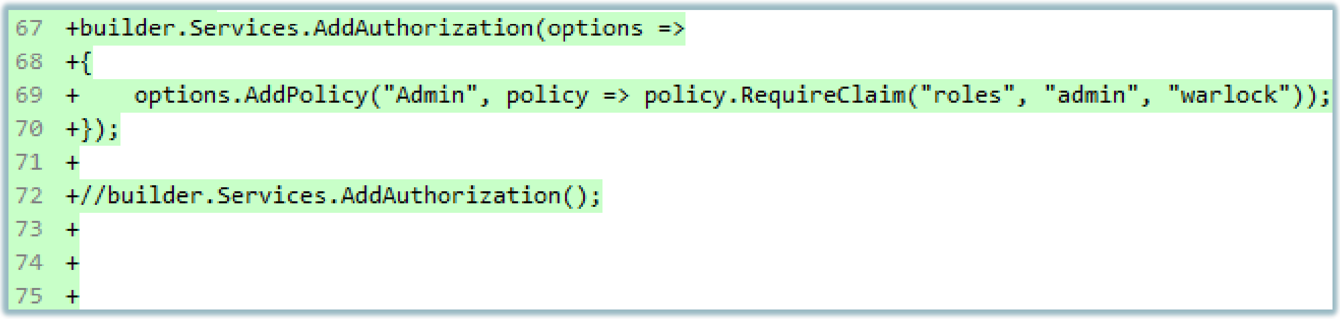
11.3.2 Authentication, line [17-76]

The API will never call the OpenID login as it receives the access token from the client (ex: MVC). Hence it only needs to know the identity of the OpenID Identity provider (Microsoft: Authority).

Line [20-68] deals with verifying the content of the received JWT. Line [48-66] was added as optional debug printing out the claims received in the event of a validated token.

11.3.3 Authorization, line [71-76]

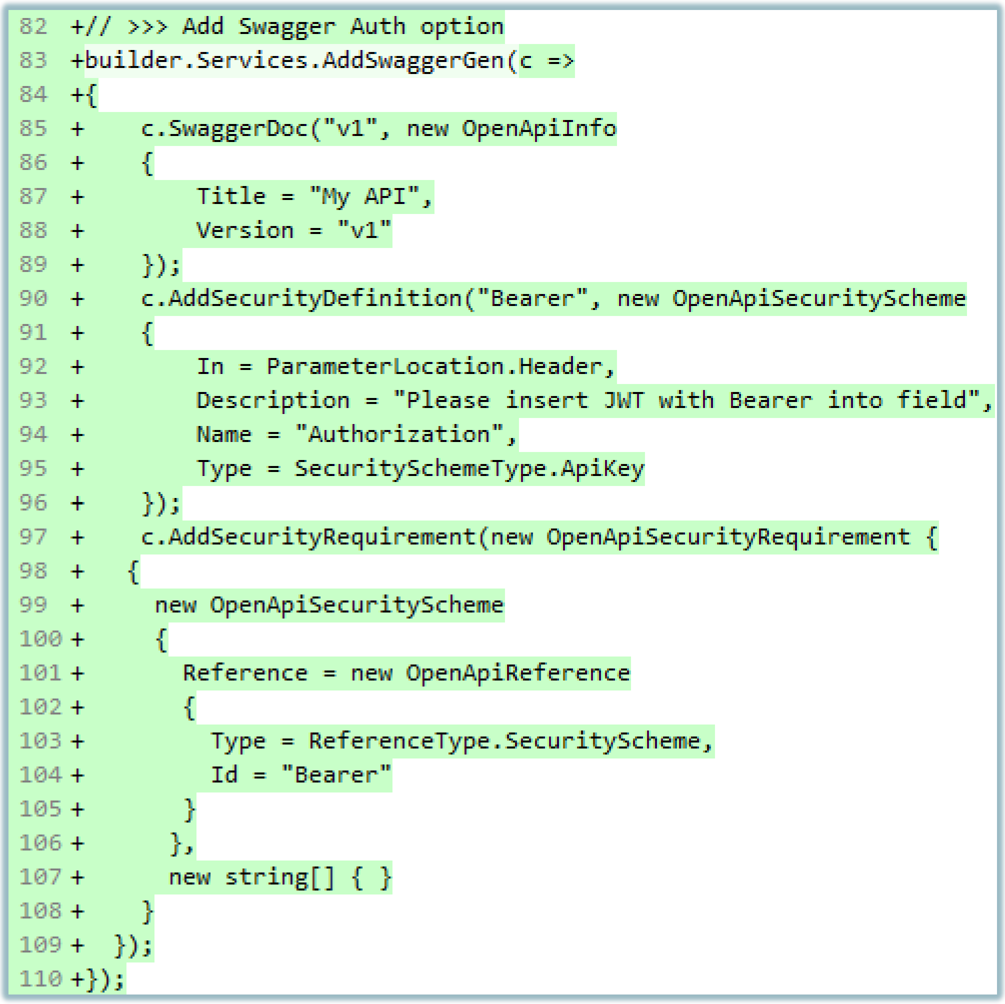
Role based authorization is optionally added as policies stating in line 69 that admin will be the policy-based role in case the “roles” claim shows either values of "admin" or "warlock".



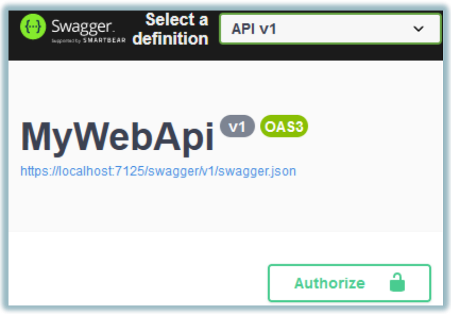
```
67  +builder.Services.AddAuthorization(options =>
68  +{
69  +    options.AddPolicy("Admin", policy => policy.RequireClaim("roles", "admin", "warlock"));
70  +});
71  +
72  +//builder.Services.AddAuthorization();
73  +
74  +
75  +
```

11.3.4 Open API (Swagger) interface line [88-117]

These lines optionally activate the Swagger Open API interface including the “Authorize” button which allows for adding the Access Token to the requests

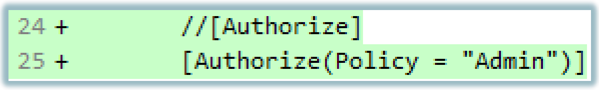


```
82  +// >>> Add Swagger Auth option
83  +builder.Services.AddSwaggerGen(c =>
84  +{
85  +    c.SwaggerDoc("v1", new OpenApiInfo
86  +    {
87  +        Title = "My API",
88  +        Version = "v1"
89  +    });
90  +    c.AddSecurityDefinition("Bearer", new OpenApiSecurityScheme
91  +    {
92  +        In = ParameterLocation.Header,
93  +        Description = "Please insert JWT with Bearer into field",
94  +        Name = "Authorization",
95  +        Type = SecuritySchemeType.ApiKey
96  +    });
97  +    c.AddSecurityRequirement(new OpenApiSecurityRequirement {
98  +   {
99  +     new OpenApiSecurityScheme
100 +     {
101 +       Reference = new OpenApiReference
102 +       {
103 +         Type = ReferenceType.SecurityScheme,
104 +         Id = "Bearer"
105 +       }
106 +      },
107 +      new string[] { }
108 +    }
109 +  });
110 +});
```



11.4 WeatherForecastController.cs

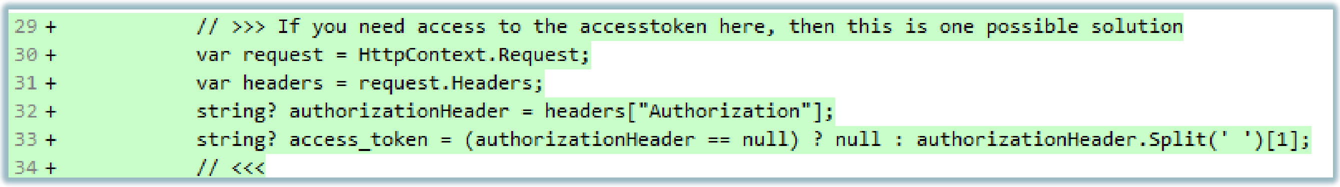
Depending on which authentication scheme is chosen, then one of the two define statements must be activated:



```
24 +        //[Authorize]
25 +        [Authorize(Policy = "Admin")]
```

11.4.1 Access_token line [27-31]

The access token itself may be acquired as shown in this snippet. Look to the MVC part for decoding of the JWT.



```
29 +            // >>> If you need access to the accesstoken here, then this is one possible solution
30 +            var request = HttpContext.Request;
31 +            var headers = request.Headers;
32 +            string? authorizationHeader = headers["Authorization"];
33 +            string? access_token = (authorizationHeader == null) ? null : authorizationHeader.Split(' ')[1];
34 +            // <<<
```

12. Securing an MVC

For this, we will use a Visual Studio 2022 scaffolding: ASP.NET Core Web API using .NET 8.0 as it is LTS.
There is too much code to show here so please refer to repository:
https://github.com/goldfingyr/OpenID-8/tree/main/WebAppMVC

To make the API and the MVC work, it must be ensured that the https ports are locked to port 8888 (API) and port 7777 (MVC).

Securing the MVC proved to be more difficult than anticipated. While there are plenty of examples to be found on the internet only a few will work. This is due to middleware incompatibility issues due to poor documentation.

Another complication is that this application will seek authentication and will have to carry this authentication to the calls it will be making to the API.

12.1 Required packages

- Microsoft.AspNetCore.Authentication.OpenIdConnect
- Microsoft.AspNetCore.Authentication.JwtBearer

12.2 Environment

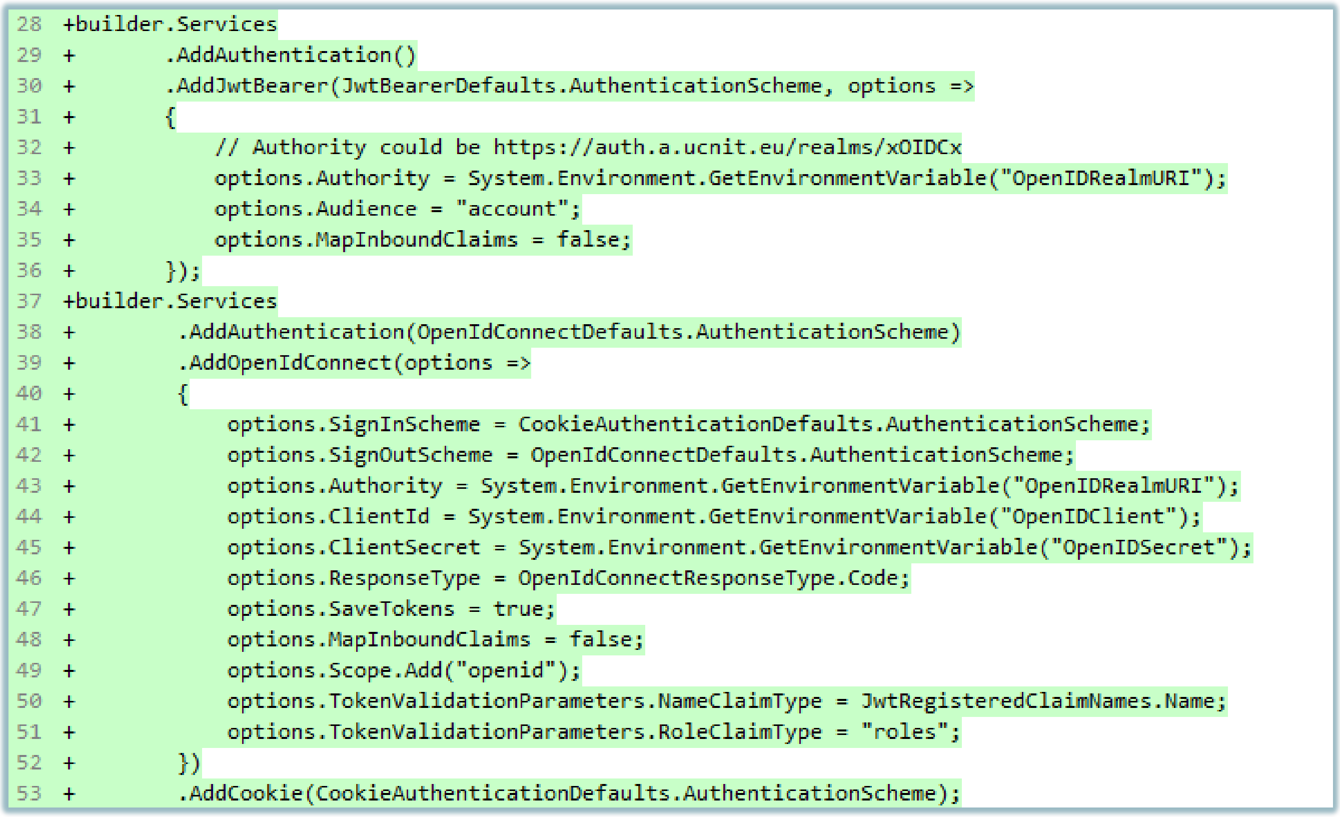
Several environment variables have been defined:

- OpenIDRealmURI: Base URI for authentication
- OpenIDRealm: The Realm to seek authentication for
- OpenIDClient: Name of the client to seek authentication for
- OpenIDSecret: OpenIDClient's secret

12.3 Program.cs

12.3.1 Authentication, line [28-53]

This section [28-36] defines that we are using token-based authentication of the JWT type.

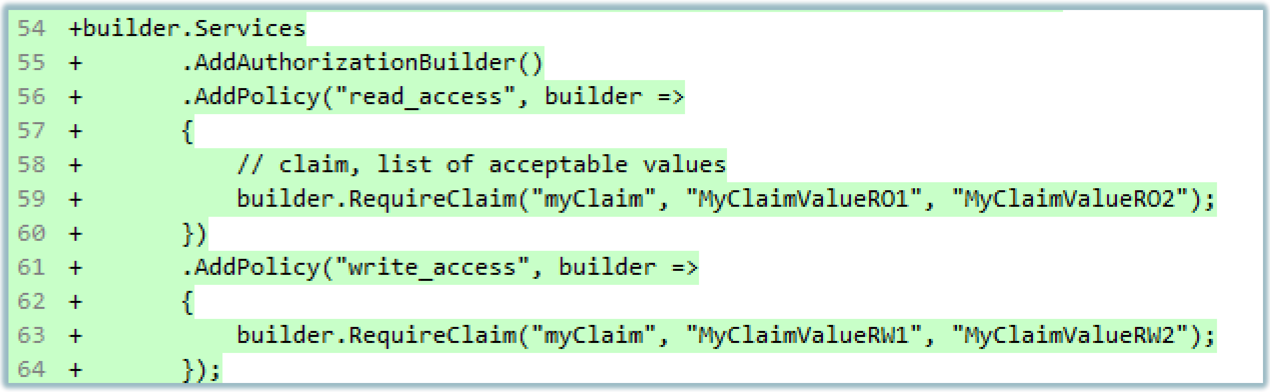


```
28 +builder.Services
29 +        .AddAuthentication()
30 +        .AddJwtBearer(JwtBearerDefaults.AuthenticationScheme, options =>
31 +        {
32 +            // Authority could be https://auth.a.ucnit.eu/realms/xOIDCx
33 +            options.Authority = System.Environment.GetEnvironmentVariable("OpenIDRealmURI");
34 +            options.Audience = "account";
35 +            options.MapInboundClaims = false;
36 +        });
37 +builder.Services
38 +         .AddAuthentication(OpenIdConnectDefaults.AuthenticationScheme)
39 +         .AddOpenIdConnect(options =>
40 +         {
41 +             options.SignInScheme = CookieAuthenticationDefaults.AuthenticationScheme;
42 +             options.SignOutScheme = OpenIdConnectDefaults.AuthenticationScheme;
43 +             options.Authority = System.Environment.GetEnvironmentVariable("OpenIDRealmURI");
44 +             options.ClientId = System.Environment.GetEnvironmentVariable("OpenIDClient");
45 +             options.ClientSecret = System.Environment.GetEnvironmentVariable("OpenIDSecret");
46 +             options.ResponseType = OpenIdConnectResponseType.Code;
47 +             options.SaveTokens = true;
48 +             options.MapInboundClaims = false;
49 +             options.Scope.Add("openid");
50 +             options.TokenValidationParameters.NameClaimType = JwtRegisteredClaimNames.Name;
51 +             options.TokenValidationParameters.RoleClaimType = "roles";
52 +         })
53 +         .AddCookie(CookieAuthenticationDefaults.AuthenticationScheme);
```

This section [37-53] sets up the authentication scheme defining the identity provider (KeyCloak). All values are fetched from the environment allowing this code eventually to be used in containers.
Additionally we attach the ability to handle the authentication by placing the tokens in cookies.

12.3.2 Authorization, line [26-36]

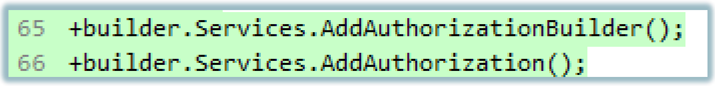
Providing policy based Authorization. This splits into two categories: “read_access”and “write_access”. “myClaim” will be the claim where roles are located. In this case it should be “roles”. The claim values list following are the roles accepted for these policies.



```
54 +builder.Services
55 +        .AddAuthorizationBuilder()
56 +        .AddPolicy("read_access", builder =>
57 +        {
58 +            // claim, list of acceptable values
59 +            builder.RequireClaim("myClaim", "MyClaimValueRO1", "MyClaimValueRO2");
60 +        })
61 +        .AddPolicy("write_access", builder =>
62 +        {
63 +            builder.RequireClaim("myClaim", "MyClaimValueRW1", "MyClaimValueRW2");
64 +        });
```

12.3.3 Authentication, line [63-64]

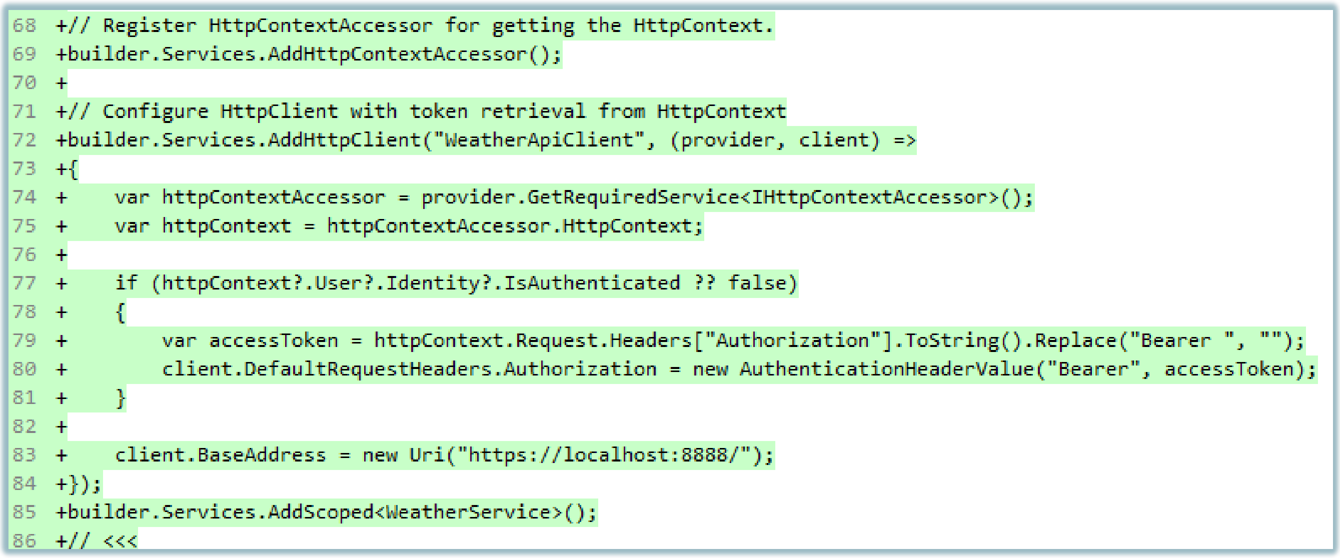
These lines merely activate the previous sections.



```
65 +builder.Services.AddAuthorizationBuilder();
66 +builder.Services.AddAuthorization();
```

12.3.4 Authentication, line [68-86]

The idea was to set up an authentication handler to make sure that all outgoing traffic would have the access_token attached. This failed somehow as the httpContextAccessor failed to retrieve the access_token. Hence it in this case is supplied in a later stage.



```
68 +// Register HttpContextAccessor for getting the HttpContext.
69 +builder.Services.AddHttpContextAccessor();
70 +
71 +// Configure HttpClient with token retrieval from HttpContext
72 +builder.Services.AddHttpClient("WeatherApiClient", (provider, client) =>
73 +{
74 +    var httpContextAccessor = provider.GetRequiredService<IHttpContextAccessor>();
75 +    var httpContext = httpContextAccessor.HttpContext;
76 +
77 +    if (httpContext?.User?.Identity?.IsAuthenticated ?? false)
78 +    {
79 +        var accessToken = httpContext.Request.Headers["Authorization"].ToString().Replace("Bearer ", "");
80 +        client.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", accessToken);
81 +    }
82 +
83 +    client.BaseAddress = new Uri("https://localhost:8888/");
84 +});
85 +builder.Services.AddScoped<WeatherService>();
86 +// <<<
```

12.4 WeatherService.cs

This business layer class takes care of the connection to the API server.
As mentioned before we must inject the access_token at this point due to that the httpContentAccessor fails to retrieve it.

12.5 AuthenticationHandler.cs

Currently this file is a non-functional relic. It was left for future use.

12.6 Views/Shared/_Layout.cshtml

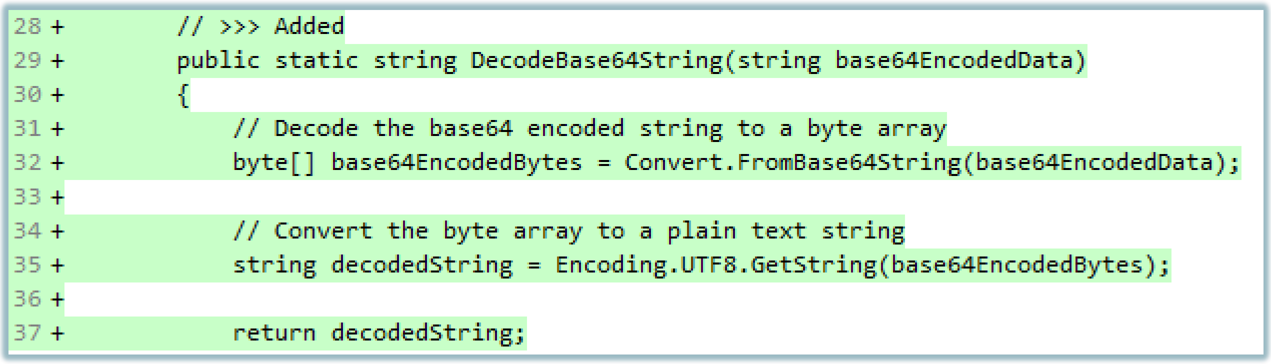
Provides the login button calling the login sequence.

12.7 HomeController.cs

Changes to this file are not very pretty, due to time constraints.

12.7.1 DecodeBase64String, line [28-37]

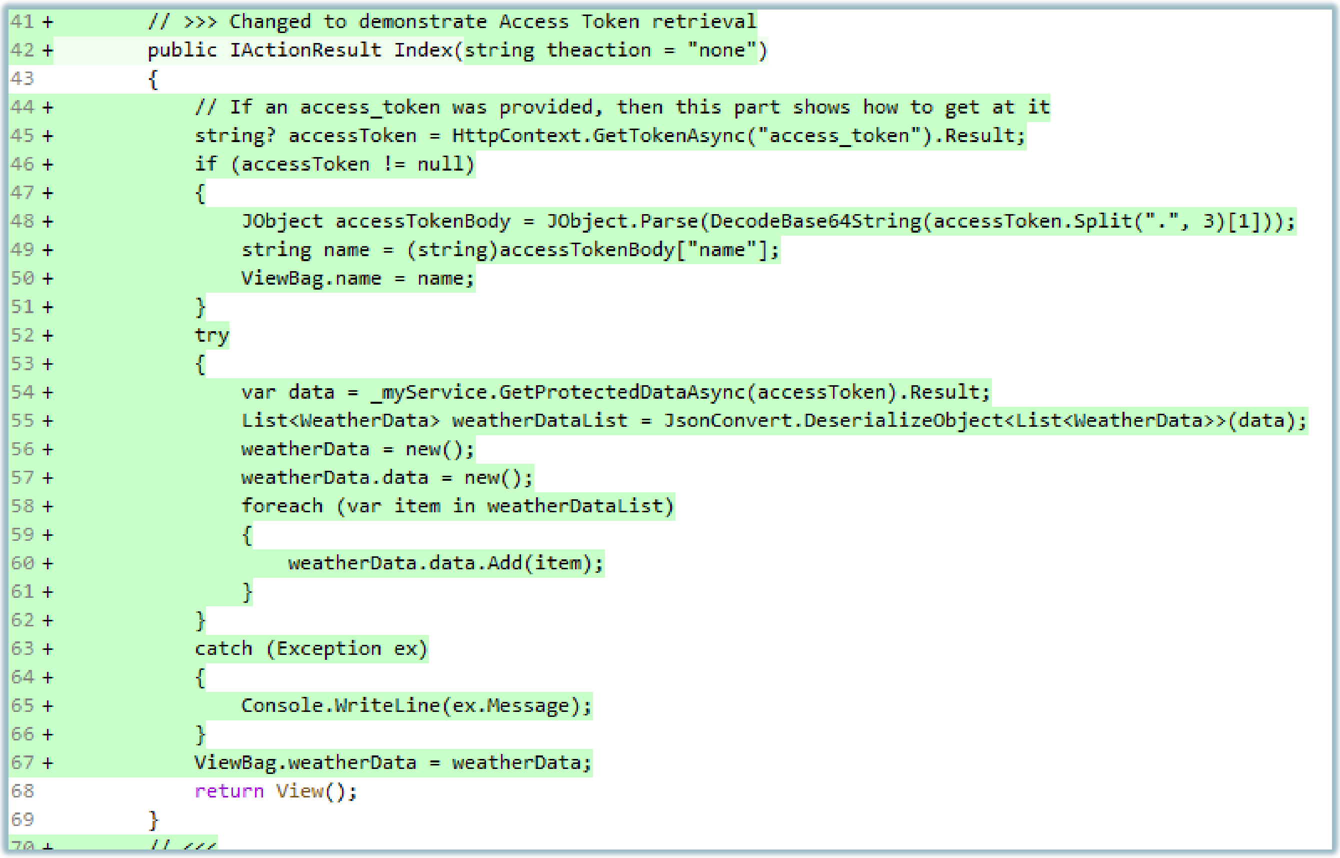
This method was added to make the JWT decoding method neath since the decoding involves conversion between byte array and string.



```
28 +        // >>> Added
29 +        public static string DecodeBase64String(string base64EncodedData)
30 +        {
31 +            // Decode the base64 encoded string to a byte array
32 +            byte[] base64EncodedBytes = Convert.FromBase64String(base64EncodedData);
33 +
34 +            // Convert the byte array to a plain text string
35 +            string decodedString = Encoding.UTF8.GetString(base64EncodedBytes);
36 +
37 +            return decodedString;
```

12.7.2 Extracting data from the access_token, line [42-69]

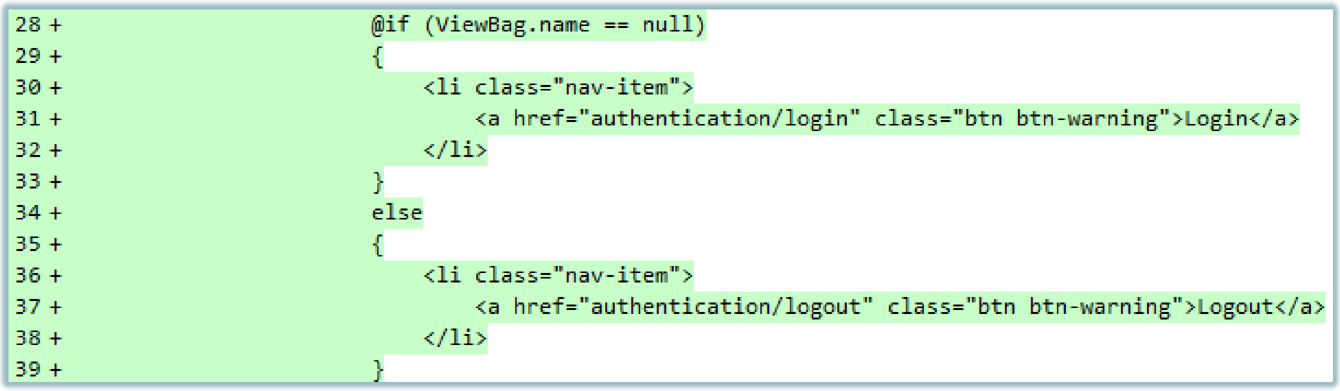
Obviously, we don't know if the access_token exists, hence the nullable string. If it does exist, then the JWT places the payload as the second part of the access_token which is then extracted in line 43.
Using LINQ based query line 44 extracts the value for the key "name" and adds it to the viewbag.



```
41 +        // >>> Changed to demonstrate Access Token retrieval
42 +        public IActionResult Index(string theaction = "none")
43          {
44 +            // If an access_token was provided, then this part shows how to get at it
45 +            string? accessToken = HttpContext.GetTokenAsync("access_token").Result;
46 +            if (accessToken != null)
47 +            {
48 +                JObject accessTokenBody = JObject.Parse(DecodeBase64String(accessToken.Split(".", 3)[1]));
49 +                string name = (string)accessTokenBody["name"];
50 +                ViewBag.name = name;
51 +            }
52 +            try
53 +            {
54 +                var data = _myService.GetProtectedDataAsync(accessToken).Result;
55 +                List<WeatherData> weatherDataList = JsonConvert.DeserializeObject<List<WeatherData>>(data);
56 +                weatherData = new();
57 +                weatherData.data = new();
58 +                foreach (var item in weatherDataList)
59 +                {
60 +                    weatherData.data.Add(item);
61 +                }
62 +            }
63 +            catch (Exception ex)
64 +            {
65 +                Console.WriteLine(ex.Message);
66 +            }
67 +            ViewBag.weatherData = weatherData;
68              return View();
69          }
70 +        // <<<
```

12.8 _Layout.cshtml

This file is processed early in the rendering process and produce a login and logout button depending on the presence of the “name” in the viewbag.



```
+                           @if (ViewBag.name == null)
+                           {
+                               <li class="nav-item">
+                                   <a href="authentication/login" class="btn btn-warning">Login</a>
+                               </li>
+                           }
+                           else
+                           {
+                               <li class="nav-item">
+                                   <a href="authentication/logout" class="btn btn-warning">Logout</a>
+                               </li>
+                           }
```

13. URLs and Keys for auth.a.ucnit.eu

Name	Value
OpenIDEndpointConfURI	https://auth.a.ucnit.eu/realms/xOIDCx/.well-known/openid-configuration
OpenIDRealm	xOIDCx
OpenIDRealmURI	https://auth.a.ucnit.eu/realms/xOIDCx
OpenIDTokenURI	https://auth.a.ucnit.eu/realms/xOIDCx/protocol/openid-connect/token
OpenIDClient	Alice
OpenIDSecret	JvDnso8O773pE9ENJdRhsrJd5pVD5Q86
OpenIDRedirectURI	https://localhost:7777/signin-oidc https://localhost:8888/signin-oidc
RS256_RSA_KEY	MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAxY8NkKFAszTBl1Iop-gbwj4LS4Bpl+i4/fCKHF2M3+gMkVAvVoB+9BWXUy-ofBbgL7NEgsMzLUH58ArPQh4NFE6Fm68bDP0Ho8HAPmM-spsO7jxXqfwIWBEeXlq37Zo/zTdgI8TpFDt029CMER3yLFAVk/hx9WB2GWF3vG98FEkOXeqy1c7WyR+486Da/QcsPqONfvpKejaWa1BK9/bisiqDyDiorfOFH0fcppTI-HyaajzDuA9oQm5jSh79jjp+pUHPf3/ej5SKyhhOQJLRoa7/pVhqk2y96hO-JqMfTX+LOmqce7JqFZzpLVGg90EEj/pIzGqftuzKhQJM7nv+2hiB8JQIDAQAB