*Asp.Net Core Identity API’s & Secure JWT \ Refresh HttpOnly Cookies within a Clean Architecture Solution*

|  |  |
| --- | --- |
| Bert O’Neill | |
| Department: | Code Project - IT\Software Support |
| Document Title: | Asp.Net Core Identity API & Secure JWT \ Refresh HttpOnly Cookies Within a Clean Architecture solution |
| Author: | Bert O’Neill |
| Date: | 03-Sept-2024 |
| Version | 1.0 |
| Email | BertONeill@Yahoo.com |
| Document Goals | Provide a complete reference for all the Asp.Net Core Identity API’s within .Net Core environment. But also securing them with JWT & Refresh Bearer Tokens using HttpOnly cookies, within a Clean Architecture solution. |

# Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| Date | Version | Description | Author |
| 03-Sept-2024 | 1.0 | Initial draft | Bert O’Neill |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Contents

[Revision History 1](#_Toc176134947)

[Introduction 3](#_Toc176134948)

[Purpose 3](#_Toc176134949)

[Scope 3](#_Toc176134950)

[Prerequisites 3](#_Toc176134951)

[Miscellaneous 4](#_Toc176134952)

[Acronym 4](#_Toc176134953)

# Introduction

This document a detailed overview on how to securely implement the various Asp.Net Core Identity Minimal API’s, using HttpOnly Cookies, incorporating JWT and Refresh Bearer tokens within a Clean Architecture approach.

## Purpose

Online you will find numerous articles, tutorials and YouTube videos detailing the various Identity APIs, but they are either disjointed (only detailing several API’s) or wanting you to subscribe to their patron site to gain access to their code!!!

I wanted to provide a one-stop-shop tutorial where you can pick and choose the API you are interested in learning more about, but also how to securely (HTTS) implement them using Bearer (JWT\Refresh) Tokens – and convey this within a Clean Architecture approach.

But also, to incorporate as many standard project components as possible (like Logging Middleware, Mappers, Fluent Validation, API Caching, Error Handling Middleware and the various Identity configurations, that need to be implemented for Asp.Net Core Identity to function smoothly).

## Scope

The scope of this document is to quickly convey the implementation and configuration steps, needed to securely incorporate the Identity APIs into your application – to authentication and authorization to your project APIs, like production code by the major multinationals).

# Prerequisites

* An understanding of [Asp.Net Core Identity API’s](https://learn.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-8.0&tabs=visual-studio)
* An understanding of [Clean Architecture](https://www.codeproject.com/Articles/5351235/Clean-Architecture-Incorporating-Repository-Patter)
* An understanding of [Minimal API’s](https://learn.microsoft.com/en-us/aspnet/core/tutorials/min-web-api?view=aspnetcore-8.0&tabs=visual-studio)
* An understanding of [Entity Framework Core](https://learn.microsoft.com/en-us/ef/) (EFC)
* An understanding of [EFC Data Migrations](https://learn.microsoft.com/en-us/ef/core/managing-schemas/migrations/managing?tabs=dotnet-core-cli)
* An understanding of [Microsoft’s SQL Server](https://learn.microsoft.com/en-us/sql/linux/new-to-sql-learning-resources?view=sql-server-ver16)
* An understanding of [Auto Mappers](https://automapper.org/)
* An understanding of [Fluent Validation](https://fluentvalidation.net/)
* An understanding of the [HttpOnly](https://owasp.org/www-community/HttpOnly) flag
* Install [Visual Studio 22 Community](https://visualstudio.microsoft.com/vs/community/)
* Install [SQL Server Developer](https://www.microsoft.com/en-us/sql-server/sql-server-downloads) and [SSMS](https://learn.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms?view=sql-server-ver16)

# Project Structure

Below, you can see that I have incorporated a Clean Architecture approach to implementing ASP Net Core Identity security APIs. The API layer has been designed using .Net8 Minimal APIs, *I am testing the API’s at this stage with Swagger only* – but a Blazor WASM (client) project has been included in the solution for completeness – which references a Shared project of DTO’s to communicate with the API layer efficiently.

A screenshot of a computer

Description automatically generated

# Explanation of Clean Code Approach for API Project (Program.cs)

The program.cs file is now the main entry point for your API layer, and this file can get very busy very quickly, with configuration settings and defining minimal API endpoints. Below is how my Program.cs looks – lean and clean:

// configure using extensions, to keep programs.cs lean

builder.Services.**AddServicesLogging**(builder);

builder.Services.**AddCORsServices**(builder.Configuration);

builder.Services.**AddServicesInitialSetup**(builder.Configuration);

builder.Services.**AddServicesJwtIdentity**(builder.Configuration);

builder.Services.**AddCustomServicesSwagger**(builder.Configuration);

var app = builder.Build();

// Register logging middleware

app.UseMiddleware<**MethodLoggingMiddleware**>();

// use extension methods to configure application middleware and custom endpoints

app.**ConfigureMiddleware**(app.Environment);

app.**ConfigureEndpoints**();

I have created static extensions to implement the configurations into separate files, making it easier to maintain, scale and debug.

# Setup And Component Configurations

## Register With Mailosaur to Send (Confirmation) Emails

Sign up for a free account email account on [Mailosaur](https://mailosaur.com/app/signup). You can then use their online inbox to view any (confirmation) emails you send to a user, and click the generated URL link within the body of the email or use the token\code you generated to complete a process step (for e.g. complete the 2FA step when a user logins).

**NB**: You can use your company’s SMTP settings, I am just using Mailosaur as it’s handy and easy to setup for POC’s at home.

A screenshot of a computer

Description automatically generated

Once you have registered, locate your account settings and record them in the Appsetting file:

"Mailosaur": {

"**ApiKey**": "1IlJ8eIY1waV0mZd7xPBqX5aIZxOGFrt", *// replace with your generated Api key*

"**ServerId**": "a3tuvq9f", *// replace with your Server Id*

"**From**": "use-save@a3tuvq9f.mailosaur.net" *// change this to your Mailosaur email address*

},

The *Server Key* and *From* details can be found on the top right of your screen.

The *API ID* can be found in Settings → Server API Keys

A screenshot of a computer

Description automatically generated

## Role Based Policy Authentication Against API’s

To enhance the security around certain API’s, I want to implement a Role Based Policy. This will ensure that only certain API’s can be called by certain role types.

Within the Identity schema, you will see a table called AspNetRoles, in here you can add in new roles appropriate to your system. When you register a user, you may want to give them a least privilege role, and they can later request a higher role from your Admin team.

For this tutorial I will create two users, one will have an *Admin* role and one will have a *User* role.

A screenshot of a computer

Description automatically generated

### Defining and Applying Role Based Policies

Below you can see that I am creating certain policies, based on the user’s role. I then apply them to the appropriate API.

// Add Custom Authorization Policies

services.AddAuthorization(options =>

{

options.AddPolicy("**AdminPolicy**", policy => policy.RequireRole("**Admin**"));

options.AddPolicy("**UserPolicy**", policy => policy.RequireRole("**User**"));

options.AddPolicy("**QAPolicy**", policy => policy.RequireRole("**Qa**"));

options.AddPolicy("**UserOrQaPolicy**", policy => policy.RequireRole("**User", "Qa**"));

});

Below, I am applying the *Admin Role Based Policy* to an Admin related API:

adminGroup.MapPost("/**Enable2faForUserAsync**", async Task<Results<Ok<string>, NotFound<string>>> (string userName, IAuthService authService) =>

{

Guard.Against.Empty(userName, "Username is missing");

var loginRegisterRefreshResponseDto = await authService.Enable2FactorAuthenticationForUserAsync(userName);

// was the email confirmation sent successfully

if (!loginRegisterRefreshResponseDto.IsStatus) return TypedResults.NotFound(loginRegisterRefreshResponseDto.Message);

else return TypedResults.Ok(loginRegisterRefreshResponseDto.Message);

})

.WithName("Enable2faForUser")

**.RequireAuthorization("AdminPolicy") // apply a security policy to API**

.WithMetadata(new AuthorizeAttribute { AuthenticationSchemes = JwtBearerDefaults.AuthenticationScheme })

.WithOpenApi(x => new OpenApiOperation(x)

{

Summary = "Admin enable 2FA for a user",

Description = "Admin can enable 2FA for a user",

Tags = new List<OpenApiTag> { new OpenApiTag { Name = "Admin - API Library" } }

});

## Setup Identity Database Using Data Migration

Create a new database using SQL Server Management Studio called *ShareMemories*

A screenshot of a computer

Description automatically generated

Update the connection string within your Appsettings file, to point to your new database:

"ConnectionStrings": {

"DefaultConnection": **"Server =(localdb)\\mssqllocaldb; Database= ShareMemories; Trusted\_Connection = True;"**

}

Open the *Package Manager Console****:***

A screenshot of a computer

Description automatically generated

Set your API project as the Start-Up project.

A screenshot of a computer

Description automatically generated

Set Infrastructure as the default project and run this command:

**Update-Database**

A screenshot of a computer

Description automatically generated

You may see warnings\errors, but check that your database has been created and seeded with some data – then you are good to go.

A screenshot of a computer

Description automatically generated

## Why Stateless Identity Session - No Caching

In previous companies I have seen variation son how security has been implemented, but the most common approach now is to use JWT Bearer, along with the HttpOnly flag (I’ll explain this later). But most often I have seen companies use a Bearer (access) Token, that is just a uniquely generated string. This token is then cached on the client side (Angular\React etc.) using LocalStorage, SessionStorage, IndexedDB or In-Memory Caching. But this approach has one fundamental flaw, they can be spied upon by erroneous (JavaScript) malware on the client’s environment.

Another scenario I have seen is that on the server side, the Bearer Token is cached as the key along with (JSON) information about the user. So, when an API call is made with the Bearer Token, it is extracted and the IMemory cache is queried for that user’s details (their username, role etc.), if the user is not present, then the Identity database is queried for the appropriate information. This too has a flaw; in that you must maintain the IMemory cache or if the server is rebooted the cache is cleared – and needs to be reseeded with new API requests from the client.

But, by using a Stateless approach, the JWT can contain all the information you need, no need to cache or hit the database – you just need to make sure it’s securely kept within the API request\response loop – this is done using the *HttpOnly flag*. No need to cache user information on the client or server.

By taking this approach, you must incorporate additional Identity APIs to allow a user to invalidate or revoke a JWT if they think it has been compromised (we will implement this too), and keep the JWT lifespan short lived.

**NB:** The lifespan of a JWT should be in minutes and the lifespan of a Refresh Token can be in days, this will accommodate a Single Sign-On approach in conjunction with the “Remember-Me” option.

## Why use the JWT and Refresh Tokens Approach to Security

### JWT

1. **Stateless Authentication:**

* **Self-Contained**: JWTs are self-contained tokens that carry all the necessary information about the user and their permissions (claims). This eliminates the need for server-side session storage, making it easier to scale your application horizontally.
* **Compact**: JWTs are compact and can be easily passed in HTTP headers or URLs, which makes them well-suited for web applications and APIs.

1. **Security:**
2. **Signed**: JWTs are digitally signed, usually with a secret key (HMAC) or a public/private key pair (RSA). This ensures that the token cannot be tampered with without invalidating the signature.
3. **Claims-Based**: JWTs can include claims that provide context about the user and their permissions, which can be verified and trusted by the server.
4. **Performance:**

* **No Server-Side Lookup**: Since JWTs are self-contained, the server can validate and parse the token without querying a database or session store, which improves performance.

### Refresh Tokens

JWTs, by design, have an expiration time, after which they become invalid. This expiration is crucial for security but poses a challenge for maintaining long-lived sessions. Here’s where refresh tokens come in:

1. **Short-Lived Access Tokens:**

* **Security**: JWTs are usually short-lived (e.g., 15 minutes to 1 hour) to minimize the impact of a compromised token. If a token is stolen, the attacker only has access for a limited time.
* **Ease of Revocation**: Short-lived tokens are easier to manage because they automatically expire, reducing the need for server-side revocation.

1. **Long-Lived Sessions with Refresh Tokens:**

* **Persistent Login**: Refresh tokens are typically long-lived (e.g., days, weeks, or even months) and are used to obtain new access tokens when the current JWT expires. This allows users to maintain their sessions without needing to log in repeatedly.
* **Less Frequent Authentication**: By using refresh tokens, you reduce the frequency of requiring the user to re-authenticate, providing a smoother user experience.

1. **Improved Security with Refresh Tokens:**

* **Stored Securely**: Refresh tokens should be stored securely on the client side (e.g., in HttpOnly cookies) to prevent XSS (Cross-Site Scripting) attacks.
* **Server-Side Validation**: Unlike JWTs, refresh tokens are often stored and validated on the server side. This allows for revocation, monitoring, and additional security checks.
* **Rotation**: When a refresh token is used to obtain a new access token, a new refresh token is often issued (rotation). This reduces the risk of token replay attacks, where a stolen refresh token is used multiple times.

## Why Use HttpOnly Flag

The HttpOnly flag is an important security feature for cookies in web applications. When a cookie has the HttpOnly flag set, it cannot be accessed or manipulated via client-side JavaScript. This helps protect the cookie from being stolen through Cross-Site Scripting (XSS) attacks.

1. **Preventing XSS Attacks:**

* **XSS Attacks**: In an XSS attack, an attacker injects malicious scripts into a web page viewed by other users. If a cookie storing sensitive information (like a session ID or authentication token) is accessible via JavaScript, the attacker can easily steal it using a script and then impersonate the user.
* **HttpOnly Protection**: By setting the HttpOnly flag on a cookie, you ensure that the cookie is not accessible via document.cookie in JavaScript, making it much harder for an attacker to steal it via XSS.

1. **Enhanced Security for Sensitive Cookies:**

* **Session Cookies**: Cookies that store session identifiers or tokens that authenticate users should always have the HttpOnly flag set to prevent unauthorized access.
* **Auth Tokens**: If you're storing tokens (like JWTs) in cookies, setting the HttpOnly flag ensures that the token cannot be accessed or modified by any JavaScript running on the page.

1. **Mitigating the Impact of Other Vulnerabilities:**

* Even if your application has other vulnerabilities, such as unpatched XSS flaws, the HttpOnly flag can limit the damage that can be done by preventing access to critical cookies from client-side scripts.

It is configured in a couple of places, initially when you are configuring the Cookie Options:

// add cookie settings

services.ConfigureApplicationCookie(options =>

{

options.LoginPath = "/LoginAsync"; // Set your login path

options.LogoutPath = "/LogoutAsync"; // Set your logout path

options.SlidingExpiration = true;

options.ExpireTimeSpan = TimeSpan.FromDays(1);

**options.Cookie.HttpOnly = true;**

options.Cookie.SecurePolicy = CookieSecurePolicy.Always;

});

And when you are refreshing or revoking the tokens, so the client has new tokens associated with its API calls going forward:

// Set the JWT as a HttpOnly cookie

**cookieOptionsJWT** = new CookieOptions

{

**HttpOnly = true,**

IsEssential = true,

Secure = true, // Ensures the cookie is sent over HTTPS

SameSite = SameSiteMode.Strict, // Helps mitigate CSRF attacks

Expires = JwtTokenExpire

};

// Set the Refresh Token as a HttpOnly cookie

**cookieOptionsRefreshJWT** = new CookieOptions

{

**HttpOnly = true,**

IsEssential = true,

Secure = true, // Ensures the cookie is sent over HTTPS

SameSite = SameSiteMode.Strict, // Helps mitigate CSRF attacks

Expires = JwtRefreshTokenExpire

};

## Performance With API Caching

This not necessarily JWT related, but I have included it for completeness, as it’s a great way to improve the response times for your API (a link to my previous [blog on caching](https://www.codeproject.com/Articles/5385070/Using-IDistributed-Cache-with-EF-Core)):

API caching is a powerful technique to improve the performance of your web applications or services. By storing responses temporarily, you can reduce the load on your servers, decrease latency, and improve the overall user experience. Below are key aspects of how API caching enhances performance, along with best practices and considerations.

1. **Reduced Latency**

* **Faster Responses**: Cached responses are delivered directly from the cache, eliminating the need to process the request on the server. This leads to faster response times, especially for APIs with heavy processing logic or frequent database access.

1. **Lower Server Load**

* **Reduced Processing**: Caching reduces the number of requests that hit your backend servers, allowing them to handle more requests or operate more efficiently. This is especially useful during traffic spikes.
* **Database Load**: If your API frequently queries a database, caching can minimize database load by serving cached results for repetitive queries.

1. **Improved Scalability**

* **Horizontal Scaling**: By reducing the number of requests that need full processing, you can scale your application more effectively, often reducing the need for additional server resources.
* **Better Resource Utilization**: Server resources, like CPU and memory, are better utilized because repetitive work is avoided.

1. **Enhanced User Experience**

* **Consistent Performance**: Users experience faster, more consistent response times, which leads to higher satisfaction, especially in applications requiring real-time interaction or where low latency is crucial.

See the second last line in this API, here we are caching by Id.

group.MapPost("/InsertPictureAsync", async (HttpContext context, ShareMemories.Domain.Entities.Picture picture, IPictureService pictureService) =>

{

// DTO validated before this line, using "PictureValidator"

var insertedPicture = await pictureService.InsertPictureAsync(picture);

// Return 200 OK with the inserted picture or 404 Not Found if insertion fails

return insertedPicture.Id > 0

? Results.Ok(insertedPicture)

: Results.NotFound("Not able to insert picture.");

})

.WithName("InsertPictureAsync")

.WithOpenApi(x => new OpenApiOperation(x)

{

Summary = "Insert a new picture",

Description = "Adds a new picture to database",

Tags = new List<OpenApiTag> { new() { Name = "Pictures API Library" } }

})

**.CacheOutput(x => x.Tag("PictureById"))**

.AddEndpointFilter<GenericValidationFilter<PictureValidator, ShareMemories.Domain.Entities.Picture>>(); // apply fluent validation to DTO model from client and pass back broken rules

## API Clean Code - Using Endpoint Extracted Extensions

To improve the maintainability of the various API endpoints, I have extracted the APIs into their own static class and eventually referenced them from within the Programs.cs file.

A screenshot of a computer

Description automatically generated

For e.g. I have moved all the Admin related APIs into their own file called **AdminEndpoints.cs**, an example of **Enable2faForUserAsync** endpoint below within the static class and method:

**public static class AdminEndpoints**

{

**public static void MapAdminEndpoints(this IEndpointRouteBuilder routes)**

{

var adminGroup = routes.MapGroup("adminGroup").WithOpenApi();

adminGroup.MapPost("/**Enable2faForUserAsync**", async Task<Results<Ok<string>, NotFound<string>>> (string userName, IAuthService authService) =>

{

Guard.Against.Empty(userName, "Username is missing");

var loginRegisterRefreshResponseDto = await authService.Enable2FactorAuthenticationForUserAsync(userName);

// was the email confirmation sent successfully

if (!loginRegisterRefreshResponseDto.IsStatus) return TypedResults.NotFound(loginRegisterRefreshResponseDto.Message);

else return TypedResults.Ok(loginRegisterRefreshResponseDto.Message);

})

.WithName("Enable2faForUser")

.RequireAuthorization("AdminPolicy") // apply a security policy to API's and a default Bearer Scheme

.WithMetadata(new AuthorizeAttribute { AuthenticationSchemes = JwtBearerDefaults.AuthenticationScheme })

.WithOpenApi(x => new OpenApiOperation(x)

{

Summary = "Admin enable 2FA for a user",

Description = "Admin can enable 2FA for a user",

Tags = new List<OpenApiTag> { new OpenApiTag { Name = "Admin - API Library" } }

});

}

}

Then within the middleware class I reference the static methods containing the endpoints:

public static void **ConfigureEndpoints**(this WebApplication app)

{

// Register Minimal API Endpoints

**app.MapAdminEndpoints();**

app.MapLockoutEndpoints();

app.MapLoginRegisterEndpoints();

app.MapPasswordEndpoints();

app.MapTokenEndpoints();

app.MapTwoFAEndpoints();

app.MapPictureEndpoints();

app.MapVideoEndpoints();

}

And then within the Programs.cs file I just reference the middleware class, to keep the class as clean as possible:

try

{

// some code removed for brevity

// use extension methods to configure application middleware and custom endpoints

app.ConfigureMiddleware(app.Environment);

app.**ConfigureEndpoints**();

app.Run();

}

catch (Exception exception)

{

Log.Logger.Error(exception,"Stopped program because of exception");

throw;

}

# APIs

## Configurations

### Extend Identity Model

The default Identity model may suffice your needs. But within a production solution, your company may have fields that you wish to add to the model and thus the database table *AspNetUsers* schema.

Below I have extended Identity model with fields I want to track (most importantly the Refresh Token and Refresh Token Expiry fields) – but also some simple fields like DateCreated and LastUpdated.

I create a new model called **ExtendIdentityUser**, that inherits from Identity and add my new properties there.

public class ExtendIdentityUser : **IdentityUser**

{

public string FirstName { get; set; }

public string LastName { get; set; }

public DateOnly DateOfBirth { get; set; }

public bool? IsArchived { get; set; } = false;

public DateTime? LastUpdated { get; set; }

public DateTime? CreatedDate { get; set; } = DateTime.Now;

public string? RefreshToken { get; set; }

public DateTime? RefreshTokenExpiry { get; set; }

}

**NB:** There is always the option to create a new model\table that will hold these values, and join the tables based on Identity Id – but for simplicity I am adding them to the *AspNetUsers* table.

Then inside your EFC DBContext class, you will extend **ExtendIdentityUser** instead of **IdentityUser**.

**public partial class ShareMemoriesContext : IdentityDbContext<ExtendIdentityUser>**

**NB:** When you create theData Migration file, it will be updated appropriately with the new model properties:

migrationBuilder.CreateTable(

name: "AspNetUsers",

columns: table => new

{

Id = table.Column<string>(type: "nvarchar(450)", nullable: false),

**FirstName = table.Column<string>(type: "nvarchar(max)", nullable: false),**

**LastName = table.Column<string>(type: "nvarchar(max)", nullable: false),**

**DateOfBirth = table.Column<DateOnly>(type: "date", nullable: false),**

**IsArchived = table.Column<bool>(type: "bit", nullable: true),**

**LastUpdated = table.Column<DateTime>(type: "datetime2", nullable: true),**

**CreatedDate = table.Column<DateTime>(type: "datetime2", nullable: true),**

**RefreshToken = table.Column<string>(type: "nvarchar(max)", nullable: true),**

**RefreshTokenExpiry = table.Column<DateTime>(type: "datetime2", nullable: true),**

UserName = table.Column<string>(type: "nvarchar(256)", maxLength: 256, nullable: true),

NormalizedUserName = table.Column<string>(type: "nvarchar(256)", maxLength: 256, nullable: true),

Email = table.Column<string>(type: "nvarchar(256)", maxLength: 256, nullable: true),

NormalizedEmail = table.Column<string>(type: "nvarchar(256)", maxLength: 256, nullable: true),

EmailConfirmed = table.Column<bool>(type: "bit", nullable: false),

PasswordHash = table.Column<string>(type: "nvarchar(max)", nullable: true),

SecurityStamp = table.Column<string>(type: "nvarchar(max)", nullable: true),

ConcurrencyStamp = table.Column<string>(type: "nvarchar(max)", nullable: true),

PhoneNumber = table.Column<string>(type: "nvarchar(max)", nullable: true),

PhoneNumberConfirmed = table.Column<bool>(type: "bit", nullable: false),

TwoFactorEnabled = table.Column<bool>(type: "bit", nullable: false),

LockoutEnd = table.Column<DateTimeOffset>(type: "datetimeoffset", nullable: true),

LockoutEnabled = table.Column<bool>(type: "bit", nullable: false),

AccessFailedCount = table.Column<int>(type: "int", nullable: false)

},

constraints: table =>

{

table.PrimaryKey("PK\_AspNetUsers", x => x.Id);

});

### JWT

Below is how to configure your JWT settings –all the code can be found within the attached solution:

#### Bearer JWT Options

#### Bearer JWT Authentication

### Identity Options

#### Password

#### Lockout

#### Default provider

#### Confirmation Email When Registering

### 2FA Token Timespan

### Authorization Policy’s

### Capture JWT Bearer in Pipeline

### Cookie Options

#### HttpOnly

#### Remember Me Timespan

### CORS Policy

### API Output Caching

### Swagger Setup with JWT Security Requirement

## List of APIs Implemented and Scenarios on Usage

Below, is a list of the Identity API’s that I have implemented – basically everything you will need to get your web site’s API’s (entry point) secure and authenticated. Some of the API’s are for administrator level security (restricting usage to Admin roles), whereas others implement a User role (a logged in User can call) and some have no authorization (as they can be called by a user when they are not logged in, for e.g. request your account to be unlocked – sends a *self-service* email to the user’s email account)

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated  
A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

## Register | Login | Logout | Confirm Registration | Verify Registration | Resend Registration Confirm

## Verify 2FA | Enable 2FA | Disable 2FA | Resend 2FA

## JWT & Refresh Bearer - Revoke Tokens | Refresh Tokens (IMemory Cache)

## Remember Me Option (Persist user after browser closed – not Logout)

## Update User Details

## Request Password Update | Update Password

## Lockout | Unlock Account

# Miscellaneous

## Acronym

|  |  |
| --- | --- |
| **Abbreviation** | **Meaning** |
| EFC | Entity Framework Core |
| SQL | Structured Query Language |
| API | Application Programming Interface |
|  |  |
|  |  |