*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](#_Toc176422400)

[Introduction 5](#_Toc176422401)

[Purpose 5](#_Toc176422402)

[Scope 5](#_Toc176422403)

[Prerequisites 5](#_Toc176422404)

[Project Structure 6](#_Toc176422405)

[Explanation of Clean Code Approach for API Project (Program.cs) 6](#_Toc176422406)

[Setup And Component Configurations 7](#_Toc176422407)

[Register With Mailosaur to Send (Confirmation) Emails 7](#_Toc176422408)

[Role Based Policy Authentication Against API’s 7](#_Toc176422409)

[Defining and Applying Role Based Policies 8](#_Toc176422410)

[Setup Identity Database Using Data Migration 9](#_Toc176422411)

[Why Stateless Identity Session - No Caching 10](#_Toc176422412)

[Why use the JWT and Refresh Tokens Approach to Security 11](#_Toc176422413)

[JWT 11](#_Toc176422414)

[Refresh Tokens 12](#_Toc176422415)

[Why Use HttpOnly Flag 12](#_Toc176422416)

[Performance With API Caching 14](#_Toc176422417)

[API Clean Code – Extracting Endpoint into Separate Extension Files 15](#_Toc176422418)

[APIs 16](#_Toc176422419)

[Configurations 16](#_Toc176422420)

[Extend Identity Model 16](#_Toc176422421)

[Setting JWT (Authentication, Options and MessageReceivedContext) 18](#_Toc176422422)

[Identity Options 20](#_Toc176422423)

[2FA Token Timespan 21](#_Toc176422424)

[Authorization Policy’s 21](#_Toc176422425)

[Cookie Options 21](#_Toc176422426)

[CORS Policy 22](#_Toc176422427)

[API Output Caching 23](#_Toc176422428)

[Swagger Setup with JWT Security Requirement 23](#_Toc176422429)

[Mailosaur 24](#_Toc176422430)

[List of APIs Implemented and Scenarios on Usage 25](#_Toc176422431)

[Register New User 27](#_Toc176422432)

[API Method 27](#_Toc176422433)

[Service Method 27](#_Toc176422434)

[Swagger UI 29](#_Toc176422435)

[Login 31](#_Toc176422436)

[API Method 31](#_Toc176422437)

[Service Method 32](#_Toc176422438)

[Swagger UI 34](#_Toc176422439)

[Logout 35](#_Toc176422440)

[Api Method 35](#_Toc176422441)

[Service Method 35](#_Toc176422442)

[Swagger UI 36](#_Toc176422443)

[Confirm Email Registration 37](#_Toc176422444)

[Api Method 38](#_Toc176422445)

[Service Method 39](#_Toc176422446)

[Resend Registration Confirm 39](#_Toc176422447)

[Api Method 39](#_Toc176422448)

[Service Method 40](#_Toc176422449)

[Verify 2FA 41](#_Toc176422450)

[Api Method 42](#_Toc176422451)

[Service Method 42](#_Toc176422452)

[Enable 2FA (Admin) 43](#_Toc176422453)

[Api Method 44](#_Toc176422454)

[Service Method 45](#_Toc176422455)

[Disable 2FA (Admin) 46](#_Toc176422456)

[Api Method 46](#_Toc176422457)

[Service Method 47](#_Toc176422458)

[Resend 2FA 47](#_Toc176422459)

[Api Method 48](#_Toc176422460)

[Service Method 48](#_Toc176422461)

[Revoke Tokens (IMemory Cache) 49](#_Toc176422462)

[Api Method 49](#_Toc176422463)

[Service Method 50](#_Toc176422464)

[Refresh Tokens 52](#_Toc176422465)

[Api Method 52](#_Toc176422466)

[Service Method 53](#_Toc176422467)

[Remember Me Option (Persist user after browser closed – not Logout) 55](#_Toc176422468)

[Api Method 55](#_Toc176422469)

[Service Method 55](#_Toc176422470)

[Swagger Test Data 56](#_Toc176422471)

[Update User Details 56](#_Toc176422472)

[Api Method 56](#_Toc176422473)

[Service Method 57](#_Toc176422474)

[Swagger Test Data 59](#_Toc176422475)

[View User details 59](#_Toc176422476)

[Api Method 59](#_Toc176422477)

[Service Method 60](#_Toc176422478)

[Request Password Reset 60](#_Toc176422479)

[Api Method 61](#_Toc176422480)

[Service Method 61](#_Toc176422481)

[Verify Password Reset 63](#_Toc176422482)

[Api Method 64](#_Toc176422483)

[Service Method 64](#_Toc176422484)

[Lockout Account (Admin) 65](#_Toc176422485)

[Api Method 66](#_Toc176422486)

[Service Method 67](#_Toc176422487)

[Request Unlock Account (User) 68](#_Toc176422488)

[Api Method 68](#_Toc176422489)

[Service Method 69](#_Toc176422490)

[Unlock Account (Admin) 69](#_Toc176422491)

[Api Method 70](#_Toc176422492)

[Service Method 71](#_Toc176422493)

[Api Method 71](#_Toc176422494)

[Service Method 72](#_Toc176422495)

[Miscellaneous 73](#_Toc176422496)

[Acronym 73](#_Toc176422497)

# 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 – Extracting Endpoint into Separate Extension Files

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();

}

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);

});

### Setting JWT (Authentication, Options and MessageReceivedContext)

Update you Appsetting file to use the correct <URL:Port> number:

"Jwt": {

"Key": "6AD2EFDE-AB2C-4841-A05E-7045C855BA22",

"Issuer": "https://localhost:**7273**/",

"Audience": "https://localhost:**7273**/",

"REFRESH\_TOKEN\_EXPIRE\_DAYS": "10",

"JWT\_TOKEN\_EXPIRE\_MINS": "30"

},

Below is how to configure your JWT Options and Authentications:

// Add Bearer JWT Authentication

services.AddAuthentication(options =>

{

**options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;**

**options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;**

})

// Add Bearer JWT Options

.AddJwtBearer(options =>

{

options.TokenValidationParameters = new TokenValidationParameters

{

**ValidateActor = true,**

**ValidateIssuer = true,**

**ValidateAudience = true,**

**RequireExpirationTime = true,**

**ValidateIssuerSigningKey = true,**

**ValidIssuer = configuration.GetSection("Jwt:Issuer").Value,**

**ValidAudience = configuration.GetSection("Jwt:Audience").Value,**

**IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(configuration.GetSection("Jwt:Key").Value!)),**

};

Capture the JWT token form the pipeline and append it to the current HttpContext, that is used later in the code to extract the current User’s JWT:

// Capture JWT Bearer in the pipeline and assign it to MessageReceivedContext

options.Events = new JwtBearerEvents

{

**OnMessageReceived** = context =>

{

if (context.Request.Cookies.ContainsKey("jwtToken")) // this cookie is assigned after "LoginAsync" endpoint called

{

**context.Token = context.Request.Cookies["jwtToken"];**

}

else

{

Log.Logger.Information("JWT token missing");

context.Fail("JWT token missing.");

}

return Task.CompletedTask;

},

OnTokenValidated = context =>

{

// This event is triggered when a token is successfully validated.

return Task.CompletedTask;

},

OnAuthenticationFailed = context =>

{

// This event is triggered when authentication fails.

Log.Logger.Error("An issue extracting JWT Bearer form HttpOnly Cookie");

return Task.CompletedTask;

}

};

});

When a User has successfully registered, confirmed their email address and then logged in. You can view the JWT token in the browser.

A screenshot of a computer

Description automatically generated

If you then manually copy the JWT token and go to the <Jwt.io> web site, enter the value, you can see what information is saved within the token – thus no need for caching data on the client or server - Stateless.

A screenshot of a computer

Description automatically generated

### Identity Options

Below are the configurations to set the Password, Confirmation Email, Lockout, Default AuthenticatorTokenProvider (currently using Email) and making the User enter a Unique Email when registering options.

// Register Identity services

services.AddIdentity<ExtendIdentityUser, IdentityRole>(options =>

{

**// Enforce password rules - For example: P@ssw0rd**

**options.Password.RequiredLength = 8;**

**options.Password.RequireNonAlphanumeric = true; // For example: !"£$%^**

**options.Password.RequireDigit = true;**

**options.Password.RequireLowercase = true;**

**options.Password.RequireUppercase = true;**

// Confirm Email options

**options.SignIn.RequireConfirmedEmail = true;** // set to false if user is not to confirm their email address when registering

// Lockout settings.

**options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(lockoutLifeSpanMinutes);**

**options.Lockout.MaxFailedAccessAttempts = lockoutAttempts;**

**options.Lockout.AllowedForNewUsers = true;**

// User settings.

**options.User.AllowedUserNameCharacters = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789-.\_@+";**

**options.User.RequireUniqueEmail = true;**

// confirm Token settings

**options.Tokens.AuthenticatorTokenProvider = TokenOptions.DefaultEmailProvider;**  // provider for 2FA

**options.Tokens.EmailConfirmationTokenProvider = TokenOptions.DefaultEmailProvider;** // provider for email confirmation

**options.Tokens.PasswordResetTokenProvider = TokenOptions.DefaultEmailProvider;** // provider for password reset

})

.AddEntityFrameworkStores<ShareMemoriesContext>()

.AddApiEndpoints()

.AddDefaultTokenProviders();

### 2FA Token Timespan

Configure the timeout for a token before it expires. Defaults to 1 day, used as part of Login process.

services.Configure<DataProtectionTokenProviderOptions>(options =>

{

**options.TokenLifespan = TimeSpan.FromMinutes(tokenLifeSpanMinutes);** // Set the email token lifespan (2FA or Confirm Email in registration)

});

### Authorization Policy’s

Add custom authorization policies, that will be used to restrict access to certain APIs.

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"));**

});

The above roles should match what we have in our identity database for Roles that can be assigned to a User:

A screenshot of a computer

Description automatically generated

### Cookie Options

#### HttpOnly

// 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); // when IsPersistent (Remember me) is true, use this timespan for SSO

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

options.Cookie.SecurePolicy = CookieSecurePolicy.Always;

});

As mentioned above, when generating a new JWT or Refresh Token, you need to inform the Cookie that it will be HttpOnly.

// 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

};

#### Remember Me Timespan

If the user logs in with the “Remember Me” option checked (true), the option below tells Identity how long the user can stay logged in without having to sign in again (if the user logs out – all cookies are removed from the browser and the user must log in as normal)

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);** // when IsPersistent (Remember me) is true, use this timespan for SSO

options.Cookie.HttpOnly = true;

options.Cookie.SecurePolicy = CookieSecurePolicy.Always;

});

### CORS Policy

Implementing a CORS policy is good practice, for the prevention of unauthorized requests. Especially when you have 2FA enabled. As you find on commercial web sites like eBay or Amazon, they will send you a code by email or phone, but you have to go back into the web page and enter it – no new browser or tab can be used for the code entry (here you have an option to resend the 2FA code as well as entering and validating it).

// retrieve the list of allowed origins from the configuration

var corsWhitelistedDomains = configuration.GetSection("CORsWhitelistedDomains").Get<string[]>();

// configure CORS

services.AddCors(options =>

{

options.AddPolicy("AllowSpecificOrigins",

policyBuilder =>

{

// apply the allowed origins from the configuration

policyBuilder.WithOrigins(corsWhitelistedDomains)

**.AllowAnyMethod() // Allow all HTTP methods**

**.AllowAnyHeader() // Allow all headers**

**.AllowCredentials(); // Allow credentials (cookies)**

});

});

### API Output Caching

Generate response output caching policies, these can then be applied to certain APIs by name.

services.AddOutputCache(options =>

{

**options.AddBasePolicy(builder => builder.Expire(TimeSpan.FromSeconds(5)));**

**options.AddPolicy("Expire30", builder => builder.Expire(TimeSpan.FromSeconds(30)));**

**options.AddPolicy("Expire60", builder => builder.Expire(TimeSpan.FromSeconds(60)));**

});

Appling a caching policy by name:

group.MapGet("/GetAllUserPicturesByUserId", **[OutputCache(PolicyName = "Expire30")]** Results<Ok<List<ShareMemories.Domain.Entities.Picture>>, NotFound> (IPictureService pictureService, int id) =>

{

return pictureService.GetPictures() is { } picture // pattern matching expression. Checking if bookService.GetBook(id) matches the pattern { } and assigns it to a variable named book.

? TypedResults.Ok(picture) // return Book if non-null value

: TypedResults.NotFound(); // if Null, return NotFound

})

.WithName("RetrieveAllUserPicturesByUserId")

.WithOpenApi(x => new OpenApiOperation(x)

{

Summary = "Retrieve all pictures by user Id",

Description = "Returns information about a selected picture from the user's library.",

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

});

### Swagger Setup with JWT Security Requirement

*This is optional, because we are using the HttpOnly flag*, the JWT will automatically be included in the API request, but for completeness, in the past if you only used a Bearer Token you had to enter the token manually - this snippet of code allows Swagger to use JWT Bearer Tokens when calling secure API endpoints and not have to enter the text "Bearer".

services.AddSwaggerGen(c =>

{

c.SwaggerDoc("v1", new OpenApiInfo { Version = "v1" });

c.AddSecurityDefinition("Bearer", new OpenApiSecurityScheme

{

Name = "Authorization",

Type = SecuritySchemeType.Http,

BearerFormat = "JWT",

Scheme = "Bearer",

In = ParameterLocation.Header

});

c.AddSecurityRequirement(new OpenApiSecurityRequirement

{

{

new OpenApiSecurityScheme

{

Reference = new OpenApiReference {

Type = ReferenceType.SecurityScheme,

Id = "Bearer"

}

},

Array.Empty<string>() // pass in empty collection

}

});

});

### Mailosaur

Register Mailosaur (dependency injection) as the email provider

**NB:** you can replace with your company SMTP or another email provider.

services.**AddSingleton**<MailosaurClient>(sp =>

{

**return new MailosaurClient(configuration["Mailosaur:ApiKey"]);** // Read from configuration

});

## 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 privileges (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 New User

### API Method

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Register a new user (adding FluentValidator to ensure data integrity from client)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

loginRegisterGroup.MapPost("/**RegisterAsync**", async Task<Results<Ok<string>, BadRequest<string>>> (IMapper mapper, RegisterUserDto registerDto, IAuthService authService) =>

{

Guard.Against.Null(registerDto, nameof(registerDto));

// convert DTO to Model

var registerUserModel = mapper.Map<RegisterUserModel>(registerDto);

var loginRegisterRefreshResponseModel = await **authService.RegisterUserAsync(registerUserModel);**

// convert model to DTO

var loginRegisterRefreshResponseDto = mapper.Map<LoginRegisterRefreshResponseDto>(loginRegisterRefreshResponseModel);

if (loginRegisterRefreshResponseDto.IsStatus) return TypedResults.Ok(loginRegisterRefreshResponseDto.Message);

else return TypedResults.BadRequest(loginRegisterRefreshResponseDto.Message);

}).WithName("RegisterAsync")

.WithOpenApi(x => new OpenApiOperation(x)

{

Summary = "Register a new user",

Description = "Registers a new user within the .Net Roles Identity DB. Must have a unique Username & Email to be valid. Returns a boolean status and an error string (if applicable).",

Tags = new List<OpenApiTag> { new OpenApiTag { Name = "Login/Register/Refresh API Library" } }

})

.CacheOutput(x => x.Tag("LoginUser")); // invalidate data when new record added, by using tag in Post API

### Service Method

public async Task<LoginRegisterRefreshResponseModel> **RegisterUserAsync**(RegisterUserModel user)

{

Guard.Against.Null(user, null, "User credentials are not valid");

const string DEFAULT\_ROLE = "User";

LoginRegisterRefreshResponseModel registerResponseModel = new() { Message = $"Username: {user.UserName} registered successfully. You can now login" };

// verify that Username and\or email have not already been registered

if (await IsUsernameOrEmailTakenAsync(user.UserName, user.Email))

{

registerResponseModel.Message = $"Username {user.UserName} or Email {user.Email}, already exists within the system";

return registerResponseModel;

}

// add these details to a new AspNetUser table instance

var identityUser = new ExtendIdentityUser

{

UserName = user.UserName,

Email = user.Email,

FirstName = user.FirstName,

LastName = user.LastName,

DateOfBirth = user.DateOfBirth,

LastUpdated = DateTimeOffset.UtcNow.UtcDateTime,

EmailConfirmed = !\_identityOptions.Value.SignIn.RequireConfirmedEmail, // configured in Services.AddIdentity - options.SignIn.RequireConfirmedEmail. Store the opposite to your setting!

TwoFactorEnabled = bool.Parse(\_config.GetSection("SystemDefaults:Is2FAEnabled").Value!) // retrieve form appsettings in API

};

if (await \_roleManager.RoleExistsAsync(DEFAULT\_ROLE)) // verify that the role exists

{

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

if (result.Errors.Any())

{

var errors = new StringBuilder();

result.Errors.ToList().ForEach(err => errors.AppendLine($"{err.Description}")); // build up a string of faults

registerResponseModel.Message = errors.ToString();

}

else // success - assign default role

{

// assign a default role (USER) to the user

var roleAssignResult = await \_userManager.AddToRoleAsync(identityUser, "User"); // Replace "User" with the desired role

if (roleAssignResult.Errors.Any())

{

var roleErrors = new StringBuilder();

roleAssignResult.Errors.ToList().ForEach(err => roleErrors.AppendLine($"{err.Description}"));

registerResponseModel.Message = $"Username: {user.UserName} registered, but there was an issue assigning roles: {roleErrors}";

}

else // success registering user & role

{

// does the user need to confirm their email (OTP)

if (\_identityOptions.Value.SignIn.RequireConfirmedEmail)

{

string verificationCode = await \_userManager.GenerateEmailConfirmationTokenAsync(identityUser); // generate token to be used in URL

await SendEmailTaskAsync(identityUser, verificationCode, EmailType.ConfirmationEmail);

registerResponseModel.Message = $"Username: {user.UserName} registered successfully. A confirmation email with a registration token has been sent to {identityUser.Email}, you will need to complete the registration process by opening the Swagger API '/loginGroup/ConfirmRegisteredEmailAsync' and enter your username and token. Check your Spam folder if it isn't in your Inbox.";

}

registerResponseModel.IsStatus = true; // doubling up the IsLoggedIn property to indicate if user was successfully registered or not

}

}

}

else

{

// notify user that the role doesn't exist

registerResponseModel.Message = $"Role: {DEFAULT\_ROLE} - doesn't exist.";

}

return registerResponseModel;

}

### Swagger UI

I’ll initially test with an email address that is already in use, and we can see the response.

#### Swagger Test Data

{

"password": "string@12345",

***"email": "bertoneill@yahoo.com",***

"firstName": "John",

"lastName": "Bloggs",

"userName": "AdminRole",

"confirmPassword": "string@12345",

"dateOfBirth": "1980-10-10"

}

Here you see that the email has already been used:

A screenshot of a computer

Description automatically generated

#### Registration API Response

If I supply a username or email address that has been already allocated, you’ll get the following response:

A close up of a computer screen

Description automatically generated

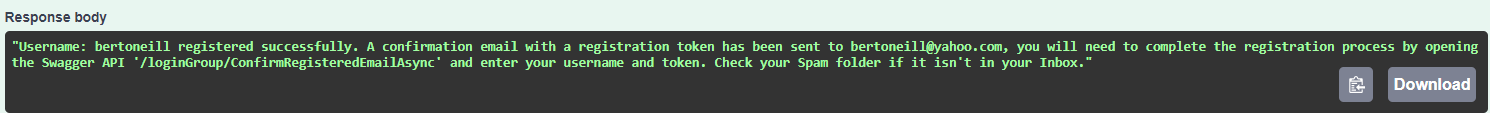
If I use a different email address and try again (I’ll update the email already in use in the database, for simplicity).

**NB** The passwords too, don’t match the Identity criteria (no capital letter):

A screen shot of a computer

Description automatically generated

If I fix the data, I will get this (200) response and database entry:



There will be a new registered user within the Identity database, note that the email confirmation task still needs to be completed by the user.

A screenshot of a computer

Description automatically generated

The user will receive an email (as we have configured Email as our provider of choice, but you can also use SMS or an Authentication app to verify\confirm a registration).

A email message with red text

Description automatically generated with medium confidence

Once the user clicks on the email link, the appropriate API to verify the newly registered user will be called, this will update the Identity database and notify the user of its success.

A screen shot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

## Login

### API Method

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Login an already registered user

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

loginRegisterGroup.MapPost("/**LoginAsync**", async (IMapper mapper, LoginUserDto loginDto, IAuthService authService, HttpContext context) =>

{

Guard.Against.Null(loginDto, nameof(loginDto));

// convert DTO to Model

var loginUserModel = mapper.Map<LoginUserModel>(loginDto);

var loginRegisterRefreshResponseModel = await authService.**LoginAsync**(loginUserModel);

// convert model to DTO

var loginRegisterRefreshResponseDto = mapper.Map<LoginRegisterRefreshResponseDto>(loginRegisterRefreshResponseModel);

if (loginRegisterRefreshResponseDto.IsStatus)

{

#if DEBUG

return Results.Ok(loginRegisterRefreshResponseDto); // testing with JWT Token in Swagger - development ONLY!!!

#else

return Results.Ok(new { message = "Logged in successfully" });

#endif

}

return Results.BadRequest(loginRegisterRefreshResponseDto.Message);

})

.WithName("LoginAsync")

.WithOpenApi(x => new OpenApiOperation(x)

{

Summary = "Login a user",

Description = "Logs in the user and returns a JWT token if successful.",

Tags = new List<OpenApiTag> { new OpenApiTag { Name = "Login/Register/Refresh API Library" } }

})

.CacheOutput(x => x.Tag("LoginUser"))

.AddEndpointFilter<GenericValidationFilter<LoginUserValidator, LoginUserDto>>(); // apply fluent validation to DTO model from client and pass back broken rules

### Service Method

public async Task<LoginRegisterRefreshResponseModel> **LoginAsync**(LoginUserModel user)

{

Guard.Against.Null(user, null, "User credentials are not valid");

int rememberMeExpireDays = int.Parse(\_config.GetSection("SystemDefaults:RememberMeLifeSpan").Value!);

var response = new LoginRegisterRefreshResponseModel(); // "IsStatus" will be false by default

var identityUser = await \_userManager.FindByNameAsync(user.UserName);

if (identityUser is null)

{

response.Message = $"Credentials are not valid";

return response;

}

// does the the user still need to confirmed their email (if applicable)

if (\_identityOptions.Value.SignIn.RequireConfirmedEmail && !await \_userManager.IsEmailConfirmedAsync(identityUser))

{

response.Message = $"User has not yet confirmed their email address. Check your Spam folder";

return response;

}

// Check if the account is locked out

if (await \_userManager.IsLockedOutAsync(identityUser))

{

response.Message = $"Account is locked. Please wait {\_identityOptions.Value.Lockout.DefaultLockoutTimeSpan.TotalMinutes} minutes before trying again.";

return response;

}

IList<string> roles = await VerifyUserRolesAsync(identityUser); // retrieve their roles (at least 1 must exist)

// try to sign the user in

await \_signInManager.SignOutAsync();

// Attempt to sign the user in

SignInResult loginResult = await \_signInManager.PasswordSignInAsync(identityUser, user.Password, user.IsPersistent, lockoutOnFailure: true);

if (loginResult.IsLockedOut || loginResult.IsNotAllowed)

{

response.Message = $"Account is either locked (wait {\_identityOptions.Value.Lockout.DefaultLockoutTimeSpan} minutes) or you are not allowed to sign-in - contact Administration.";

}

else if (loginResult.RequiresTwoFactor) // valid user at this stage - determine if 2FA is enabled and halt login- send 2fa email

{

await SendTwoFactorAuthenticationAsync(identityUser!);

response.Message = "Two-factor authentication is enabled on your account. You have been sent an email with a OTP, click on the Swagger API '/2FAGroup/Verify2FactorAuthenticationAsync' and enter your username and the code supplied to complete your login.";

}

else if (loginResult.Succeeded)

{

// if "remember me" is true from client, extent their login

if (user.IsPersistent)

{

var authProperties = new AuthenticationProperties

{

IsPersistent = true,

ExpiresUtc = DateTimeOffset.UtcNow.AddDays(rememberMeExpireDays) // Set expiration to days

};

// sign-in again with extended duration

await \_signInManager.SignInAsync(identityUser, authProperties);

}

await AssignJwtTokensResponse(response, identityUser, roles); // create JWT bearer

}

else // failed to login

{

response.Message = "Invalid login attempt. Please check your username and password.";

}

return response;

}

### Swagger UI

#### Swagger Test Data

Now I’ll try to login with an invalid password:

{

"userName": " bertoneill",

"password": "**String@12345xxx**",

"isPersistent": true

}

#### Login API Response

A screen shot of a computer screen

Description automatically generated

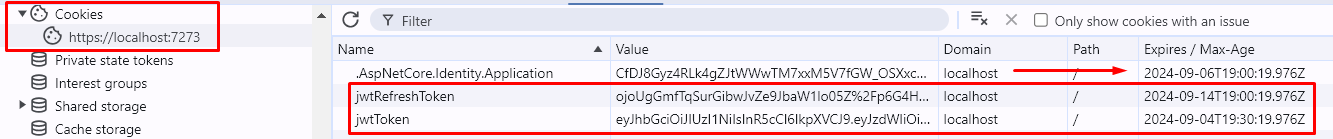
If I correct the password and retry to login, I will get a valid response.

**NB**: I am passing back the JWT and Refresh Token’s in development mode (for testing only):

A black screen with a red line

Description automatically generated

You can now see the JWT and Refresh Tokens within the browser’s Cookie settings, notice that because we selected to persist our login, that *AspNetCore.Identity.Application* has an expiry date too, this will facilitate the *Single Sign On* process.



## Logout

### Api Method

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Allow user to logout and delete their JWT Token \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Define the logout endpoint

loginRegisterGroup.MapPost("/**LogoutAsync**", async Task<Results<Ok<string>, NotFound<string>>> (IMapper mapper, HttpContext context, IAuthService authService) =>

{

VerifyRequestCookiesExist(context);

var loginRegisterRefreshResponseModel = await authService.**LogoutAsync**(context.Request.Cookies["jwtToken"]!);

// convert model to DTO

var loginRegisterRefreshResponseDto = mapper.Map<LoginRegisterRefreshResponseDto>(loginRegisterRefreshResponseModel);

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

else return TypedResults.NotFound(loginRegisterRefreshResponseDto.Message);

})

.WithName("Logout")

.RequireAuthorization()

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

.WithOpenApi(x => new OpenApiOperation(x)

{

Summary = "Logout a user",

Description = "Logout user and delete their cached JWT token.",

Tags = new List<OpenApiTag> { new OpenApiTag { Name = "Login/Register/Refresh API Library" } }

});

### Service Method

public async Task<LoginRegisterRefreshResponseModel> **LogoutAsync**(string jwtToken)

{

Guard.Against.Null(jwtToken, null, "Token is not valid");

var response = new LoginRegisterRefreshResponseModel() { Message = "Successfully logged out" }; // default message

var principal = \_jwtTokenService.GetPrincipalFromExpiredToken(jwtToken);

// not able to retrieve user from Jwt bearer token

if (principal?.Identity?.Name is null)

{

response.IsStatus = true; // user is still logged in

response.Message = "Jwt Bearer not valid, during logout process";

}

else

{

var identityUser = await \_userManager.FindByNameAsync(principal.Identity.Name); // retrieve user principal

**// clear the refresh token**

**identityUser!.RefreshToken = null;**

**identityUser.RefreshTokenExpiry = null;**

**identityUser.LastUpdated = DateTimeOffset.UtcNow.UtcDateTime;**

var result = await \_userManager.**UpdateAsync**(identityUser); // Update the user in the database

// handle a database fail

if (!result.Succeeded)

{

response.IsStatus = true; // user is still logged in

var errors = new StringBuilder();

result.Errors.ToList().ForEach(err => errors.AppendLine($"{err.Description}")); // build up a string of faults

response.Message = errors.ToString();

}

else

{

**// remove cookies form response to client**

**await \_signInManager.SignOutAsync(); // this will clear the "AspNetCore.Identity.Application" IsPersistent cookie**

**\_httpContextAccessor.HttpContext?.Response.Cookies.Delete("jwtToken");**

**\_httpContextAccessor.HttpContext?.Response.Cookies.Delete("jwtRefreshToken");**

}

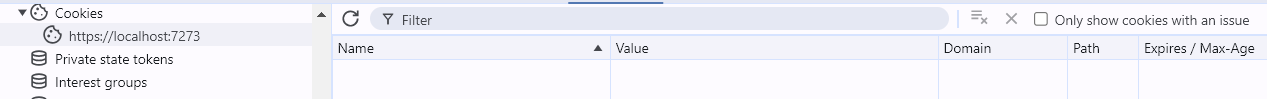
}

return response;

}

### Swagger UI

When a user logs out, the JWT and Refresh tokens are cleared from their browser cookies cache. Even if they have persisted their login.



If the user has persisted their login, and just closes the browser or the browser tab (not calling the Logout API), they will be able to make API calls when they revisit the web site without having to log back in – the JWT and Refresh Token cookies are kept alive.

To test this, Run the Visual Studio solution to start Swagger. Login in with *isPersistent* set to true. Hit F12 in your browser and you will see the tokens. Then close your browser (do not call the Logout API). Rerun Visual Studio solution to bring Swagger back up, hit F12 and you will see the Cookies have persisted. You can now call PictureAsync with the Id of 1 to see that you are able to retrieve data – without having to relog back in:



A computer screen shot of a computer code

Description automatically generated

## Confirm Email Registration

As seen earlier, when a new user registers their details, an email will be sent to the email address they registered with, prompting them to verify it. They can do this by clicking on an embedded link (which has their Username and Code within the URL) – once they click on the link, they will be redirected to the API below:

A email message with red text

Description automatically generated with medium confidence

### Api Method

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Allow user to confirm their registered email address, stop fraud and bogus accounts \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

loginRegisterGroup.MapGet("/**ConfirmRegisteredEmailAsync**", async Task<Results<Ok<string>, NotFound<string>>> (IMapper mapper, IAuthService authService, string userName, string token) =>

{

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

Guard.Against.Empty(token, "Confirm token is missing");

var loginRegisterRefreshResponseModel = await authService.**VerifyEmailConfirmationAsync**(userName, token);

// convert model to DTO

var loginRegisterRefreshResponseDto = mapper.Map<LoginRegisterRefreshResponseDto>(loginRegisterRefreshResponseModel);

// was the email confirmation successful

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

else return TypedResults.Ok(loginRegisterRefreshResponseDto.Message);

})

.WithName("ConfirmRegisteredEmailAsync")

.WithOpenApi(x => new OpenApiOperation(x)

{

Summary = "Confirm a new registered user's email",

Description = "Confirms the user's email address with the provided token, after they have registered.",

Tags = new List<OpenApiTag> { new OpenApiTag { Name = "Login/Register/Refresh API Library" } }

});

### Service Method

public async Task<LoginRegisterRefreshResponseModel> **VerifyEmailConfirmationAsync**(string userName, string token)

{

Guard.Against.Null(userName, null, "User credentials are not valid");

Guard.Against.Null(token, null, "Token is not valid");

var response = new LoginRegisterRefreshResponseModel() { Message = "Email confirmation successful, you can now login." }; // default message

var identityUser = await \_userManager.FindByNameAsync(userName);

if (identityUser == null) response.Message = "Invalid credentials or token";

else

{

var result = await \_userManager.**ConfirmEmailAsync**(identityUser, token);

if (result.Succeeded)

{

response.IsStatus = true;

}

else

{

var errors = new StringBuilder();

result.Errors.ToList().ForEach(err => errors.AppendLine($"{err.Description}")); // build up a string of faults

response.Message = errors.ToString();

}

}

return response;

}

## Resend Registration Confirm

### Api Method

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Request that a new confirmation email be sent (to complete registration process \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

loginRegisterGroup.MapPost("/**ResendConfirmationEmailAsync**", async Task<Results<Ok<string>, NotFound<string>>> (IMapper mapper, string userName, IAuthService authService) =>

{

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

var loginRegisterRefreshResponseModel = await authService.**RequestConfirmationEmailAsync**(userName);

// convert model to DTO

var loginRegisterRefreshResponseDto = mapper.Map<LoginRegisterRefreshResponseDto>(loginRegisterRefreshResponseModel);

// was the email confirmation sent successfully

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

else return TypedResults.Ok(loginRegisterRefreshResponseDto.Message);

})

.WithName("ResendConfirmationEmail")

.WithOpenApi(x => new OpenApiOperation(x)

{

Summary = "Request a new user's registered email confirmation",

Description = "Request a new email confirmation, to complete registration",

Tags = new List<OpenApiTag> { new OpenApiTag { Name = "Login/Register/Refresh API Library" } }

});

### Service Method

public async Task<LoginRegisterRefreshResponseModel> **RequestConfirmationEmailAsync**(string userName)

{

Guard.Against.Null(userName, null, "User credentials are not valid");

var response = new LoginRegisterRefreshResponseModel() { Message = $"A new confirmation email has been sent - check your Spam folder." }; // default message

var identityUser = await \_userManager.FindByNameAsync(userName);

if (identityUser == null) response.Message = "Invalid credentials supplied.";

else

{

if (identityUser.EmailConfirmed)

{

response.Message = "Email address associated with your Username, has already been confirmed.";

}

else

{

string verificationCode = await \_userManager.**GenerateEmailConfirmationTokenAsync**(identityUser); // generate token to be used in URL

await SendEmailTaskAsync(identityUser, verificationCode, EmailType.ConfirmationEmail);

response.IsStatus = true;

}

}

return response;

}

## Verify 2FA

In the Appsetting file I have configured if 2FA should be enabled when registering a new user. If set to true, it will mean that when a user logs in, they will be sent an email with a code, that they must enter before it expires:

"SystemDefaults": {

"ProviderTokenLifeSpan": "30", // minutes

"LockoutLifeSpan": "10", // minutes

"LockoutAttempts": "3",

**"Is2FAEnabled": "True",**

"RememberMeLifeSpan": "2", // days

"AdminLocksAccountLifeSpan": "7" // duration when Admin locs your account in days

},

Below, is where I configure the timespan for the 2FA code to expire (reading a value in minutes form the Appsetting file):

// configure the timeout for a token (Confirmation email , 2FA etc.), before it expires. Defaults to 1 day.

services.Configure<DataProtectionTokenProviderOptions>(options =>

{

**options.TokenLifespan = TimeSpan.FromMinutes(tokenLifeSpanMinutes);** // Set the email token lifespan (2FA or Confirm Email in registration)

});

An example of the email that is sent to the user, with the 2FA code, they must take this code and go back to the web site to enter it – no email link is possible here due to CORS security:

A screenshot of a computer

Description automatically generated

Below is the response the user will receive when they login and 2FA is enabled on their account:

A black rectangular object with red lines

Description automatically generated

### Api Method

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Verify user's 2FA (API not secure as user must be able to use it as part of logged in process) \*

\*To verify 2FA code, it has to be form the same browser tab that initiated - thus an code entry screen\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

twoFAGroup.MapPost("/**Verify2FactorAuthenticationAsync**", async Task<Results<Ok<string>, NotFound<string>>> (IAuthService authService, string userName, string code) =>

{

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

Guard.Against.Empty(code, "Code is missing");

var loginRegisterRefreshResponseDto = await authService.**Verify2FactorAuthenticationAsync**(userName, code);

// was the email confirmation sent successfully

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

else return TypedResults.Ok(loginRegisterRefreshResponseDto.Message);

})

.WithName("Verify2FactorAuthentication")

.WithOpenApi(x => new OpenApiOperation(x)

{

Summary = "Verify user login with 2FA",

Description = "Allow user to authenticate themselves using a code that was emailed to them, as part of the login process (if enabled)",

Tags = new List<OpenApiTag> { new OpenApiTag { Name = "2FA - API Library" } }

});

### Service Method

public async Task<LoginRegisterRefreshResponseModel> **Verify2FactorAuthenticationAsync**(string userName, string verificationCode)

{

Guard.Against.Null(userName, null, "User credentials are not valid");

Guard.Against.Null(verificationCode, null, "Token is not valid");

var response = new LoginRegisterRefreshResponseModel() { Message = "2FA verification successful. You have been verified, you can now call the (secure) API's" }; // default message

var identityUser = await \_userManager.FindByNameAsync(userName);

if (identityUser == null)

{

response.Message = "User not found.";

}

else

{

// retrieve existing providers and check that Email is one of them (you may want to use SMS etc.)

var providers = await \_userManager.**GetValidTwoFactorProvidersAsync**(identityUser);

if (!providers.Contains("Email"))

{

response.Message = "Expected 2FA Provider doesn't exist!";

}

var result = await \_signInManager.TwoFactorSignInAsync(TokenOptions.DefaultEmailProvider, verificationCode, false, false); // Replace "Email" with the actual provider name if different

if (result.Succeeded)

{

IList<string> roles = await VerifyUserRolesAsync(identityUser); // retrieve their roles (at least 1 must exist)

await AssignJwtTokensResponse(response, identityUser, roles);

response.IsStatus = true;

}

else if (result.IsLockedOut) response.Message = "User account locked out";

else response.Message = "Invalid 2FA code";

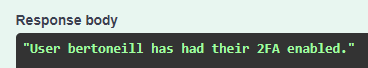
}

return response;

}

## Enable 2FA (Admin)

A user with the Admin privileges, will be able to assign 2FA to another account:



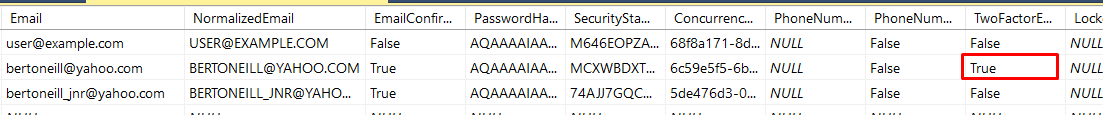
A screenshot of a email

Description automatically generated

If you try to enable 2FA for an account that already has it enabled, you will be notified with the appropriate response:

A black and white text

Description automatically generated



### Api Method

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Enforce 2FA for a specific user (called by Admin) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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" } }

});

### Service Method

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Enable & Disable 2FA on account \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public async Task<LoginRegisterRefreshResponseModel> **Enable2FactorAuthenticationForUserAsync**(string userName)

{

Guard.Against.Null(userName, null, "User credentials are not valid");

var response = new LoginRegisterRefreshResponseModel() { Message = $"User {userName} has had their 2FA enabled." }; // default message

var identityUser = await \_userManager.FindByNameAsync(userName);

if (identityUser == null) response.Message = "Invalid credentials supplied.";

else

{

var is2faEnabled = await \_userManager.**GetTwoFactorEnabledAsync**(identityUser);

if (is2faEnabled)

{

response.Message = $"2FA is already enabled for user {userName}.";

}

else

{

var result = await \_userManager.SetTwoFactorEnabledAsync(identityUser, true); // enable 2FA in DB

if (result.Succeeded)

{

response.IsStatus = true;

// send user email to notify them that they have 2FA enabled

await SendEmailTaskAsync(identityUser, string.Empty, EmailType.TwoFactorAuthenticationEnabled);

}

}

}

return response;

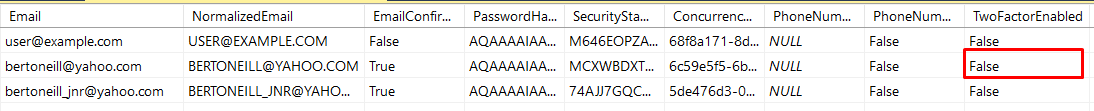
}

## Disable 2FA (Admin)

A user with the Admin privileges can disable 2FA on a user’s account, the user will receive an email stating that their 2FA has been disabled on their account:

A screenshot of a computer

Description automatically generated



### Api Method

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Revoke 2FA for a specific user (called by Admin) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

adminGroup.MapPost("/**Disable2faForUserAsync**", async Task<Results<Ok<string>, NotFound<string>>> (string userName, IAuthService authService) =>

{

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

var loginRegisterRefreshResponseDto = await authService.**Disable2FactorAuthenticationForUserAsync**(userName);

// was the email confirmation sent successfully

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

else return TypedResults.Ok(loginRegisterRefreshResponseDto.Message);

})

.WithName("Disable2faForUser")

.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 disable 2FA for a user",

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

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

});

### Service Method

public async Task<LoginRegisterRefreshResponseModel> **Disable2FactorAuthenticationForUserAsync**(string userName)

{

Guard.Against.Null(userName, null, "User credentials are not valid");

var response = new LoginRegisterRefreshResponseModel() { Message = $"User {userName} has had their 2FA disabled." }; // default message

var identityUser = await \_userManager.FindByNameAsync(userName);

if (identityUser == null) response.Message = "Invalid credentials supplied.";

else

{

var is2faEnabled = await \_userManager.**GetTwoFactorEnabledAsync**(identityUser);

if (!is2faEnabled)

{

response.Message = $"2FA has already been disabled for user {userName}.";

}

else

{

var result = await \_userManager.SetTwoFactorEnabledAsync(identityUser, false); // disable 2FA in DB

if (result.Succeeded)

{

response.IsStatus = true;

// send user email to notify them that they have 2FA enabled

await SendEmailTaskAsync(identityUser, string.Empty, EmailType.TwoFactorAuthenticationDisabled);

}

}

}

return response;

}

## Resend 2FA

A user can request that a new 2FA code be resent, due to an expiry issue or the user deletes the original email by accident.

### Api Method

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Request that a new 2FA code be sent to the user \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

twoFAGroup.MapPost("/**Request2FACodeAsync**", async Task<Results<Ok<string>, NotFound<string>>> (string userName, IAuthService authService) =>

{

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

var twoFaResponseDto = await authService.**Request2FACodeAsync**(userName);

// was the 2FA code sent successfully

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

else return TypedResults.Ok(twoFaResponseDto.Message);

})

.WithName("Request2FACodeAsync")

.WithOpenApi(x => new OpenApiOperation(x)

{

Summary = "Request new 2FA code",

Description = "Request a new 2FA code to be sent to the user",

Tags = new List<OpenApiTag> { new OpenApiTag { Name = "2FA - API Library" } }

});

### Service Method

public async Task<LoginRegisterRefreshResponseModel> **Request2FACodeAsync**(string userName)

{

Guard.Against.Null(userName, null, "User credentials are not valid");

var response = new LoginRegisterRefreshResponseModel(); // "IsStatus" will be false by default

var identityUser = await \_userManager.FindByNameAsync(userName);

if (identityUser is null)

{

response.Message = $"Credentials are not valid";

return response;

}

await SendTwoFactorAuthenticationAsync(identityUser!);

response.Message = "Two-factor authentication is enabled on your account. You have been sent an email with a OTP, click on the Swagger API '/2FAGroup/Verify2FactorAuthenticationAsync' and enter your username and the code supplied to complete your login.";

return response;

}

## Revoke Tokens (IMemory Cache)

### Api Method

When a user Revokes the JWT and Refresh Tokens, the original JWT token will be added to an in-memory cache, and this cache is challenged for every API call to determine if the request is from an invalid JWT – thus cancelling the request and logging that information. The cache entry is timespan based, so it cleans up the cache automatically (when the JWT would normally be invalid).

For e.g. you can see the JWT for a user in the browser Cookies below (before a Revoke is executed).

A screen shot of a computer

Description automatically generated

Once a Revoke has been invoked, the JWT will be removed from the client’s browser – thus forcing them to log in again (*you could supply them with a new JWT and Refresh token at this stage too*).



A black and white screen with white text

Description automatically generated

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Allow user to revoke the Refresh Token if they think it has been compromised \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

tokenGroup.MapPost("/**RevokeAsync**", async Task<Results<Ok<string>, NotFound<string>>> (HttpContext context, IAuthService authService) =>

{

VerifyRequestCookiesExist(context); // before revoking, make sure they exist

var loginRegisterRefreshResponseDto = await authService.**RevokeTokenLogoutAsync**(context.Request.Cookies["jwtToken"]!);

// was the Refresh Token revoked successfully

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

else return TypedResults.Ok(loginRegisterRefreshResponseDto.Message);

})

.WithName("RevokeAsync")

.RequireAuthorization()

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

.WithOpenApi(x => new OpenApiOperation(x)

{

Summary = "Revoke JWT & Refresh token",

Description = "Revokes the JWT refresh token for the specified user.",

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

});

### Service Method

public async Task<LoginRegisterRefreshResponseModel> **RevokeTokenLogoutAsync**(string jwtToken)

{

Guard.Against.Null(jwtToken, null, "Token is not valid");

var response = new LoginRegisterRefreshResponseModel() { Message = "Successfully revoked JWT token", IsStatus = true }; // default message

var claimsPrincipal = \_jwtTokenService.GetPrincipalFromExpiredToken(jwtToken);

// not able to retrieve user from Jwt bearer token

if (claimsPrincipal?.Identity?.Name is null)

{

response.IsStatus = false;

response.Message = "Jwt Bearer is not valid, during revoke process";

}

else

{

var identityUser = await \_userManager.FindByNameAsync(claimsPrincipal.Identity.Name); // retrieve user principal

if (identityUser is null)

{

response.IsStatus = false;

response.Message = $"User '{claimsPrincipal.Identity.Name}' not found during token revoke";

return response;

}

**// clear the refresh token & update database**

**identityUser!.RefreshToken = null;**

**identityUser.RefreshTokenExpiry = null;**

**identityUser.LastUpdated = DateTimeOffset.UtcNow.UtcDateTime;**

var result = await \_userManager.**UpdateAsync**(identityUser); // update database

// handle a database fail

if (!result.Succeeded)

{

response.IsStatus = false;

var errors = new StringBuilder();

result.Errors.ToList().ForEach(err => errors.AppendLine($"{err.Description}")); // build up a string of faults

response.Message = errors.ToString();

}

else // success

{

// sign out & remove cookies from response to client - force user to log back in (thus generating new tokens)

await \_signInManager.**SignOutAsync**();

**\_httpContextAccessor.HttpContext?.Response.Cookies.Delete("jwtToken");**

**\_httpContextAccessor.HttpContext?.Response.Cookies.Delete("jwtRefreshToken");**

**CacheRevokedToken**(jwtToken, response, claimsPrincipal); // cache revoked Jwt so that an imposter can't use it (middleware checks API calls)

}

}

return response;

}

The original JWT is cached so that if it is used again, you can cancel the request. If the server is rebooted this cache will be invalidated – but with short lived JWT’s this should be a minimal scenario.

A screenshot of a computer

Description automatically generated

private void **CacheRevokedToken**(string jwtToken, LoginRegisterRefreshResponseModel response, ClaimsPrincipal principal)

{

// store revoked JWT Id in IMemory cache (with sliding timespan)

var tokenHandler = new JwtSecurityTokenHandler();

var jwtTokenObject = tokenHandler.ReadJwtToken(jwtToken);

var jti = jwtTokenObject?.Claims.FirstOrDefault(x => x.Type == JwtRegisteredClaimNames.Jti)?.Value;

if (jti == null)

{

response.IsStatus = false;

response.Message = "Invalid JWT to revoke";

}

else

{

var expiration = jwtTokenObject!.ValidTo - DateTime.UtcNow; // calculate the token's remaining validity period

// Cache the revoked token's JTI with an sliding (expiry) timespan equal to the token's remaining validity (thus keeping cache clean)

\_memoryCache.Set(jti, true, expiration);

\_logger.LogInformation($"JWT Bearer {jti} was revoked by user {principal.Identity.Name}. Currently {expiration} left before expires");

}

}

## Refresh Tokens

The user may want to extend their online activity by requesting a new JWT, rather than logging out and back in again when their original JWT expires. This is where the Refresh token comes into play. The Refresh token has a longer expiry date than the JWT and is used to allow the user to refresh as seemingly as possible. Once refreshed, a new JWT and Refresh token are generated (with the new Refresh token having a new extended lifespan).

Before the Refresh:

A screenshot of a computer

Description automatically generated

After the Refresh:

A screenshot of a computer

Description automatically generated

### Api Method

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Refresh a user's login instance, without having to pass the credentials again

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

tokenGroup.MapPost("/**RefreshTokenAsync**", async (IAuthService authService, HttpContext context) =>

{

VerifyRequestCookiesExist(context);

var loginRegisterRefreshResponseDto = await authService.**RefreshTokenAsync**(context.Request.Cookies["jwtToken"]!, context.Request.Cookies["jwtRefreshToken"]!);

if (loginRegisterRefreshResponseDto.IsStatus)

{

#if DEBUG

return Results.Ok(loginRegisterRefreshResponseDto); // testing with JWT Token in Swagger - development ONLY!!!

#else

return Results.Ok("Successfully refreshed JWT Bearer");

#endif

}

return Results.Unauthorized();

})

.WithName("RefreshTokenAsync")

.RequireAuthorization()

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

.WithOpenApi(x => new OpenApiOperation(x)

{

Summary = "Renew JWT & Refresh token",

Description = "Using refresh & JWT token, you can request to be logged back in again, without having to supply credentials.",

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

});

### Service Method

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* JWT Refresh & Revoke \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public async Task<LoginRegisterRefreshResponseModel> **RefreshTokenAsync**(string jwtToken, string refreshToken)

{

var response = new LoginRegisterRefreshResponseModel();

try

{

if (!\_isRefreshing) // stop user refresh saturation \ disable client side button that calls API too

{

\_isRefreshing = true;

var principal = \_jwtTokenService.GetPrincipalFromExpiredToken(jwtToken);

// not able to retrieve user from Jwt token

if (principal?.Identity?.Name is null)

{

await LogoutAsync(jwtToken); // call logout

response.Message = "Jwt Bearer is not valid";

return response;

}

var identityUser = await \_userManager.FindByNameAsync(principal.Identity.Name);

if (identityUser is null || identityUser.RefreshToken != refreshToken || identityUser.RefreshTokenExpiry < DateTime.Now)

{

await LogoutAsync(jwtToken); // call logout

response.Message = "Jwt Bearer invalid or invalid Refresh Token or Refresh Token expired - Use the login screen again";

return response;

}

var roles = await \_userManager.GetRolesAsync(identityUser); // retrieve role(s) to append to Claims in JWT bearer token

response.IsStatus = true;

**response.JwtToken = \_jwtTokenService.GenerateJwtToken(identityUser, roles, int.Parse(\_config["Jwt:JWT\_TOKEN\_EXPIRE\_MINS"]!));**

**response.JwtRefreshToken = \_jwtTokenService.GenerateRefreshToken();**

**response.JwtRefreshTokenExpire = DateTimeOffset.UtcNow.AddDays(int.Parse(\_config["Jwt:REFRESH\_TOKEN\_EXPIRE\_DAYS"]!)).UtcDateTime;**

**response.JwtTokenExpire = DateTimeOffset.UtcNow.AddMinutes(int.Parse(\_config["Jwt:JWT\_TOKEN\_EXPIRE\_MINS"]!)).UtcDateTime;**

// update AspNetUser DB table with latest details

identityUser.RefreshToken = response.JwtRefreshToken;

identityUser.RefreshTokenExpiry = response.JwtRefreshTokenExpire; // refresh token should be longer than JWT bearer token

identityUser.LastUpdated = DateTimeOffset.UtcNow.UtcDateTime;

var result = await \_userManager.**UpdateAsync**(identityUser);

if (!result.Succeeded)

{

response.IsStatus = true; // user is still logged in

var errors = new StringBuilder();

result.Errors.ToList().ForEach(err => errors.AppendLine($"{err.Description}")); // build up a string of faults

response.Message = errors.ToString();

}

else

{

UpdateResponseTokens(response);

}

return response;

}

else

{

// Return a response in case the token is already refreshing

response.Message = "Token is already refreshing.";

return response;

}

}

finally { \_isRefreshing = false; } // reset for next refresh call

}

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

### Api Method

The *Remember Me* option is part of the login process, where the user will tick a box to keep them *Signed In* for a longer period (this time period is taken from the Appsetting file).

public class LoginUserDto()

{

[Required]

public string UserName { get; set; } = string.Empty;

[Required]

public string Password { get; set; } = string.Empty;

**public bool IsPersistent { get; set; } = false;**

}

### Service Method

Then within the LoginAsync service method, there is the PasswordSignInAsync method that will take the **IsPersistent** property to set this option (or not) when a user logs in.

// Attempt to sign the user in

SignInResult loginResult = await \_signInManager.**PasswordSignInAsync**(identityUser, user.Password, **user.IsPersistent**, lockoutOnFailure: true);

If true, the Cookie *AspNetCore.Identity.Application* is set in the client’s browser with an expiry date (until when the user can automatically login without having to supply credentials).

A screenshot of a computer

Description automatically generated

If false, the Cookie *AspNetCore.Identity.Application* only lasts as long as session (browser open).

A screenshot of a computer

Description automatically generated

### Swagger Test Data

{

"userName": "userRole",

"password": "String@1234567",

**"isPersistent": true**

}

If the User logs out (calling the LogoutAsync API) - they clear all their browser cookies - thus making the User non-persistent and having to log in again.

## Update User Details

### Api Method

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Update User's details (Name, email, phone number for e.g.) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

loginRegisterGroup.MapPut("/**UpdateUserDetailsAsync**", async Task<Results<Ok<string>, BadRequest<string>>> (IMapper mapper, HttpContext context, UpdateUserDetailsDto userUpdateDetails, IAuthService authService) =>

{

Guard.Against.Null(userUpdateDetails, nameof(userUpdateDetails));

// convert DTO to Model

var registerUserModel = mapper.Map<RegisterUserModel>(userUpdateDetails);

var loginRegisterRefreshResponseModel = await authService.**UpdateUserDetailsAsync**(context.Request.Cookies["jwtToken"]!, registerUserModel);

// convert model to DTO

var loginRegisterRefreshResponseDto = mapper.Map<LoginRegisterRefreshResponseDto>(loginRegisterRefreshResponseModel);

// was the email confirmation sent successfully

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

else return TypedResults.Ok(loginRegisterRefreshResponseDto.Message);

})

.RequireAuthorization()

.WithName("UpdateUserDetailsAsync")

.WithOpenApi(x => new OpenApiOperation(x)

{

Summary = "Update user's details",

Description = "Update Email, Phone, First Name, Last Name or DOB",

Tags = new List<OpenApiTag> { new OpenApiTag { Name = "Login/Register/Refresh API Library" } }

});

### Service Method

public async Task<LoginRegisterRefreshResponseModel> **UpdateUserDetailsAsync**(string jwtToken, RegisterUserModel userUpdateDetails)

{

Guard.Against.Null(userUpdateDetails, null, "User details are not valid");

LoginRegisterRefreshResponseModel registerResponseModel = new() { Message = $"Username: {userUpdateDetails.UserName} registered successfully. You can now login" };

var claimsPrincipal = \_jwtTokenService.GetPrincipalFromExpiredToken(jwtToken);

// not able to retrieve user from Jwt bearer token

if (claimsPrincipal?.Identity?.Name is null)

{

registerResponseModel.IsStatus = false;

registerResponseModel.Message = "Jwt Bearer is not valid, during update process";

}

else

{

var identityUser = await \_userManager.FindByNameAsync(claimsPrincipal.Identity.Name); // retrieve user principal

var roles = await \_userManager.GetRolesAsync(identityUser);

if (identityUser is null)

{

registerResponseModel.IsStatus = false;

registerResponseModel.Message = $"User '{claimsPrincipal.Identity.Name}' not found during details update";

return registerResponseModel;

}

// verify that email is not in use already by another account

if (!String.IsNullOrEmpty(userUpdateDetails.Email))

{

var user = await \_userManager.FindByEmailAsync(userUpdateDetails.Email);

if (user is not null)

{

if (user.UserName != claimsPrincipal.Identity.Name)

{

registerResponseModel.IsStatus = false;

registerResponseModel.Message = "Email has already been assigned to another user";

return registerResponseModel;

}

}

}

// update user details (that have been supplied)

identityUser.Email = userUpdateDetails.Email != string.Empty ? userUpdateDetails.Email : identityUser.Email;

identityUser.PhoneNumber = userUpdateDetails.PhoneNumber != string.Empty ? userUpdateDetails.PhoneNumber : identityUser.PhoneNumber;

identityUser.FirstName = userUpdateDetails.FirstName != string.Empty ? userUpdateDetails.FirstName : identityUser.FirstName;

identityUser.LastName = userUpdateDetails.LastName != string.Empty ? userUpdateDetails.LastName : identityUser.LastName;

identityUser.DateOfBirth = userUpdateDetails.DateOfBirth != default(DateOnly) ? userUpdateDetails.DateOfBirth : identityUser.DateOfBirth;

// create new tokens for clients browser based on their updated details

registerResponseModel.JwtToken = \_jwtTokenService.GenerateJwtToken(identityUser, roles, int.Parse(\_config["Jwt:JWT\_TOKEN\_EXPIRE\_MINS"]!));

registerResponseModel.JwtRefreshToken = \_jwtTokenService.GenerateRefreshToken();

registerResponseModel.JwtRefreshTokenExpire = DateTimeOffset.UtcNow.AddDays(int.Parse(\_config["Jwt:REFRESH\_TOKEN\_EXPIRE\_DAYS"]!)).UtcDateTime;

registerResponseModel.JwtTokenExpire = DateTimeOffset.UtcNow.AddMinutes(int.Parse(\_config["Jwt:JWT\_TOKEN\_EXPIRE\_MINS"]!)).UtcDateTime;

// update AspNetUser DB table with latest details

identityUser.RefreshToken = registerResponseModel.JwtRefreshToken;

identityUser.RefreshTokenExpiry = registerResponseModel.JwtRefreshTokenExpire; // refresh token should be longer than JWT bearer token

identityUser.LastUpdated = DateTimeOffset.UtcNow.UtcDateTime;

var result = await \_userManager.**UpdateAsync**(identityUser); // update database

// handle a database fail

if (!result.Succeeded)

{

registerResponseModel.IsStatus = false;

var errors = new StringBuilder();

result.Errors.ToList().ForEach(err => errors.AppendLine($"{err.Description}")); // build up a string of faults

registerResponseModel.Message = errors.ToString();

}

else // success

{

// notify user of update (return new JWT Bearer Token and refresh Tokens)

registerResponseModel.IsStatus = true;

registerResponseModel.Message = "User details have been updated";

// notify user that details have been updated

await SendEmailTaskAsync(identityUser, string.Empty, EmailType.DetailsUpdated);

}

}

return registerResponseModel;

}

### Swagger Test Data

{

"firstName": "Bert",

"lastName": "Admin",

"email": "YourEmailAddress@gmail.com",

"dateOfBirth": "1970-10-10",

"phoneNumber": "0044 (0)12345678"

}

## View User details

### Api Method

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* View a User's details (Name, email, phone number for e.g.) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

loginRegisterGroup.MapGet("/**ViewUserDetailsAsync**", async Task<Results<Ok<string>, NotFound<string>>> (IMapper mapper, string userName, IAuthService authService) =>

{

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

var loginRegisterRefreshResponseModel = await authService.**ViewUserDetailsAsync**(userName);

// convert model to DTO

var loginRegisterRefreshResponseDto = mapper.Map<LoginRegisterRefreshResponseDto>(loginRegisterRefreshResponseModel);

// was the email confirmation sent successfully

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

else return TypedResults.Ok(loginRegisterRefreshResponseDto.Message);

})

.RequireAuthorization()

.WithName("ViewUserDetailsAsync")

.WithOpenApi(x => new OpenApiOperation(x)

{

Summary = "View user's details",

Description = "View Email, Phone, First Name, Last Name, Roles and DOB",

Tags = new List<OpenApiTag> { new OpenApiTag { Name = "Login/Register/Refresh API Library" } }

});

### Service Method

public async Task<LoginRegisterRefreshResponseModel> **ViewUserDetailsAsync**(string userName)

{

Guard.Against.Null(userName, null, "User credentials are not valid");

var response = new LoginRegisterRefreshResponseModel() { Message = $"Details for {userName}" }; // default message

var identityUser = await \_userManager.FindByNameAsync(userName);

if (identityUser == null) response.Message = "Invalid credentials supplied.";

else

{

response.IsStatus = true;

response.Message += Environment.NewLine;

response.Message += $"User ID: {identityUser.Id}\n";

response.Message += $"Username: {identityUser.UserName}\n";

response.Message += $"Email: {identityUser.Email}\n";

response.Message += $"Phone: {identityUser.PhoneNumber}\n";

response.Message += $"First Name: {identityUser.FirstName}\n";

response.Message += $"Last Name: {identityUser.LastName}\n";

response.Message += $"DOB: {identityUser.DateOfBirth}\n";

var roles = await \_userManager.GetRolesAsync(identityUser);

response.Message += "Roles: " + string.Join(", ", roles);

}

return response;

}

## Request Password Reset

A user can request to update their password, with an email being sent to them with a valid reset code (which is time restricted). Your application would then have an entry page to capture their new password along with the reset code.

A black and white text

Description automatically generated

A screenshot of a computer

Description automatically generated

### Api Method

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Request password reset (this will send an email with a link to reset password) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

passwordGroup.MapPost("/**RequestPasswordResetAsync**", async Task<Results<Ok<string>, NotFound<string>>> (HttpContext context, IAuthService authService) =>

{

var loginRegisterRefreshResponseDto = await authService.**RequestPasswordResetAsync**(context.Request.Cookies["jwtToken"]!);

// was the password reset email sent successfully

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

else return TypedResults.Ok(loginRegisterRefreshResponseDto.Message);

})

.WithName("RequestPasswordResetAsync")

.RequireAuthorization()

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

.WithOpenApi(x => new OpenApiOperation(x)

{

Summary = "User requests an email with a link to reset the password - calls VerifyPasswordResetAsync API",

Description = "Request a password reset email.",

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

});

### Service Method

public async Task<LoginRegisterRefreshResponseModel> **RequestPasswordResetAsync**(string jwtToken)

{

var response = new LoginRegisterRefreshResponseModel();

var claimsPrincipal = \_jwtTokenService.GetPrincipalFromExpiredToken(jwtToken);

// not able to retrieve user from Jwt bearer token

if (claimsPrincipal?.Identity?.Name is null)

{

response.Message = "Jwt Bearer is not valid, during password request";

return response;

}

response.Message = $"A password reset request has been sent to user - {claimsPrincipal.Identity.Name}."; // default message

var identityUser = await \_userManager.FindByNameAsync(claimsPrincipal.Identity.Name);

if (identityUser == null) response.Message = "An invalid email or the email is not register to your account";

else

{

var token = await \_userManager.**GeneratePasswordResetTokenAsync**(identityUser);

await SendEmailTaskAsync(identityUser, token, EmailType.PasswordReset);

response.IsStatus = true; // double up for validating password sent successfully

}

return response;

}

## Verify Password Reset

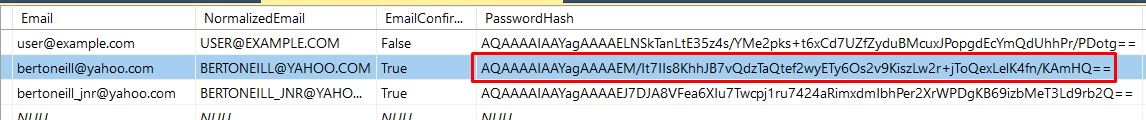
A screenshot of a computer

Description automatically generated

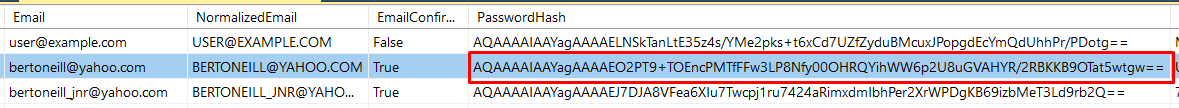
A close up of a screen

Description automatically generated

Password Hash in database before the update:



Password Hash in database after the update:



If I, then try to log in with the old password:

A black and white background with white text

Description automatically generated

### Api Method

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Verify password reset (this will be called by email link to with new password) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

passwordGroup.MapPost("/**VerifyPasswordResetAsync**", async Task<Results<Ok<string>, NotFound<string>>> (HttpContext context, string token, string newPassword, string oldPassword, IAuthService authService) =>

{

Guard.Against.Empty(oldPassword, "Old password is missing");

Guard.Against.Empty(newPassword, "New password is missing");

Guard.Against.Empty(token, "Password reset token is missing");

var loginRegisterRefreshResponseDto = await authService.**VerifyPasswordResetAsync**(context.Request.Cookies["jwtToken"]!, token, newPassword, oldPassword);

// was the email confirmation successful

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

else return TypedResults.Ok(loginRegisterRefreshResponseDto.Message);

})

.WithName("VerifyPasswordResetAsync")

.WithOpenApi(x => new OpenApiOperation(x)

{

Summary = "Redirected to API by a link within an email - thus appropriate user gets link",

Description = "Request a password reset email.",

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

});

### Service Method

public async Task<LoginRegisterRefreshResponseModel> **VerifyPasswordResetAsync**(string jwtToken, string token, string newPassword, string oldPassword)

{

Guard.Against.Null(jwtToken, null, "JWT Bearer is not valid");

Guard.Against.Null(token, null, "Token is not valid");

Guard.Against.Null(newPassword, null, "Password is not valid");

var response = new LoginRegisterRefreshResponseModel() { Message= "Password was reset successfully." }; // default message

var claimsPrincipal = \_jwtTokenService.GetPrincipalFromExpiredToken(jwtToken);

// not able to retrieve user from Jwt bearer token

if (claimsPrincipal?.Identity?.Name is null)

{

response.Message = "Jwt Bearer is not valid, during password request";

return response;

}

var identityUser = await \_userManager.FindByNameAsync(claimsPrincipal.Identity.Name);

if (identityUser == null) response.Message = "Invalid credentials or token";

else

{

// Verify the old password

var passwordCheck = await \_userManager.**CheckPasswordAsync**(identityUser, oldPassword);

if (!passwordCheck)

{

response.Message = "Old password is incorrect";

return response;

}

var result = await \_userManager.**ResetPasswordAsync**(identityUser, token, newPassword);

if (result.Succeeded) response.IsStatus = true;

else

{

var errors = new StringBuilder();

result.Errors.ToList().ForEach(err => errors.AppendLine($"{err.Description}")); // build up a string of faults

response.Message = errors.ToString();

}

}

return response;

}

## Lockout Account (Admin)

The **Identity Lockout API** in ASP.NET Core Identity is an important security feature that helps protect your application from **brute force attacks** and unauthorized access attempts. I have restricted access to this API with a Policy (only for Admins to use).

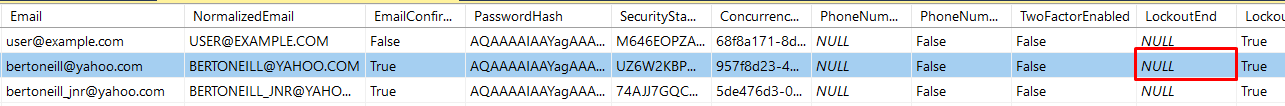
A black and white text

Description automatically generated

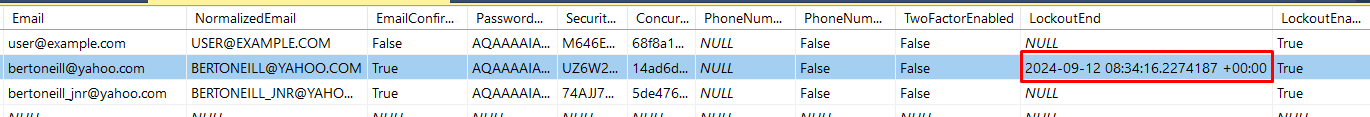
A screenshot of a email

Description automatically generated

Database before lockout:



Database after lockout:



The user will be notified by email that their account has been locked out, and they have access to an API to request that their account is unlocked.

### Api Method

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Lock a user's account (called by Admin) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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

{

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

var loginRegisterRefreshResponseDto = await authService.**LockAccountAsync**(userName);

// was the email confirmation sent successfully

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

else return TypedResults.Ok(loginRegisterRefreshResponseDto.Message);

})

.WithName("Lock Account - Admin")

**.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 lock a user's account",

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

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

});

### Service Method

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Locking & Unlocking an account \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public async Task<LoginRegisterRefreshResponseModel> **LockAccountAsync**(string userName)

{

Guard.Against.Null(userName, null, "User credentials are not valid");

int adminLockoutDuration = int.Parse(\_config.GetSection("SystemDefaults:AdminLocksAccountLifeSpan").Value); // locked for days

var response = new LoginRegisterRefreshResponseModel() { Message = $"User {userName}'s account has been locked." }; // default message

var identityUser = await \_userManager.FindByNameAsync(userName);

if (identityUser == null) response.Message = "Invalid credentials supplied.";

else

{

// lock user's account for a longer period (appsetting value in days - user can request unlock email)

var result = await \_userManager.**SetLockoutEndDateAsync**(identityUser, DateTimeOffset.UtcNow.AddDays(adminLockoutDuration));

if (!result.Succeeded)

{

response.Message = $"Not able to lock the account of {userName}.";

var errors = new StringBuilder();

result.Errors.ToList().ForEach(err => errors.AppendLine($"{err.Description}")); // build up a string of faults

response.Message = errors.ToString();

}

else

{

response.IsStatus = true;

}

}

return response;

}

## Request Unlock Account (User)

A black and white text

Description automatically generated

A screenshot of a computer

Description automatically generated

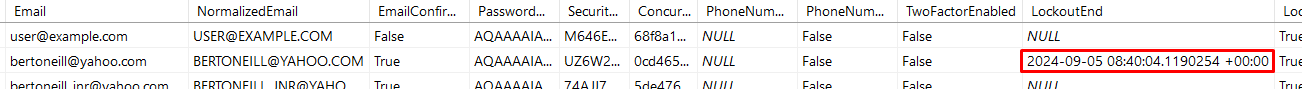
When the user clicks on the link, it will unlock their account:

A screen shot of a computer

Description automatically generated

Database after unlocking:

The LockoutEnd value is set to Now, so when the user tries to log back in, the restriction doesn’t apply.



### Api Method

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* User request's their account to be unlocked (email conformation sent) - user has no JWT at this stage \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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

{

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

var loginRegisterRefreshResponseDto = await authService.**RequestUnlockAsync**(userName);

// was the email confirmation sent successfully

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

else return TypedResults.Ok(loginRegisterRefreshResponseDto.Message);

})

.WithName("UnlockRequestAsync")

.WithOpenApi(x => new OpenApiOperation(x)

{

Summary = "User request to unlock their account",

Description = "A user requests to unlock their account - email sent with link",

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

});

### Service Method

public async Task<LoginRegisterRefreshResponseModel> **RequestUnlockAsync**(string userName)

{

Guard.Against.Null(userName, null, "User credentials are not valid");

var response = new LoginRegisterRefreshResponseModel() { Message = $"An unlock email request has been sent to user - {userName}." }; // default message

var identityUser = await \_userManager.FindByNameAsync(userName);

if (identityUser == null) response.Message = "An invalid email or the email is not register to your account";

else

{

// Generate an unlock token for the user

var unlockToken = await \_userManager.**GenerateUserTokenAsync**(identityUser, TokenOptions.DefaultProvider, TokenOptions.DefaultEmailProvider);

await SendEmailTaskAsync(identityUser, unlockToken, EmailType.UnlocKAccountRequested);

response.IsStatus = true; // double up for validating password sent successfully

}

return response;

}

## Unlock Account (Admin)

An Admin user also has the privileges to unlock a user’s account – an email will be sent to the user stating that their account has been unlocked.

A black and white text on a black and white background

Description automatically generated

A screenshot of a email

Description automatically generated

### Api Method

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Unlock a user's account (called by Admin) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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

{

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

var loginRegisterRefreshResponseDto = await authService.**UnlockAccountVerifiedByAdminAsync**(userName);

// was the email confirmation sent successfully

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

else return TypedResults.Ok(loginRegisterRefreshResponseDto.Message);

})

.WithName("Unlock Account - Admin")

.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 unlock a User's account",

Description = "A feature where an Admin can unlock a User's account",

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

});

### Service Method

public async Task<LoginRegisterRefreshResponseModel> **UnlockAccountVerifiedByAdminAsync**(string userName)

{

Guard.Against.Null(userName, null, "User credentials are not valid");

var response = new LoginRegisterRefreshResponseModel() { Message = $"User {userName}'s account has been unlocked." }; // default message

var identityUser = await \_userManager.FindByNameAsync(userName);

if (identityUser == null) response.Message = "Invalid credentials supplied.";

else

{

var result = await \_userManager.**SetLockoutEndDateAsync**(identityUser, DateTimeOffset.UtcNow);

if (!result.Succeeded)

{

response.Message = $"Not able to unlock the account of {userName}.";

var errors = new StringBuilder();

result.Errors.ToList().ForEach(err => errors.AppendLine($"{err.Description}")); // build up a string of faults

response.Message = errors.ToString();

}

else

{

response.IsStatus = true;

}

}

return response;

}

**Unlock Verification by Email**

### Api Method

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Verify email link to unlock account (user will have gotten an email to verify) - user has no JWT at this stage \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

lockoutGroup.MapGet("/**UnlockRequestVerifiedByEmailAsync**", async Task<Results<Ok<string>, NotFound<string>>> (string userName, string token, IAuthService authService) =>

{

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

Guard.Against.Empty(token, "Token is missing");

var loginRegisterRefreshResponseDto = await authService.**UnlockAccountVerifiedByEmailAsync**(userName, token);

// was the email confirmation sent successfully

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

else return TypedResults.Ok(loginRegisterRefreshResponseDto.Message);

})

.WithName("UnlockRequestVerifiedByEmailAsync")

.WithOpenApi(x => new OpenApiOperation(x)

{

Summary = "Unlock a user's account",

Description = "Admin can unlock a user's account",

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

});

### Service Method

public async Task<LoginRegisterRefreshResponseModel> **UnlockAccountVerifiedByEmailAsync**(string userName, string token)

{

Guard.Against.Null(userName, null, "User credentials are not valid");

Guard.Against.Null(token, null, "Token is not valid");

var response = new LoginRegisterRefreshResponseModel() { Message = $"User {userName}'s account has been unlocked." }; // default message

var identityUser = await \_userManager.FindByNameAsync(userName);

if (identityUser == null) response.Message = "Invalid credentials supplied.";

else

{

// Verify the token

var isTokenValid = await \_userManager.**VerifyUserTokenAsync**(identityUser,

TokenOptions.DefaultProvider,

TokenOptions.DefaultEmailProvider,

token);

if (!isTokenValid)

{

response.Message = "Invalid or expired token.";

}

else

{

var result = await \_userManager.**SetLockoutEndDateAsync**(identityUser, DateTimeOffset.UtcNow);

if (!result.Succeeded)

{

response.Message = $"Not able to unlock the account of {userName}.";

var errors = new StringBuilder();

result.Errors.ToList().ForEach(err => errors.AppendLine($"{err.Description}")); // build up a string of faults

response.Message = errors.ToString();

}

else

{

response.IsStatus = true;

}

}

}

return response;

}

# Miscellaneous

## Acronym

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