Thoughts.io

Development process documentation

# Introduction

A full-stack web application where users can securely share their thoughts with the world or keep them private.

## Project Goal

The goal of **Thoughts.io** is to create a clean and user-friendly platform where registered users can create short text entries, or "thoughts". The system allows for controlling the visibility of these entries (public or private), providing a space for both personal notes and community interactions. The project showcases modern web development techniques through a well-structured, maintainable codebase, strictly separating front-end and back-end logic.

## Key Features

### For All visitors (Unauthenticated)

* **Browse Public Thoughts:** The main page lists the latest public entries.
* **View Reactions:** Visitors can see how others have reacted to public thoughts.
* **Registration & Login:** Ability to create a new account and log into an existing one.

### For Logged-in Users

* **Thought Management (CRUD):** *Create*, *edit*, *read* and *delete* your own thoughts on a dedicated user dashboard.
* **Set Visibility:** Each thought's visibility can be individually set to private (only the author can see it) or public (everyone can see it).
* **Interaction:** Add reactions (e.g., likes) to other users' public thoughts.
* **Personal Dashboard:** Users can view all their created thoughts, including private ones.

### Admin Panel

* **User Management:** Administrators can view and manage registered users' data.
* **Content Moderation:** Ability to edit or delete any thought from any user, ensuring the platform's integrity.
* **Statistics:** An overview dashboard to monitor key metrics (e.g., number of registered users, number of thoughts created).

## Preparation

Firstly, I assessed what kind of packages I would need for the entire project. Obviously, the number of packages could extend later on depending on my needs.

These are the packages I have selected in the beginning:

* **FluentValidation:** Library to validate user inputs easily.
* **Bcrypt.Net-Next:** Password hashing library to protect against rainbow table attacks.
* **Microsoft.AspNetCore.Authentication.JwtBearer:** Authentication library that uses JsonWebTokens (JWTs) to authenticate and authorize users.
* **Microsoft.EntityFrameWorkCore:** Object-relational mapper (ORM) to work with a database using .NET objects.
* **Microsoft.EntityFrameworkCore.SqlServer:** Extension library for EF (core) to be able to connect various SQL servers.
* **ZiggyCreatures.FusionCache:** Easy to use, fast and robust hybrid cache with advanced resiliency features.
* **Serilog:** Provides diagnostic logging to files, the console, and elsewhere.

## Configuring services and app

In my recent projects I have always had a ConfigureServices.cs and a ConfigureApp.cs file. This separation mirrors the two core phases of the modern ASP.NET Core application lifecycle: service registration and request pipeline configuration.

ConfigureServices.cs would add all the necessary services to the Dependency Injection (DI) container via extension methods on the WebApplicationBuilder.

Goal: To register all required components and services (e.g., database context, authentication mechanisms, custom services, logging configuration, caching) that the application will need during runtime.

ConfigureApp.cs would configure the application's request processing pipeline (middleware) using extension methods on the fully built WebApplication object.

Goal: To define how incoming HTTP requests are processed. Middleware components are added in a specific order, as they handle the request sequentially (e.g., logging, error handling, routing, authentication, CORS).