HAMK
Hämeen ammatti-
korkeakoulu

Project planner

.NET MVC

Information and Communication Technology

Summer 2025

Aleksi Häyrynen

HAMK
Hämeen ammatti-
korkeakoulu

Abstract

Koulutus	Koulutuksen nimi
Tekijä	Aleksi Häyrynen
Työn nimi	FullStack Project Planner
Ohjaaja	-

Vuosi 2025

This project is a full-stack web application developed using Microsoft's **.NET Core** framework, designed to follow the **Model-View-Controller (MVC)** architectural pattern while integrating modern **Blazor** components to enhance interactivity and user experience. The application serves as a task and project management system that supports multiple user roles such as Owners, Reviewers, and Contributors, each with distinct permissions to ensure secure and appropriate access control.

The backend is implemented in **C#**, utilizing **MongoDB** as a NoSQL database for flexible, scalable data storage. Security is a primary concern, with user credentials protected using the **Argon2** encryption algorithm. The application supports all essential **CRUD** (Create, Read, Update, Delete) operations, enabling users to manage projects, tasks, and profile information dynamically.

Deployment is handled on **Microsoft Azure**, providing a reliable and scalable environment suitable for real-world applications. Throughout the project, emphasis was placed on secure data handling, role-based authorization, and a clean separation of concerns in the system architecture.

While the application is fully functional and demonstrates a comprehensive range of web development skills, there is potential for future improvements, including further unification of the frontend design and optimization of backend services.

NET Core, C#, MongoDB, Azure, Argon2 encryption, and Blazor components.

.NET, Azure, C#, Encryption, Web Development

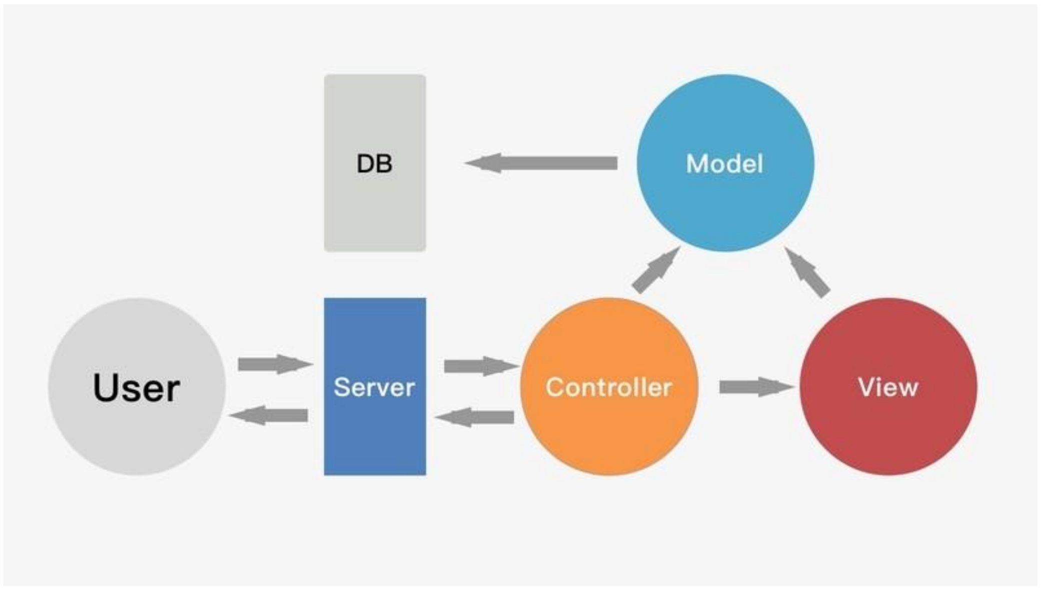
Pages 12 Pages

Tabel of Contents

1 System Architecture

The application is built using Microsoft's .NET Core 8.0, a modern, high-performance, and cross-platform framework ideal for building scalable web applications. At its core, the project follows the Model-View-Controller (MVC) architectural pattern, which separates the application into three primary components—Models, Views, and Controllers—to ensure a clean separation of concerns, making the codebase more maintainable, testable, and scalable.

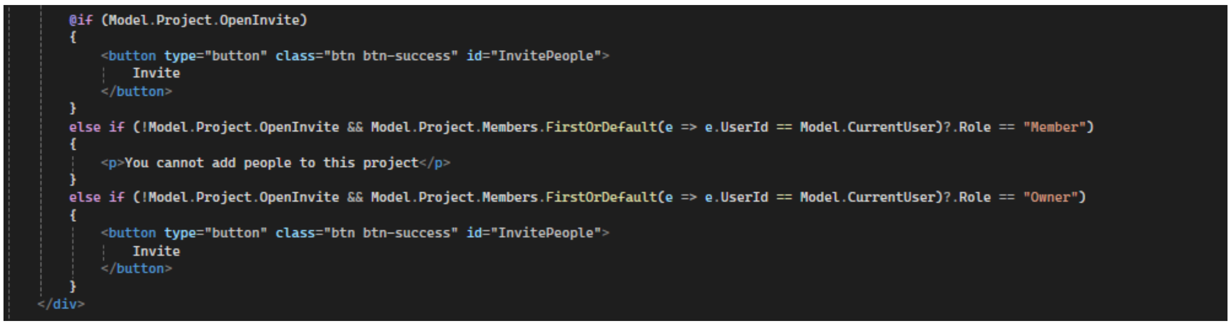
Figure 1 Model-View-Controller (MVC) Architecture



1.1 View

The View component is responsible for rendering the user interface and presenting data to the user. In this application, Views are primarily implemented using Razor pages, which combine HTML markup with C# code to dynamically generate content. Razor syntax allows seamless integration of server-side data into the HTML, making the UI responsive to changes in the application's state.

Figure 2 Razor syntax example



```
        @if (Model.Project.OpenInvite)
        {
            <button type="button" class="btn btn-success" id="InvitePeople">
                Invite
            </button>
        }
        else if (!Model.Project.OpenInvite && Model.Project.Members.FirstOrDefault(e => e.UserId == Model.CurrentUser)?.Role == "Member")
        {
            <p>You cannot add people to this project</p>
        }
        else if (!Model.Project.OpenInvite && Model.Project.Members.FirstOrDefault(e => e.UserId == Model.CurrentUser)?.Role == "Owner")
        {
            <button type="button" class="btn btn-success" id="InvitePeople">
                Invite
            </button>
        }
    </div>
```

1.2 Controller

Controllers handle HTTP requests, including user authentication and authorization by verifying user roles and permissions before granting access. They manage form data submissions and route parameters, passing this data through action methods to services and views. This enables dynamic interaction across the application while maintaining security and clean data flow. I personally used these exactly like this limiting access from certain users and passing parameters around the program

1.3 Models

In this project, Models are C# classes that represent the main data entities, such as User, Project, or Task. They define the structure and properties of the objects we want to save and retrieve from the MongoDB database. Models serve as the blueprint for the data throughout the application, ensuring consistency between the database and the program’s logic.

1.4 Services

I used Services extensively to abstract and separate business logic from the Controllers, keeping the code cleaner and easier to maintain. Instead of placing database queries or complex operations directly inside Controllers, I moved them into dedicated Service classes. This improved clarity and allowed better reusability across the application.

The services are injected into Controllers using dependency injection, which is built into .NET Core 8.0. This approach simplifies testing and promotes loose coupling

. Key services I implemented include:

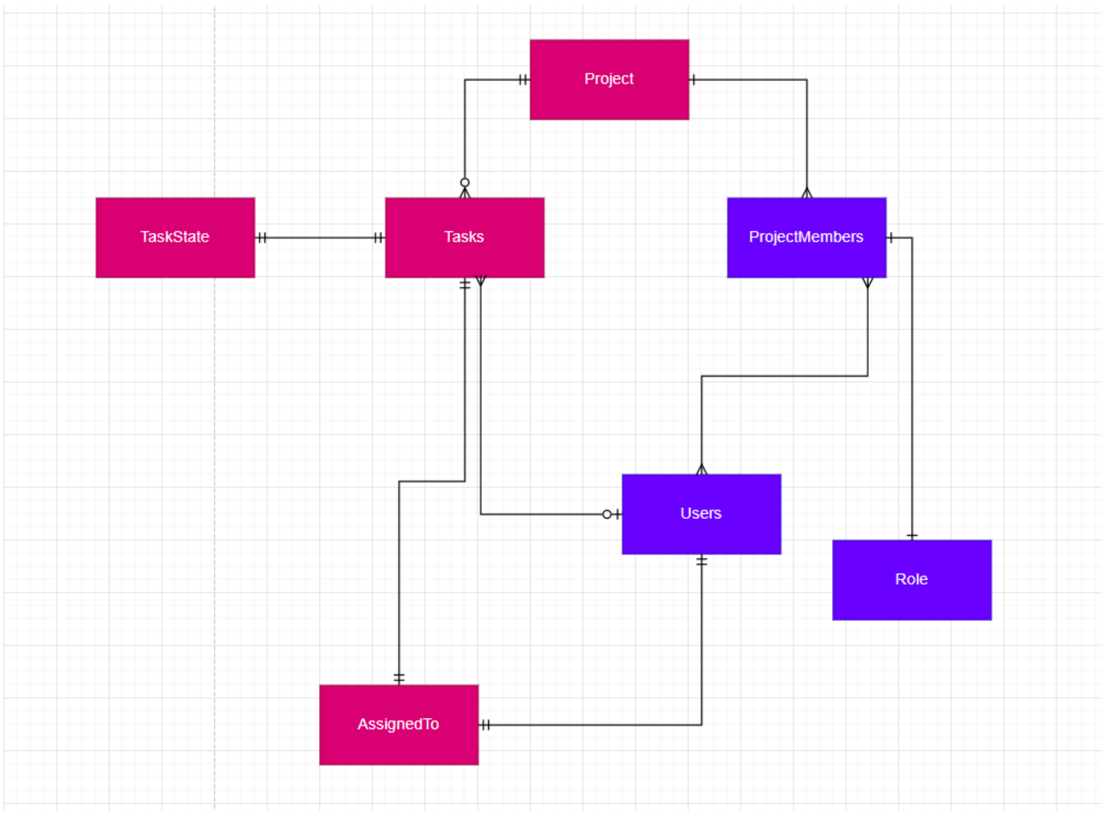
- **TaskService** – handles task-related operations and logic
- **ProjectService** – manages project data and actions
- **ProfilePictureService** – deals with uploading, retrieving, and managing user images
- **DatabaseService** – general database interaction and data access

.

2 Data Model Design

I utilized multiple specialized models to represent distinct roles and functionalities across different project contexts. To simulate relationships between these models, I leveraged MongoDB’s **ObjectId** as a unique identifier, effectively linking related data. MongoDB’s flexible schema design enabled me to manage complex data structures with ease, while its powerful indexing capabilities ensured fast querying. Additionally, its high scalability supports the application’s growth and performance demands seamlessly.

Figure 3 Very simple version of my model structure



In the picture above I have an extremely simplified Model structure of my application. You can see very well how most of the classes are connected to either the “User” or the “Project” itself. This allowed me to manage user roles and task in separate environments.

3 Security

Security is a critical aspect of this application to protect user data, maintain privacy, and ensure that only authorized users can access or modify resources. The system combines robust authentication and role-based authorization with additional safeguards to prevent common web vulnerabilities and abuse.

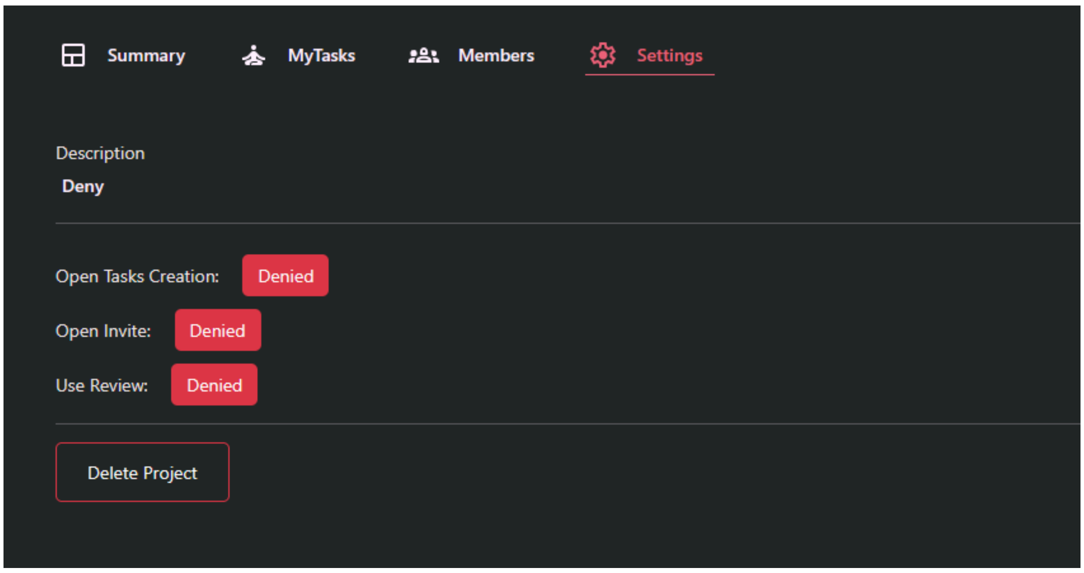
3.1 Role-Based Access Control

The application enforces strict role-based access control both at the UI level and within backend controllers. Users are assigned roles such as “Owner,” “Reviewer,” or general member, which determine their permissions:

- Owners have full control over projects and members.
- Reviewers can access and manage tasks marked as “InReview.”
- Access to project resources is validated in controllers by checking if the authenticated user belongs to the relevant project group and role.
- Functions unavailable to a user’s role are hidden in the UI, but backend checks prevent unauthorized operations even if the UI restrictions are bypassed.

Projects include additional settings that enable all members to create and manage tasks, as well as invite new participants. Currently, these settings are fixed and cannot be modified. Only the project Owner can view these settings and delete the project.

Figure 4 Project settings for the owner



Description
Deny

Open Tasks Creation: Denied
Open Invite: Denied
Use Review: Denied

3.2 Additional Security Measures

To strengthen the overall security posture, the application implements multiple safeguards addressing common threats and input validation:

- Anti-Spam and Anti-Forgery: The use of anti-forgery tokens on form submissions protects against Cross-Site Request Forgery (CSRF) attacks.
- File Upload Validation: Uploaded files are validated for allowed types (e.g., .jpg, .png, .gif) and restricted by size limits to prevent malicious files and resource abuse.
- Input Validation: All critical inputs such as user IDs and task IDs are validated to avoid injection attacks and unauthorized data manipulation.
- Fallback Defaults: The system uses default values, like a default profile picture, to handle missing or invalid data gracefully without exposing sensitive information.

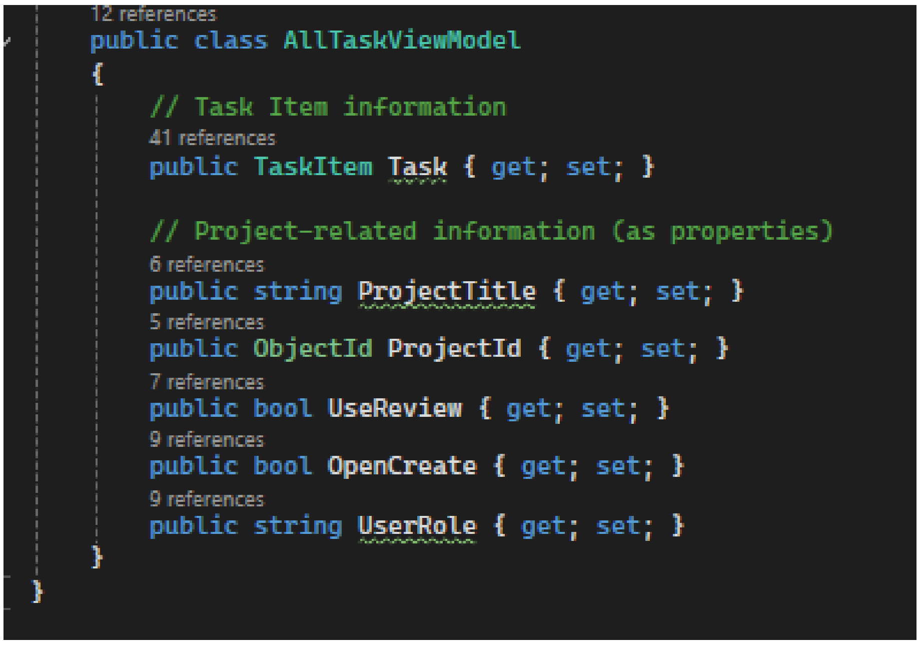
3.3 AppSettings abstraction

To protect sensitive configuration values such as database connection strings and encryption keys, I excluded them from version control by placing them in appsettings.json and adding them to .gitignore. This ensures that critical information like IP addresses and encryption settings are not exposed in the source code or shared repositories. Looking ahead, a more secure approach would be to migrate these settings to environment variables managed through Microsoft's secure configuration system, such as Azure Key Vault or user secrets, to further strengthen the application's security posture.

4 ViewModels

ViewModels played a crucial role in this project by simplifying and structuring data specifically for the user interface. They allowed me to tailor the data sent from the back-end to exactly what each view required, improving both clarity and performance.Key ViewModels like **ProjectWithTaskViewModel** and **AllTaskViewModel** encapsulate related data — such as projects, tasks, and user roles — making it straightforward to bind this data in Razor views and Blazor components.

Figure 5AllTaskViewModel



```
12 references
public class AllTaskViewModel
{
    // Task Item information
    41 references
    public TaskItem Task { get; set; }

    // Project-related information (as properties)
    6 references
    public string ProjectTitle { get; set; }
    5 references
    public ObjectId ProjectId { get; set; }
    7 references
    public bool UseReview { get; set; }
    9 references
    public bool OpenCreate { get; set; }
    9 references
    public string UserRole { get; set; }
}
}
```

In the picture above (Figure 5) I have combined properties from multiple Models like **Project, TaskItem, ProjectMembers**

4.1 Data Flow & Parameter Passing

The application follows a clear data flow pipeline:

Database → Service → Controller → ViewModel → View

When a user navigates to a Project page, the current Project ID is passed through various layers and retained throughout the user’s interaction with the project-related pages. This approach avoids redundant queries by reusing the Project ID and cached data within the session or page context.

Most queries are performed on larger collections or aggregated objects — for example, fetching a list of projects with their related tasks — instead of multiple smaller, repeated queries. This reduces database load and improves response times, ensuring a smoother user experience.

5 Packages and File upload

The project had some dependencies, and they are either connected to the MongoDB or the Image handling. The user can upload and use profile pictures, and I opted to handle this without an external storage

5.1 Project Dependencies

The project relies on several key NuGet packages to support core functionality:’

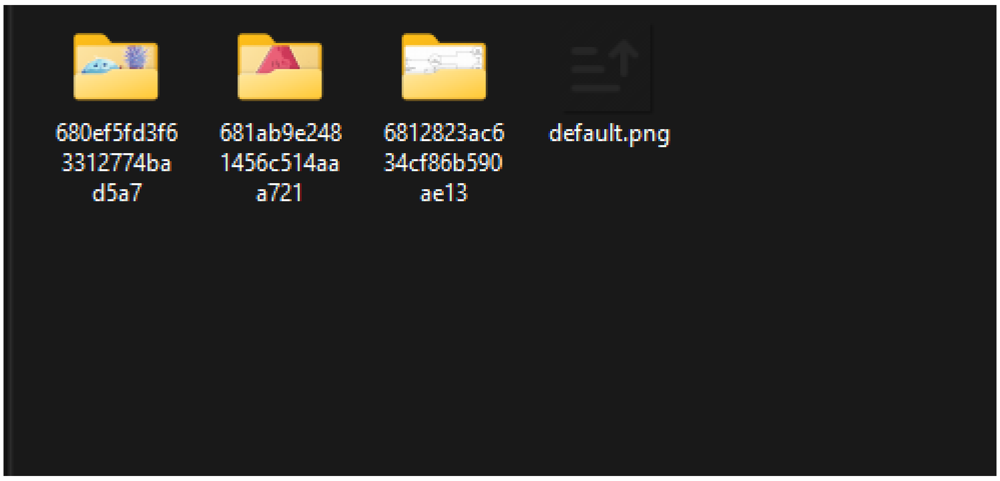
- MongoDB.Driver, MongoDB.Bson, and MongoDB.Driver.Core: These are the standard packages for interacting with MongoDB, providing a powerful and flexible database solution ideal for the app’s dynamic data needs.
- SixLabors.ImageSharp: Used for advanced image processing tasks such as resizing and format conversion, allowing consistent handling of user-uploaded images.
- Microsoft.AspNetCore.StaticFiles (for IContentTypeProvider): Helps detect and serve correct MIME types for files, ensuring proper content delivery.

5.2 File Upload Constraints & Processing

Profile image uploads are limited to **.jpg**, **.png**, and **.gif** formats, with a maximum size of **5 MB** to ensure consistency and protect performance. Uploaded images are resized to **150x150 pixels** using ImageSharp, optimizing them for display across the application.

Images are saved in user-specific folders and renamed with GUIDs to avoid filename conflicts (Figure6). If an image is missing or invalid, the system defaults to serving a default.png. The correct MIME type is detected using IContentTypeProvider to ensure safe and proper rendering in browsers.

Figure 6 different users in the image system

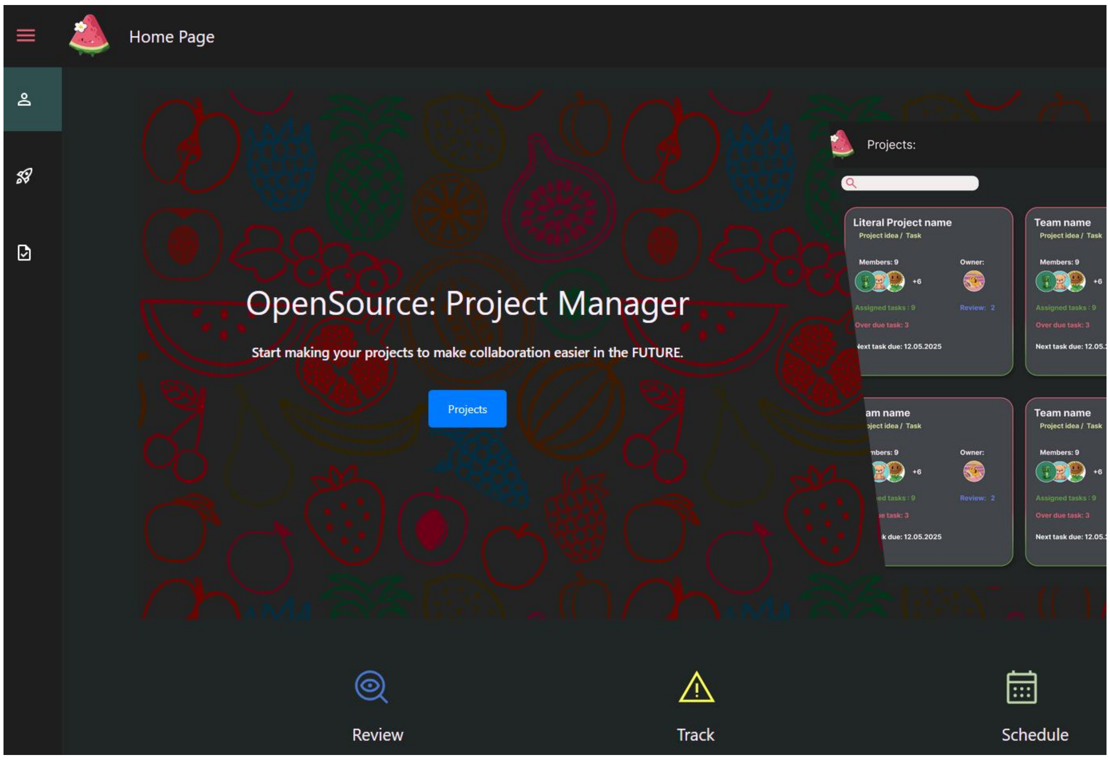


680ef5fd3f63312774bad5a7
681ab9e2481456c514aaa721
6812823ac634cf86b590ae13
default.png

6 UI & UX

The application features a dark theme aimed at providing a modern, visually consistent experience. Main inspiration came for the cure watermelon pictures initially. I personally like having a bit fun with the design and all its colour choices. To improve usability, basic filtering and search functionality were implemented for task lists and projects. While these features cover essential use cases, there is still room for enhancement—such as more dynamic filters and customizable views—to further improve user experience. In (Figure7) You can see the front-page design.

Figure 7Front-page design



OpenSource: Project Manager

Start making your projects to make collaboration easier in the FUTURE.

7 Publishing & Deployment

The application is published and hosted using IIS on a Windows Virtual Machine (VM). Deployment was handled through Visual Studio's publish profiles, targeting the production environment.

Key Considerations:

- MongoDB is hosted externally (e.g., MongoDB Atlas) to ensure scalability and remote access.
- User file storage uses absolute paths (e.g., C:\UserImages\) on the server, with proper read/write permissions configured.
- Sensitive configuration values like connection strings and encryption keys are excluded from the codebase and managed securely in the appsettings.json file.
- The hosting environment is configured to serve static files safely while preventing access to private directories.

This setup provides a realistic production environment with full control over hosting, file access, and security. In the future you could move the database and filesystem to be external parts like Blob storage.

8 Conclusion

This project successfully demonstrates the development of a full-featured, secure, and maintainable web application using .NET 8.0 and the MVC pattern, enhanced with Blazor components and MongoDB integration. Through thoughtful separation of concerns, clear architectural planning, and role-based access control, I was able to implement key functionalities like task management, image handling, and group collaboration in a scalable way. The system's structure—paired with strong service abstraction, encryption, and responsive UI elements—ensures both robustness and user-friendliness.

Moving forward, there are clear directions for improvement to further evolve the platform. One important step is migrating file storage and the database to external services, which would improve scalability and simplify deployment. Enhancing the user experience is also a key priority — this includes making filtering more dynamic, converting repeated structures into partial views or Blazor components, and unifying the UI design across all pages. Finally, introducing real-time features like online user tracking and basic messaging would significantly increase the application's interactivity and overall value. These enhancements will help transform the system into a more polished, production-ready product.