Chapter 4: Implementation

# 4.1: Software Architecture

Our software architecture is a client-server architecture, and we implemented it as follows:

Our website is divided into two parts.

The first part is the client

which we implemented with.NET 6 Razor Pages .NET 6 Razor Pages offers several compelling reasons for developers to choose it as their preferred framework. Here are a few key points highlighting why it is a good choice:

1. Simplicity and Productivity: Razor Pages embraces a minimalist approach, making it simple and straightforward to build web applications. With its convention-based model, developers can quickly create pages without the need for complex configurations. This simplicity enhances productivity, allowing developers to focus more on building the application's features rather than getting caught up in boilerplate code.
2. Seamless Integration with ASP.NET Core: Razor Pages is an integral part of ASP.NET Core, which provides a robust and versatile web development framework. It seamlessly integrates with other ASP.NET Core components, such as middleware, routing, and dependency injection, enabling developers to leverage the full power of the ecosystem while building their applications.
3. Powerful Templating Engine: Razor, the templating engine used in Razor Pages, offers a powerful and intuitive syntax that combines HTML markup with C# code seamlessly. This fusion allows developers to create dynamic web pages easily, with the ability to embed C# code directly into the views. Razor's syntax provides excellent readability, maintainability, and code reusability.
4. Powerful Templating Engine: Razor, the templating engine used in Razor Pages, offers a powerful and intuitive syntax that combines HTML markup with C# code seamlessly. This fusion allows developers to create dynamic web pages easily, with the ability to embed C# code directly into the views. Razor's syntax provides excellent readability, maintainability, and code reusability.
5. Powerful Templating Engine: Razor, the templating engine used in Razor Pages, offers a powerful and intuitive syntax that combines HTML markup with C# code seamlessly. This fusion allows developers to create dynamic web pages easily, with the ability to embed C# code directly into the views. Razor's syntax provides excellent readability, maintainability, and code reusability.
6. Smooth Migration and Compatibility: If you're already familiar with ASP.NET Web Forms, migrating to Razor Pages is relatively straightforward. Razor Pages provides a similar programming model, allowing you to gradually migrate your existing applications while taking advantage of modern web development practices and technologies.
7. Vibrant Community and Ecosystem: Razor Pages is backed by a vibrant and active community of developers. It benefits from the extensive ecosystem of libraries, tools, and extensions available for ASP.NET Core. The community support ensures that you can find resources, tutorials, and assistance whenever you encounter any challenges during development.

## The second part is the server

Here, our backend is divided into two projects

* .NET 6 Web API

NET 6 Web API provides a robust and versatile framework for building backend services, offering several compelling reasons to choose it for your application's backend:

1. High Performance and Scalability: .NET 6 Web API is built on top of the high-performance ASP.NET Core framework, optimized for handling heavy workloads and delivering exceptional performance. It leverages the latest enhancements in .NET 6, such as minimal API templates and performance optimizations, enabling you to build scalable and efficient backend services capable of handling large volumes of requests.
2. Cross-Platform Compatibility: .NET 6 Web API is a cross-platform framework, allowing you to develop and deploy your backend services on Windows, macOS, and Linux environments. This flexibility enables you to target a wide range of platforms, ensuring your API can be consumed by various clients, including web browsers, mobile applications, and IoT devices.
3. Rich Ecosystem and Tooling: .NET has a mature and extensive ecosystem with a wide range of libraries, packages, and tools. This ecosystem enables developers to accelerate development by leveraging existing components, such as authentication frameworks, database connectors, logging tools, and more. Additionally, Visual Studio and Visual Studio Code provide excellent integrated development environments (IDEs) with robust debugging and profiling capabilities.
4. Security and Authentication: .NET 6 Web API offers built-in support for authentication and authorization mechanisms, making it easier to secure your backend services. It supports various authentication schemes, including JWT (JSON Web Tokens), OAuth, and OpenID Connect. Additionally, the framework provides mechanisms for role-based access control (RBAC) and claims-based authorization, allowing you to implement fine-grained security policies.
5. Flexible Routing and Middleware: Web API in .NET 6 uses a powerful routing system that allows you to define flexible and customizable routes. This routing system enables you to map incoming requests to appropriate controller actions efficiently. Additionally, the middleware pipeline in ASP.NET Core gives you granular control over request processing, allowing you to add custom middleware for logging, error handling, caching, and more.
6. Strongly Typed and Productive Development: With .NET 6, Web API supports C# 10 language features, enabling developers to write clean, readable, and maintainable code. The use of strongly typed models and validation attributes enhances code correctness and reduces errors. Furthermore, features like automatic model binding, input validation, and content negotiation simplify the development process, leading to increased productivity.

* Flask API (Python)

We used Flask API for a small part of our project only to handle our AI.

# 4.2: Flowchart

