**Technical Documentation**

**1. Architecture Overview**

This project follows a layered architecture, structured into the following components:

**1.1 Presentation Layer (UI)**

* Framework: Blazor (or Razor Pages)
* Role: Responsible for rendering the UI and handling user interactions. This layer communicates with the Application Layer to manage user input (such as creating and managing to-do tasks).

**1.2 Application Layer**

* Role: Contains the business logic. It processes user requests, enforces business rules, and interacts with the data through the Data Access Layer (Repository).

**1.3 Data Access Layer (DAL)**

* Framework: Entity Framework Core
* Role: Responsible for communicating with the SQLite database. It uses repositories to encapsulate data access logic, allowing the Application Layer to interact with the database in a cleaner manner.

**1.4 Database**

* Type: SQLite
* Purpose: Stores user data and to-do items. SQLite is used for simplicity and ease of deployment.

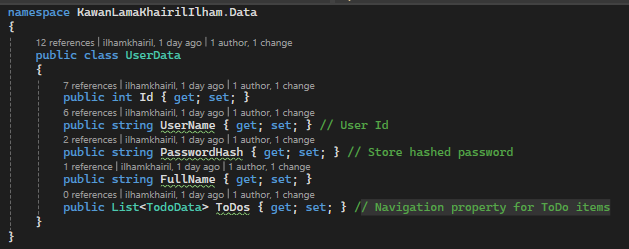
**1.5 Containerization**

* Tool: Docker
* Purpose: Containerizes the application to ensure it runs consistently across different environments.

**2. Data Models**

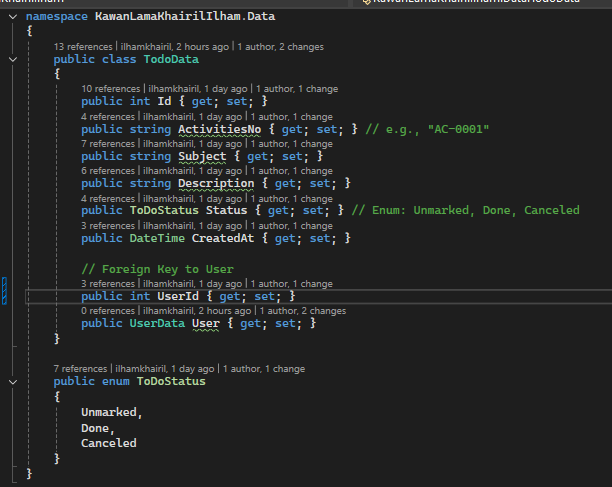
**2.1 User Model**

* Purpose: Represents a user in the system who can register, log in, and manage to-do items.



**2.2 To-Do Model**

* Purpose: Represents a to-do item with a subject, description, status, and a unique activity number.



**2.3 Enum ToDoStatus**

* Purpose: Represents the possible states of a to-do item.

**public enum ToDoStatus**

**{**

**Unmarked,**

**Done,**

**Canceled**

**}**

**3. Key Decisions and Trade-offs**

**3.1 Use of SQLite**

* Decision: SQLite was chosen for simplicity, given its easy integration and portability with ASP.NET Core applications. It does not require a separate database server, which simplifies the deployment process, especially within Docker.
* Trade-off: SQLite may not be suitable for very large-scale applications due to performance constraints with high concurrency. For larger applications, a more robust database like SQL Server or PostgreSQL could be used.

**3.2 Docker Containerization**

* Decision: Docker was used to ensure that the application runs consistently in any environment by isolating dependencies.
* Trade-off: Docker adds some overhead during setup and deployment, but it greatly simplifies managing dependencies and environments for development, testing, and production.

**3.3 Use of Blazor/Razor Pages for UI**

* Decision: Blazor (or Razor Pages) was chosen for its tight integration with ASP.NET Core, reducing the need for third-party libraries and making it easy to build interactive web applications.
* Trade-off: Blazor Server can introduce higher latency due to round-trip communication with the server for UI interactions. Alternatively, Blazor WebAssembly could be used for a more responsive user experience, but it requires more resources on the client side.

**3.4 Sequential Activities Number**

* Decision: The ActivitiesNo field is a unique sequential number in the format “AC-XXXX”. A simple sequence generator is implemented for this, ensuring uniqueness across to-do items.
* Trade-off: For scalability, this mechanism should be made more robust with database-level sequence management, especially in highly concurrent environments.

**3.5 Error Handling and Logging**

* Decision: The application uses a basic error handling mechanism with custom error messages shown to the user. A more advanced logging system could use ASP.NET Core's built-in logging framework to log errors to external services like ELK Stack, Datadog, or Serilog.
* Trade-off: While simple logging works in smaller applications, more complex systems should integrate structured logging and error tracking for better maintainability and scalability.

**4. Unit Testing**

**Framework: xUnit**

* Test Coverage: Unit tests focus on the core business logic of the application, particularly CRUD operations for to-do items and user authentication.
* Example Test: Testing the creation of a to-do item.

public void CreateToDo\_ShouldAddToDatabase()

{

// Arrange

var todo = new TodoData { Subject = "Test", Description = "Test Desc", ActivitiesNo = "AC-0001" };

var service = new ToDoService(context); // Assume context is pre-configured

// Act

service.CreateToDo(todo);

// Assert

Assert.NotNull(context.Todos.Find(todo.Id));

}

* Trade-off: Unit tests provide high coverage for isolated functions, but integration tests or functional tests may be needed to validate the full user experience.

**5. Performance Considerations**

**Performance Goal: Response time under 1 second**

* Database Query Optimization: Queries are kept simple and optimized using Entity Framework Core’s best practices.
* Caching: To meet the sub-second response time requirement with large data volumes (e.g., 1 million records), caching mechanisms (such as in-memory or distributed caching) could be introduced. However, for this test application, caching is not yet implemented.
* Concurrency: SQLite may have limitations with high concurrency (10+ users), so in a production scenario, switching to a more scalable database might be necessary.

**6. Future Improvements**

* Pagination: For the To-Do List, adding pagination to handle large datasets would improve performance and user experience.
* Advanced Authentication: Implementing external authentication providers (OAuth, OpenID Connect) for better security.
* Asynchronous Task Handling: Offloading heavy operations to background workers using a job queue (e.g., Hangfire or Azure Functions) could further optimize performance under heavy load.