Project Documentation: Authentication Module - Design Patterns & Architecture

Project Name: Qalb Connect
Module: User Authentication & Authorization
Date: June 8, 2025
Student: Abdul Rafay (Software Engineering Student, Second Year, Fourth Semester)

1. Module Overview and Architectural Approach

The Authentication Module of the Qalb Connect application is built using a standard **Layered Architecture** (also known as N-tier architecture), which is common in Spring Boot applications. This approach promotes separation of concerns, maintainability, and scalability.

Layers Implemented:

- **Presentation Layer (Controller):** Handles incoming HTTP requests, interacts with the Service Layer, and prepares data for the views (HTML templates).
- **Service Layer (Service):** Contains the core business logic, orchestrates operations, and acts as an intermediary between the Presentation and Data Access Layers.
- **Data Access Layer (Repository):** Manages data persistence and retrieval from the database, typically using Spring Data JPA.
- **Domain Layer (Model):** Represents the core business entities and their properties.
- **Configuration Layer (Config):** Sets up application-wide configurations, such as security settings.
- **Data Transfer Objects (DTOs):** Simple objects used for transferring data between layers, decoupling the domain model from external interfaces (like web forms).

2. Components Developed

Below is a detailed description of each component within the authentication module and its role.

2.1. User Entity (src/main/java/com/qalbconnect/qalbconnect/model/User.java)

- **Layer:** Domain Layer (Model)
- **Description:** Represents the core user entity in the application's database. It defines the structure and attributes of a user, such as id, username, password (hashed), email, and registrationDate.
- **Key Responsibilities:**
  - Map to a table in the database (via JPA annotations like @Entity, @Table).

    - Hold user-specific data.

2.2. UserRepository Interface (src/main/java/com/qalbconnect/qalbconnect/repository/UserRepository.java)

- **Layer:** Data Access Layer (Repository)
- **Description:** An interface that extends JpaRepository, providing powerful CRUD (Create, Read, Update, Delete) operations out-of-the-box for the User entity. It also includes custom query methods (findByUsername, existsByUsername, existsByEmail) for specific data access needs.
- **Key Responsibilities:**
    - Abstract database interactions.
    - Provide methods for persisting, retrieving, and querying User entities.

2.3. UserService Class (src/main/java/com/qalbconnect/qalbconnect/service/UserService.java)

- **Layer:** Service Layer
- **Description:** Implements the core business logic for user management, including user registration. Crucially, it also implements UserDetailsService, making it compatible with Spring Security's authentication mechanism.
- **Key Responsibilities:**
    - Handle user registration, including password hashing.
    - Act as Spring Security's UserDetailsService to load user details for authentication.
    - Validate unique usernames and emails before registration.

2.4. SecurityConfig Class (src/main/java/com/qalbconnect/qalbconnect/config/SecurityConfig.java)

- **Layer:** Configuration Layer
- **Description:** Configures the security rules for the entire web application using Spring Security. It defines how authentication providers work, sets up password encoding, and specifies URL-based authorization rules (e.g., which paths require authentication, which are public).
- **Key Responsibilities:**
    - Define the BCryptPasswordEncoder bean.
    - Configure DaoAuthenticationProvider to use UserService and the password encoder.
    - Set up the SecurityFilterChain to manage HTTP request security.
    - Define access rules for various URLs (e.g., /register, /login are public; /home is secured).

2.5. Data Transfer Objects (DTOs) (src/main/java/com/qalbconnect/qalbconnect/dto/)

- **Layer:** Data Transfer Layer
- **Description:** Lightweight, plain Java objects used to transfer data between the web (HTML forms) and the service layer. They are specifically designed for the data required by frontend forms, which may differ from the full User entity.
  - UserRegistrationDto.java: Captures username, password, and email for new user registration.
  - UserLoginDto.java: Captures username and password for user login.
- **Key Responsibilities:**
  - Decouple the domain model from the presentation layer.
  - Facilitate data validation using jakarta.validation annotations.
  - Ensure type-safe data transfer.

2.6. AuthController Class (src/main/java/com/qalbconnect/qalbconnect/controller/AuthController.java)

- **Layer:** Presentation Layer (Controller)
- **Description:** Handles incoming HTTP requests related to user authentication. It maps URLs like /, /register, /login, and /home to specific methods, renders appropriate HTML templates, and delegates business logic to the UserService.
- **Key Responsibilities:**
  - Serve the index, registration, login, and home HTML pages.
  - Process user registration form submissions, including input validation.
  - Orchestrate interactions between the web view and the UserService.

3. Design Patterns Implemented

Multiple object-oriented design patterns have been strategically applied to enhance the architecture, maintainability, and extensibility of the authentication module.

3.1. Singleton Pattern

- **Location:**
  - UserService (annotated with @Service)
  - UserRepository (extended from JpaRepository, managed by Spring)
  - SecurityConfig (annotated with @Configuration)
  - AuthController (annotated with @Controller)
  - Beans created via @Bean methods in SecurityConfig (e.g., BCryptPasswordEncoder, DaoAuthenticationProvider, SecurityFilterChain).
- **Explanation:** In Spring, classes annotated with @Service, @Repository, @Configuration, or @Controller are, by default, managed by the Spring IoC

container as **Singletons**. This means that Spring creates only **one instance** of these classes (and the beans they define) throughout the application's lifecycle. Whenever another component (like SecurityConfig or a controller) needs to use them, Spring injects this same single instance.

- **Benefit:** Ensures that there is a single, global point of control for critical components, preventing inconsistencies, optimizing resource usage, and reducing object creation overhead.

3.2. Adapter Pattern

- **Location:** UserService class.
- **Implementation:** UserService implements Spring Security's UserDetailsService interface, and its loadUserByUsername() method transforms our domain User object into Spring Security's UserDetails object.
- **Explanation:** The UserDetailsService is an interface required by Spring Security to load user details for authentication. Our internal User entity has a different structure than what UserDetailsService expects. The UserService acts as an **Adapter**, allowing our application's User model to be used seamlessly by the Spring Security framework without modifying the original User entity.
- **Benefit:** Enables collaboration between incompatible interfaces. It allows us to integrate our existing domain model with a third-party framework (Spring Security) without tight coupling, promoting modularity and reusability.

3.3. Null Object Pattern (Implicit via Optional)

- **Location:** UserService class (loadUserByUsername() method and registerUser() method's existence checks).
- **Implementation:** The use of java.util.Optional with orElseThrow() and explicit if (null) checks in registerUser().
- **Explanation:** While a dedicated NullUser object isn't explicitly created, the use of Optional in methods like findByUsername effectively handles the absence of a user. Instead of returning null (which often leads to NullPointerExceptions), Optional forces the developer to explicitly handle the "not found" scenario, either by providing an alternative (e.g., throwing UsernameNotFoundException) or by chaining operations. Similarly, in registerUser, explicit checks for null after attempting registration (due to username/email existence) manage the failure state clearly.
- **Benefit:** Improves code robustness by reducing the chances of NullPointerExceptions, makes the handling of "no value" scenarios explicit, and simplifies error management.

3.4. Factory Method Pattern

- **Location:** SecurityConfig class.
- **Implementation:** The @Bean annotated methods: passwordEncoder(), authenticationProvider(), and filterChain().
- **Explanation:** Each @Bean method serves as a **Factory Method**. Instead of requiring client code (e.g., other parts of SecurityConfig or other services) to know how to instantiate complex objects like BCryptPasswordEncoder or DaoAuthenticationProvider, these methods encapsulate the object creation logic. Spring calls these methods to "manufacture" and manage the lifecycle of these beans.
- **Benefit:** Decouples the object creation process from the code that uses these objects. This makes the system more flexible (e.g., swapping password encoders is simple), centralizes configuration, and adheres to the Single Responsibility Principle by giving the configuration class the responsibility of creating these essential beans.

3.5. Chain of Responsibility Pattern (Implicit)

- **Location:** SecurityConfig class, specifically the SecurityFilterChain bean.
- **Implementation:** The configuration of HttpSecurity with csrf(), authorizeHttpRequests(), formLogin(), and logout().
- **Explanation:** Spring Security internally operates on the principle of a **Chain of Responsibility**. When an HTTP request enters the application, it passes through a series of security filters (SecurityFilterChain). Each filter in this chain has a specific responsibility (e.g., handling authentication, authorization, CSRF protection). A filter processes the request and either handles it completely or passes it to the next filter in the chain. The filterChain method configures the order and behavior of these filters.
- **Benefit:** Promotes loose coupling by allowing multiple objects (filters) to handle a request without them explicitly knowing each other. It simplifies the extension or modification of processing steps and provides a flexible way to process incoming requests.

3.6. Facade Pattern

- **Location:** AuthController interacting with UserService.
- **Implementation:** The AuthController directly calls methods on UserService (e.g., userService.registerUser(...)).
- **Explanation:** The UserService acts as a **Facade** for the AuthController. The controller doesn't need to understand the intricate details of how user registration involves hashing passwords, interacting with the UserRepository, or handling unique constraint violations. Instead, it interacts with a simplified, unified interface

provided by UserService, which encapsulates all this complexity.
- **Benefit:** Reduces complexity and dependencies by providing a simpler, high-level interface to a subsystem (the business logic and data access for users). It decouples the client (controller) from the complex inner workings of the service layer.

3.7. Data Transfer Object (DTO) Pattern

- **Location:** dto package (UserRegistrationDto, UserLoginDto).
- **Implementation:** Dedicated POJO classes (UserRegistrationDto, UserLoginDto) used to transport data between the web forms (presentation layer) and the AuthController / UserService.
- **Explanation:** DTOs are used to carry data between processes to avoid exposing the internal domain model directly. For example, UserRegistrationDto contains only the fields required for registration (username, password, email) and includes validation annotations, which is distinct from the full User entity that might have id, registrationDate, etc.
- **Benefit:** Decouples the domain model from the presentation layer, provides a clear contract for data transfer, simplifies validation, and improves security by preventing over-exposure of internal entity details.

3.8. Command Pattern (Implicit)

- **Location:** Methods within AuthController (e.g., showRegistrationForm, registerUser, showLoginForm).
- **Implementation:** Each method in the controller represents a specific action or "command" that the application can perform in response to a user's request.
- **Explanation:** While not implemented with explicit Command objects, the methods in the controller implicitly follow the spirit of the **Command Pattern**. Each method encapsulates a request as an object (the method call itself), allowing for parameterization of clients with different requests (e.g., a form submission triggers a "register user" command). In more complex scenarios, you might define separate Command classes (e.g., RegisterUserCommand) and dedicated CommandHandlers to fully decouple the action from its execution.
- **Benefit:** Promotes separation of concerns and can lead to more flexible systems, allowing for features like undo/redo, logging, or queuing of actions in more advanced contexts. For a controller, it ensures each method has a clear, single responsibility.