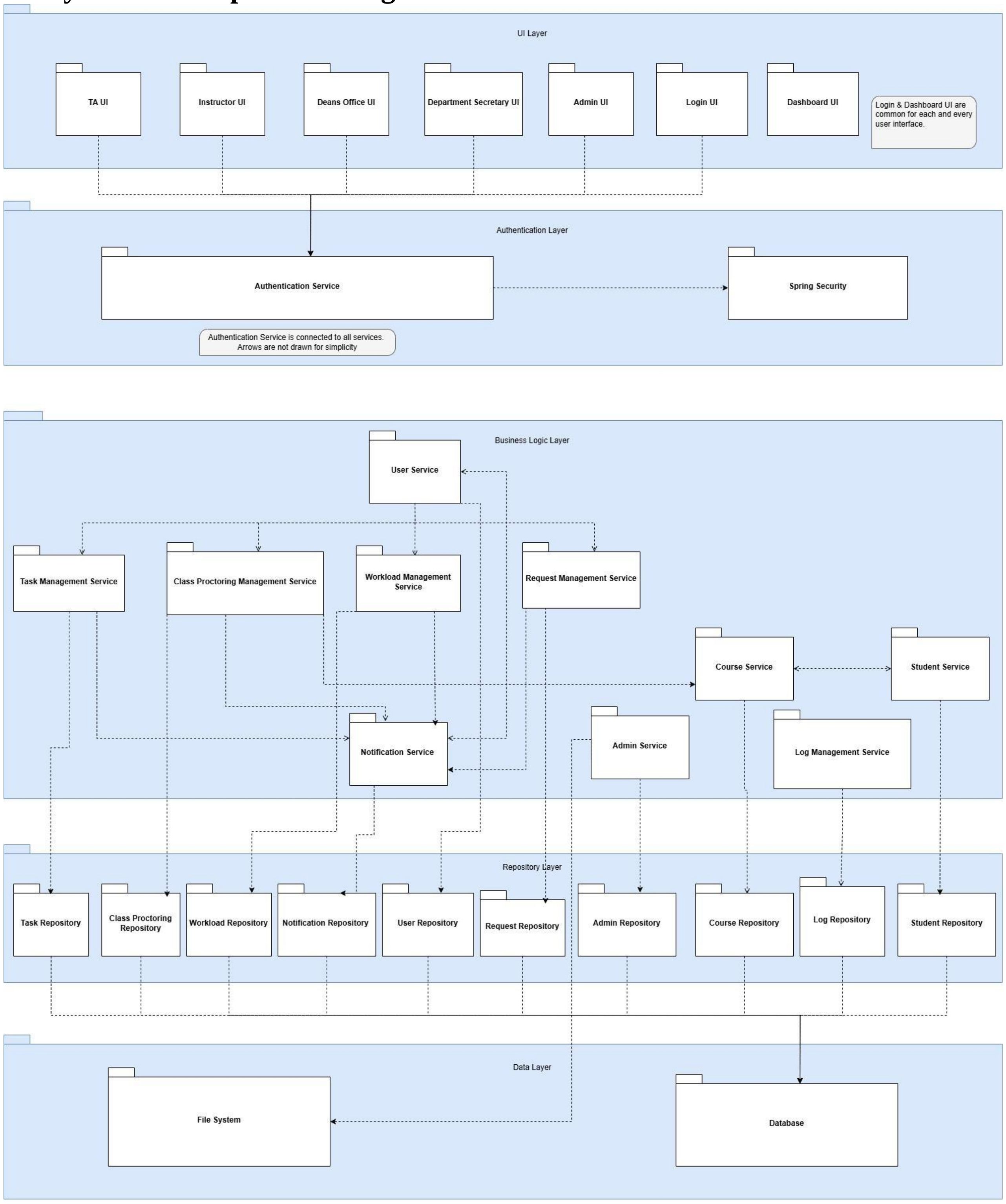
Subsystem Decomposition Diagram



1. Design Goals

1.1 Maintainability

Why It Matters?

The real-world complexity of university systems grows over time. That’s because maintainability matters. Evolving Requirements, from the project brief, we already anticipate possible changes (e.g., new types of TA tasks, changes in proctoring logic & swap rules, or integration with other departments). A maintainable codebase makes it easier to accommodate these changes without breaking the entire system. Long-Term Ownership requires outlive our development team. If the department or future students maintain the system, it needs to be easy to understand, extend, and debug. Another important aspect is bug fixes and enhancements: A maintainable system should be easily modified to fix bugs or introduce enhancements (like new reporting tools or more intelligent assignment algorithms) without extensive rewrites. Onboarding New Developers is also important. Clear structure, good documentation, and modular design help new developers onboard quickly, reducing the ramp-up time when new contributors join.

How to Achieve?

To ensure maintainability, we are adopting a layered architecture that separates concerns into controller, service, and repository layers, making each component easier to manage and update independently. Features will be organized into modular packages to improve clarity and reduce interdependencies. We’ll follow consistent naming conventions and Spring Boot best practices, using Lombok to reduce boilerplate code. Clear documentation and comments will be maintained to support future developers.

1.2 Rapid Development

Why It Matters?

We have only two months for implementation, and a functioning system by the deadline is more valuable than a exact-perfect one that never ships. Rapid development helps us deliver working software fast, iterate based on feedback, and meet our deadline. Since we are developing our app deadline-driven, without a deployable system, none of the features or architectural dreams matter. Rapid development enables us to reach the working-app fast. Also agile feedback loops, a working system — even a basic one — lets stakeholders (faculty, TAs, etc.) test features early, provide feedback, and refine the vision. By rapid development we can also avoid burnout. Spending too much time perfecting early features might leave us scrambling later. Rapid development emphasizes working smart and delivering just enough to move forward.

How to Achieve?

To achieve rapid development, we will prioritize building a minimal viable system first, focusing on core features that demonstrate end-to-end functionality. We'll follow agile principles such as iterative development, frequent feedback, and continuous integration (CI) to improve quickly. Using Spring Boot and React enables us to leverage rapid scaffolding, built-in configurations, and hot reloading to speed up development. We will also use tools like Postman and Swagger for quick API testing and validation. Clear task breakdowns, frequent commits, and team coordination will help us avoid bottlenecks and keep the project on track.

2 Trade-Offs

2.1 Maintainability Trade-Offs

Although maintainability is really important, it does not come free cost. Having this maintainability feature results in some losing other aspects.

Maintainability vs. Performance

- Writing highly maintainable code often means favoring clarity over efficiency. For instance, we might use abstract layers or ORM (like Hibernate) to make code easier to read and modify — but this can come at the cost of raw performance compared to hand-optimized SQL or streamlined procedural logic

Maintainability vs. Increased Productivity (Short-term)

Focusing on maintainability can reduce short-term productivity for us because:

- More time is spent on structure and documentation: Instead of just building features, we spend time creating abstractions, writing clean code, and documenting it for future developers.
- We may avoid "quick wins": Writing temporary code that "just works" is often faster, but maintainable design encourages longer-term thinking — leading to slower short-term delivery.
- Extra work on modularity: Maintainable code involves setting up proper architectures (like MVC), separating concerns — all of which require effort that doesn't immediately show user-facing results.

2.2 Rapid Development Trade-Offs

Same as maintainability, rapid-development comes with cost.

Rapid Development vs. Functionality

- When we prioritize speed, we often have to defer or cut lower-priority features, leading to reduced functionality in the initial version.

- This is a classic trade-off: We deliver faster, but the system might not fully cover all edge cases, exceptions, or nice-to-have features.

Rapid Development vs. Reliability

When we prioritize Rapid Development, we often sacrifice some level of reliability in the initial stages.

- Less time for thorough testing: We may skip writing comprehensive unit tests, integration tests, or corner case handling.
- More shortcuts and assumptions: We might hard-code certain values, ignore error-checking, or assume ideal user behavior just to “make it work.”
- Higher risk of bugs or breakdowns in real-world usage, especially under unexpected inputs or workflows.

3 Subsystem Decomposition

We implemented a five-layer subsystem architecture to improve the system's structure and functionality. These layers—UI, Business Logic, Authentication, Repository, and Data—each play a unique role in supporting the application’s overall performance and organization.

3.1 UI Layer

This layer is the front-facing part of the system where users interact. It communicates with the Business Logic Layer and is responsible for usability, responsiveness, and user experience.

TA UI

- Purpose: Interface for Teaching Assistants (TAs) to interact with scheduling, exam, and proctoring systems.
- Associated With:
    - Dashboard UI
    - Login UI
    - Authentication Layer

Instructor UI

- Purpose: Interface for instructors to manage their tasks.
- Associated With:
    - Dashboard UI
    - Login UI
    - Authentication Layer

Deans Office UI

- Purpose: Allows the Dean's Office to view staff, manage exams and TAs.

- Associated With:
    - Dashboard UI
    - Login UI
    - Authentication Layer

Department Secretary UI

- Purpose: Enables department secretaries to coordinate exam and TA-related tasks.
- Associated With:
    - Dashboard UI
    - Login UI
    - Authentication Layer

Admin UI

- Purpose: Admin-specific interface for system-wide controls.
- Associated With:
- Dashboard UI
- Login UI
- Authentication Layer

Login UI

- Purpose: Login screen for all users.
    - Being used by every other UI subsystem (except dashboard)

Dashboard UI

- Purpose: General interface summarizing essential tasks and actions for multiple roles.
    - Being used by every other UI subsystem (except login)

3.2 Authentication Layer

The authentication layer handles the user login process by verifying credentials such as email and password. In addition to authentication, it identifies the user's role or type (e.g., student, TA, instructor, admin), which is essential for determining access permissions and directing users to the appropriate parts of the system. By doing so, it performs as a bridge between UI and Service Layer in the system, so that user can only access permitted services and use the application. In order to build such layer, we benefit from Spring Security.

Authentication Service with Spring Security

- The authentication service in our system uses **Spring Security** with **JWT** to handle user login and access control. It verifies user credentials (email and password), assigns roles (e.g., student, TA, instructor, admin), and generates a JWT for secure, stateless sessions. while passwords are hashed with **BCrypt**. The JWT token is included in each request to authenticate and authorize access to protected resources. Based on roles, users are routed to the appropriate parts of the system, ensuring role-based access.

3.3 Business Logic Layer

The Business Logic Layer handles the core functionalities of the application by using control objects to process data received from the UI Layer. It can modify this data as needed while ensuring the consistency and integrity of the system and its information throughout the application. These control objects then interact with the Repository Layer to communicate with the database.

Class Proctoring Management Service

- The Class Proctoring Management Service consists of service objects responsible for managing the lifecycle of class proctoring sessions. This includes the creation, modification, and deletion of proctoring records. It also handles the assignment of Teaching Assistants (TAs) to specific proctoring sessions and oversees proctoring-related operations such as TA swap requests, ensuring that all scheduling and assignment rules are respected.

Log Management Service

- The Log Management Service automatically records system interactions between users and the application. It captures key events such as user logins, requests, and other important actions to ensure traceability, support debugging, and enhance system monitoring.

Notification Service

- The Notification Service is responsible for informing users about important system events. Every significant action or request in the application is followed by a corresponding notification, such as confirmation messages, approvals, rejections, or reminders. This ensures that users stay informed and engaged with real-time updates throughout the system.

User Service

- The User Service is responsible for managing user entities across the system. It handles operations such as user creation, retrieval, updating, and deletion. This service is also utilized by features like user profile pages, ensuring that user-related data is consistently and securely managed throughout the application. And also uses notification service in order to keep other users informed all the time.

Workload Management Service

- The Workload Management Service is responsible for tracking and managing the workloads of Teaching Assistants (TAs). It allows TAs to submit their workload information, which is then reviewed by instructors for approval or rejection. This service ensures that workload records are accurately maintained and that instructors can monitor and regulate TA responsibilities effectively.

Task Management Service

- The Task Management Service manages the definition and organization of task types within the system. It enables instructors to create, update, and delete various task categories—such as quizzes, exams, or assignments—that are used in proctoring and workload tracking. This service ensures consistency and flexibility in defining academic responsibilities.

Request Management Service

- The Request Management Service is responsible for handling all types of proctoring-related requests within the system. This includes the creation, submission, approval, rejection, and cancellation of requests made by instructors, TAs, or administrators. Moreover, it handles swap requests and the assignment of TAs from other departments through the Dean's Office. It ensures that each request follows the appropriate workflow based on user roles and that all decisions are logged and communicated to relevant parties through the Notification Service.

Course Service

- The Course Service is responsible for managing all course-related operations within the system. This includes creating, updating, retrieving, and deleting course records, as well as associating courses with departments, instructors, and proctoring sessions.

Student Service

- The Student Service is responsible for managing all student-related operations within the system. This includes creating, updating, retrieving, and deleting student records, as well as associating students with courses, departments, and relevant system activities such as proctoring or requests.

Admin Service

- This service allows administrators to perform high-level operations such as updating system records, managing users, modifying course or proctoring data, and overseeing application-wide settings. It ensures that only users with admin privileges can access and execute these sensitive operations.

3.4 Repository Layer

The Repository Layer serves as an intermediary between the Business Logic Layer and the underlying data storage. It is responsible for handling all data retrieval, persistence, and query operations, while providing a clean and consistent interface for the Business Logic Layer to interact with the database. This layer ensures separation of concerns by isolating data access logic from business rules. Notice that every repository connects with data layer.

- Task Repository: handles database operations related to TA tasks such as assignments, deadlines, and descriptions.
- Class Proctoring Repository: manages the storage and retrieval of data related to class proctoring schedules and assignments.
- Workload Repository: is responsible for tracking and querying TA workload records across various activities.
- Notification Repository: deals with storing and fetching system-generated messages or alerts for users.
- User Repository: manages CRUD operations for all system users, including TAs, faculty, and admins.
- Request Repository: handles access to request data such as leave requests, swaps, or special approvals.
- Admin Repository: provides access to administrative user data and any related configurations.
- Course Repository: manages course-related data, including course lists, codes, and instructors.
- Log Repository: is responsible for storing system logs and audit trails for monitoring and debugging.
- Student Repository: deals with student-related records such as enrollment and identification data.

3.5 Data Layer

The Data Layer is responsible for the physical storage and retrieval of data from databases or other external sources. It ensures data security, integrity, and persistence through two main components:

FileSystem

- Manages file-based storage operations (e.g., storing and retrieving images, documents), managed by the admin.
- Directly interacts with the Image Service (Business Logic Layer) for file handling.

Database

- Handles structured data storage and retrieval (e.g., user records, application data).

- Connects with the Database Repository (Business Logic Layer) for data operations.