Course Management System: Backend Architecture and Implementation Report

Subject: *Advanced Web Application Development*
Date: *January 5, 2026*

1. Introduction

The objective of this project was to design and implement a robust backend for a full-stack **Course Management System (CMS)**, often referred to as an LMS (Learning Management System). The system is designed to facilitate interaction between three primary roles: **Students**, **Instructors**, and **Administrators**. It allows for course creation, material distribution, assignment submission, grading, and real-time discussion.

The backend is built using **Java Spring Boot**, chosen for its dependency injection capabilities, robust security features (Spring Security), and seamless database integration via Spring Data JPA.

2. System Architecture

The application follows a standard **Layered Architecture** (Multi-tier Architecture) to ensure separation of concerns, maintainability, and scalability. The request flow is structured as follows:

Client (React) → Controller Layer → Service Layer → Repository Layer → Database (MySQL)

2.1 The Layers

- **Controller Layer**: Handles incoming HTTP requests, validates input using DTOs (Data Transfer Objects), and returns HTTP responses. It acts as the entry point.
- **Service Layer**: Encapsulates the business logic. This is where permission checks, data processing, and complex operations occur.
- **Repository Layer**: An abstraction over the database, leveraging Hibernate and JPA to perform CRUD operations without writing raw SQL.

- **Entity Layer**: Defines the data model and relationships mapping directly to database tables.

3. Detailed Implementation

3.1 Data Modeling (Entities)

The core of the application is the relational data model defined in the `entity` package. We utilized JPA annotations (`@Entity` , `@ManyToOne` , `@OneToMany`) to define relationships.

- **Utilisateur (User)**: The central entity representing all actors. We used an `Enum` strategy for `Role` (ETUDIANT, INSTRUCTEUR, ADMINISTRATEUR) and `Status` (ACTIF, EN_ATTENTE, BANNED), allowing for strict type safety and role-based logic.
- **Course**: The central hub for content. It has a `@ManyToOne` relationship with an Instructor, ensuring clear ownership.
- **Submission**: This entity links a Student and a Project. A key design decision here was the implementation of a composite unique constraint (`@UniqueConstraint`) on `etudiant_id` and `projet_id` . This prevents the logical error of a student submitting multiple times for the same assignment at the database level.
- **Polymorphic Comments**: To avoid creating separate tables for "Course Comments" and "Submission Feedback", the `Comment` entity considers both but enforces logic in the service layer where only one relationship is active at a time.

3.2 Security Configuration

Security is a critical component of any CMS. We implemented `SecurityConfig.java` to manage authentication and authorization.

- **Password Encryption**: We utilized the `BCryptPasswordEncoder` bean. Passwords are salted and hashed before storage, ensuring that even if the database is compromised, user credentials remain secure.
- **Authentication Flow**: The `AuthController` exposes a `/login` endpoint. Upon a login attempt, the system retrieves the user by email via `UtilisateurRepository` and uses the encoder's `.matches()` method to verify credentials.
- **Gatekeeping**: We implemented logic to block users with `EN_ATTENTE` or `BANNED` status, ensuring that administrative actions (like banning a user) take immediate effect.

3.3 Business Logic (Services)

The service layer enforces the rules of the university domain:

- **Role Validation**: In `CourseService` and `ProjectService` , we explicitly check `user.getRole()` before allowing creation actions. This prevents students from elevating privileges (e.g., creating a course) simply by calling an API endpoint.
- **Instructor Ownership**: The `ProjectService` verifies that an instructor creating an assignment is indeed the teacher of that specific course (`course.getInstructeur().getId().equals(id)`). This prevents horizontal privilege escalation where Instructor A modifies Instructor B's course.
- **File Management**: `MaterialService` handles file uploads by generating UUID-based filenames. This design choice prevents filename collisions and sanitizes inputs before saving to the local entity disk.

3.4 Exception Handling

To provide a clean API experience, we implemented a `GlobalExceptionHandler` using `@RestControllerAdvice` .

- **Validation Handling**: Instead of passing generic 500 errors, we intercept `MethodArgumentNotValidException` (triggered by `@Valid` annotations) to return detailed field-level errors (e.g., "Email is required").
- **Runtime Logic**: Custom exceptions thrown in services (e.g., "Student not found") are caught and transformed into readable JSON 400 Bad Request responses, protecting the stack trace from being exposed to the client.

4. Key Design Decisions & Trade-offs

1. **Interfaces for Repositories**: We used Spring Data JPA interfaces (e.g., `CourseRepository extends JpaRepository`).
   - *Decision*: This allows us to declare query methods like `findByEmail` without writing implementations. Spring generates the proxy beans at runtime.
   - *Benefit*: Drastically reduced boilerplate code and development time.
2. **Stateless API**: The backend is designed as a RESTful API.
   - *Decision*: We disabled CSRF (`csrf.disable()`) in `SecurityConfig` .
   - *Justification*: Since we are not using session cookies for a browser-heavy session flow but

rather a modern token-based approach (implied for future JWT integration), CSRF protection adds unnecessary complexity for this phase of development.

3. **Local File Storage**:
    - *Decision*: Files are stored in a local `uploads/` directory.
    - *Trade-off*: While simple for development, this is not stateless (harder to scale horizontally across multiple servers). A future improvement would be to migrate to cloud storage (e.g., AWS S3).

5. Conclusion

The implemented backend successfully meets all functional requirements. It provides a secure, role-aware environment for educational management. The modular architecture ensures that future features—such as attendance tracking or automated grading—can be integrated with minimal refactoring. The system is robust, handling edge cases and security threats effectively through its centralized exception handling and validation logic.