Java OOP Semester Project – Desktop Application

Student Information Management System

Group Members	**ID**
1. Filipos Tesera	UGR/7799/17
2. Hilina Getnet	UGR/4377/17
3. Ermias Leulseged	UGR/1423/17
4. Henok Assefa	UGR/9446/17
5. Etsubdink Alemu	UGR/1344/17
6. Gelila Getabalew	UGR/2507/17

Submission Date: Feb, 01, 2026

Project Title: Student Information Management System (SIMS)

Description:

The Student Information Management System is a Java desktop application designed to efficiently manage university student records, courses, enrollments, and grades. Built using JavaFX, the application provides role-based access with distinct interfaces for administrators and students. Administrators can perform full CRUD operations on all entities, while students can view their information and enrollments. The system demonstrates professional software development practices with proper file-based persistence, input validation, and a clean, user-friendly interface.

Section 2: Problem Statement

Traditional student management in educational institutions often involves manual record-keeping or disparate systems that lack integration. This leads to:

- Inefficient management of student records, courses, and grades
- Difficulty in tracking student enrollments and academic progress
- Lack of real-time access to academic information for students
- Manual errors in grade calculations and record maintenance
- No centralized system with proper access control

The SIMS addresses these challenges by providing a unified platform where:

1. Administrators can efficiently manage all academic records
2. Students can view their academic information in real-time
3. Course enrollments and grade assignments are systematically tracked
4. Data is persistently stored and easily retrievable
5. Role-based access ensures data security and appropriate permissions

Section 3: System Features

Core Features:

1. Role-Based Authentication System
    - Two user roles: Admin and Student
    - Secure login with username/password validation
    - Different dashboards and permissions per role
2. Student Management
    - Add, view, update, and delete student records
    - Store student ID, name, date of birth, email, and age calculation
    - Unique student ID format: UGR/XXXX/YY
3. Course Management
    - CRUD operations for courses
    - Course code, name, and credits management
    - Pre-loaded sample courses for demonstration
4. Enrollment System
    - Enroll students in courses for specific semesters
    - Track enrollment IDs, student-course relationships
    - Semester-based enrollment tracking
5. Grade Management
    - Assign and manage grades for enrollments
    - Marks calculation and grade assignment (A-F scale)
    - Grade persistence and retrieval

Technical Features:

6. File-Based Persistence
    - All data stored in text files (students.txt, courses.txt, etc.)
    - Automatic file creation on first run
    - Data persistence between application sessions

7. Input Validation & Error Handling
    - Comprehensive validation using `Validation` utility class
    - User-friendly error messages
    - Exception handling for file operations
8. User Interface
    - Clean, responsive JavaFX GUI
    - Intuitive navigation between modules
    - Consistent design across all views

Security Features:

9. Access Control
    - Admin: Full CRUD access to all modules
    - Student: Read-only access to personal information and grades
    - Session management with logout functionality

Section 4: Explanation of Basic OOP Concepts Used

4.1 Encapsulation

All class fields are declared `private` with controlled access via public getters/setters:

```
// BaseEntity.java - Complete encapsulation
private String id;
public String getId() { return id; }
public void setId(String id) { this.id = id; }

// StudentRepository.java - Internal state hidden
private List<Student> students = new ArrayList<>();
```

4.2 Inheritance

Logical inheritance hierarchy with `BaseEntity` as parent to `Student`:

```
// BaseEntity provides common ID functionality
public abstract class BaseEntity {
    private String id;
    // ... getters/setters
}

// Student inherits and extends functionality
public class Student extends BaseEntity {
    private String name;
    private LocalDate dateOfBirth;
    // ... student-specific methods
}
```

4.3 Polymorphism

Interface-based polymorphism:

```
// Declare as interface type
private StudentService studentService;

// Can assign any implementation
studentService = new StudentServiceImpl();
// OR studentService = new MockStudentServiceImpl();

// Runtime polymorphism through method calls
studentService.addStudent(...); // Calls appropriate implementation
```

Also we have implemented basic overriding polymorphism through `Basic Entity` to `Student` inheritance.

4.4 Abstraction

Interfaces define contracts without implementation:

```
public interface StudentService {
    void addStudent(String id, String name, LocalDate dateOfBirth, String
email);
    List<Student> getAllStudents();
    void deleteStudent(String id);

}
```

Abstract class provides common implementation:

```
public abstract class BaseEntity {

    // Common functionality for all entities that extend from it
```

Section 5: SOLID Principles Implementation

5.1 Single Responsibility Principle (SRP)

Each class has one clearly defined responsibility:

Repository Classes - Only handle data persistence:

```
// StudentRepository.java - ONLY data storage/retrieval
public class StudentRepository {
    private List<Student> students = new ArrayList<>();
    public void addStudent(Student student) { /* save to file */ }
    public List<Student> getAllStudents() { /* load from file */ }

}
```

Service Classes - Only handle business logic:

```
// StudentServiceImpl.java - ONLY business logic
public class StudentServiceImpl implements StudentService {
    public void addStudent(String id, String name, ...) {
        // Validation and business rules
        studentRepo.addStudent(new Student(id, name, ...));
    }
}
```

Controller Classes - Only handle UI interaction:

```
// StudentController.java - ONLY UI logic
public class StudentController {
    @FXML private void handleAdd() {
        // Get user input, validate, call service
        studentService.addStudent(id, name, ...);
    }
}
```

Utility Class - Only validation logic:

```
// Validation.java - ONLY validation
public class Validation {
    public static boolean notEmpty(String text) { ... }
    public static boolean isNumber(String text) { ... }
}
```

5.2 Open/Closed Principle (OCP)

Classes are open for extension but closed for modification:

Service Interfaces allow new implementations without changing existing code:

```
// StudentService.java - Interface (OPEN for extension)
public interface StudentService {
    void addStudent(String id, String name, LocalDate dateOfBirth, String
email);
}
```

```
// Current implementation
public class StudentServiceImpl implements StudentService { ... }

// Can add without modifying existing code
public class CachedStudentServiceImpl implements StudentService {
    // Adds caching functionality
}

public class LoggingStudentServiceImpl implements StudentService {
    // Adds logging functionality
}
```

5.3 Liskov Substitution Principle (LSP)

Subclasses can substitute base classes without breaking functionality:

Inheritance Hierarchy maintains substitutability:

```
// BaseEntity provides common functionality
public abstract class BaseEntity {
    private String id;
    public String getId() { return id; }
}

// Student extends and adds functionality WITHOUT breaking contracts
public class Student extends BaseEntity {
    private String name;
    private LocalDate dateOfBirth;
    // Adds new methods while maintaining BaseEntity contract
    public int getAge() { ... }
}

// Anywhere BaseEntity is expected, Student can be used
public void displayEntity(BaseEntity entity) {
    System.out.println("ID: " + entity.getId());
    // Works for Student objects too
}
```

5.4 Interface Segregation Principle (ISP)

Small, focused interfaces instead of large "god" interfaces:

Each service has its own specific interface:

```
// GOOD: Small, focused interfaces
public interface StudentService {
    void addStudent(...);
    List<Student> getAllStudents();
    void deleteStudent(String id);
}


public interface CourseService {
    void addCourse(...);
    List<Course> getAllCourses();
    void deleteCourse(String code);
}
```

5.5 Dependency Inversion Principle (DIP)

High-level modules depend on abstractions, not concrete implementations:

Controller depends on Service Interface:

```
// StudentController.java - Depends on ABSTRACTION
private StudentService studentService;  // Interface, not implementation

// Dependency injected via setter
public void setStudentService(StudentService studentService) {
    this.studentService = studentService;
}

// In DashboardController - Injection point
((StudentController) controller).setStudentService(new StudentServiceImpl());
```

Section 6: System Architecture

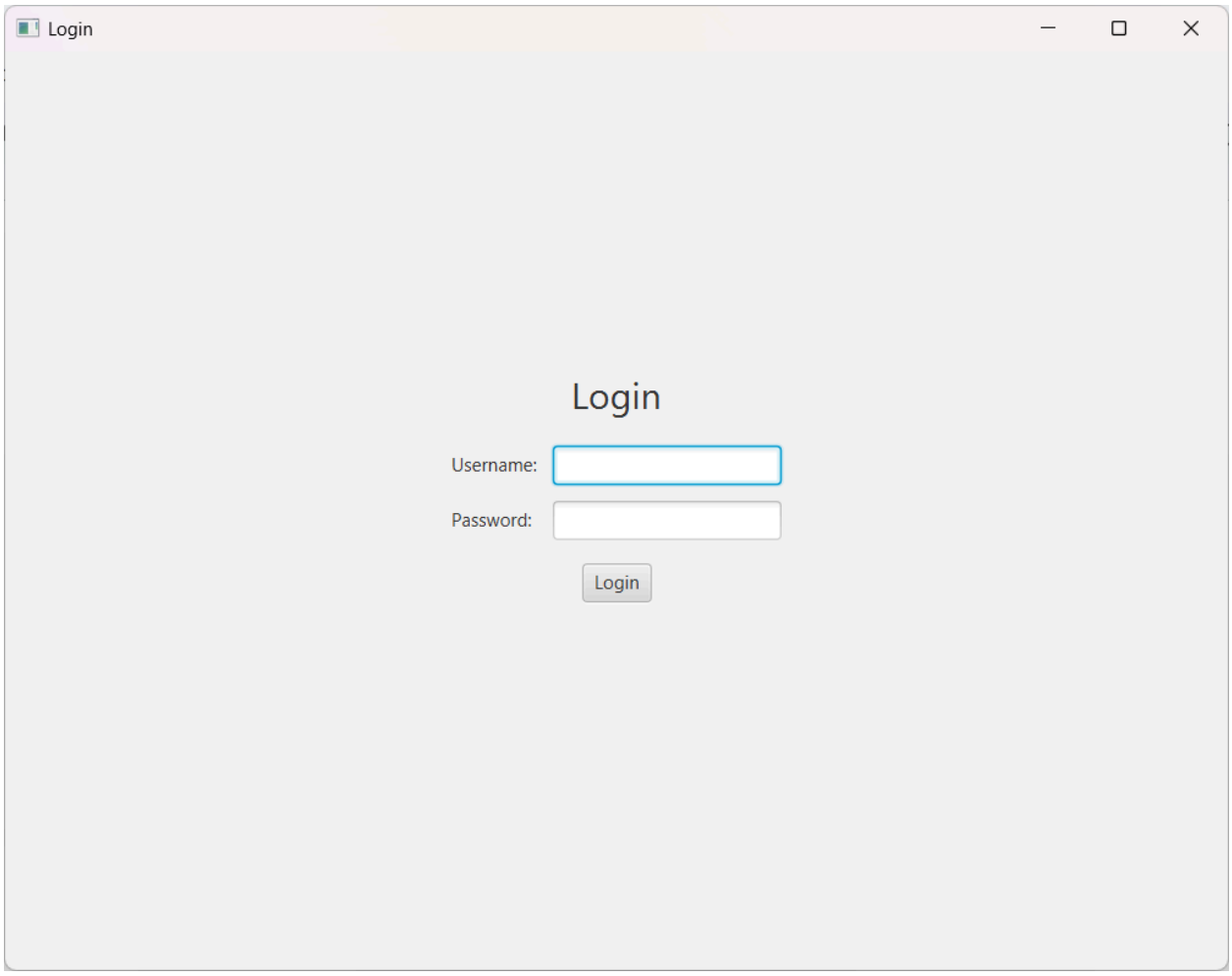
6.1 Layered Architecture

VIEW LAYER FXML Files + Controllers (JavaFX)
SERVICE LAYER Interfaces → Implementations (Business Logic)
REPOSITORY LAYER Data Access & Persistence (File Storage)
MODEL LAYER Entity Classes (Student, Course, etc.)

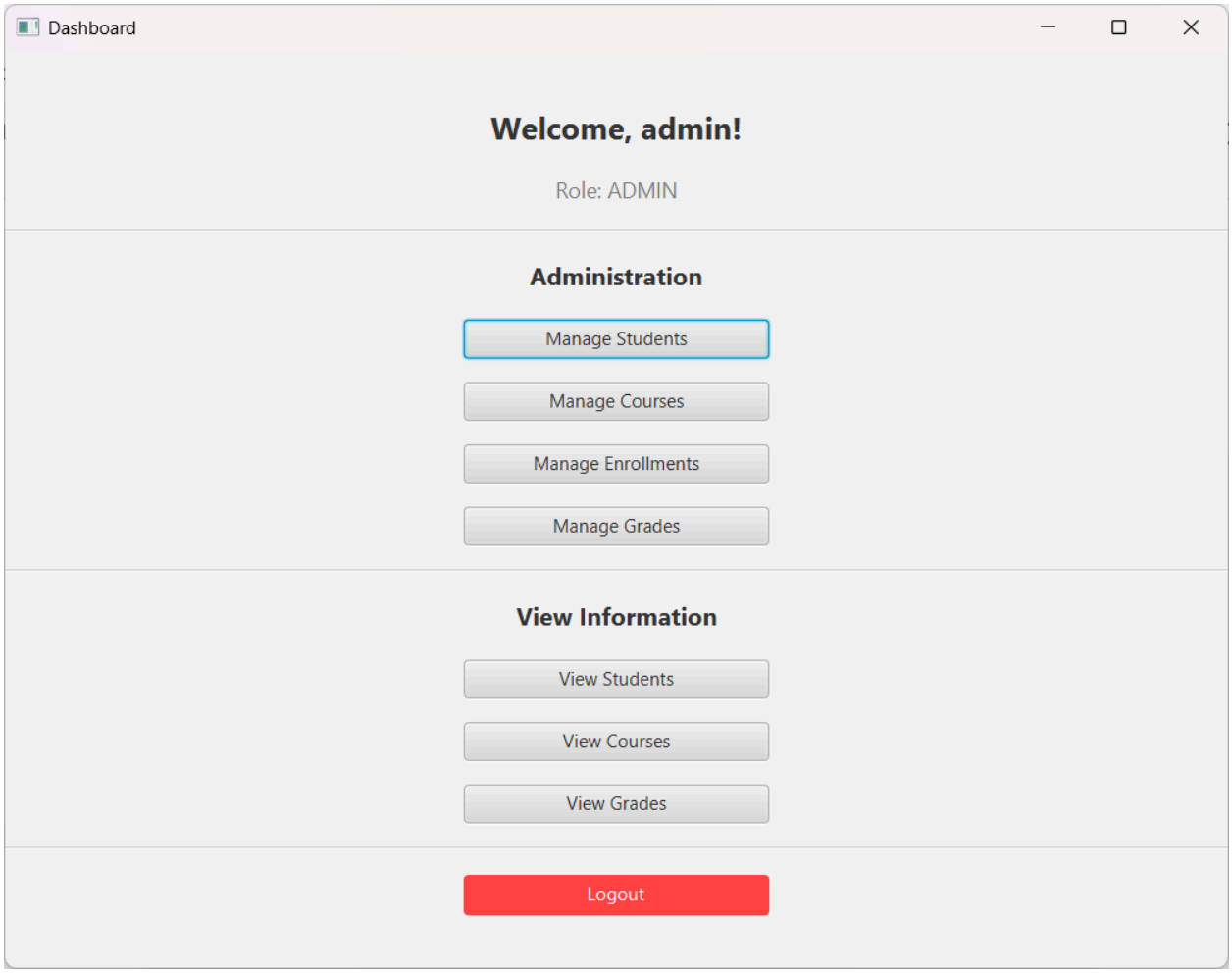
6.2 Data Flow

1. User Input → Controller (validates) → Service (business logic) → Repository (saves) → File
2. Data Retrieval ← Repository (loads) ← Service (processes) ← Controller (displays)

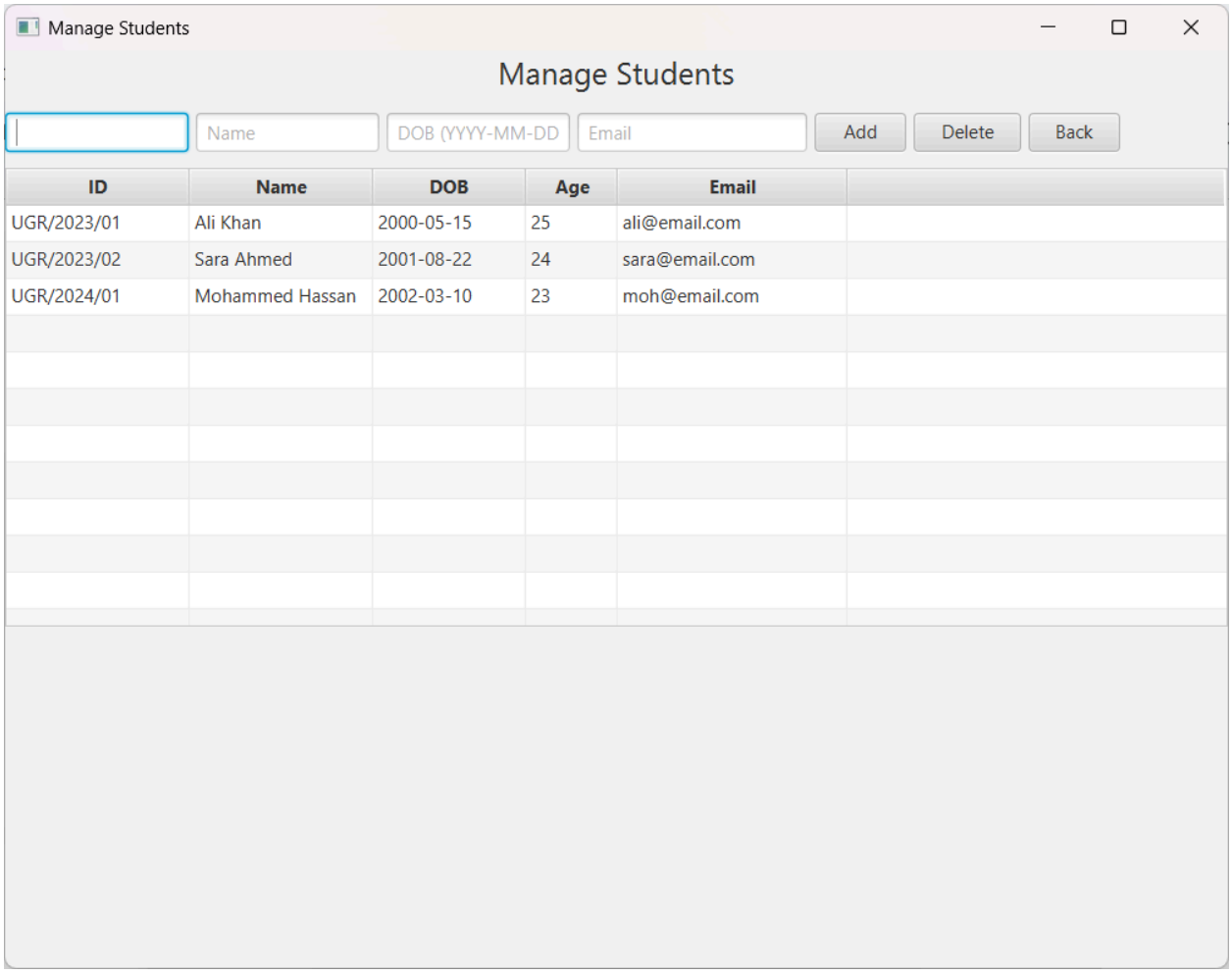
Section 7: Application Screenshots



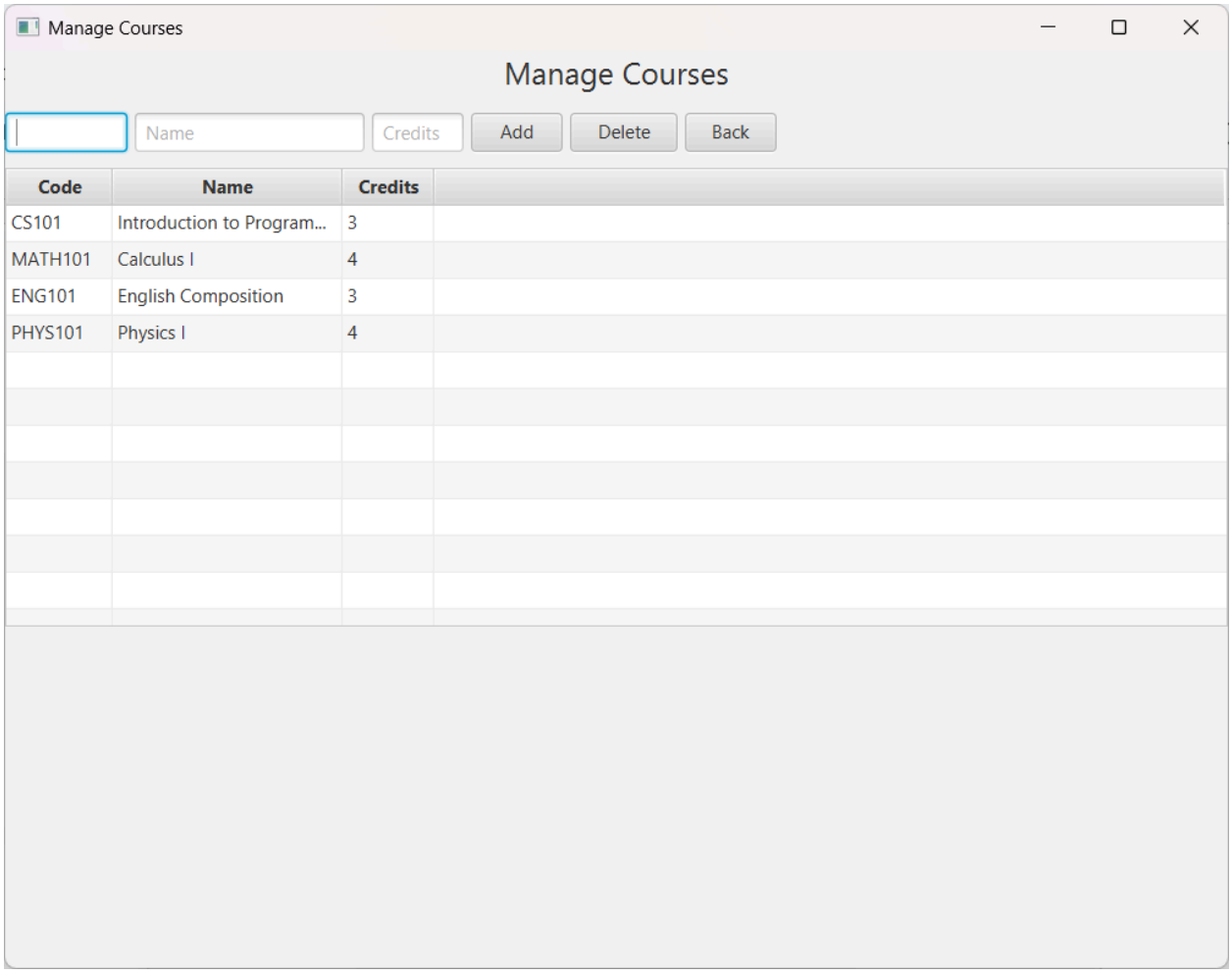
Login Interface - Secure authentication system with role-based access control. Users can login as Admin or Student with different permission levels.



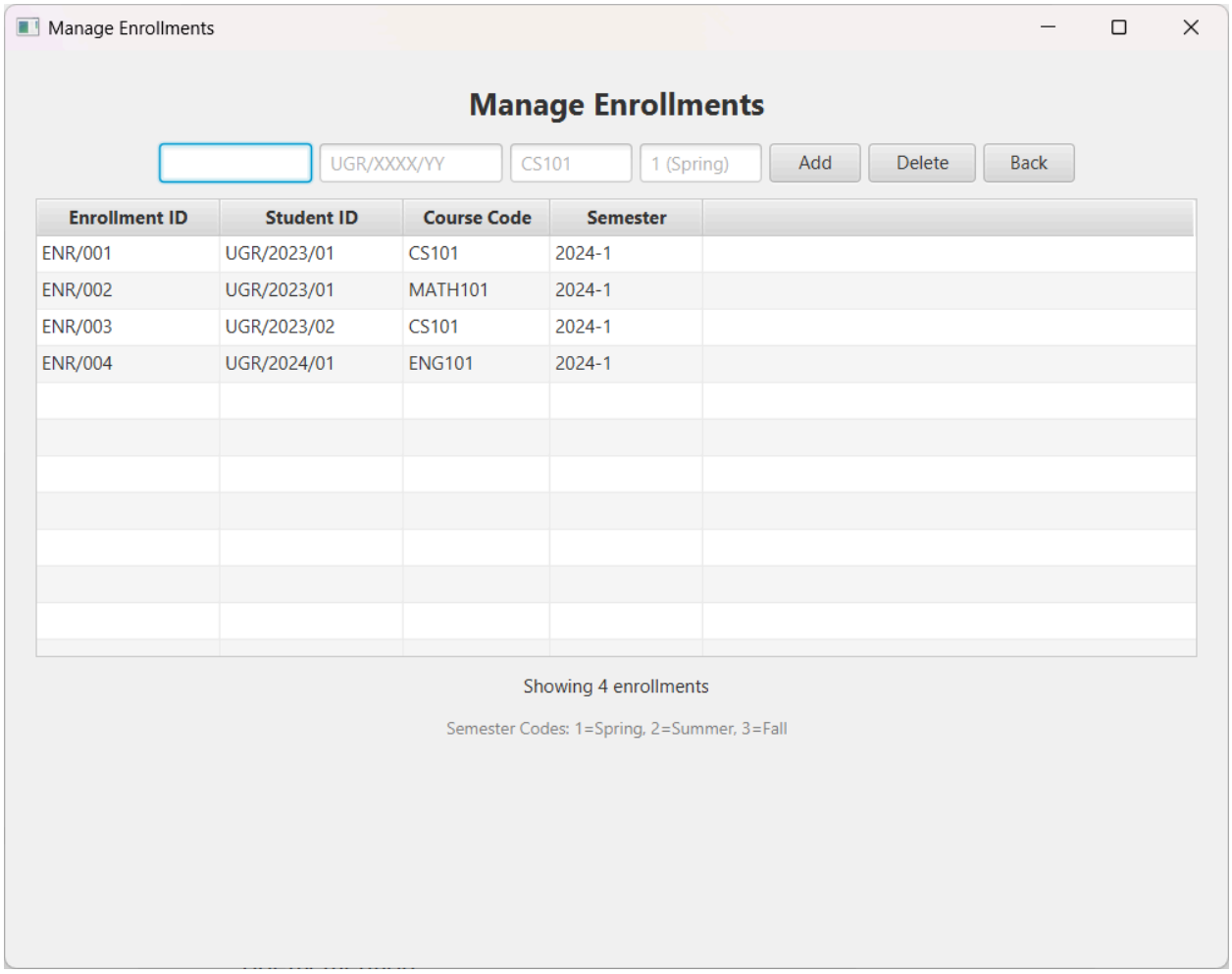
Administrator Dashboard - Role-based interface showing full administrative privileges including student management, course management, enrollment handling, and grade administration.



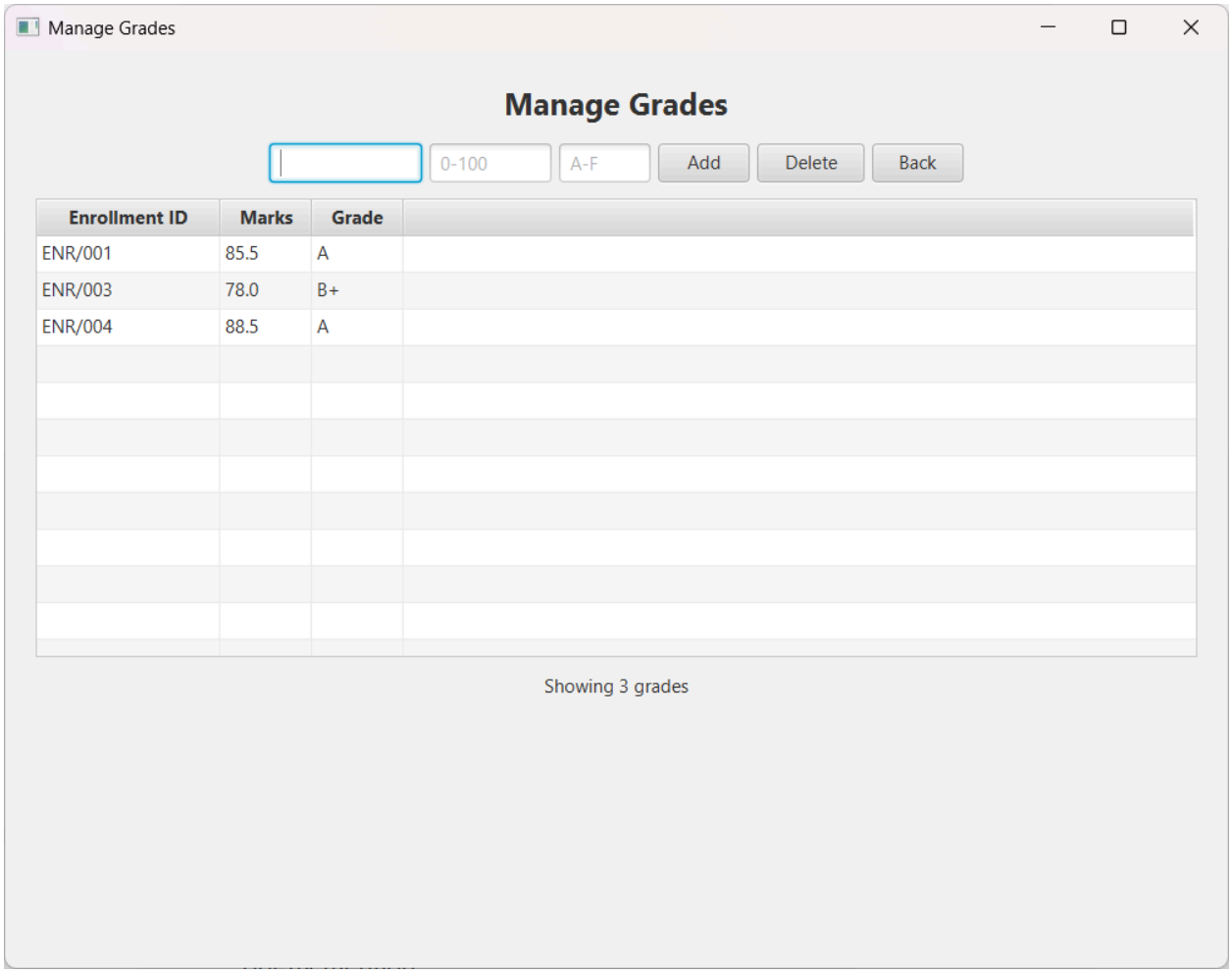
Student Management Module - Complete CRUD operations for student records showing ID, name, date of birth, age (calculated), and email with validation.



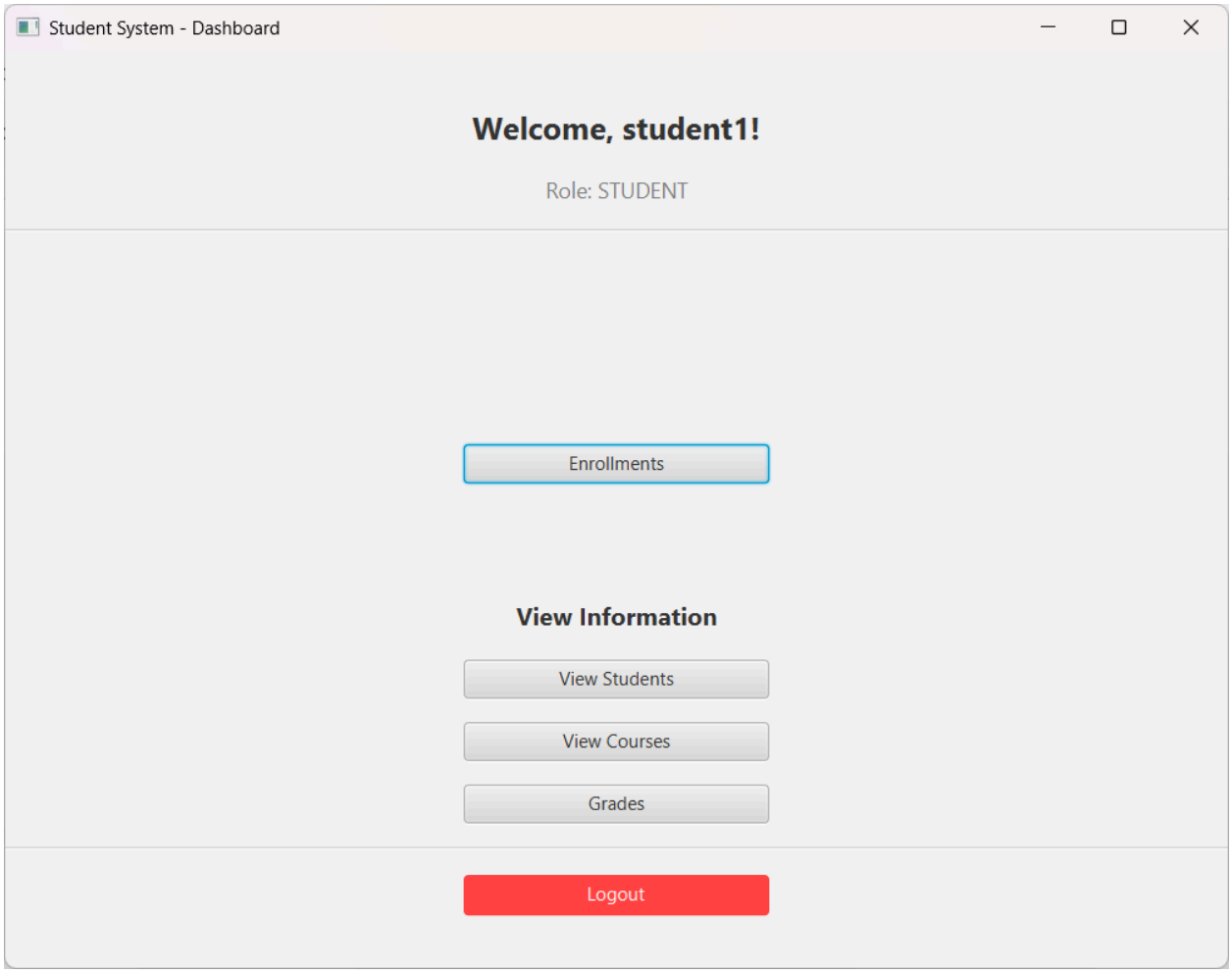
Course Management Module - Interface for managing academic courses with code, name, and credit hours. Demonstrates proper data persistence and validation.



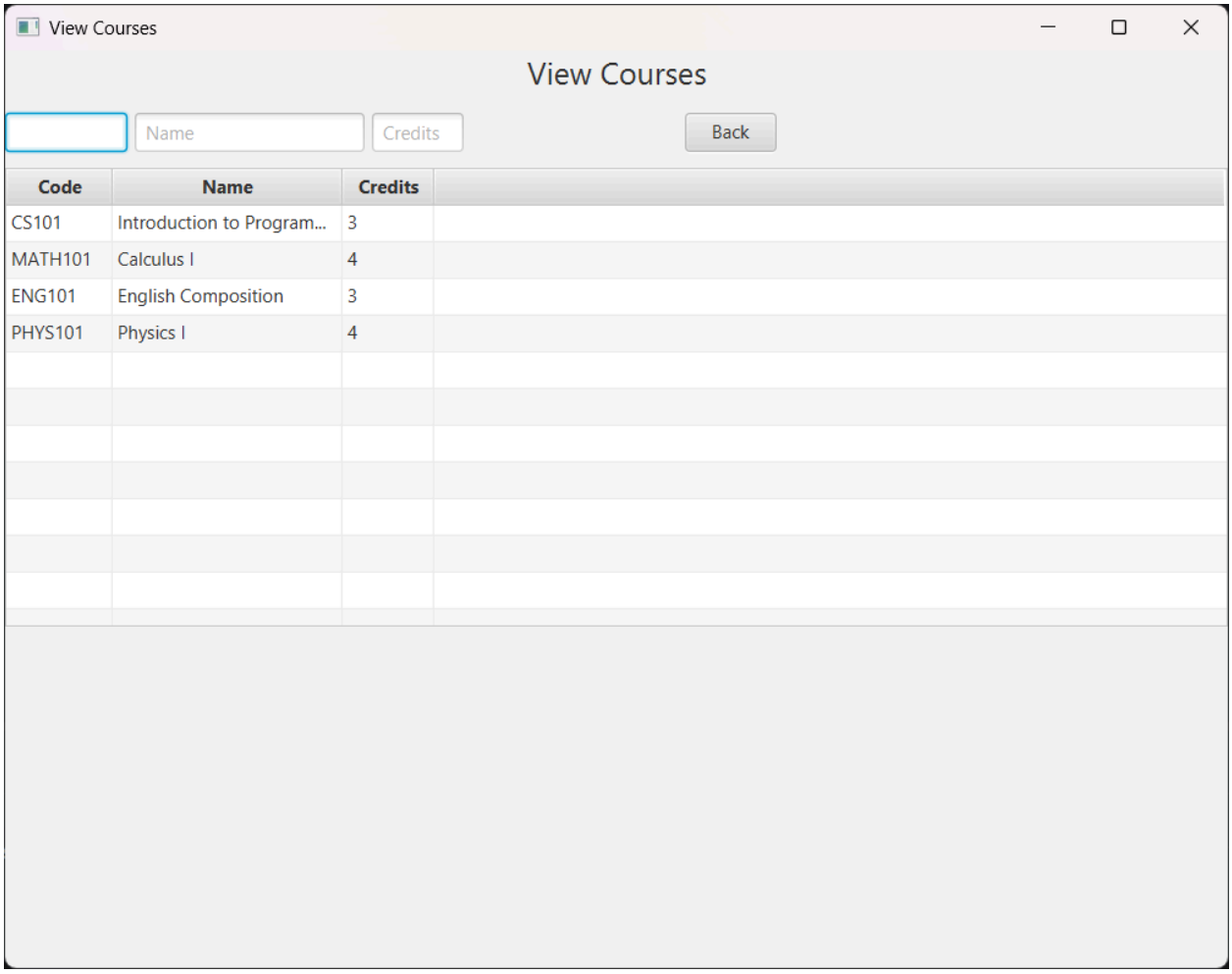
Enrollment Management - System for enrolling students in courses with semester tracking. Shows relational data management between students and courses.



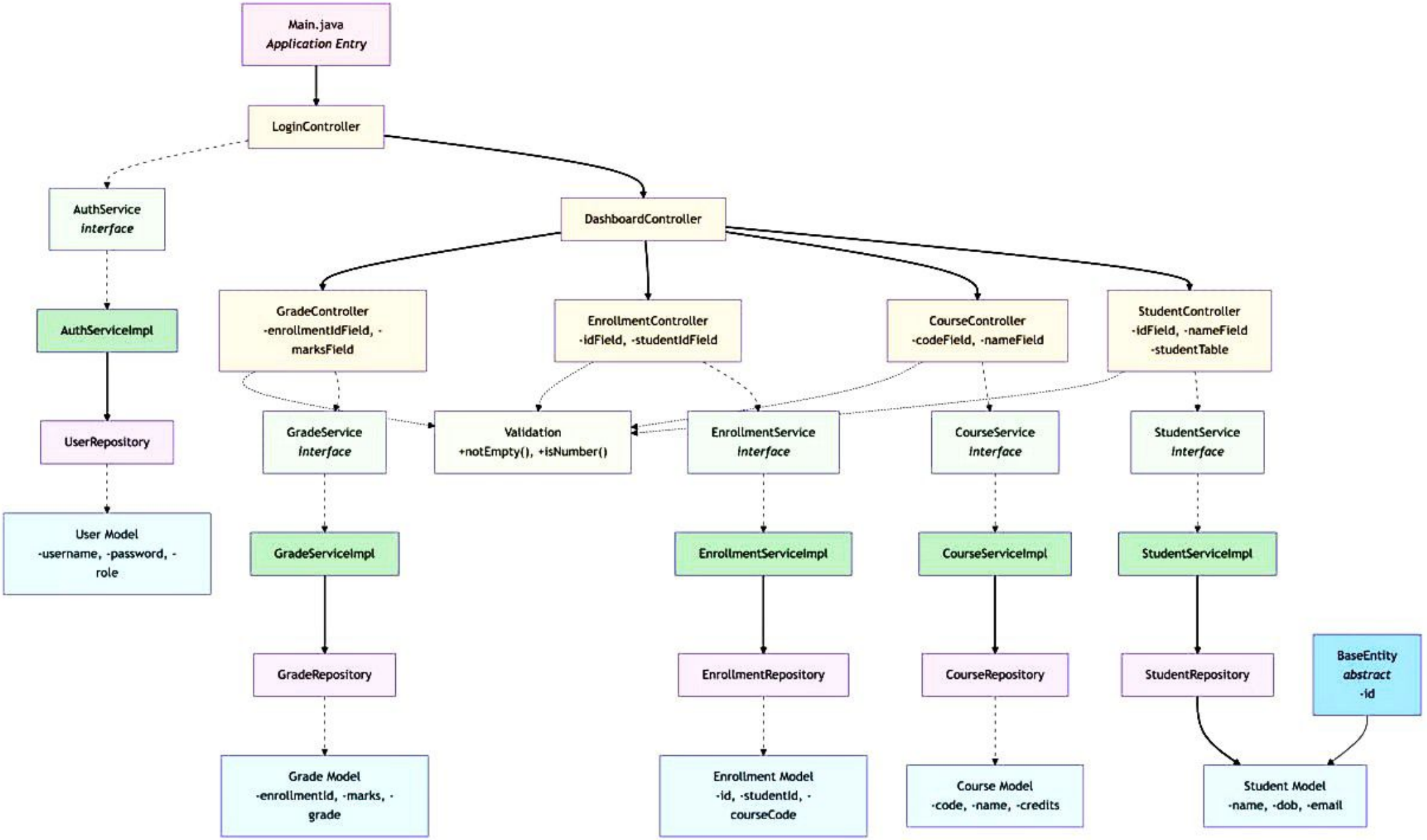
Grade Assignment Module - Allows administrators to assign and manage grades for student enrollments with marks validation (0-100) and grade calculation.



Student Dashboard - Limited-access interface showing role-based permissions. Students can only view information and manage their own enrollments, with all administrative functions disabled.



Course Catalog View - Read-only interface for students to browse available courses. Shows course details without modification capabilities, demonstrating proper role-based access control.


Main.java
Application Entry
LoginController
AuthService
interface
DashboardController
AuthServiceImpl
GradeController
-enrollmentIdField, -marksField
EnrollmentController
-idField, -studentIdField
CourseController
-codeField, -nameField
StudentController
-idField, -nameField
-studentTable
UserRepository
GradeService
interface
Validation
+notEmpty(), +isNumber()
EnrollmentService
interface
CourseService
interface
StudentService
interface
User Model
-username, -password, -role
GradeServiceImpl
EnrollmentServiceImpl
CourseServiceImpl
StudentServiceImpl
GradeRepository
EnrollmentRepository
CourseRepository
StudentRepository
BaseEntity
abstract
-id
Grade Model
-enrollmentId, -marks, -grade
Enrollment Model
-id, -studentId, -courseCode
Course Model
-code, -name, -credits
Student Model
-name, -dob, -email