**Online Examination Software Technical Manual**

[**Overview 2**](#_heading=h.7rk37vwtsrn5)

[Software Architecture 2](#_heading=h.tnchb9vlw8sj)

[User Roles and Permissions 2](#_heading=h.j3b89n8pj091)

[**UML Overviews 4**](#_heading=h.e0v3q2g9wzm0)

[**Repository Structure 11**](#_heading=h.zhynkbv9kela)

[**Class Details 13**](#_heading=h.inzhkskdwyf4)

[OnlineExamManagementSystemApplication.java 13](#_heading=h.2piii6h1r42r)

[Class Methods: 14](#_heading=h.molu5qajfhcg)

[CustomAuthenticationSuccessHandler.java 15](#_heading=h.iggh8pm2ovv7)

[OpenAIConfig.java 16](#_heading=h.9fm0zhxg2w4v)

[SecurityConfig.java 16](#_heading=h.k4jw1zprss0r)

[Class Methods: 16](#_heading=h.xrlddrahzhmr)

[Autowired Components: 17](#_heading=h.gj2q5nh1ax51)

[AdministratorController 18](#_heading=h.f2xft1v994vo)

[Autowired Components: 18](#_heading=h.bzmpkvhkb9ai)

[Handler Methods: 18](#_heading=h.9qrrh3y5izk)

[Annotations: 19](#_heading=h.ke1i97jxuakl)

[CustomBotController.java 20](#_heading=h.bciyy0onpjsb)

[Autowired Components: 20](#_heading=h.nh41s9pq6ev)

[Request Handler Methods: 20](#_heading=h.7sprbidkqtj)

[Private Methods: 21](#_heading=h.v0xlkuwkbved)

[ExamController 21](#_heading=h.hhimi76uf25c)

[Request Handler Methods: 22](#_heading=h.oeucassk3n86)

[Autowired Components: 22](#_heading=h.k58t071oh1ed)

[Request Handler Methods: 23](#_heading=h.e6u6kbuzurm8)

[Autowired Components: 23](#_heading=h.k4xzbgrmntsf)

[ExcelExportController 23](#_heading=h.9x0fn92avpna)

[Request Handler Method: 24](#_heading=h.3frsrvqetwa8)

[Autowired Components: 24](#_heading=h.76chwxnbwmjm)

[**InstructorController: 24**](#_heading=h.671ciylrzen0)

[**ScheduleManagerController 26**](#_heading=h.hdl6kdciangg)

[**StudentController 28**](#_heading=h.pg567otarea5)

[**UserRegistrationController 29**](#_heading=h.e7qwbqttx2dt)

[**Conclusion 1**](#_heading=h.5qlo9l5jrsix)

# Overview

This technical manual provides an overview of the Online Examination Software, designed for managing and conducting online exams. The software features different security permissions for various user types such as students, instructors, schedule managers, and administrators.

## Software Architecture

* **Source Code Structure:** The software is structured as SpringBoot MVC. It includes various directories and files, including configurations, controllers, domain models, services, and web interfaces.
* **Database Integration:** It integrates with a MySQL database for storing exam results and user data.

## User Roles and Permissions

Students (SV - StudentView)

* Can take exams.
* Access and complete online exams.
* View and track their exam results.

Instructors(IV - InstructorView)

* Responsible for creating, modifying, and publishing exams.
* Can design exams with various question types.
* Monitor and evaluate student performance.

Schedule Managers(SMV - ScheduleManagerView)

* Manage exam schedules.
* Coordinate with instructors and students for smooth exam administration.
* Oversee the allocation of resources for exams.

Administrators(AV - AdministratorView)

* Oversee the entire system and manage user permissions.
* Perform system-wide updates and maintenance.
* Ensure the integrity and security of the examination system.

Exam Management

* **Exam Creation:** Instructors can create exams with various question types.
* **Publishing Exams:** Exams are published and made available to students.
* **Evaluation:** Automated or manual evaluation of exam responses.

Student Interface

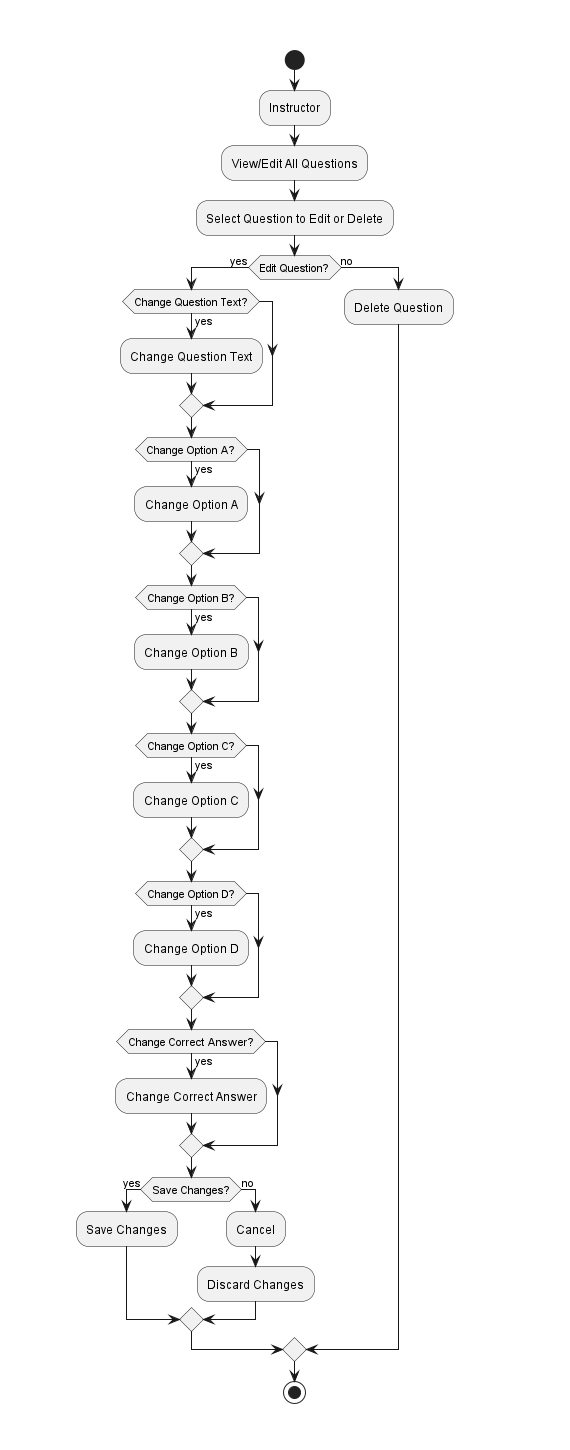
* **Taking Exams:** Students can access and take exams online.
* **Results:** Exam results are displayed and stored in the database.

Technical Specifications

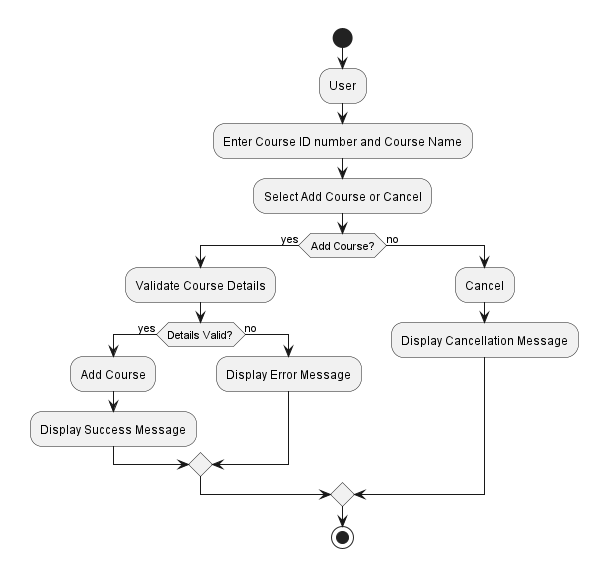
* **Programming Language:** Java.
* **Web Technologies:** HTML, CSS, JavaScript
* **Database:** MySQL.

# UML Overviews

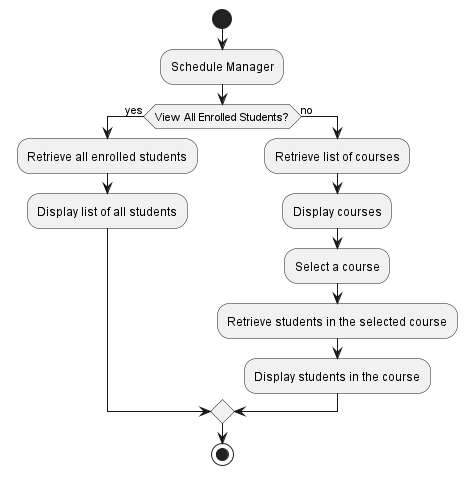
Activity Diagram for an Instructor Editing Questions



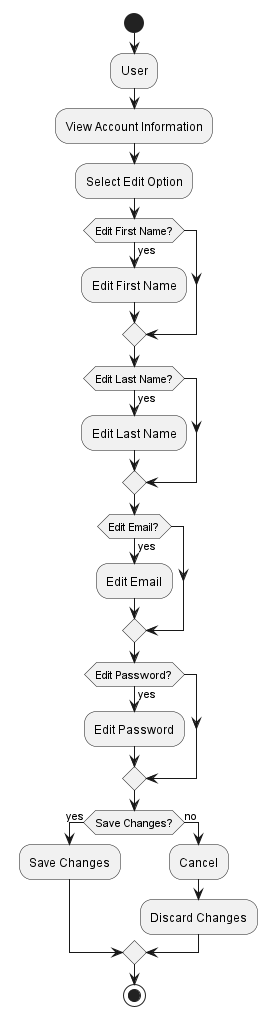
Activity Diagram for adding a course



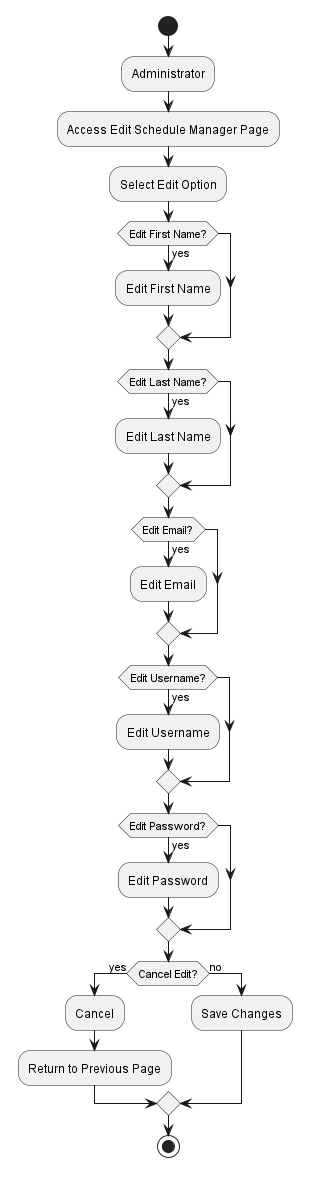
Activity Diagram for schedule manager viewing the Student lists



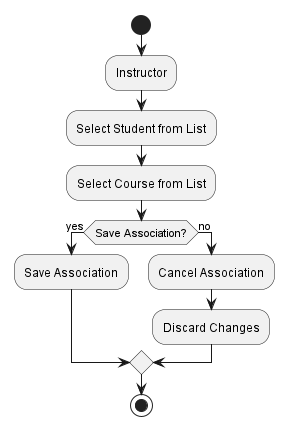
Activity Diagram for Account Management



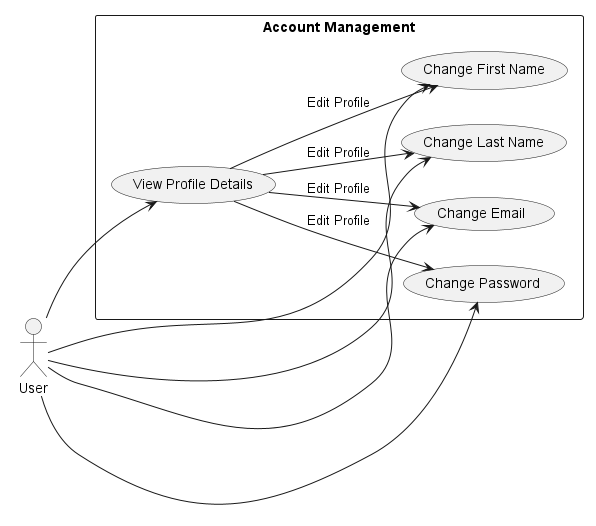
Activity Diagram for Administrator Editing a Schedule Manager



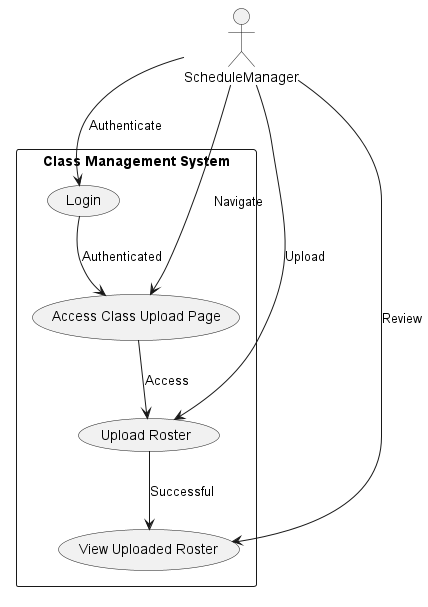
Activity Diagram for Instructor Associating a Student with a course



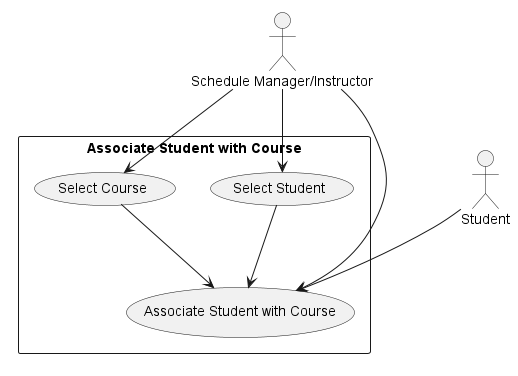
Use Case Diagram for Account Management



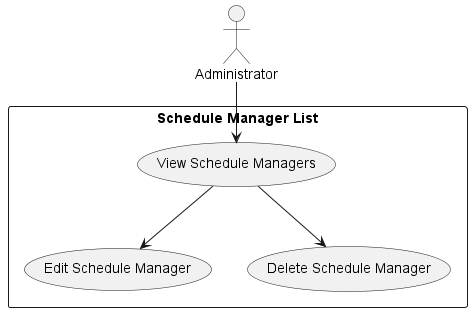
Use Case Diagram for Roster Upload



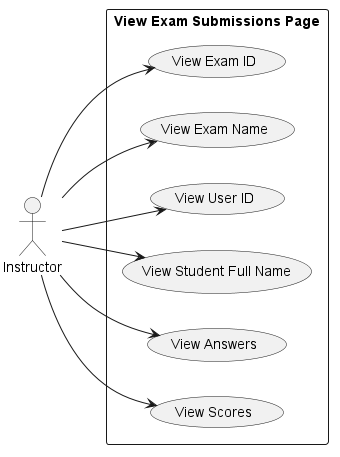
Use Case Diagram for Associating a Student with the Course



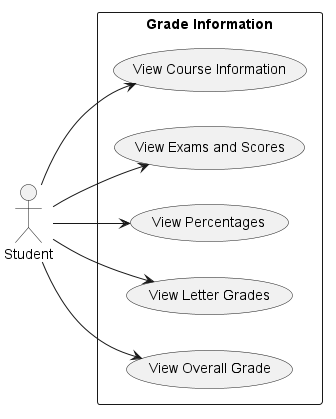
Use Case Diagram for Administrator Accessing the Schedule Manager List



Use Case Diagram for Instructor Viewing the Exam Submission Page



Use Case Diagram for Student Viewing their Grade Information



# Repository Structure

Below is an ASCII diagram representing the structure of the entire repository.

OnlineExamManagmentSystem

|

|-- Documents

| |-- PlantUML

| |-- chapters

| `-- rosters

|

|-- src

| |-- main

| | |-- java

| | | `-- edu

| | | `-- sru

| | | `-- thangiah

| | | |-- config

| | | |-- controller

| | | |-- domain

| | | |-- exception

| | | |-- model

| | | |-- repository

| | | |-- service

| | | `-- web

| | | `-- dto

| | |-- resources

| | | |-- chapters

| | | |-- references

| | | |-- static

| | | | |-- css

| | | | `-- icons

| | | `-- templates

Below is a detailed ASCII diagram representing the structure of the thangiah folder, which contains the SpringBoot MVC structure for the program.

thangiah  
│  
├── config  
│ ├── CustomAuthenticationSuccessHandler.java  
│ ├── OpenAIConfig.java  
│ └── SecurityConfig.java  
│  
├── controller  
│ ├── AdministratorController.java  
│ ├── ConfirmationController.java  
│ ├── CourseController.java  
│ ├── CustomBotController.java  
│ ├── ExamController.java  
│ ├── ExcelController.java  
│ ├── ExcelExportController.java  
│ ├── FileDownloadController.java  
│ ├── FileUploadController.java  
│ ├── InstructorController.java  
│ ├── MenuController.java  
│ ├── ScheduleManagerController.java  
│ ├── SearchController.java  
│ ├── StudentController.java  
│ └── UserRegistrationController.java  
│  
├── domain  
│ ├── Administrator.java  
│ ├── Chapter.java  
│ ├── Course.java  
│ ├── Exam.java  
│ ├── ExamDetails.java  
│ ├── ExamQuestion.java  
│ ├── ExamQuestionDisplay.java  
│ ├── ExamResult.java  
│ ├── ExamSubmission.java  
│ ├── ExamSubmissionEntity.java  
│ ├── Instructor.java  
│ ├── Question.java  
│ ├── ScheduleManager.java  
│ └── Student.java  
│  
├── exception  
│ └── ResourceNotFoundException.java  
│  
├── model  
│ ├── Roles.java  
│ └── User.java  
│  
├── repository  
│ ├── AdministratorRepository.java  
│ ├── CourseRepository.java  
│ ├── ExamQuestionRepository.java  
│ ├── ExamRepository.java  
│ ├── ExamSubmissionRepository.java  
│ ├── InstructorRepository.java  
│ ├── RoleRepository.java  
│ ├── ScheduleManagerRepository.java  
│ ├── StudentRepository.java  
│ └── UserRepository.java  
│  
├── service  
│ ├── CourseService.java  
│ ├── EmailService.java  
│ ├── ExamQuestionService.java  
│ ├── ExamQuestionServiceImpl.java  
│ ├── ExamService.java  
│ ├── ExcelExportService.java  
│ ├── ExcelGeneratorService.java  
│ ├── ExcelParserService.java  
│ ├── InstructorService.java  
│ ├── ScheduleManagerService.java  
│ ├── StudentService.java  
│ └── UserService.java  
│

│  
└── web  
 └── dto  
 ├── ChatGPTRequest.java  
 ├── ChatGptResponse.java  
 ├── CourseGradeDTO.java  
 ├── ExamGradeDTO.java  
 ├── Message.java  
 └── UserRegistrationDto.java

# Class Details

## OnlineExamManagementSystemApplication.java

The OnlineExamManagementSystemApplication is structured to perform initial setup tasks essential for the online exam management system. It uses CommandLineRunner beans to populate the database with essential roles and default users. These tasks are crucial for setting the stage for the application's functionality, ensuring that upon starting the system, the necessary foundational data is in place. The application employs Spring Boot's inversion of control to wire dependencies and manage the lifecycle of beans, simplifying the configuration and bootstrapping of the application.

## Class Methods:

**public static void main(String[] args)**

**Description**: This is the main method that Spring Boot uses to start the application. It launches the Spring application context with the given class.

**public CommandLineRunner setupRoles()**

**Description**: A Spring Boot CommandLineRunner bean that ensures the necessary roles (ADMINISTRATOR, STUDENT, INSTRUCTOR, SCHEDULE\_MANAGER) are present in the database upon initialization.

**public CommandLineRunner setupDefaultUser()**

**Description**: Another CommandLineRunner bean that creates default users for each role if they are not already present in the database.

**private void createRoleIfNotFound(String roleName)**

**Description**: A helper method used by setupRoles to check if a role is present in the database and create it if it is not found.

**private void createUserIfNotFound(String username, String password, String roleName, String email, String firstName, String lastName)**

**Description**: A helper method used by setupDefaultUser to create a user with the given details if they do not exist. Depending on the role, it also initializes the corresponding domain objects such as Administrator, Student, Instructor, and ScheduleManager.

**Autowired Components:**

**PasswordEncoder passwordEncoder**

Usage: Used for encoding passwords before storing them in the database.

**RoleRepository roleRepository**

Usage: Facilitates interaction with the roles stored in the database.

**UserRepository userRepository**

Usage: Allows querying and manipulation of user data in the database.

**AdministratorRepository administratorRepository**

Usage: Manages persistence for Administrator entities.

**StudentRepository studentRepository**

Usage: Handles data access for Student entities.

**InstructorRepository instructorRepository**

Usage: Used for accessing and persisting Instructor entities.

**ScheduleManagerRepository scheduleManagerRepository**

Usage: Facilitates operations related to ScheduleManager entities.

## CustomAuthenticationSuccessHandler.java

The CustomAuthenticationSuccessHandler is a crucial part of the Spring Security configuration that ensures users are redirected to the appropriate pages after logging in. It also handles the case where a user logs in with default credentials, prompting them to change their password to enhance security. This class is annotated with @Component, signifying that it is automatically detected and managed by Spring's container as a bean, allowing it to be injected and used where necessary in the application.

**public void onAuthenticationSuccess(HttpServletRequest request, HttpServletResponse response, Authentication authentication)**

**Description**: This method is overridden from the parent class to provide custom logic after a successful authentication. It retrieves the password entered by the user and the user's role to perform additional checks. If the entered password matches a default password for a particular role (e.g., "instructor" for the INSTRUCTOR role), it prompts a password change and redirects the user to an account management page specific to their role. Otherwise, it redirects the user to their respective homepage based on their role.

## OpenAIConfig.java

The OpenAIConfig class is a configuration class that defines how to create a RestTemplate bean pre-configured with an API key for authenticating requests to the OpenAI API. It leverages Spring's @Value annotation to inject configuration properties and @Bean to define the RestTemplate as a managed bean within the application context. The ASCII art comment block in the class appears to be a stylistic addition, possibly representing the OpenAI logo or related imagery, serving no functional purpose in the code.

**public RestTemplate template()**

**Description**: Defines a bean for RestTemplate, which is a synchronous client to perform HTTP requests, exposing a simple, template method API over underlying HTTP client libraries. This method configures the RestTemplate to include an "Authorization" header with a Bearer token, which is the OpenAI API key, for authentication purposes when making API calls.

## SecurityConfig.java

The SecurityConfig class is essential for configuring the security aspects of the application. It sets up how passwords are encoded, configures the security filter chain for processing HTTP requests, and defines the user details service and success handler for authentication. It leverages Spring Security's mechanisms to protect web applications from common vulnerabilities and provides a way to customize authentication and authorization to the needs of the application. The disabling of CSRF in the configuration is noteworthy, as it is a security measure that should typically be enabled, but it can be disabled for development or testing purposes. The class also indicates where the security configuration is loaded and how the application's security context is set up.

## Class Methods:

**public PasswordEncoder passwordEncoder()**

**Description**: This method declares a bean for password encoding, using the BCryptPasswordEncoder which is a password encoder that uses the BCrypt strong hashing function.

**protected DefaultSecurityFilterChain configure(HttpSecurity http)**

**Description**: This method configures the HttpSecurity for the application. It disables CSRF (Cross-Site Request Forgery) which is not recommended for production but may be necessary during development. It restricts access to certain URLs based on authority (e.g., "/student\_homepage/\*\*" can only be accessed by users with "STUDENT" authority). It also configures form-based login with a custom success handler and custom logout behavior.

**public void configureGlobal(AuthenticationManagerBuilder auth)**

**Description**: This method sets up the global AuthenticationManagerBuilder to use a custom UserDetailsService for user authentication and the passwordEncoder for password hashing.

## Autowired Components:

**UserDetailsService userDetailsService**

**Usage**: Autowired to use a custom UserDetailsService implementation for loading user-specific data in the security framework.

**CustomAuthenticationSuccessHandler successHandler**

Usage: Autowired to use a custom authentication success handler which is invoked when a user successfully logs in.

**PasswordEncoder passwordEncoder**

Usage: This field is intended to hold a reference to a PasswordEncoder bean, which should be autowired as well. However, it's missing the @Autowired annotation and the initialization, which could be a bug or an oversight.

# AdministratorController

## Autowired Components:

Various repository components for accessing data related to administrators, students, instructors, courses, etc.

**EmailService** for sending emails.

**PasswordEncoder** for encoding passwords.

## Handler Methods:

**createAdministrator**: Handles creating a new administrator.

**viewAdminHomepage**: Returns the admin homepage view.

**accountManager**: Displays the account management view for the administrator.

**editingCurrentUser**: Handles the request to edit the current administrator's details.

**saveCurrentUserEdits**: Processes the form submission for editing an administrator's details.

**examsPage**: Returns the view for the exams page.

**classesPage**: Returns the view for the classes page.

**Various quiz page handlers**: Return views for different quiz pages like math, history, and science.

**showCreateStudentForm**: Returns the view for creating a new student.

**showCreateInstructorForm**: Returns the view for creating a new instructor.

**showCreateCourseForm**: Returns the view for adding a new course.

**importStudents**: Returns the view for importing students.

**associateStudentWithCourseForm**: Returns the view for associating students with courses.

**associateInstructorWithCourse**: Handles the association of an instructor with a course.

**uploadSuccess**: Returns the view for a successful upload.

**uploadFail**: Returns the view for a failed upload.

**showStudentList**: Displays the list of students.

**showStudentsListAV**: Displays the list of students for the admin view.

**associateStudentWithCourse**: Associates a student with a course.

**showRegistrationForm**: Returns the registration form view.

**showRegistrationFormAV**: Returns the admin version of the registration form view.

**registerUserAV**: Handles the registration of a new user (admin view).

**registerUser**: Handles the registration of a new user.

**showUpdateFormAV**: Displays the form for updating student details (admin view).

**updateStudentAV**: Updates a student's details (admin view).

**deleteStudentAV**: Deletes a student (admin view).

**registerConfirm**: Returns the registration confirmation view.

**registerConfirmAV**: Returns the admin version of the registration confirmation view.

**showCourseSuccessForm**: Returns the course success page view.

**showInstructorSuccessForm**: Returns the instructor success page view.

**sendVerificationEmail**: Sends a verification email to the user.

**showUpdateFormInstructorAV**: Displays the form for updating instructor details (admin view).

**deleteInstructorAV**: Deletes an instructor (admin view).

**updateInstructorAV**: Updates an instructor's details (admin view).

**showInstructorsAV**: Displays the list of instructors (admin view).

**deleteScheduleManagerAV**: Deletes a schedule manager (admin view).

**showUpdateFormScheduleManagerAV**: Displays the form for updating schedule manager details (admin view).

**updateScheduleManagersAV**: Updates a schedule manager's details (admin view).

**showScheduleManagersAV**: Displays the list of schedule managers (admin view).

## Annotations:

@**Controller**: Indicates that the class is a Spring MVC controller.

@**PreAuthorize**: Specifies that a method should only be invoked if the user has a given authority.

@**GetMapping** and @**PostMapping**: Used to map HTTP GET and POST requests onto specific handler methods.

@**PathVariable**: Indicates that a method parameter should be bound to a URI template variable.

@**ModelAttribute**: Indicates that a method parameter should be bound to a model attribute.

@**RequestParam**: Indicates that a method parameter should be bound to a web request parameter.

@**Transactional**: Indicates that a method should execute within a transaction.

@**Validated**: Marks a parameter to be validated, for example, a form object.

## CustomBotController.java

The CustomBotController is a component of the web layer in the application, providing endpoints for user interactions with a chatbot, including chatting, selecting topics, generating quizzes, and downloading quiz content. It utilizes the OpenAI API for generating content and offers a user-friendly interface for engaging with the bot. The controller encapsulates the logic for the chatbot's functionality and interacts with services for generating and downloading quiz data in various formats. The ASCII art in the class comments resembles the OpenAI logo, serving as a stylistic representation associated with the chatbot's functionality.

## Autowired Components:

**excelGeneratorService**: Service for generating Excel documents from quiz data.

**template**: A RestTemplate object used for making HTTP requests, configured to interact with OpenAI's API.

**quizData**: A list of maps to hold quiz question data.

## Request Handler Methods:

**getProgress**: Returns the current progress of the quiz generation.

**chooseTopic**: Displays the view for topic selection.

**showChatbox**: Returns the chatbot interface view.

**chat**: Provides chat functionality by sending user prompts to OpenAI and returning the response.

**setTopic**: Sets the current topic for quiz generation and confirms it to the user.

**getQuiz**: Generates a quiz based on the current topic and displays it.

**downloadQuiz**: Facilitates the download of the generated quiz in Excel format.

**generateQuestion**: Generates a single multiple-choice question using OpenAI based on a given topic.

**downloadQuizTxt**: Facilitates the download of the generated quiz in plain text format.

## Private Methods:

**generateTxtContent**: Converts quiz data into a plain text format for downloading.

**Annotations:**

@**Controller**: Specifies that the class is a Spring MVC controller.

@**RequestMapping**("/bot"): Indicates that all request mappings in this controller are prefixed with /bot.

@**GetMapping** and @**PostMapping**: Map HTTP GET and POST requests onto specific handler methods.

@**ResponseBody**: Indicates that a handler method's return value should be bound to the web response body.

@**RequestParam**: Indicates that a method parameter should be bound to a web request parameter.

# ExamController

The ExamController facilitates various functionalities associated with exams in an educational application. It handles the full lifecycle of an exam, including creation, modification, deletion, and taking an exam, as well as reviewing submissions and results. The controller interacts with services and repositories to perform business logic and data persistence operations, providing a link between the front-end views and the back-end data models. It showcases typical Spring MVC patterns and is designed to work within a larger Spring Boot application with user authentication and session management capabilities.

## Request Handler Methods:

**filterExamQuestionsByChapter**: Filters exam questions by chapter.

**editExam**: Edits an existing exam.

**updateExamQuestions**: Updates the questions associated with an exam.

**deleteExam**: Deletes an exam.

**viewExamSubmissions**: Displays exam submissions.

**viewExamDetails**: Shows the details of an exam.

**generateExam**: Manually or automatically generates an exam.

**confirmExam**: Confirms the creation of an exam.

**addMoreChapters**: Adds more chapters to an existing exam.

**generateExam**: Generates an exam with selected chapters.

**showExamLink**: Displays the link for an exam.

**selectChapter**: Selects a chapter for exam generation.

**handleChapterSelection**: Handles the chapter selection for exam generation.

**generateExam**: Generates an exam based on selected chapters.

**takeExam**: Handles the exam-taking process for a student.

**generateExamLink**: Generates a link to take an exam.

**submitExam**: Processes the exam submission and calculates the score.

**generateExam**: Generates an exam for a given chapter.

**pickExam**: Picks an exam for a specific chapter.

**submitEditedExam**: Submits edited exam questions and generates an Excel file.

**submitAnswers**: Receives submitted exam answers and evaluates them.

**showResults**: Displays the exam results to the user.

## Autowired Components:

**examService**: Service for exam-related operations.

**examQuestionService**: Service for managing exam questions.

**userRepository**: Repository for accessing user data.

**examSubmissionRepository**: Repository for accessing exam submission data.

**examRepository**: Repository for accessing exam data.

ExcelController  
The ExcelController serves as a data import tool within the application, providing a way for users to upload Excel files containing course and user data, which are then processed and integrated into the system's data models. It offers a streamlined approach for mass data entry, reducing the need for manual input and thereby improving efficiency. The controller demonstrates a practical application of file handling, data parsing, and transactional database operations in a Spring MVC context.

## Request Handler Methods:

**showClassImportPage**: Displays the page to import class data from an Excel file.

**upload**: Processes the uploaded Excel file and imports the data into the system.

## Autowired Components:

**roleRepository**: Repository for accessing roles data.

**studentRepository**: Repository for accessing student data.

**instructorRepository**: Repository for accessing instructor data.

**courseRepository**: Repository for accessing course data.

**userRepository**: Repository for accessing user data.

**passwordEncoder**: Service for encoding passwords.

## ExcelExportController

The ExcelExportController provides a RESTful endpoint for exporting student data to an Excel file. It simplifies the process of data export by automatically saving the file to the user's downloads directory and informing the user of the outcome through a simple message. This controller is part of the application's backend layer, facilitating the interaction between the front-end and the services that handle data processing and file creation.

## Request Handler Method:

**exportStudentData**: Exports the data of all students to an Excel file and saves it in the user's download directory.

## Autowired Components:

**excelExportService**: Service for exporting data to Excel files.

**studentRepository**: Repository for accessing student data.

# InstructorController:

The InstructorController is a comprehensive controller that facilitates the interaction between instructors and various aspects of the educational platform. It provides a wide range of functionalities such as managing exams, editing personal and student information, handling Excel file uploads for bulk data processing, and overseeing the courses and student lists pertinent to the instructor. The controller ensures secure and efficient operations by integrating with multiple services and repositories, offering a robust toolset for instructors to effectively manage their academic responsibilities.  
  
Methods:  
showInstructorHomepage: Returns the homepage view for an instructor.

**listExamQuestions**: Displays a list of exam questions to the instructor.

**showExamLandingPage**: Prepares the view for the exam landing page, including the courses handled by the instructor.

**showExamUploadLandingPage**: Returns the view for uploading exams.

**showAllExams**: Lists all exams created by the instructor.

**captureExamLandingPageData**: Captures data from the exam landing page form to create or update exams.

**showAutoExamPage**: Returns the view for automatically generating exams.

**showExamQuestionForm**: Returns the view to create a new exam question.

**editExamQuestion**: Returns the view to edit an existing exam question.

**updateExamQuestion**: Processes the updated exam question data and persists it to the database.

**removeSelectedQuestion**: Removes a question from the exam being created.

**deleteExamQuestion**: Deletes an exam question from the database.

**selectExamQuestions**: Provides a view to select questions for an exam.

**generateExam**: Handles the creation of an exam with selected questions.

**showInstructorList**: Lists all instructors.

**createInstructor**: Creates a new instructor record.

**findInstructor**: Retrieves details of a specific instructor.

**showStudentList**: Displays a list of all students.

**updateStudentIV**: Updates student details, including password changes.

**deleteStudent**: Removes a student record.

**showStudentsListIV**: Lists students related to the instructor's courses.

**enableDisableStudent**: Toggles the enabled status of a student.

**showCreateStudentFormIV**: Returns the view to create a new student.

**createIV**: Processes the creation of a new student and an associated user.

**associateStudentWithCourseFormIV**: Shows the form to associate students with a course.

**handleAssociateStudentWithCourse**: Associates a student with a selected course.

**exportStudentDataIV**: Exports student data to an Excel file.

**uploadExcelIV**: Processes the uploaded Excel file for class data.

**Autowired:**

**UserRepository**: For user data access.

**RoleRepository**: To handle roles associated with users.

**ScheduleManagerRepository**: For accessing scheduling data.

**ExcelExportService**: To export data into Excel format.

**ExamService**: For exam-related operations.

**InstructorRepository**: To interact with instructor data.

**CourseRepository**: For course data access.

**StudentRepository**: To manage student records.

**ExamQuestionService**: For operations related to exam questions.

**ExamRepository**: To handle exam data persistence.

# ScheduleManagerController

The ScheduleManagerController is a specialized controller that serves the administrative role of a schedule manager within an educational platform. It provides an interface for managing courses, instructors, and student rosters, alongside account management for schedule managers themselves. It also handles importing and exporting data through Excel, allowing for bulk operations that can streamline the administrative workload. This controller is essential for maintaining the organization and scheduling of classes, ensuring that instructors and students are appropriately associated with their courses.

**Methods**  
**showScheduleManagerHomepage**: Displays the homepage for the schedule manager.

**passwordReset**: Presents the password reset and account management page for the schedule manager.

**editingCurrentUser**: Displays a form for the schedule manager to update their own account details.

**saveCurrentUserEdits**: Processes and saves the updated account details of the schedule manager.

**showCreateCourseForm**: Displays a form for adding a new course.

**showInstructorList**: Lists all instructors in the system.

**showCreateInstructorForm**: Displays a form to create a new instructor account.

**associateStudentWithCourseFormSMV**: Shows a form to associate students with courses.

**handleAssociateStudentWithCourse**: Associates a selected student with a selected course.

**associateInstructorWithCourseFormSMV**: Shows a form to associate instructors with courses.

**handleAssociateInstructorWithCourse**: Associates a selected instructor with a selected course.

**showCreateInstructorFormSMV**: Displays a form for schedule managers to create new instructors.

**showStudentsListSMV**: Lists all students for the schedule manager.

**getInstructors**: Returns a list of all instructors as a JSON response.

**showSMSMV**: Lists all schedule managers.

**showUpdateFormIV**: Displays a form to update student information by ID.

**updateInstructorSMV**: Updates instructor details and processes password changes.

**deleteInstructorSMV**: Deletes an instructor record and disassociates them from courses.

**showCreateStudentFormSMV**: Displays a form to create a new student.

**createSMV**: Processes the creation of a new student and an associated user account.

**updateStudentSMV**: Updates a student's details and handles password changes.

**deleteStudentSMV**: Deletes a student record and removes them from associated courses.

**exportStudentDataSMV**: Exports student data to an Excel file.

**importStudentsSMV**: Displays a form for importing student data from an Excel file.

**uploadExcelSMV**: Processes the uploaded Excel file and imports student and class data.

**deleteClassSMV**: Deletes a class and disassociates its students.

**showClassListSMV**: Lists all classes and shows the number of students enrolled in each.

**showUpdateClassSMV**: Displays a form to update class information.

**updateClassSMV**: Processes and saves updates to class information.

**Autowired:**

**CourseRepository**: Interface for CRUD operations on course data.

**StudentRepository**: Interface for CRUD operations on student data.

**InstructorRepository**: Interface for CRUD operations on instructor data.

**RoleRepository**: Interface for accessing role data.

**UserRepository**: Interface for CRUD operations on user accounts.

**ScheduleManagerRepository**: Interface for CRUD operations on schedule manager accounts.

**PasswordEncoder**: Utility for encoding passwords.

**ExcelExportService**: Service for exporting data to Excel files.

# StudentController

The StudentController provides an interface for student-specific functionalities within the platform. It enables students to manage their accounts, enroll in courses, and view their grades. The controller also allows for the creation and management of student records by the platform's administrators. This includes updating student details, managing account passwords, and handling student course enrollments. The integration with PasswordEncoder ensures secure password management, while the interaction with various repositories facilitates the retrieval and storage of student, course, and exam-related data.

**Methods**

**showStudentHomepage**: Renders the homepage for students.

**showStudentList**: Fetches and displays a list of all students.

**saveStudentWithCourse**: Saves a student's details along with course associations.

**findStudent**: Retrieves a specific student by their ID.

**findStudentsContainingByStudentFirstName**: Finds students whose names contain a given string.

**findByIdContaining**: Searches for courses by ID.

**mathQuizPage**: Displays a page for a math quiz.

**accountManager**: Shows account management options for students.

**editingCurrentUser**: Displays a form for students to edit their personal details.

**saveCurrentUserEdits**: Saves changes to a student's account after validation.

**showStudentCourses**: Lists all courses associated with the logged-in student.

**showStudentGrades**: Compiles and shows grade details for the logged-in student's courses and exams.

**Autowired:**

**StudentRepository**: Interface for JPA operations on student data.

**CourseRepository**: Interface for accessing course data.

**UserRepository**: Interface for JPA operations on user accounts.

**RoleRepository**: Interface for accessing role data.

**ExamRepository**: Interface for accessing exam data.

**PasswordEncoder**: Utility for encoding passwords.

**ExamSubmissionRepository**: Interface for JPA operations on exam submission data.

**InstructorRepository**: Interface for JPA operations on instructor data.

# UserRegistrationController

The UserRegistrationController is a skeletal structure for managing user registration within the application. It injects UserService to utilize its functionalities. As it stands, the controller does not define any request handling methods, implying it might be a work in progress or used as a part of a larger framework where request mappings are defined elsewhere or inherited. The actual registration logic is expected to reside within the UserService, which is abstracted from this controller.

**Administrator.java**The Administrator class is a JPA entity representing an administrator with fields for personal details, credentials, and associated roles. It uses annotations to define how it maps to a database table and includes methods for setting and retrieving the properties of an administrator, along with custom methods to populate an administrator's data from a User object. It also includes constructors for creating instances of the class.

**Parameters:  
adminId** - The primary key of the administrator.

**adminFirstName** - The first name of the administrator.

**adminLastName** - The last name of the administrator.

**adminEmail** - The email of the administrator.

**adminPassword** - The password for the administrator's account.

**adminUsername** - The username of the administrator.

**roles** - A list of Roles entities associated with the administrator.

**Methods:**  
**setRoot** - A custom method to set the administrator's details from a User entity.

**setUser** - Similar to setRoot, it sets the administrator's details from another User entity.

**Constructors** - Two constructors are defined, a no-argument constructor (public Administrator()) and a parameterized constructor that initializes all the fields.

**Course.java**

The Course class represents a course entity in a database for an educational institution, with fields for its unique ID, name, associated students, the instructor, and exams. It includes annotations for defining the database table mapping and relationships with other entities like Student, Instructor, and Exam. The class provides getters and setters for accessing and modifying these properties. It is designed to be part of a larger system managing educational information, such as a student information system.

**Parameters**

**id** - The unique identifier of the course.

**courseName** - The name of the course.

**students** - A set of Student entities that are enrolled in the course.

**instructor** - The Instructor entity that teaches the course.

**exams** - A set of Exam entities that are associated with the course.

**Methods**Getters and Setters

**Exam.java**

The Exam class models an exam entity with properties such as ID, name, duration, start and end times, course association, and related questions. It includes annotations for entity definition, table mapping, and establishing relationships with Course and ExamQuestion entities. The class also provides a method to format the start time of the exam and includes standard Java bean methods to get and set the properties of an exam. This setup facilitates the creation, storage, and retrieval of exam-related data within the system.  
  
**Parameters  
id** - A unique identifier for the exam.

**submissionCount** - The number of submissions for the exam.

**examName** - The name of the exam.

**durationInMinutes** - The duration of the exam in minutes.

**startTime** and **endTime** - The start and end times of the exam, represented as LocalDateTime objects.

**course** - The Course entity to which the exam belongs.

**formattedStartTime** - A non-persisted string to hold the formatted start time.

**questions** - A list of ExamQuestion entities associated with the exam.

**Methods:**

**Constructor** - Initializes startTime with the current date and time.

**formatStartTime** - A utility method to format startTime as a string according to a specific pattern.

Standard **getters** and **setters** for all fields.

**ExamDetails.java**

The ExamDetails class is designed to hold information about an exam, including the selected questions' IDs, exam name, duration, and chapter information. It provides a set of getters and setters for accessing and modifying these properties. This class likely serves as a means to transfer data between different layers of an application, such as from a user interface to business logic or vice versa. It is not a JPA entity, so it is not directly used for database operations but may be involved in the process of preparing data for persistence or after retrieval from persistence.

**Parameters**

**selectedExamQuestionIds** - A list of Long values that likely represent the IDs of questions selected for the exam.

**examName** - A String containing the name of the exam.

**durationInMinutes** - An int representing the duration of the exam in minutes.

**chapter** - An int that could represent a chapter number or identifier that the exam is associated with.

**Methods:**Traditional setters and getters for all fields.

**ExamQuestion.java:**The ExamQuestion class is intended to represent an exam question entity in a database. It includes fields for question details, options, and the correct answer, with a mechanism to represent different types of questions. The class provides methods to manipulate and retrieve these properties, with additional logic to handle the correct answer text depending on the type of question. This entity is likely used in conjunction with other entities, such as Exam or Course, to form a complete examination system. The use of Lombok simplifies the code by auto-generating boilerplate code like the no-argument constructor.

**Parameters:  
id** - The unique identifier of the exam question.

**isAiGenerated** - A boolean flag to indicate whether the question was generated by an AI system.

**questionText** - The text of the question itself.

**userAnswer** - A non-persistent field to hold the user's answer to the question.

**optionA**, **optionB**, **optionC**, **optionD** - The multiple-choice options for the question.

**correctAnswer** - The correct answer to the question, stored as a string.

**chapter** - An integer representing the chapter number associated with the question.

**questionType** - An enumeration QuestionType which indicates the type of question (multiple choice, true/false, fill in the blank).

**Inner Enum:**

**QuestionType** - An enumeration that defines constants for different types of questions.

**Methods:**Traditional setters and getters for all fields.

**ExamQuestionDisplay.java**

The ExamQuestionDisplay class is a simple POJO enhanced with Lombok annotations to automate the creation of mundane methods like getters, setters, and others. The class holds data relevant to an exam question for display purposes, such as the question text, the user's answer, and the correct answer text. It is likely used to transfer this specific subset of exam question data between different parts of the application, especially for presentation layers or client-side views.

**Parameters:**

**id** - A Long object that likely represents the unique identifier for the exam question.

**questionText** - A String containing the text of the question.

**userAnswer** - A String that holds the user's answer to the question.

**correctAnswerText** - A String that stores the correct answer text for the question.

**Methods:**The @Data annotation eliminates the need to manually code the getters and setters, equals, hashCode, and toString methods. They will be automatically generated during the compilation.

**getCorrectAnswerText** and **setCorrectAnswerText** - These are explicitly defined methods for accessing and modifying the **correctAnswerText** field. However, due to the @Data annotation, these are technically redundant unless they are intended to add additional behavior in the future.

**ExamResults.java**

The ExamResult class is designed to store and provide access to the results of an exam, including the total score and details about the answers given. The class provides methods to retrieve and modify these properties. It is likely used to convey exam result data within an application, possibly for reporting or analysis purposes. The class's structure suggests it could be used in the application's service layer or as part of a data handling mechanism to process exam outcomes.

**Parameters:**

**core** - An integer representing the numeric score achieved on the exam.

**correctAnswers** - A list of strings, each presumably containing the text of an answer that was correct.

**incorrectAnswers** - A list of strings, each containing the text of an answer that was incorrect.

**incorrectAnswersWithCorrections** - A map where the keys are the text of incorrect answers and the values are the corresponding correct answers.

**Methods:**Traditional setters and getters for all fields.

**ExamSubmissionEntity.java  
  
Parameters:**

**id** - The unique identifier for the exam submission.

**user** - The User entity that submitted the exam.

**exam** - The Exam entity to which this submission relates.

**userAnswers** - A list of strings to store the user's answers.

**submissionTime** - A LocalDateTime object capturing the time when the submission was made.

**score** - An integer representing the score obtained by the user.

**Methods:**

Standard **getters** and **setters** (getId, setId, getUser, setUser, getExam, setExam, getUserAnswers, setUserAnswers, getSubmissionTime, setSubmissionTime, getScore, setScore).

Placeholder setter methods for **userId** and **examId**, which do not contain any implementation. These are intended to be filled in to allow setting of foreign keys directly without needing to fetch the associated User or Exam entity.

**Instructor.java**

The Instructor class models the attributes and relationships of an academic instructor. It includes fields for identification, personal details, roles, and course associations. Through its JPA annotations, it is mapped to a database table and includes relationships to other entities such as Course and Roles. It is equipped with getters and setters for all fields, allowing for straightforward manipulation of instructor data. The presence of constructors and the setUser method provides various ways to instantiate and configure the state of an Instructor object.

**Parameters:  
instructorId** - The unique ID of the instructor.

**courses** - A set of Course entities associated with the instructor.

**roles** - A list of Roles entities linked to the instructor.

**instructorFirstName**, **instructorLastName**, **instructorEmail**, **instructorPassword**, **instructorUsername** - Fields representing personal and authentication details of the instructor.

**creditsTaught** - A float value representing the number of credits taught by the instructor.

**Methods:**

Standard **getters** and **setters** for all the fields (getInstructorId, setInstructorId, getRoles, etc.).

The **setUser** method is a convenience method for setting multiple fields from a User object.

**Constructors** include a no-argument default constructor and a parameterized constructor for initializing an Instructor object with specific values.

**Question.java**

The Question class is a simple Java bean with properties to hold a question's text, possible answers as options, and the correct answer. The getters and setters provide access to these properties, allowing for the creation, retrieval, and update of question details within the application. The class is likely used to facilitate operations such as displaying questions to users, collecting answers, and evaluating responses.

**Parameters:**

**questionText** - A String representing the text of the question.

**options** - A **Map**<**String**, **String**> where the key-value pairs represent the options for the question. The keys might be identifiers for the options (like "**A**", "**B**", "**C**", "**D**"), and the values are the actual text of the options.

**correctAnswer** - A String that holds the correct answer to the question. This could either be the text of the correct answer or an identifier corresponding to the correct key in the options map

**Methods**:  
  
Standard getters and setters for all the fields

**ScheduleManager.java**

The ScheduleManager class encapsulates the data for a schedule manager in a system, including their identification, personal details, and associated roles. The class is designed to interact with a database using JPA for operations such as creating, reading, updating, and deleting records. The included constructors and methods provide multiple ways to create and manage instances of this class. **Parameters:**

**managerId** - A unique identifier for the manager.

**managerFirstName**, **managerLastName**, **managerEmail**, **managerPassword**, **managerUsername** - Personal and authentication details of the manager.

**roles** - A list of Roles entities associated with the manager.

**Methods:**

Standard **getters** and **setters** for all fields (getManagerId, setManagerId, getManagerFirstName, etc.).

**setUser** - A method to set the fields of ScheduleManager from a given User entity. This method facilitates the population of ScheduleManager details directly from a User object.

**Constructors** include a no-argument default constructor and a parameterized constructor to instantiate the ScheduleManager object with initial values.

**Student.java**

The Student class models a student entity with personal details, enrollment information, roles, and account status within an academic system. It provides methods for accessing and modifying these properties. The class is designed to interact with a relational database through JPA for operations such as creating, reading, updating, and deleting student records. The class's structure suggests its use in a larger system that manages student accounts, course enrollments, and roles.

**Parameters:**

**studentId** - A unique identifier for the student.

**courses** - A set of Course entities representing the courses in which the student is enrolled.

**studentFirstName**, **studentLastName**, **studentEmail**, **studentPassword**, **studentUsername** - Fields containing the student's personal and account information.

**creditsTaken** - A float value representing the total credits in which the student is enrolled.

**roles** - A list of Roles entities associated with the student.

**enabled** - A boolean flag indicating whether the student's account is enabled.

**user** - A User entity that may represent additional user information or account details.

**Methods:**

Standard **getters** and **setters** for all fields (getStudentId, setStudentId, getStudentFirstName, etc.).

A **parameterized** **constructor** for initializing a Student object with **specific** **values**.

A **default** **constructor** that also initializes the **courses** **set**.

**toggleEnabled** - A method to switch the enabled state of the student's account.

**Roles.java**

The Roles class is used to create, retrieve, update, and delete role records in the database. It encapsulates the role's unique identifier and name, providing the necessary accessors and mutators to manipulate these properties. The simplicity of this entity suggests that its primary function is to categorize or group users within the application by their roles, which can then be used to grant or restrict access to various parts of the application based on these roles.

**Parameters:  
id** - A unique identifier for the role.

**name** - A String representing the name of the role.

**Methods:**

A no-argument constructor, which is a requirement for JPA entities.

A parameterized constructor to initialize the Roles object with an ID and name.

Standard **getters** and **setters** (getId, setId, getName, setName).

**User.java**

The User class encapsulates the user's details and integrates with Spring Security for authentication and authorization purposes. It maps to a database table and includes relationships with Student and Roles. The class is tailored to fit the requirements of the system's security configuration, supporting user verification, role-based access control, and account management features. The implementation of UserDetails allows instances of User to be used directly by Spring Security for handling user information.

**Parameters:**

**id** - The unique identifier for the user.

**firstName**, **lastName**, **email**, **password**, **username** - The personal and authentication details of the user.

**student** - A reference to a Student entity associated with the user, if applicable.

**verificationCode** - A code used for account verification purposes.

**verified** - A boolean flag indicating whether the user has been verified.

**enabled** - A boolean flag indicating whether the user's account is enabled.

**roles** - A list of Roles entities representing the roles associated with the user.

**Methods:**

A no-argument constructor and a parameterized constructor for creating instances of User.

Standard **getters** and **setters** for all fields (getId, setId, getFirstName, etc.).

**getAuthorities** - A method from the UserDetails interface that returns a collection of GrantedAuthority objects based on the user's roles.

**isAccountNonExpired**, **isAccountNonLocked**, **isCredentialsNonExpired** - Methods from the UserDetails interface that return true by default (indicating the account is not expired, not locked, and credentials not expired).

**isEnabled** - A method from the UserDetails interface that returns the value of the enabled field.

**toString** - An overridden method that provides a string representation of the User object.

**setVerified** - A method that sets the verified field, which is hardcoded to true in the method's body.

**AdministratorRepository.java**

The AdministratorRepository provides a way to retrieve Administrator entities using their username as the search criteria, with support for exact matches and substring matches. By extending JpaRepository, it leverages Spring Data JPA to reduce the amount of boilerplate code needed to interact with the database and supports a wide range of database operations and query methods. Custom query methods can be easily added to this interface to fit the specific needs of the application it is used in.

**Query Methods:**

**Optional<Administrator> findByAdminUsername(String username)** - This method declares a query to find an Administrator by their username. It returns an Optional which will contain the Administrator if found or be empty if no match was found, thus helping to handle null values gracefully.

**List<Administrator> findByAdminUsernameContaining(String instructorUser)** - This method allows searching for administrators whose usernames contain a specified string. It returns a list of matching Administrator entities.

**CourseRepository.java**

The CourseRepository provides methods to fetch Course entities based on their ID or associated Instructor. The usage of JpaRepository allows leveraging Spring Data JPA's functionalities for data access and manipulation, reducing boilerplate code and increasing efficiency. The interface allows for the extension of functionality through custom methods and JPQL queries, enabling more complex data retrieval tailored to the application's requirements.

**Query Methods:**

**List<Course> findByIdContaining(Long Id)** - This method is intended to find courses by their ID, where the ID contains the passed long value. However, typically the Containing keyword is used for string comparisons in repository query methods, and using it with a Long type may not be appropriate unless the method signature or the way the IDs are stored and queried is customized to fit such a usage.

**@Query("SELECT c FROM Course c WHERE c.instructor = :instructor") -** This is a JPQL (Java Persistence Query Language) custom query that selects all courses taught by a specific instructor. The :instructor is a parameter placeholder that will be replaced by the actual Instructor entity passed to the findAllByInstructor method.

**@Param("instructor")** - An annotation that binds the parameter of the findAllByInstructor method to the :instructor placeholder in the query.

**ExamQuestionRepository.java**

The ExamQuestionRepository provides a set of methods for querying ExamQuestion entities from the database based on various criteria such as question text, chapter, and question type. It includes the ability to perform case-insensitive searches and retrieve random questions, which can be useful for creating varied exam experiences. The Pageable parameter in the random question query suggests that the application may have functionality to present a subset of questions, such as for a quiz or practice test. The interface extends the capabilities of standard JPA repositories with custom queries to meet the specific needs of the application.

**Query Methods:**

**ExamQuestion findByQuestionText(String questionText)** - Retrieves an ExamQuestion entity where the questionText exactly matches the provided string.

**List<ExamQuestion> findByChapter(int chapter) -** Returns a list of ExamQuestion entities that are associated with a specific chapter number.

**@Query("SELECT DISTINCT eq.chapter FROM ExamQuestion eq") -** Uses JPQL to select distinct chapter numbers from all exam questions in the database.

**List<ExamQuestion> findQuestionsByChapter(int chapter) -** An alternative method to findByChapter which may be redundant unless it has a different implementation or specific reason for being separate.

**void save(List<ExamQuestion> aiQuestions) -** An overloaded version of the save method that allows for saving a list of ExamQuestion entities, typically used for batch insertions or updates.

**List<ExamQuestion> findByQuestionTextContainingIgnoreCase(String searchText)** - Finds all ExamQuestion entities where the questionText contains the given search text, ignoring case.

**Optional<ExamQuestion> findById(Long id)** - Retrieves an ExamQuestion entity by its ID, wrapped in an Optional to handle the possibility of the entity not being found.

**@Query("SELECT q FROM ExamQuestion q WHERE q.questionType = ?1 ORDER BY FUNCTION('RAND')")** - Executes a custom JPQL query to retrieve a list of random ExamQuestion entities of a specific QuestionType. The Pageable parameter allows for pagination and limiting the results.

**ExamRepository.java**

The ExamRepository is designed to streamline interactions with the database for operations concerning Exam entities, especially in relation to their associated Course entities. By extending JpaRepository, it inherits several useful methods for entity management, such as saving, finding, and deleting exams. The custom query method findByCourse adds specific functionality to retrieve exams based on the course, which can be particularly useful in an academic or educational application where organizing and accessing exams based on courses is a common requirement.

**Query Methods:**

**Set<Exam> findByCourse(Course course) -** This method provides a way to retrieve all Exam entities associated with a specific Course entity. The method returns a Set of Exam instances, ensuring that each exam is unique within the set and directly related to the given course.

**ExamSubmissionRepository.java**

The ExamSubmissionRepository offers tailored methods for querying, counting, and deleting ExamSubmissionEntity records based on user and exam identifiers. The repository takes advantage of Spring Data JPA's capabilities to minimize the need for boilerplate code while allowing for transactional operations and entity-specific queries to support the application's business logic related to exam submissions.

**Query Methods:**

**ExamSubmissionEntity findByUser\_IdAndExam\_Id(Long userId, Long examId) -** This method retrieves a single ExamSubmissionEntity based on a composite query that includes both the user's ID and the exam's ID. It is useful when you need to find a particular exam submission for a specific user.

**@Transactional void deleteByExam(Exam exam) -** Annotated with @Transactional to ensure the method is executed within a transaction context. It deletes all ExamSubmissionEntity instances associated with the provided Exam entity.

**long countByExam(Exam exam) -** Counts the number of ExamSubmissionEntity instances for a given Exam, which can be useful for statistical or administrative purposes, such as determining the number of submissions for a particular exam.

**boolean existsByUser\_IdAndExam\_Id(Long userId, Long examId) -** Checks whether an exam submission exists for a particular user and exam combination, returning a boolean value. This can be used to enforce rules like "one submission per student per exam."

**Instructor.java**

The InstructorRepository facilitates CRUD operations on Instructor entities and defines additional methods for finding instructors by username, as well as searching for instructors whose first name or username contains a specified string. These methods are particularly useful for implementing features like search and user profile retrieval based on partial input. The JpaRepository inheritance provides a rich set of default database interaction methods, while the custom methods enable more specific queries tailored to application requirements.

**Query Methods:**

**Optional<Instructor> findByInstructorUsername(String username) -** Retrieves an Instructor by their username. The use of Optional indicates that the query might not find an instructor, in which case it will return an empty Optional.

**List<Instructor> findByInstructorFirstNameContaining(String searchParam) -** Returns a list of Instructor entities where the instructorFirstName contains the specified search parameter. This is useful for implementing search features where the exact first name is not known.

**List<Instructor> findByInstructorUsernameContaining(String searchParam) -** Similar to the previous method, this returns a list of Instructor entities where the instructorUsername contains the specified search parameter. This can be used for a username-based search functionality.

**RoleRepository.java**

The RoleRepository simplifies the interaction with the roles table in the database by providing an abstraction layer that automatically implements common operations, such as finding a role by its name. The use of Optional in the custom query method is a good practice for handling the absence of a result in a clean and safe manner. This repository interface would be used within the service layer of an application to perform data access operations related to roles, which are typically used for authorization and access control within the application.

**Query Methods:**

**Optional<Roles> findByName(String name) -** This method provides the ability to find a Roles entity based on its name attribute. The return type Optional<Roles> is used to handle the case where the role might not be found, thus avoiding the possibility of a NullPointerException.

**ScheduleManagerRepository.java**

The ScheduleManagerRepository allows for the retrieval and manipulation of ScheduleManager records in the database. It uses the convenience of Spring Data JPA to reduce the need for boilerplate code, providing pre-implemented methods for standard operations. In addition, it includes custom methods for searching managers by username and first name, enhancing the repository's utility for client applications that require dynamic search capabilities based on user input.

**Query Methods:**

**Optional<ScheduleManager> findBymanagerUsername(String username) -** Retrieves a ScheduleManager entity based on the managerUsername. The method's return type is Optional, which means it will either contain the retrieved ScheduleManager or be empty if no matching record is found.

**List<ScheduleManager> findByManagerFirstNameContaining(String searchParam)** - Searches for ScheduleManager entities whose managerFirstName contains the given substring. This can be particularly useful for implementing search functionalities where users can look for managers by parts of their first name.

**List<ScheduleManager> findByManagerUsernameContaining(String searchParam)** - Similar to the above method, but it searches within the managerUsername field. This method is useful when you need to find schedule managers based on a partial match of their username.

**StudentRepository.java**

The StudentRepository provides methods for querying Student entities by various criteria, such as username and first name substrings for search functionalities. It also allows for querying students based on their associated User entity or the courses they are enrolled in. The repository makes use of Spring Data JPA's features to streamline data access operations and custom query creation, thus minimizing boilerplate code and simplifying the implementation of complex database queries.

**Query Methods:**

**Optional<Student> findByStudentUsername(String username) -** Retrieves a Student entity based on the studentUsername. The use of Optional is a way to potentially avoid NullPointerException by wrapping the result.

**List<Student> findBystudentFirstNameContaining(String name) -** Returns a list of Student entities where the studentFirstName field contains the specified string, which is useful for search functionality.

**List<Student> findBystudentUsernameContaining(String studentUsername) -** Finds students whose studentUsername contains the specified substring.

**Optional<Student> findByUserId(User user) -** Retrieves a Student entity that is associated with a given User entity. This method may be redundant given the following method.

**@Query("SELECT s FROM Student s WHERE s.user.id = :userId") -** This is a JPQL (Java Persistence Query Language) custom query to find a Student based on the user's ID. The @Param("userId") annotation is used to bind the method parameter userId to the placeholder in the query.

**List<Student> findAllByCoursesIn(List<Course> courses) -** Retrieves a list of Student entities that are enrolled in any of the Course entities provided in the list.

**UserRepository.java**

The UserRepository provides data access and manipulation capabilities for User entities in the system. It includes methods for retrieving users by username and verification code, with the flexibility to extend or modify with additional query methods as needed. By using Spring Data JPA, it significantly reduces boilerplate code for database operations, allowing for straightforward and efficient data access patterns within the application.

**Query Methods:**

**Optional<User> findByUsername(String username) -** This method retrieves a User by their username. Wrapping the result in an Optional is a common practice to handle the possibility of the User not being found without resulting in a NullPointerException.

**User findByVerificationCode(String verificationCode) -** This custom method returns a User entity based on a provided verificationCode. Unlike the previous method, it returns a direct User object, assuming that the verification code is unique and a corresponding user will be found. However, using Optional<User> might be a safer approach to handle cases where no user is found for the given verification code.

**CourseService.java**

The CourseService class provides business operations for Course entities through methods that utilize the CourseRepository. This setup separates business logic (in the service layer) from data access logic (in the repository layer), following the common pattern of Spring applications to maintain clean architecture and separation of concerns. The service is responsible for defining actions like saving individual courses or a collection of them, while the repository handles the actual database interaction.

**Methods:  
save(Course course) -** This method delegates to CourseRepository to save a single Course entity to the database.

**saveAll(List<Course> courses) -** This method uses CourseRepository to save a list of Course entities to the database in a batch, which is more efficient for multiple entities.

**Autowired:**

**Course**: This is the domain class that represents the Course entity, containing course-related attributes and is used to interact with the database via JPA.

**CourseRepository**: This is a Spring Data JPA repository interface for Course entities. It abstracts away the boilerplate data access code and provides convenient methods to perform CRUD operations and other queries.

**EmailService.java**

The EmailService class is responsible for sending emails in a Spring application. It leverages Spring's mail abstraction framework, particularly JavaMailSender for mail sending functionality and SimpleMailMessage for creating email messages. The service provides a method sendEmail that clients of this service can use to send emails by simply providing the necessary parameters. This class abstracts the complexity involved in email communications, offering a simple interface for sending emails to other components or services in the application.

**Methods:  
sendEmail(String recipientEmail, String subject, String message) -** This is the key method of the service that crafts and sends an email. It takes the recipient's email address, email subject, and message body as parameters, uses SimpleMailMessage to compose an email, and then sends it using JavaMailSender.

**Autowired:**

**JavaMailSender -**This is an interface provided by Spring Framework that abstracts away the mailing functionality, allowing for sending emails. It is typically configured with SMTP settings to connect to a mail server.

**SimpleMailMessage -** This is a class provided by Spring Framework that represents a simple mail message, including data such as the recipient's email, subject, and text content.

**ExamQuestionService.java**

The ExamQuestionService interface specifies a collection of methods for managing ExamQuestion entities, including CRUD operations, data import from files, question generation, and retrieval of questions based on various criteria like chapter and content. Implementations of this service would be responsible for the business logic associated with these operations, such as interfacing with the persistence layer, processing files, and generating questions. This service acts as a bridge between the domain model (ExamQuestion entities) and the mechanisms used to store, retrieve, and manipulate these entities within an application.

**Methods:  
getAllExamQuestions()-** Retrieves a list of all ExamQuestion entities.

**getExamQuestionById(Long id)-** Fetches a single ExamQuestion by its ID.

**saveExamQuestion(ExamQuestion examQuestion)-** Saves an ExamQuestion to the repository.

**deleteExamQuestion(Long id)-** Deletes an ExamQuestion from the repository based on its ID.

**readExamQuestionsFromFile()-** Loads exam questions from a file.

**getQuestionsByChapter(int chapter)-** Retrieves a list of ExamQuestion entities associated with a specific chapter.

**generateQuestionsForChapter(int chapter)**- Generates questions for a given chapter.

**getAllChapters()-** Retrieves a list of all chapter numbers that have associated questions.

**readBlanksFromFile()** Reads fill-in-the-blank questions from a file.

**generateFillInTheBlanksQuestions(int numBlanks)-** Generates a specified number of fill-in-the-blank questions.

**readTrueFalseFromFile()-** Reads true/false questions from a file.

**readAIQuestionsFromFile(MultipartFile file)-** Loads AI-generated questions from a file.

**findQuestionsContainingText(String searchText)-** Finds questions that contain the specified text.

**getRandomTrueFalseQuestions(int numTrueFalse)-** Retrieves a random set of true/false questions.

**getRandomFillInTheBlanksQuestions(int numBlanks)-** Retrieves a random set of fill-in-the-blanks questions.

**ExamQuestionServiceImpl.java**

ExamQuestionServiceImpl provides a concrete implementation of services related to exam question management. It leverages ExamQuestionRepository for CRUD operations and extends the functionality by implementing methods to initialize the database with questions from files, retrieve random questions, and perform search operations. The service acts as a middle layer between the controllers and the data access layer, encapsulating the business logic required to manipulate ExamQuestion data. The @Transactional annotation ensures that operations like saving and deleting are conducted within a transaction, maintaining data integrity.

**Methods:  
initializeQuestions()-** Initializes the database with questions upon application startup if the database is empty. It loads questions from files for multiple-choice, true/false, and fill-in-the-blanks formats.

**getRandomTrueFalseQuestions(int numQuestions)-** Retrieves a random set of true/false questions limited by the given number.

**getRandomFillInTheBlanksQuestions(int numQuestions)-** Retrieves a random set of fill-in-the-blank questions limited by the given number.

**loadTrueFalseQuestions()-** Loads true/false questions from a specific file.

**loadBlanksQuestions()-** Loads fill-in-the-blank questions from a specific file.

**generateFillInTheBlanksQuestions(int numberOfQuestions)-** Generates a list of fill-in-the-blank questions based on the specified number.

**loadQuestionsFromFile(int chapter)-** Loads multiple-choice questions from a file corresponding to a specific chapter.

**readAIQuestionsFromFile(MultipartFile file)-** Reads AI-generated questions from an uploaded file.

**readBlanksFromFile()-** Reads fill-in-the-blank questions from a file.

**readTrueFalseFromFile()-** Reads true/false questions from a file.

**transformForDisplay(List<ExamQuestion> questions, Map<Long, String> userAnswers)-** Transforms a list of ExamQuestion entities into display format.

**getAllChapters()-** Retrieves all distinct chapter numbers from the database.

**generateQuestionsForChapter(int chapter)-** Generates questions for a specific chapter.

**getQuestionsByChapter(int chapter)-** Retrieves questions by chapter.

**getAllExamQuestions()-** Retrieves all exam questions.

**getExamQuestionById(Long id)-** Fetches a single ExamQuestion by its ID.

**saveExamQuestion(ExamQuestion examQuestion)-** Saves an ExamQuestion to the database.

**deleteExamQuestion(Long id)-** Deletes an ExamQuestion by ID.

**findQuestionsContainingText(String searchText)-** Finds questions containing a specific text.

**Autowired:**

**ExamQuestion**- The domain entity representing an exam question.

**ExamQuestionRepository-** The repository interface providing data access operations for ExamQuestion entities.

**ResourceLoader-** Spring's utility for loading resources (like files) from the classpath, filesystem, etc.

**MultipartFile-** A representation of an uploaded file received in a web request.

**ExamService.java**

The ExamService class is responsible for the business logic associated with exams and exam questions. It handles the creation and evaluation of exams, managing exam questions, and generating reports based on exam results. It performs initial loading of questions from files and offers functionalities to evaluate user responses, generate random questions for exams, and maintain a count of exam submissions. The service relies on repository interfaces for data access, ensuring a separation of concerns between the business logic and data persistence layers.

**Methods:  
getAllQuestions()-** Returns all questions available in the service.

**getExamById(Long id)-** Retrieves a specific exam by its ID.

**evaluateAnswers(Map<Integer, String> userAnswers)-** Evaluates the user's answers and computes a score.

**storeExamResultForUser(ExamResult result)-** Stores the result of an exam for a user.

**getStoredExamResultForUser()-** Retrieves the stored exam result for a user.

**getRandomQuestions()-** Selects a random set of questions for an exam.

**readTrueFalseQuestions(String resourcePath)-** Reads true/false questions from a resource path.

**generateExam(int chapterOrExamType, int numberOfQuestions)-** Generates an exam with questions based on the chapter or exam type.

**createExcelFile(List<Question> questions)-** Creates an Excel file from a list of questions.

**getAllChapters()-** Retrieves a list of all unique chapters from the questions.

**generateQuestionsForChapter(int chapter)-** Generates questions for a specific chapter.

**deleteExam(Long examId)-** Deletes an exam if there are no submissions for it.

**getAllExamQuestions()-** Retrieves all exam questions.

**updateExamQuestions(Long examId, List<Long> questionIds)-** Updates an exam's questions with a list of question IDs.

**getAllQuestions()-** Returns all questions available in the service.

**getExamById(Long id)-** Retrieves a specific exam by its ID.

**evaluateAnswers(Map<Integer, String> userAnswers)-** Evaluates the user's answers and computes a score.

**storeExamResultForUser(ExamResult result)-** Stores the result of an exam for a user.

**getStoredExamResultForUser()-** Retrieves the stored exam result for a user.

**getRandomQuestions()-** Selects a random set of questions for an exam.

**readTrueFalseQuestions(String resourcePath)-** Reads true/false questions from a resource path.

**generateExam(int chapterOrExamType, int numberOfQuestions)-** Generates an exam with questions based on the chapter or exam type.

**createExcelFile(List<Question> questions)-** Creates an Excel file from a list of questions.

**getAllChapters()**- Retrieves a list of all unique chapters from the questions.

**generateQuestionsForChapter(int chapter)-** Generates questions for a specific chapter.

**deleteExam(Long examId)-** Deletes an exam if there are no submissions for it.

getAllExamQuestions(): Retrieves all exam questions.

**updateExamQuestions(Long examId, List<Long> questionIds)-** Updates an exam's questions with a list of question IDs.

**getAllExamsWithSubmissionCount()-** Retrieves all exams and includes the count of submissions for each.

**Autowired:**

**Exam, ExamQuestion, ExamResult, Question-** Domain entities representing different aspects of the examination system.

**ExamRepository, ExamQuestionRepository, ExamSubmissionRepository-** Repository interfaces that provide data access operations for the corresponding entities.

**ResourceLoader-** Spring's utility for loading resources (like files) from the classpath, filesystem, etc.

**MultipartFile-** A representation of an uploaded file received in a web request.

**ExcelExportService.java**

The ExcelExportService is a utility service for exporting student information to an Excel file, which can be useful for generating reports or backups. It demonstrates how to use Apache POI to interact with Excel files programmatically, including checking for the existence of files, reading and writing data, and appending to existing files. The service is designed to be flexible, allowing for both the creation of new files and the updating of existing ones.

**Methods:  
exportStudentData(List<Student> students, String filePath, boolean fileExists)**- Exports a list of Student objects to an Excel file at the specified filePath. If **fileExists** is true, it loads an existing file; **otherwise**, it creates a new one. It **adds** **a** **new** sheet with header rows if necessary and appends student data as new rows to the sheet.

**Autowired:**

**Student:** This is the domain class representing a student. It contains student-related information such as **ID**, **first name, last name, email, password, username, and credits taken.**

**Workbook, Sheet, Row, Cell, XSSFWorkbook:** These are classes from the Apache POI library that represent various components of an Excel workbook.

**ExcelGeneratorService.java**

The ExcelGeneratorService is responsible for creating Excel files based on quiz and question data, showcasing two approaches—one with a generic list of maps and the other with a list of domain-specific Question objects. It abstracts the complexities of file creation and data formatting behind simple method calls, providing an easy-to-use interface for generating downloadable Excel files in a system that manages quizzes or tests. This service would typically be used in the context of an educational application or a platform that requires reporting and data export capabilities.

**Methods:  
generateExcel(List<Map<String, Object>> quizData)-** Generates an Excel file from a list of maps representing quiz data. Each map contains question data including the choices. The method returns a byte array representing the Excel file's content.

**createExcelFile(List<Question> questions)-** Creates an Excel file from a list of Question objects and returns it as a ByteArrayInputStream. This stream can then be used to output the file in various ways, such as through a file download response in a web application.

**ExcelParserService.java**

The ExcelParserService is designed to process Excel files containing data for courses, instructors, and students. It reads the Excel file using Apache POI, extracts relevant data, and uses Spring Data repositories to persist this data in the database. The service is annotated with @Transactional to ensure that all operations are completed successfully or rolled back in case of an error, maintaining data integrity. This service acts as a bridge between the presentation layer (file upload) and the persistence layer (database), enabling data import functionality within the application.

**Methods:  
parseExcelFile(MultipartFile file)-** Takes an uploaded Excel file (MultipartFile), reads its contents, and parses the data to create and store Course, Instructor, and Student entities in the database.

**Autowired:**

**StudentRepository-** The JPA repository for Student entities. It is used to perform database operations like checking if a student exists and saving student data.

**InstructorRepository-** The JPA repository for Instructor entities. It is used to save instructor data to the database.

**CourseRepository-** The JPA repository for Course entities. It is used to save course data to the database.

**Workbook, Sheet, Row-** Classes from the Apache POI library that represent various components of an Excel workbook.

**InstructorService.java**

The InstructorService provides functionality for saving individual instructors, saving a list of instructors, and deleting an instructor by ID. It handles cases where instructors might not exist in the database by throwing a ResourceNotFoundException. This service simplifies interactions with instructor data and encapsulates the logic required to manage instructor records in the application's database. The use of @Transactional ensures that operations such as deleting an instructor are completed successfully or fully rolled back in case of failure.

**Methods:  
deleteInstructor(Long instructorId)-** Deletes an instructor by their ID. If the instructor is not found, it throws a ResourceNotFoundException.

**saveAll(List<Instructor> instructors)-** Saves a list of instructors to the database. This is typically used for bulk inserts or updates.

**save(Instructor instructor)-** Saves a single instructor entity to the database. This can be used for both creating new records and updating existing ones.

**Autowired:**

**InstructorRepository-** A Spring Data JPA repository interface that provides data access operations for Instructor entities.

**ResourceNotFoundException-** A custom exception that is thrown when a requested resource, in this case, an instructor, is not found in the database.

**ScheduleManagerService.java**

The ScheduleManagerService class is designed to handle operations related to course scheduling management. It includes methods for creating or updating instructors and courses, as well as deleting them. However, the presence of the @PostMapping annotation suggests that this class also contains a controller layer operation, which typically should be in a separate @Controller or @RestController class. This service directly interacts with repositories to perform CRUD operations and manipulate relationships between entities. The associateInstructorWithCourse method, in particular, embodies a business operation that should be part of the service layer but is structured as a web controller method, which can lead to confusion and potentially problematic application design.

**Methods:  
createOrUpdateInstructor(Instructor instructor)-** Saves or updates an Instructor entity in the database.

**createOrUpdateCourse(Course course)-** Saves or updates a Course entity in the database.

**deleteInstructor(Long id)-** Deletes an Instructor entity from the database by its ID.

**deleteCourse(Long id)-** Deletes a Course entity from the database by its ID.

**associateInstructorWithCourse(...)-** An endpoint method (annotated with @PostMapping) to associate an instructor with a course. It retrieves the instructor and course by their IDs, associates them, and persists the changes.

**Autowired:**

**InstructorRepository-** A Spring Data JPA repository for Instructor entities.

**CourseRepository-** A Spring Data JPA repository for Course entities.

**Course, Instructor-** Domain entities representing course and instructor data, respectively.

**Model-** A Spring MVC object that holds the model attributes (data) for a view in web applications.

**StudentService.java**

The InstructorService provides functionality for saving individual instructors, saving a list of instructors, and deleting an instructor by ID. It handles cases where instructors might not exist in the database by throwing a ResourceNotFoundException. This service simplifies interactions with instructor data and encapsulates the logic required to manage instructor records in the application's database. The use of @Transactional ensures that operations such as deleting an instructor are completed successfully or fully rolled back in case of failure.

**Methods:  
saveAll(Set<Student> students)-** This method takes a Set of Student objects and persists them all to the database using the StudentRepository. Using a Set ensures that all Student objects are unique before they are saved.

**Autowired:**

**StudentRepository-** A Spring Data JPA repository interface used for data access operations involving Student entities.

**UserService.java**

The UserService interface is designed to be part of the service layer in a Spring application and to interact with the Spring Security framework. It provides a contract for services that handle user registration and retrieval of user details for security purposes. The actual business logic will be in the implementation of this interface, where save will create and store a new User entity based on the registration details, and loadUserByUsername will be used by Spring Security to load a UserDetails object necessary for authentication and authorization processes.

**Methods:  
save(UserRegistrationDto registrationDto)-** This method is intended to handle the registration of new users. It takes a UserRegistrationDto object, which presumably contains user registration information, and returns a User entity after saving it to the database.

**loadUserByUsername(String username)-** This method overrides the loadUserByUsername from UserDetailsService. It is used by Spring Security during the authentication process to load details about the user by their username.

**Autowired:**

**User-** The domain entity that represents a user in the system. It likely contains credentials and other user-related information.

**UserRegistrationDto-** A Data Transfer Object (DTO) used for registering a new user. It typically holds information from a registration form such as username, password, email, etc.

**UsernameNotFoundException-** A Spring Security exception indicating that a user with the given username could not be found.

**UserServiceImpl.java**

UserServiceImpl is responsible for the concrete implementation of user-related services, including saving new users to the database and loading user details by username for authentication purposes. The service interacts directly with the UserRepository to perform operations on the User entity. It is an integral part of the application's security configuration, providing the necessary user details to Spring Security during the authentication process. The commented out method registerNewUserAccount suggests an alternate or previous implementation of the registration logic.

**Methods:  
save(UserRegistrationDto registrationDto)-** This method takes a UserRegistrationDto object containing user registration details, creates a User entity, and saves it using the UserRepository. This is typically used for registering new users.

**loadUserByUsername(String username)-** This method overrides the UserDetailsService method to fetch user details based on the username. If the user is not found, it throws a UsernameNotFoundException.

**Autowired:**

**UserRepository-** A Spring Data JPA repository for User entities, providing CRUD operations and custom queries.

**User-** The domain entity representing a user within the system.

**UserRegistrationDto-** A DTO that carries data required for user registration.

**UsernameNotFoundException-** An exception indicating that a user lookup by username failed, commonly used in Spring Security when a user cannot be found during the authentication process.

**ChatGTPRequest.java**

The ChatGPTRequest DTO is designed to encapsulate the data necessary to make a request to a chat model, containing the model identifier and a list of messages. It uses Lombok to reduce boilerplate code, and Jackson annotations to define JSON serialization and deserialization behavior. However, there seems to be a constructor that does not align with the typical usage of the class, which may require attention to ensure the class works as intended within the application. The class is a good example of a DTO that simplifies the transfer of data between different parts of a Spring web application, especially when dealing with REST APIs or external services.

**Fields:**

**model-** A String representing the name or identifier of the chat model to be used for processing the request.

**messages-** A List of Message objects that represent the conversation history or the current message being sent to the model.

**ChatGTPResponse.java**

The ChatGptResponseDTO is designed to encapsulate the response from a ChatGPT-like model or service. It consists of a list of choices, where each choice is represented by an index and a message. This structure allows for multiple response options to be returned by the model, and the client can choose the most suitable response based on the context. The use of Lombok annotations reduces boilerplate code, and the @JsonInclude annotation ensures that only non-null properties are included in the JSON representation of the object. This DTO is useful for simplifying the handling of responses from chat-based models within a Java application.

**Fields:**

**choices-** A List of Choice objects, representing the various choices or responses generated by the model.

**Choice is a nested class within ChatGptResponse-** It represents an individual choice or response generated by the model.

**index**: An integer representing the index or order of the choice.

**message**: A Message object representing the content of the choice.

**CourseGradeDTO.java**

The CourseGradeDTO class serves as a DTO for course grades. It allows for the storage of course-related information, including exam grades, and provides methods for adding exam grades and calculating the total score and percentage. This DTO is useful for representing and manipulating course grade data within a Java application.

**Fields:**

**courseName-** A String representing the name of the course.

**examGrades-** A List of ExamGradeDTO objects, representing the exam grades associated with the course.

**totalScore-** An Integer representing the total score achieved in the course.

**totalPossibleScore-** An Integer representing the total possible score in the course.

**percentage-** A Double representing the percentage score in the course.

**Methods:**

**addExamGrade(String examName, Integer score, Integer totalQuestions)-** A method to add an exam grade to the list of exam grades. It takes the exam name, score, and the total number of questions.

**addExamGrade(ExamGradeDTO examGrade)-** An overloaded method to add an ExamGradeDTO object directly to the list of exam grades.

**calculateTotalScoreAndPercentage()-** A method to calculate the total score and percentage based on the exam grades. It iterates through the exam grades, sums up the scores, and calculates the percentage.

**ExamGradeDTO.java**

The ExamGradeDTO class appears to be a Data Transfer Object (DTO) used to represent information related to exam grades. It contains details about an exam, including its name, score, total questions, and provides methods to format and calculate grades.

**Fields:**

**examName**- A String representing the name of the exam.

**score-** An Integer representing the score achieved in the exam. It uses Integer to handle null values if there's no submission.

**totalQuestions-** An Integer representing the total number of questions for the exam.

**Methods:**

**getFormattedScore()-** A method that returns the formatted score as a string, e.g., "5/10". It handles cases where the score or total questions are null.

**getLetterGrade()-** A method to calculate the letter grade based on the percentage score. It returns "A" for scores >= 90, "B" for scores >= 80, "C" for scores >= 70, "D" for scores >= 60, and "F" for lower scores.

**getPercentage()-** A method that returns the percentage score as a string, e.g., "50%". It handles cases where the score or total questions are null.

**Message.java**

The Message class is a lightweight DTO used to encapsulate message information. It can be used to represent messages with sender roles and message content. The Lombok annotations help reduce boilerplate code typically associated with Java classes, making the code more concise and readable.

**Fields:**

**role-** A String field representing the role or sender of the message.

**content-** A String field representing the content or text of the message.

**UserRegistrationDto.java**

The UserRegistrationDto class is used to gather and transfer user registration information in a structured manner. It contains fields for common registration details such as first name, last name, email, password, and username. However, the getVerificationCode() method is incomplete and currently returns null. This class is likely used in the context of a user registration process within a web application.

**Fields:**

**firstName**- A String field representing the user's first name.

**lastName-** A String field representing the user's last name.

**email-** A String field representing the user's email address.

**password-** A String field representing the user's chosen password.

**username-** A String field representing the user's chosen username.

**Methods:**

**getVerificationCode()-**intended to return a verification code

# Conclusion

This manual provides a basic understanding of the Online Examination Software's functionality and structure. For detailed implementation and code specifics, direct access to the source code is recommended.