Project Overview

The objective of this project is to create a University Management System capable of role-based navigation, importing and exporting data, as well as managing faculty, students, subjects, courses, and events. The program contains interactive modules to provide a cohesive system for managing academic institutions.

Introduction

This document provides a detailed user manual outlining the implementation steps and architectural design of the University Management Application. The document specifies details on the application's structure, its constituent components (including directories, packages, classes, and functions), and their interactions. This manual is intended for developers and technical users seeking an in-depth understanding of the application's codebase for maintenance, extension, or learning purposes. It describes the organization of the codebase, the purpose of different modules, the logic flow, and functionality of the various software components that constitute the application, ensuring all aspects of the repository structure are covered.

Project Structure

The University Management Application follows a standard directory structure to organize its components, facilitating navigation and maintenance of the codebase.

Root Level: At the root of the repository, essential files and directories define the project:

File/Directory	Contents	Purpose
src/	Java source code organized into packages	Contains the main implementation logic of the application.
test/	Java test files mirroring the src/ structure	Holds unit tests and integration tests.
resources/	fxml UI files and static	Stores non-Java code

	assets	resources required by the application.
pom.xml/	Project build configuration file	Defines project dependencies, build settings, and metadata.
.gitignore	List of files and directories to be ignored by Git	Specifies files not to be tracked by version control.
README.md (This file)	Project description, setup, usage instructions	Provides an overview and essential information.

Directory Breakdown

The src/ directory is the heart of the application, containing the Java source code organized as follows:

- main/java/: Contains the main application code. Within this, packages represent different modules, classes, and architectural layers.
- main/resources/: Holds resources specific to the main application (e.g., configuration files, fxml, static assets).

Similarly, the test/ directory contains:

- test/java/: Contains the test code, mirroring the package structure of the main application.
- test/resources/: Holds resources specific to the tests.

Package Structure (within main/java/)

Within the src/main/java/ directory, the source code is organized into the following packages based on the application's modules and architectural layers:

- **com.example.universitymanagementapp.model**: Contains Plain Old Java Objects (POJOs). The model code creates the generic structure for each module representing the core entities:
    - **Student.java:** A subclass of the superclass **User.java.** The student class represents a student with attributes (e.g. studentId, name, academicLevel, registeredCourses etc.)

and methods for setting and getting information (setters and getters).
    - **Course.java:** Represents a course with attributes (e.g. courseCode, courseName, instructor, enrolledStudents etc.) and methods for setting and getting information (setters and getters) as well as a method to enroll students.
    - **Faculty.java:** A subclass of the superclass **User.java.** The faculty class represents a faculty member with attributes (e.g. name, degree, email, coursesOffered etc.) and methods for setting and getting information (setters and getters).
    - **Subject.java:** Represents a subject with attributes subjectCode and subjectName as well as methods for setting and getting information (setters and getters).
    - **Event.java:** Represents a university event with attributes (e.g. eventCode, eventName, eventDescription, eventLocation etc.) and methods for setting and getting information (setters and getters).
    - **Admins.java:** A subclass of the superclass **User.java.** The admins class represents an admin with the attribute name as well as methods for setting and getting information (setters and getters).
    - **Grade.java:** Represents grades with attributes (e.g. courseCode, finalGrade, midtermGrade, assignmentGrade etc.) and methods for setting and getting information (setters and getters).
    - **User.java:** Superclass for all users (admin, faculty, student). Represents a user with the attributes username, hashedPassword, plaintextPassword, and role. The user model contains methods for setting and getting information (setters and getters).
    - **Activity.java:** Represents an activity which displays on the dashboard. The model contains the attributes type, description, and date. It contains methods for setting and getting information (setters and getters).
    - **Notification.java:** Represents a notification which displays on the dashboard. The model contains the attributes type, message, and date. It contains the methods for setting and getting information (setters and getters).
    - **Registration.java:** Represents a recent registration which displays on the dashboard. The model contains the attributes studentId, courseCode, courseName, and date. It contains methods for setting and getting information (setters and getters).

- **com.example.universitymanagementapp.dao**: Contains the business logic and CRUD operations of the application:
    - **StudentDAO.java:** Interface defining methods for CRUD operations on student entities. Provides methods for operations related to students, such as addStudent(), getStudentById(), getAllStudents(), updateStudent(), deleteStudent(),

enrollStudentInCourse() etc. Ensures correct input data to match with the corresponding model.

- **CourseDAO.java:** Interface defining methods for CRUD operations on course entities. Provides methods for course-related operations, such as addCourse(), getCourseByCode(), getAllCourses(), updateCourse(), removeCourse(), getCoursesTaught() etc. Ensures correct input data to match with the corresponding model.
- **FacultyDAO.java:** Interface defining methods for CRUD operations on faculty entities. Provides methods for faculty operations, such as addFaculty(), getFacultyById(), getAllFaculty(), updateFaculty(), deleteFacultyById(), getLoggedInFaculty() etc. Ensures correct input data to match with the corresponding model.
- **SubjectDAO.java:** Interface defining methods for CRUD operations on subject entities. Provides methods for managing subjects, such as addSubject(), removeSubject(), updateSubject(), getSubjectByName(), getAllSubjects(), clearSubjects(). Ensures correct input data to match with the corresponding model.
- **EventDAO.java:** Interface defining methods for CRUD operations on event entities. Provides methods for managing events, such as addEvent(), getEventById(), getAllEvents(), updateEvent(), and deleteEvent().
- **AdminDAO.java:** Interface defining methods for CRUD operations on admin entities. Creates a static list of admins and provides methods for managing admins, such as addAdmin(), getAllAdmins(), and clearAdmins(). Ensures correct input data to match with the corresponding model.

- **com.example.universitymanagementapp.controller**: Handles user requests and responses. Acts as the connection between the UI and the backend logic:
  - **StudentController**
    - **AdminStudentController.java:** Exposes endpoints for student-related operations as an admin. Connects the UI to the backend and handles underlying logic for user actions. Includes action methods (e.g. handleAddStudent(), handleEditStudent(), handleDeleteStudent(), handleEnrollCourse() etc.) as well as other backend and helper methods for displaying information and changing data (e.g. filterStudents(), loadAllStudents(), showStudentDetails(), updateStudentTuition() etc.).
    - **FacultyStudentController.java:** Exposes endpoints for student-related operations as a faculty member. Connects the UI to the backend and handles underlying logic for user actions. Includes methods for displaying information (e.g. populateTables(), setFacultyUsername(), setFacultyName(), studentInfo() etc.).

- **StudentDashboard.java:** Exposes endpoints for student-related operations in the student dashboard. Connects the UI to the backend and handles underlying logic for user actions. Includes action methods for the menu (e.g. handleDashboardAction(), handleCourseSelection(), handleSubjectSelection(), handleStudentSelection() etc.) as well as methods for displaying and getting updated information (e.g. refreshProfileTab(), setStudentId(), loadMyCourses(), loadSummaryData() etc.).
- **StudentSettingsController.java:** Exposes endpoints for student settings related operations. Connects the UI to the backend and handles underlying logic for user actions. Includes action methods for changing student info (e.g. handleChangePassword(), handleSelectImage(), handleSaveImage(), handleClearPassword() etc.) as well as methods for displaying and gathering data (e.g. setStudentId(), loadStudentData(), loadProfilePicture(), etc.).
- **StudentStudentController:** Exposes endpoints for profile related operations. Connects the UI to the backend and handles underlying logic for user actions. Includes methods for displaying and calculating data (e.g. setStudentId(), loadStudentData(), calculateTuitionFees() etc.).

○ **CourseController**
- **CourseAdminController:** Exposes endpoints for course-related operations as an admin. Connects the UI to the backend and handles underlying logic for user actions. Includes action methods (e.g. handleAddCourse(), handleEditCourse(), handleDeleteCourse(), handleSaveCourse() etc.) as well as methods for updating and displaying data (e.g. loadEnrolledStudents(), filterCourses(), loadAllCourses(), refreshCourses() etc.).
- **CourseFacultyController:** Exposes endpoints for course-related operations as a faculty member. Connects the UI to the backend and handles underlying logic for user actions. Includes methods for displaying data (e.g. loadMyCourses(), filterCourses(), setLoggedInFacultyName(), loadAllCourses() etc.).
- **CourseStudentController:** Exposes endpoints for course-related operations as a student. Connects the UI to the backend and handles underlying logic for user actions. Includes methods for displaying data (e.g. loadEnrolledCourses(), filterEnrolledCourses(), handleViewDetails(), setStudentId() etc.).

○ **FacultyController**
- **AdminFacultyController.java:** Exposes endpoints for faculty-related operations. Connects the UI to the backend and handles underlying logic for user actions. Includes action methods (e.g. handleAddFaculty(), handleEditFaculty(), hanldeEditFaculty(), handleAssignCourse() etc.) as well as methods for loading and displaying data (e.g. loadAllFaculty(), filterFaculty(),

showFacultyDetails(), clearForm() etc.).

- **FacultyDashboard.java:** Exposes endpoints for faculty-related operations. Connects the UI to the backend and handles underlying logic for user actions. Includes action methods to handle menu selection (e.g. handleDashboardAction(), handleSubjectSelection(), handleCourseSelection(), handleStudentSelection() etc.) as well as methods for loading and displaying data (e.g. loadSummaryData(), loadRecentActivities(), loadRecentRegistrations(), loadUpcomingEvents() etc.).
- **FacultyFacultyController.java:** Exposes endpoints for faculty-related operations when logged in as a faculty member. Connects the UI to the backend and handles underlying logic for user actions. Includes methods filterFacultyList(), showFacultyDetails, and initialize().
- **FacultySettingsController.java:** Exposes endpoints for faculty settings related operations. Connects the UI to the backend and handles underlying logic for user actions. Includes action methods (e.g. handleChangePassword(), handleClearPassword(), handleChangeLocation(), handleBrowsePicture() etc.) as well as methods for displaying and updating data (e.g. loadFacultyData(), clearProfilePictureFields(), showAlert() etc.).
- **StudentFacultyController.java:** Exposes endpoints for faculty-related operations when logged in as a student. Connects the UI to the backend and handles underlying logic for user actions. Includes methods for loading and displaying data (e.g. getStudentCoursesTaughtByFaculty(), setStudentId(), loadFacultyMembers(), filterFaculty() etc.).

○ **SubjectController**

- **AdminSubjectController.java:** Exposes endpoints for subject-related operations. Connects the UI to the backend and handles underlying logic for user actions. Includes action methods (e.g. handleAddSubject(), handleDeleteSubject(), handleEditSubject(), handleSaveSubject() etc.) as well as methods for loading and displaying data (e.g. loadAllSubjects(), filterSubjects() etc.).
- **FacultySubjectController.java:** Exposes endpoints for subject-related operations for faculty members. Connects the UI to the backend and handles underlying logic for user actions. Includes methods loadAllSubjects() and filterSubjects().
- **StudentSubjectControler.java:** Exposes endpoints for subject-related operations for students. Connects the UI to the backend and handles underlying logic for user actions. Includes methods loadAllSubjects() and filterSubjects().

- **EventController**
  - **EventAdminController.java:** Exposes endpoints for event-related operations for admins. Connects the UI to the backend and handles underlying logic for user actions. Includes action methods (e.g. handlePreviousMonth(), handleNextMonth(), handleAddEvent(), hanldeEditEvent() etc.) as well as methods for updating, loading, and displaying data (e.g. filterEvents(), showEventDetails() populateCalendar(), loadAllEvents() etc.).
  - **EventFacultyController.java:** Exposes endpoints for event-related operations for faculty members. Connects the UI to the backend and handles underlying logic for user actions. Includes action methods (e.g. handlePreviousMonth(), handleNextMonth(), handleRegisterFacultyDialog(), handleUnregisterFacultyDialog() etc.) as well as methods for updating, loading, and displaying data (e.g. showEventDetails(), filterEvents(), populateCalendar(), loadAllEvents() etc.).
  - **StudentEventController.java:** Exposes endpoints for event-related operations for students. Connects the UI to the backend and handles underlying logic for user actions. Includes action methods (e.g. handleRegisterStudentDialog(), handleUnregisterStudentDialog(), handlePreviousMonth(), handleNextMonth() etc.) as well as methods for updating, loading and displaying data (e.g. populateCalendar(), loadAllEvents(), loadRegisteredEvents(), filterEvents() etc.).
- **AdminController**
  - **AdminDashboard.java:** Exposes endpoints for admin dashboard related operations. Connects the UI to the backend and handles underlying logic for user actions. Includes action methods (e.g. handleSubjectSelection(), handleCourseSelection(), handleLogoutAction() etc.) as well as backend page methods (e.g. loadRecentActivities(), loadRecentRegistrations(), loadNotifications() etc.).
  - **AdminSettingsController.java:** Exposes endpoints for admin settings related operations. Connects the settings UI to the backend and handles underlying logic for user actions. Includes action methods (e.g. handleSaveProfile(), handleClearProfile(), handleChangePassword(), handleClearPassword() etc.).

- **com.example.universitymanagementapp.auth:** Contains the implementation and logic behind authentication and validating a student, faculty, or admin login:
  - **authenticator**
    - **UserAuthentication.java:** Loads user data through initializing DAO. Authenticates users by checking the passwords with passwords in the DAO.

- **LoginSelectionController.java:** Handles the users selection of logging is as an admin or user (faculty or student). Contains an initialize method to stage the page. Contains an action method handleUserLoginAction () which takes the user selection and loads the corresponding page. The controller also consists of an alert method to alert user of incorrect login attempts.
- **UserLoginController.java:** Connects the UI to the backend and handles underlying logic for user actions. Includes action methods handleUserBack() and handleUserLogin() which correspond to the back button and login button. The controller also includes the loadDashboard() method which loads the corresponding controller based on the users login information.

- **com.example.universitymanagementapp.util:** Contains utility classes for importing and exporting data as well as password verification:
  - **ExExporter.java:** Handles exporting data to an excel sheet as well as creating notifications, recent activities, and recent registrations. Contains backend logic for validating exported data before exporting. Includes methods for writing each modules data (e.g. writeStudents(), writeCourses(), writeFaculty() etc.) as well as methods for recording changes and updating student tuition prior to export (e.g. updateStudentTuition(), recordChanges()).
  - **ExImporter.java:** Handles importing data from an excel sheet and uses the modules DAO's for validating data and implements the constructors of each module to write the imported data into the constructors to create objects of the module classes. Contains methods for reading and validating data (e.g. checkFilePath(), readCourses(), readStudents(), readFaculty() etc.) as well as helper methods (e.g. isRowEmpty(), findStudentByName(), getNumericValue, getStringValue() etc.).
  - **PasswordHasher.java:** Utilizes SHA-256 algorithm for hashing passwords before using Base64 encoding to convert to human readable string format used for storage and transmission. Includes methods hashPassword() and verifyPassword().
- **UniversityManagementApp.java:** Main point of launch for the application. Creates static objects of each modules DAO to keep the data from the importer. Stages the login page through the start() method. Contains method importDataOnStartup() which clears the DAO's to ensure that there is no data prior to import and initializes an admin. Contains logging for debugging reading the excel file.
- **resources/com.example.universitymanagementapp:** Contains all the FXML files connected to their respective controller for the UI layout and operation. Contains a CSS file for button and text styling throughout the entire program. Contains images used for the program as well as datasheets.

Class Documentation (within each Package)

As outlined above, each class within these packages includes documentation explaining its purpose, attributes, and methods. The outline above states the specific purpose of each class and along with the general function and layout of the methods within each class.

Class Interactions (Logic Flow)

The classes within the University Management Application interact following a layered architecture pattern:

1. **Controller Layer (com.example.universitymanagementapp.controller)**: Receives incoming requests, validates input, and delegates processing to the DAO layer and on occasions the controller's own helper methods if not available in DAO. It formats responses to send back to the client.
2. **DAO Layer (com.example.universitymanagementapp.dao)**: Contains the core business logic as well as CRUD operations and interacts with the importer to manage imported data.
3. **Model Layer (com.example.universitymanagementapp.model)**: Consists of data-carrying objects used to transfer data between layers.
4. **Utility (com.example.university.util)**: Packages provide support functionalities for handling data and validation.

This layered approach promotes separation of concerns, making the application more modular, testable, and maintainable.