UE22CS352B - Object Oriented Analysis & Design

Mini Project Report

Attendance Management System

Submitted by:

Motamarri Sai Sathvik : PES2UG22CS321
Narayan Sreekumar : PES2UG22CS339
Nikhil Srivatsa : PES2UG22CS357
Nitheesh Pugazhanthi : PES2UG22CS371

Semester : 6 Section: F

Faculty Name:
Divya Ebenezer Nathaniel

January - May 2025

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
FACULTY OF ENGINEERING
PES UNIVERSITY
(Established under Karnataka Act No. 16 of 2013)
100ft Ring Road, Bengaluru – 560 085, Karnataka, India

Problem Statement

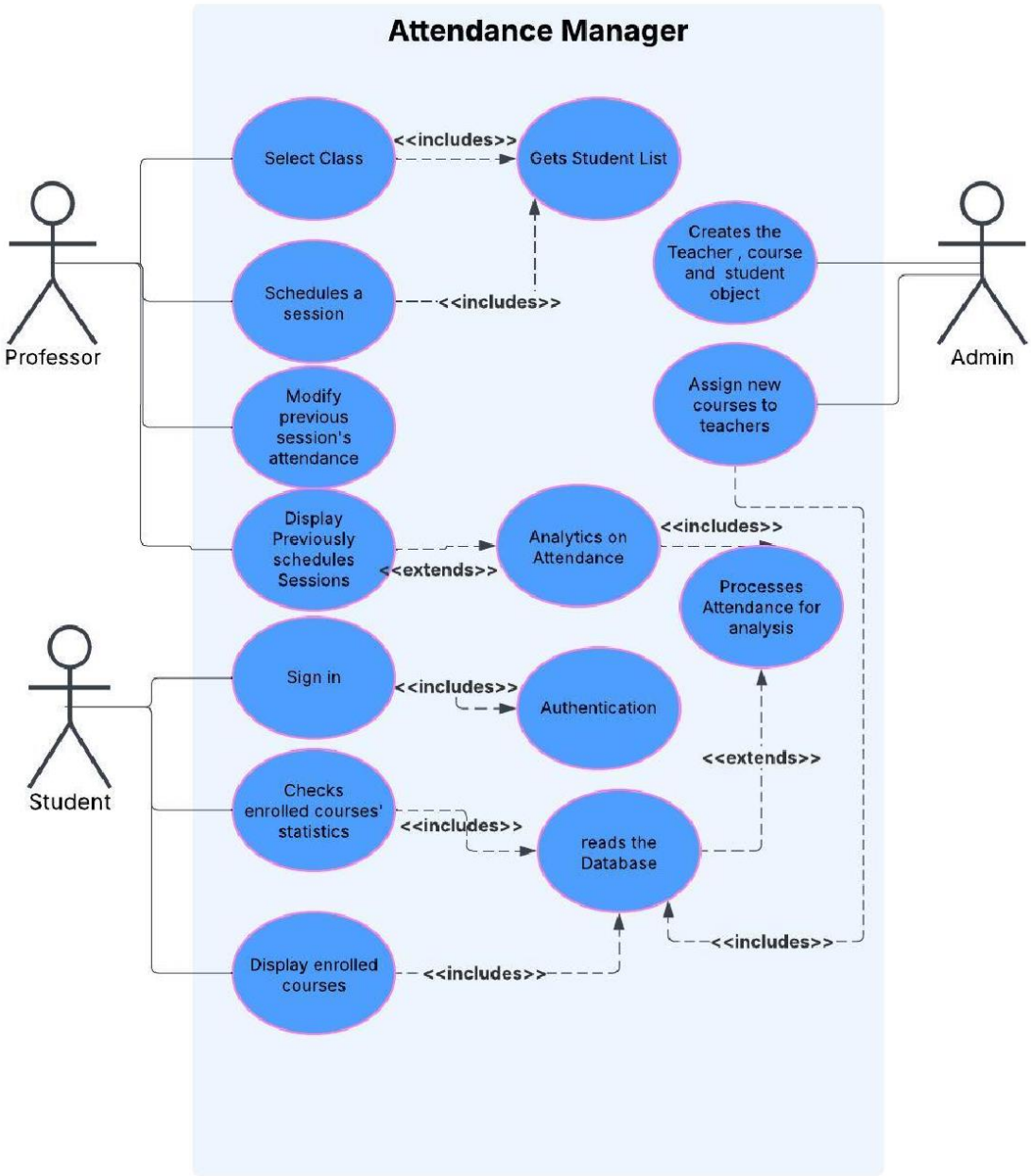
We are building an Attendance Manager system for college students using Spring Boot for the backend, MySQL for the database, and React for the frontend. The primary goal of this application is to simplify and digitize the process of attendance tracking in academic institutions. Traditional attendance methods—such as maintaining physical registers or using spreadsheets—are time-consuming, error-prone, and lack real-time accessibility. Our application addresses these issues by offering a user-friendly interface for students and faculty to manage attendance seamlessly. Faculty members can efficiently mark and update attendance records, generate subject-wise or student-wise reports, and monitor trends over time. Students, on the other hand, can log in securely to view their attendance status, receive notifications about low attendance, and stay informed about their academic standing. By leveraging modern technologies, this system enhances transparency, reduces administrative workload, and promotes accountability, making it a vital tool for educational institutions aiming to modernize their operations.

Key Features:

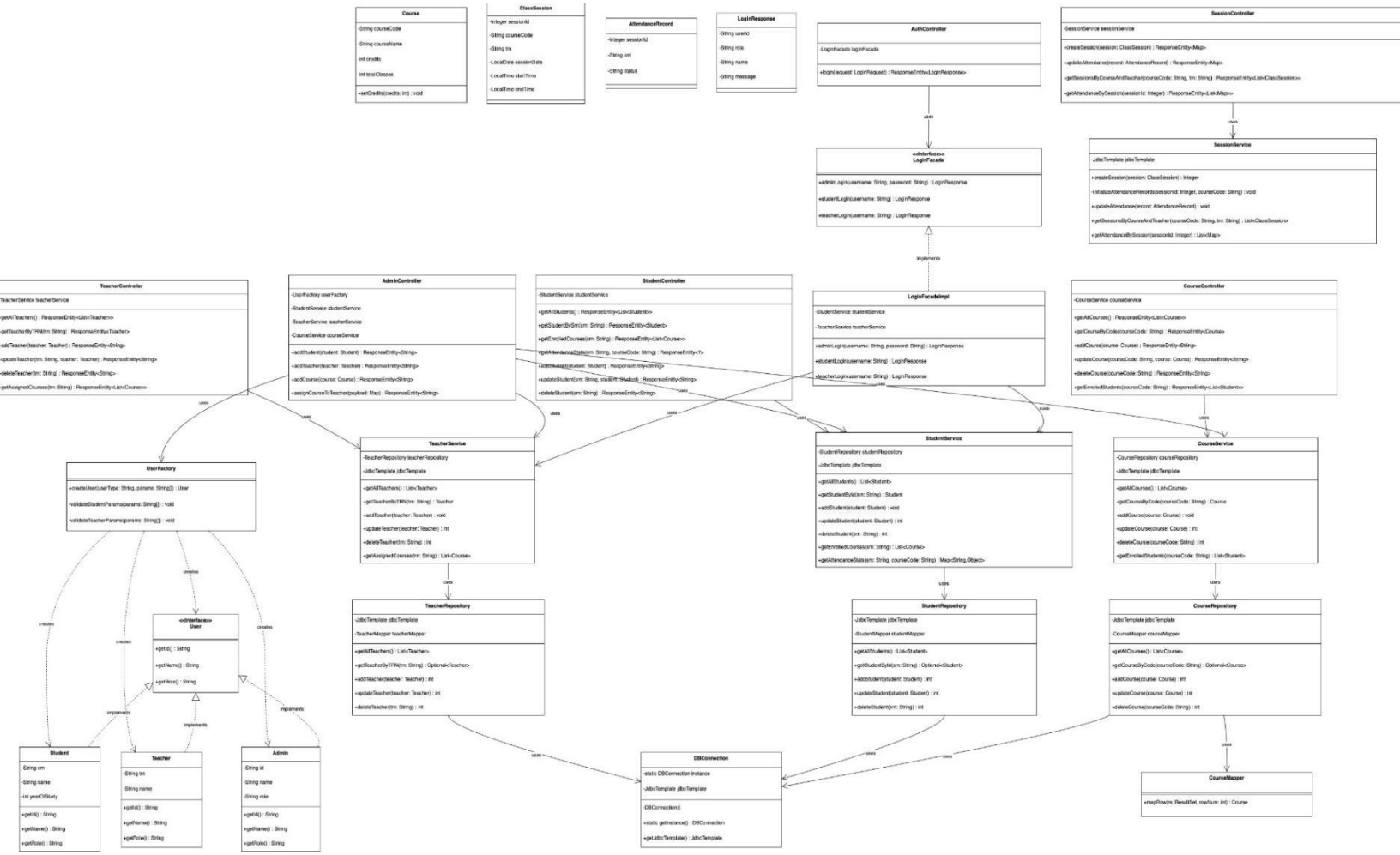
1) **Student Feature : Display the courses the student has enrolled in.**
2) **Student Feature : Display the attendance statistics for each enrolled course.**
3) **Admin Feature : Add new Students,Teachers and courses.**
4) **Authentication : Handles the authentication for all the different users.**
5) **Admin Feature : Assign new courses to the teachers.**
6) **Teacher Feature : Display previous course sessions.**
7) **Teacher Feature : Schedule new course sessions and mark the attendance.**
8) **Teacher Feature : Update the attendance for the previous sessions.**

Models:

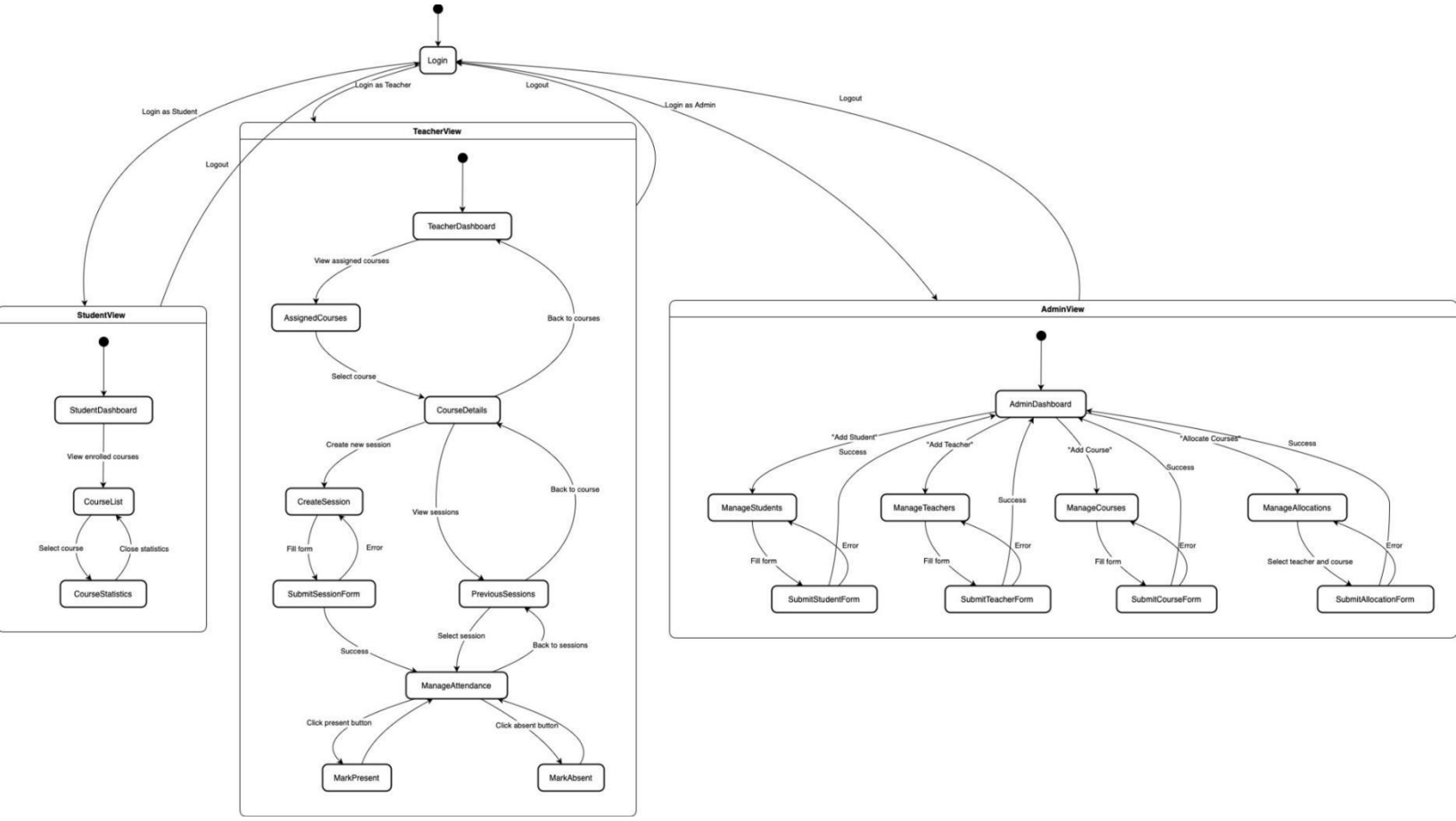
Use Case Diagram:



Class Diagram:

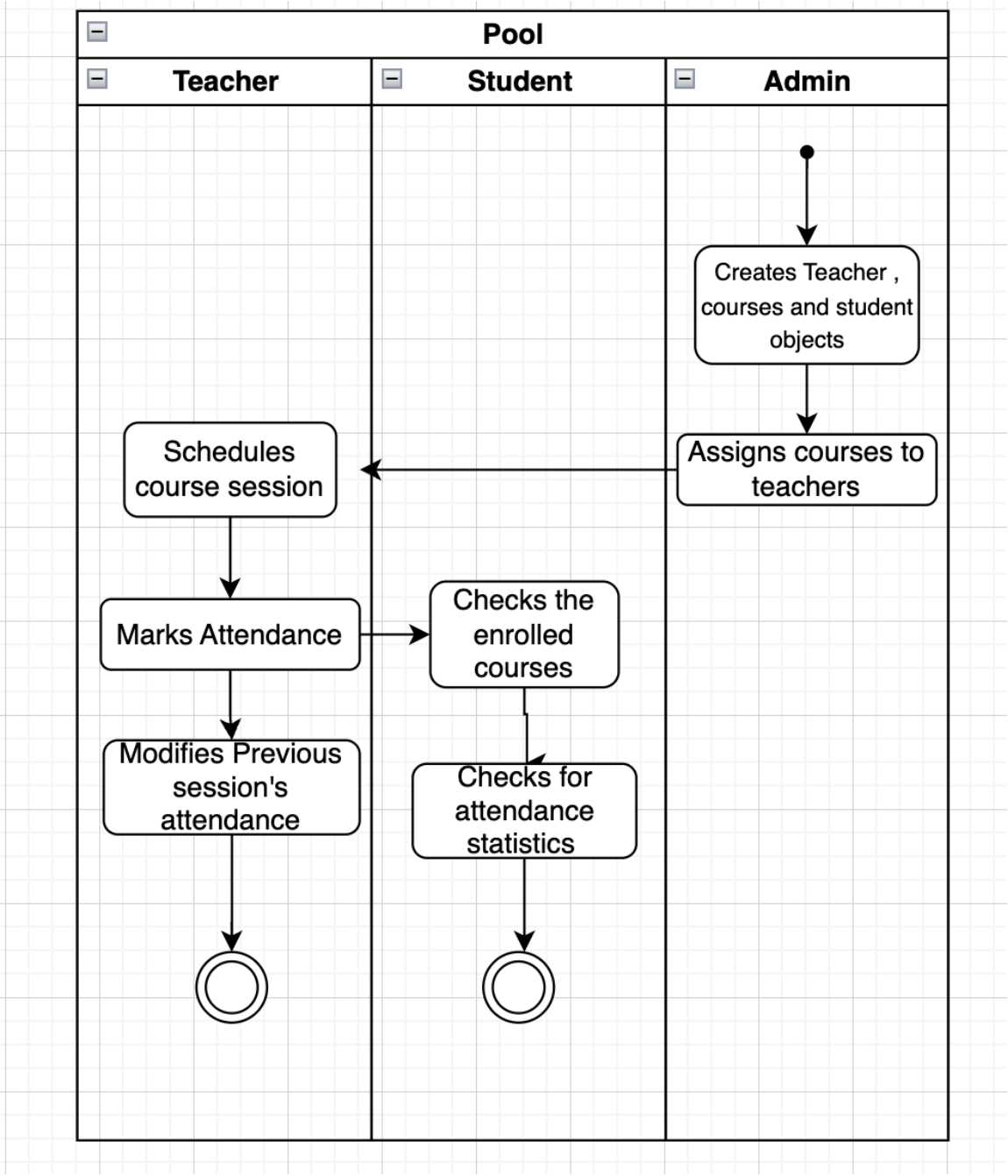


State Diagram:

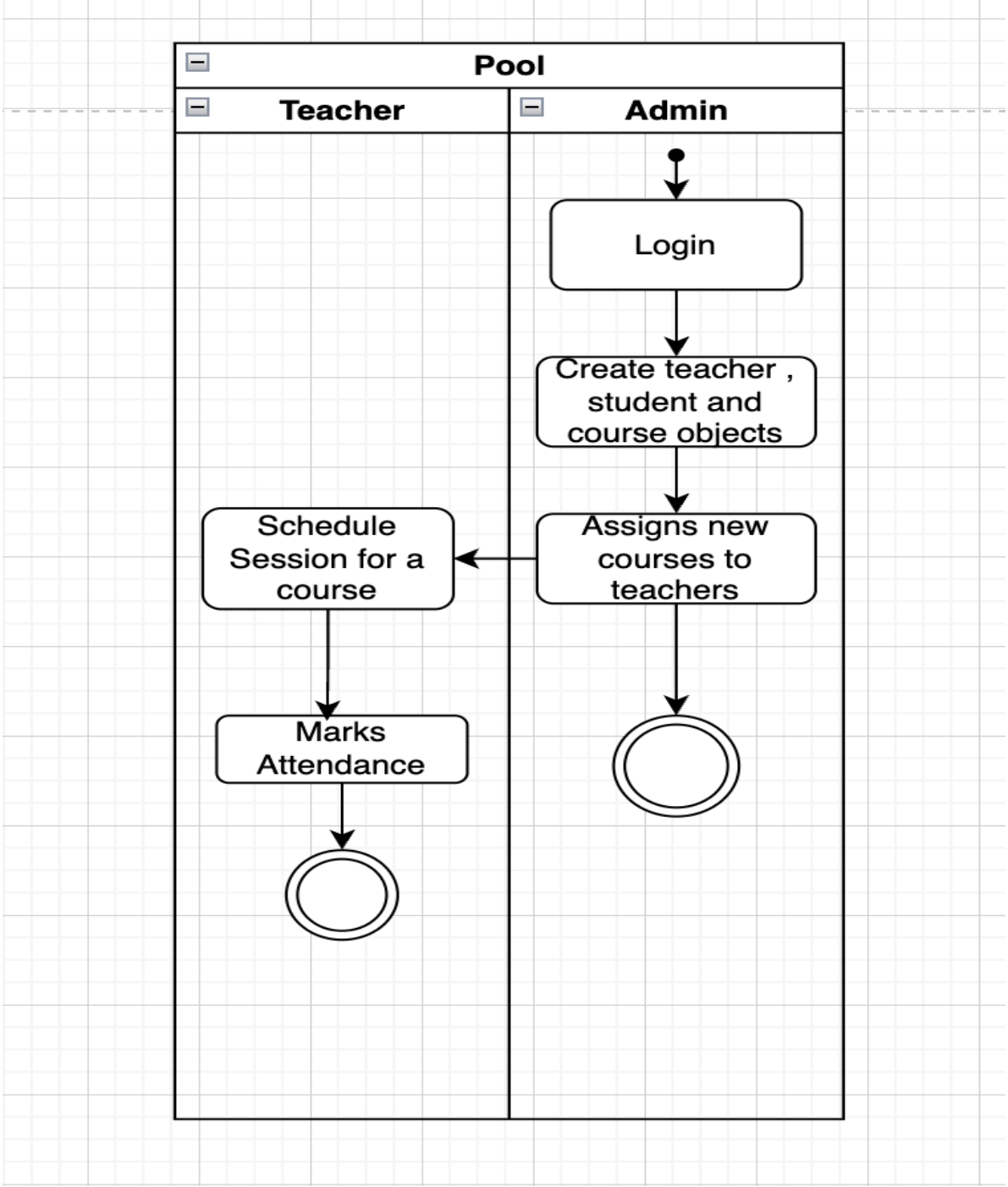


Activity Diagrams:

1. Major Use Case



Minor Activity Diagram :



Architecture Patterns, Design Principles, and Design Patterns:

Architecture Patterns

Model – View – Controller Pattern (MVC)

The Model-View-Controller (MVC) architecture pattern is widely used in software development to separate the concerns of an application into three interconnected components: the Model, View, and Controller. Each component has a specific role and responsibility within the system.

Components of MVC:

1. Model:
    - The Model represents the application's data and business logic. It encapsulates the data access, manipulation, and validation operations, independent of the user interface or presentation logic.
    - In our Attendance Manager System, the Model includes entities such as Student, Faculty, Subject, AttendanceRecord, and Course, along with the corresponding service classes responsible for performing CRUD operations, managing attendance data, validating rules (like minimum attendance requirements), and communicating with the MySQL database.

2. View :
    - The View represents the presentation layer, responsible for displaying data to users and capturing their interactions. It provides a user-friendly interface for both students and faculty to interact with the system's features.
    - In our system, the View is built using React, providing responsive and interactive web components such as login/signup forms, dashboards, attendance marking screens, and reports.. It communicates with the backend via JDBC template and reflects the data and operations managed by the Model.

3. Controller :

    - The Controller acts as the intermediary between the Model and the View. It handles incoming get requests from the frontend, processes user inputs, applies business logic, and invokes the necessary services in the Model layer to fetch or update data.
    - We have used a facade design pattern in auth controller , which is a clean approach and ensures that the session handling is set up.

Benefits of MVC:

- **Separation of Concerns**: MVC promotes a clear separation of concerns, allowing each component to focus on its specific responsibilities. This enhances code maintainability, reusability, and testability by minimizing dependencies and facilitating modular design.
- **Parallel Development**: The modular structure of MVC enables parallel development of different components by different teams or developers, promoting collaboration and accelerating the development process.
- **Flexibility and Scalability**: MVC facilitates flexibility and scalability by allowing modifications or extensions to one component without affecting the others. This enables the system to adapt to changing requirements and scaling up.

Design Principles

1) Open Closed SOLID Principle :

   The OCP is one of the five SOLID principles which states that classes, modules, microservices, and other code units should be open for extension but closed for modification. We should be able to extend the existing code using OOP features like inheritance via subclasses and interfaces. While adding a new feature extend the code rather than modifying it, so that the risk of failure is minimized.

   Importance of Open-Closed Principle(OCP):

- Flexibility: OCP enables greater flexibility in software design and architecture. Changes in requirements or business logic can be accommodated by adding new functionality through extension rather than modifying existing code.
- Reuse: By designing components to be open for extension, developers can create reusable building blocks that can be applied in different contexts without modification. This promotes code reuse and reduces duplication.
- Testability: OCP supports better testability by encouraging the creation of modular and loosely coupled components. Unit testing becomes easier as each component can be tested in isolation without relying on the implementation details of other components.

   Implementation :

   By using the Factory design pattern we are able to centralize the creation of the objects of the teacher and student class. Without modifying the creational logic , we are able to add new logic to create a new type of user if required.

2) Liskov Substitution

It states that objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program. In other words, a derived class must be substitutable for its base class without altering the desirable properties of the program. This means that every subclass or derived class should be substitutable for their base or parent class.

Importance of Liskov Substitution Principle (LSP):

- Maintainability: By adhering to the LSP, classes within a hierarchy become more interchangeable, making it easier to maintain and extend the system over time. Subclasses can be added or modified without affecting the existing behavior of the system.
- Flexibility: LSP promotes a flexible design where objects can be substituted seamlessly, allowing for easier reuse of code and components. This flexibility enables developers to build systems that are more modular, scalable, and extensible.
- Design Clarity: LSP encourages a clear and understandable class hierarchy by promoting the consistency of behavior across subclasses. This clarity enhances code readability and comprehensibility.

Implementation :

All three classes : admin , teacher and student implement the base user interface.
So whenever a User class is expected , these subclasses can easily be substituted in-place of the expected User parent class.

This can be done without altering the correctness or the semantics of the program.

3) Interface Segregation Principle

The Interface Segregation Principle (ISP) is one of the five SOLID principles of object-oriented design. It states that a client should not be forced to depend on interfaces it does not use. In other words, interfaces should be specific to the needs of the clients that use them, rather than being overly general and including methods that are not relevant to all clients.

Importance of Interface Segregation Principle (ISP):

- Reduced Dependency: By segregating interfaces based on client-specific functionalities, ISP reduces the dependency of clients on irrelevant methods. This promotes loose coupling between components and facilitates easier maintenance and evolution of the codebase.
- Improved Cohesion: ISP leads to interfaces that are more focused and cohesive, containing only the methods that are relevant to a specific client or group of clients. This improves the readability and understandability of the code, as well as making it easier to reason about and maintain.
- Enhanced Testability: Segregating interfaces based on client-specific requirements allows for more targeted and efficient testing. Each client can be tested independently, leading to more robust and reliable testing processes.

Implementation :

We have implemented the ISP principle to ensure user specific interfaces are tailored to the needs of the respective users like the admin , student and teacher, thus preventing the need of users to be dependent on the interfaces they don't use.

The StudentService class contains all the services that are needed by the student like get_attendance_stats, ge_enrolled_courses().

Similarly it has been implemented for the teacher as well.

4) Single Responsibility Principle

The Single Responsibility Principle (SRP) is one of the SOLID principles of object-oriented design, proposed by Robert C. Martin. SRP states that a class should have only one reason to change, meaning that it should have only one responsibility or job within the system.

Importance of SRP:

- Enhanced Maintainability: By adhering to SRP, classes become more focused and less complex. Each class is responsible for a single aspect of the system, making it easier to understand, modify, and maintain.
- Improved Reusability: Classes with well-defined responsibilities are more likely to be reusable in other parts of the system or in different projects.

Implementation :

Admin class : Specifically deals with the consistency of the database. Only the admin can add new students or new courses.

Teacher class : They are responsible only for marking attendance. They have not given any more responsibilities like adding new students into the database. We have the admin taking care of that.

Thus we ensure there is separation of concerns for easy debugging and maintain the modularity of the application.

Design Patterns

1) Facade Design Pattern to handle logins:

The Facade pattern is a structural design pattern that provides a simplified interface to a complex system of classes, hiding its internal implementation details. It promotes loose coupling between clients and subsystems, improving maintainability and ease of use.

In our Attendance Management System, we've implemented the Facade design pattern within the AuthController to streamline the login process for different user roles—admin, student, and teacher. The LoginFacade serves as a unified interface, abstracting the underlying authentication logic specific to each role and exposing a cleaner, centralized method for handling login requests. This approach not only simplifies the controller code but also promotes separation of concerns, making the system more modular and easier to maintain. By encapsulating the complexities of role-based authentication, the Facade pattern enhances both readability and scalability of the authentication module.

Implementation :

We have created a LoginFacade class , this class handles all the login logic for each user type : admin, students and teachers.

Based on the username , the facade class delegates which logic will be called upon , thus completely abstracting the login logic from the user.

Benefits of Facade:

- Simplified Interface: Clients interact with a single, easy-to-use interface, hiding the complexities of the underlying authentication subsystem.
- Encapsulation: The Facade encapsulates the login process, promoting information hiding and reducing dependency on internal subsystem components.
- Flexibility: Allows for changes in the authentication mechanism without affecting client code, as long as the Facade interface remains unchanged.

2. Singleton Design Pattern for Database connection

Java Singleton Pattern is one of the Gangs of Four Design patterns and comes in the Creational Design Pattern category. This design pattern ensures that only one instance of a class is being made. It gives global access to this instance to all those classes which need it.

The constructor of this class must be made private and a get_instance method must be made to ensure that global access is given. This must be implemented along with a private static instance of the same class within itself.

Implementation :

The Singleton design pattern is used in the DBConnection class to ensure that only one instance of the DBConnection object is created throughout the application's lifecycle.

The constructor is private : private DBConnection() to prevent direct instantiation of the class from outside.

A private static variable instance of type DBConnection holds the single instance of the class. This variable is only initialized once.

The getInstance() method is synchronized and checks whether the instance is null. If it is null, a new instance of DBConnection is created and assigned to instance. This ensures that the DBConnection instance is created only once, even when multiple threads try to access it via the synchronized keyword.

Benefits of the Singleton Design Pattern :

- Resource Conservation: By restricting the instantiation of the DBConnection class to a single instance, the Singleton pattern mitigates the risk of resource exhaustion associated with creating multiple database connections. It ensures that resources are utilized efficiently, optimizing the performance of the system.
- Connection Pooling: The Singleton instance of DBConnection encapsulates connection pooling logic, enabling the reuse of existing connections and minimizing the responsiveness of the system, especially in scenarios with a high volume of database operations.

3. Factory Design Pattern to create teacher and student objects :

The factory design pattern cleanly abstracts the creation of various objects. It is a creational design pattern that provides an interface to create objects in the super class. It works best with simple object creation , i.e. objects with lesser parameters.

The builder design pattern was chose to simplify and centralize the creation of teacher and student class.Instead of instantiating the objects throughout the code , a call is given to the UserFactory object along with the respective parameters.If needed we can also add a new user type without modifying the whole codebase as this supports the open/closed principle as well.

Implementation :

In our Attendance Management System, the Factory design pattern is implemented through the UserFactory class to streamline the creation of different user types such as students, teachers, and admins. The createUser method acts as a centralized creator that dynamically instantiates the appropriate subclass based on the userType input. Using a switch expression, it handles parameter validation and ensures type-safe object creation.

Benefits of the Factory Design pattern :

- Factory design pattern provides approach to code for interface rather than implementation.
- Factory pattern removes the instantiation of actual implementation classes from client code. Factory pattern makes our code more robust, less coupled and easy to extend. For example, we can easily change PC class implementation because the client program is unaware of this.
- Factory pattern provides abstraction between implementation and client classes through inheritance.

4. Strategy Pattern to handle row mapping :

Strategy design pattern is one of the behavioral design patterns. Strategy pattern is used when we have multiple algorithms for a specific task and the client decides the actual implementation to be used at runtime.
Strategy pattern is also known as Policy Pattern. We define multiple algorithms and let client application pass the algorithm to be used as a parameter.

Implementation :

The Strategy Pattern is implicitly implemented through the use of the RowMapper interface, which defines a family of strategies for mapping rows from a database ResultSet to Java objects.

The CourseMapper class implements the RowMapper interface, providing a specific strategy to map a row from the Courses table into a Course object. This allows the mapping logic to be encapsulated in the CourseMapper, making it easily interchangeable.

This has also been implemented for the teacher row to teacher object and similarly for student row to student objects.

Benefits of the Strategy Pattern :

- The Strategy pattern encapsulates different algorithms or behaviors within separate classes. Each strategy class focuses on a specific algorithm, making it easier to understand, maintain, and modify the behavior independently.
- By using the Strategy pattern, you can dynamically select and switch between different algorithms at runtime.This flexibility allows you to adapt the behavior of an object or system based on changing requirements or environmental factors.
- The Strategy pattern promotes code reuse by encapsulating algorithms in separate strategy classes. Once a strategy is defined, it can be used across multiple contexts or objects without duplicating code.

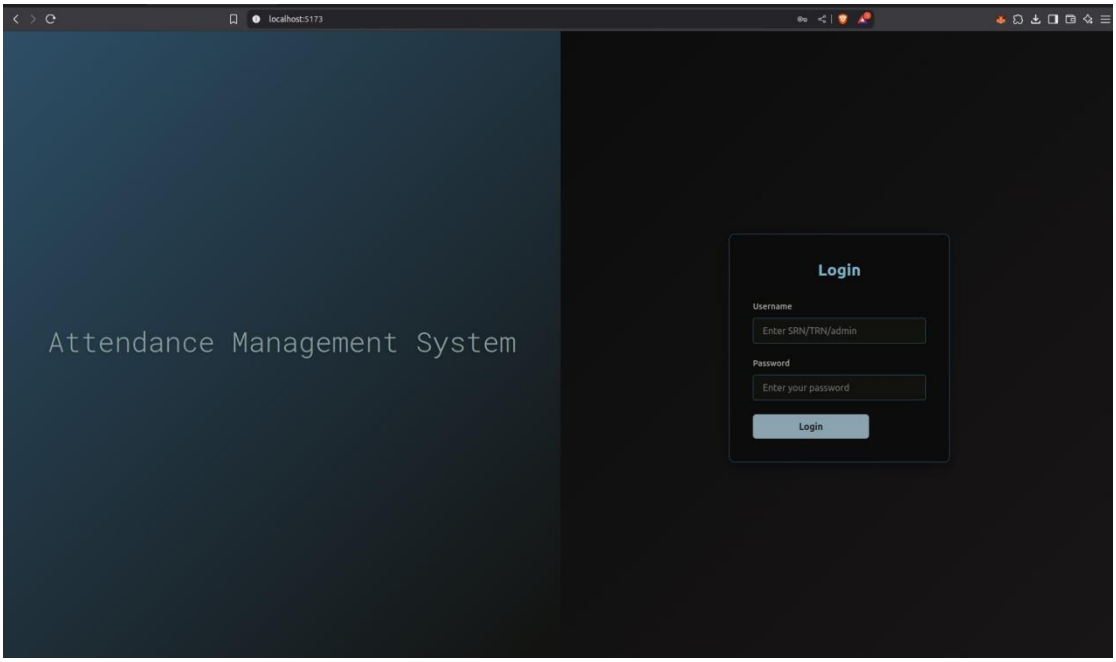
Github link to the Codebase:

https://github.com/kdb04/attendance-mgmt

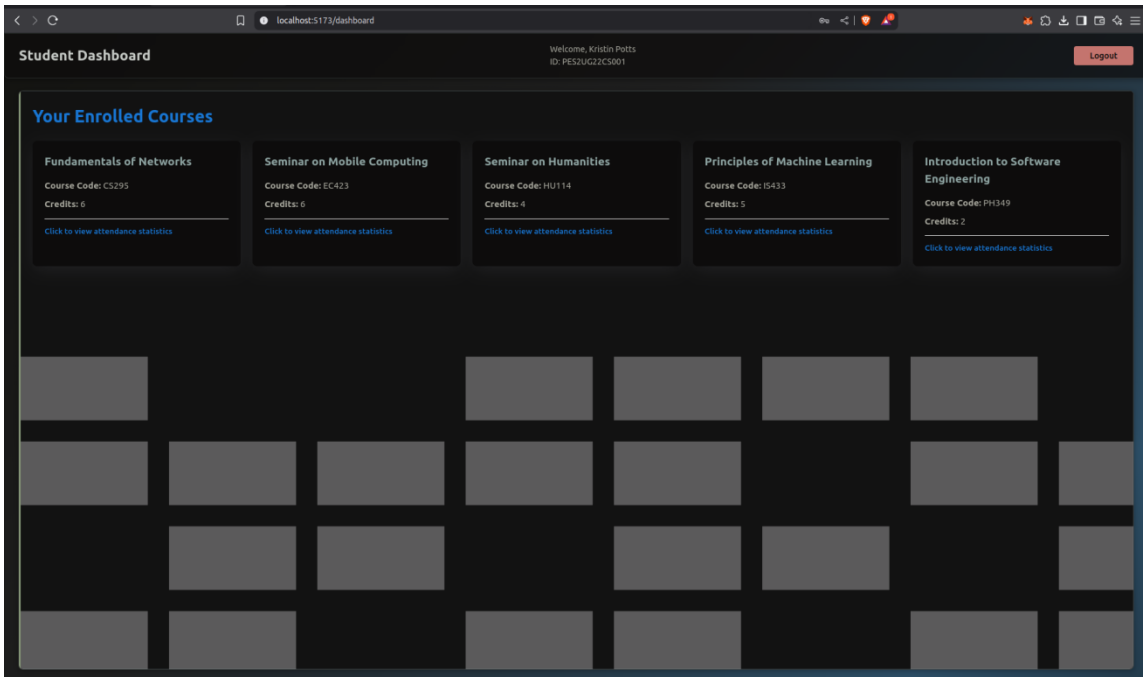
Screenshots

UI:

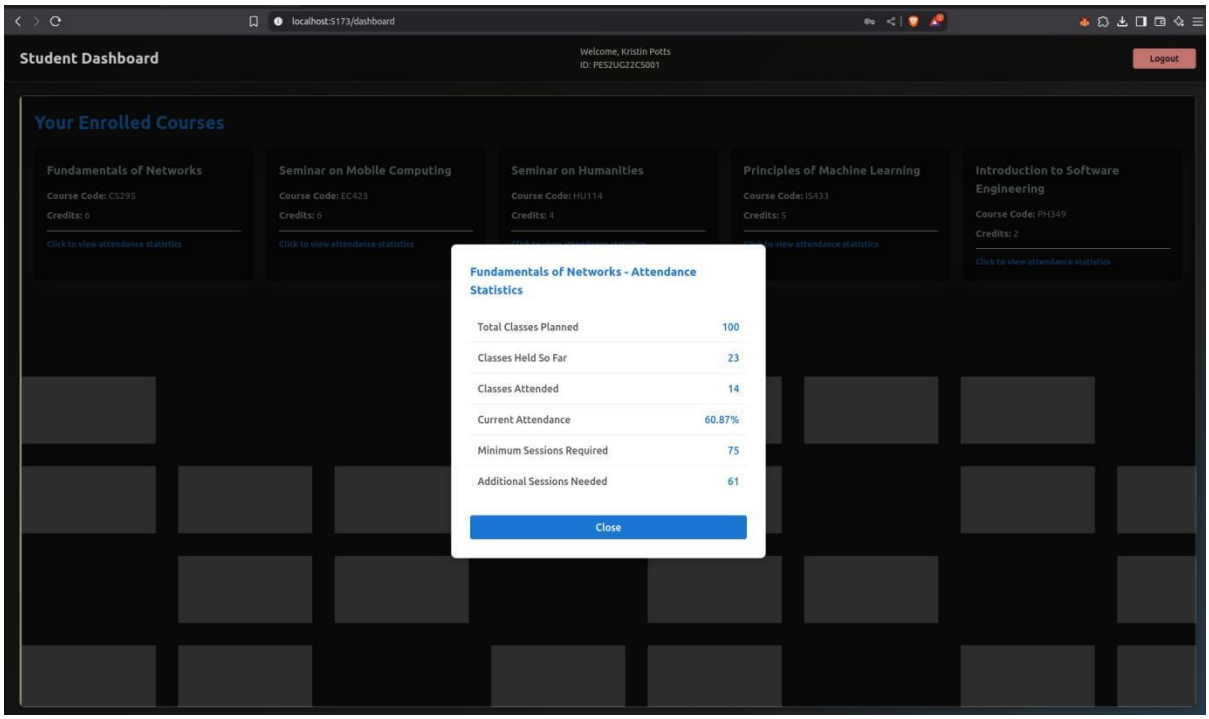
Login page :



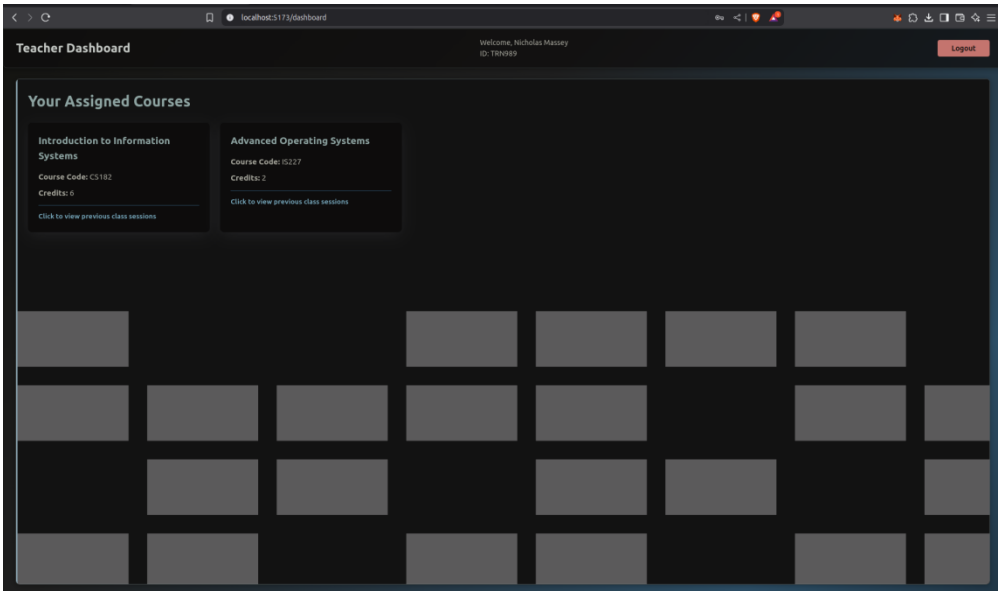
Student Dashboard with Current Enrolled Courses



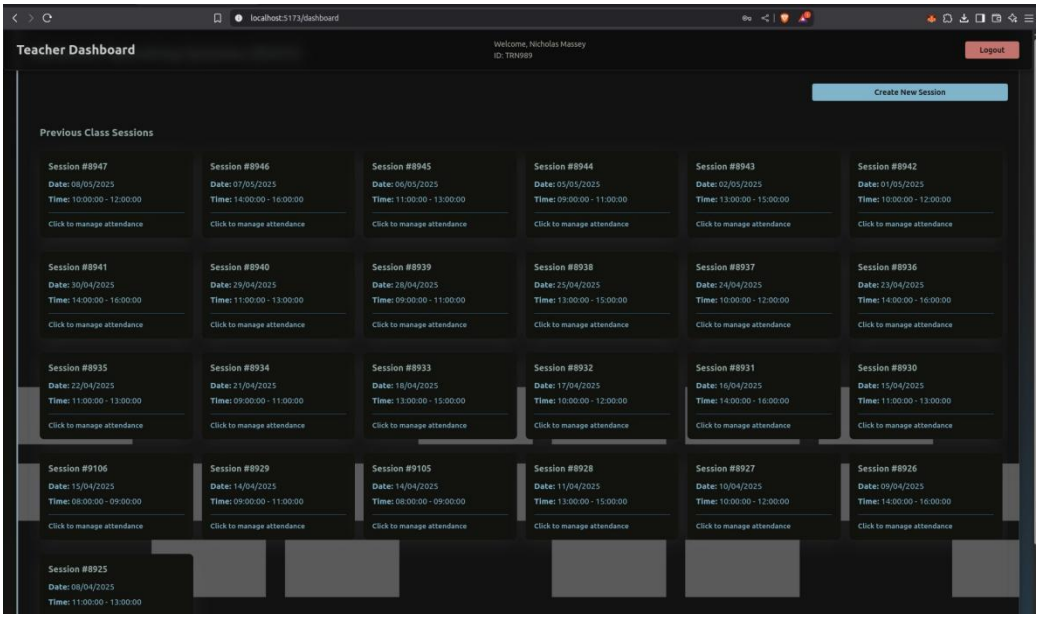
Attendance Statistics for a particular course



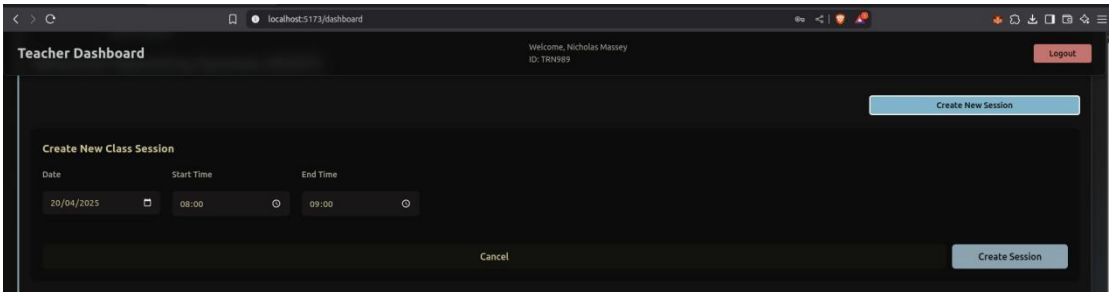
Teacher Dashboard



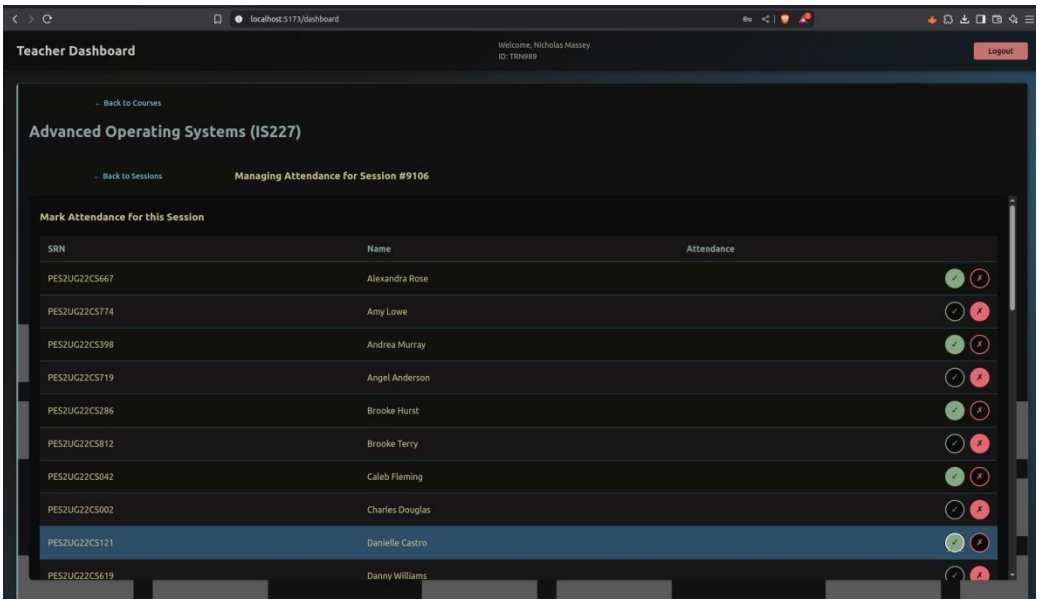
Sessions completed by a teacher for a particular course



Create new session

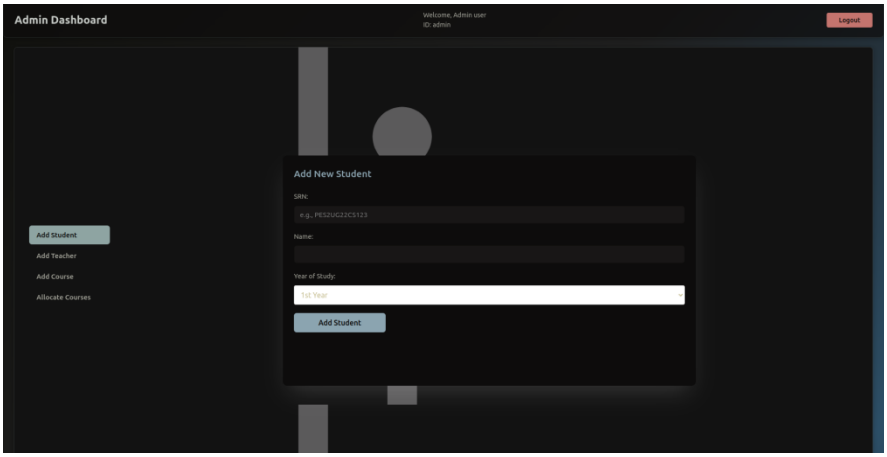


marking attendance for previous session

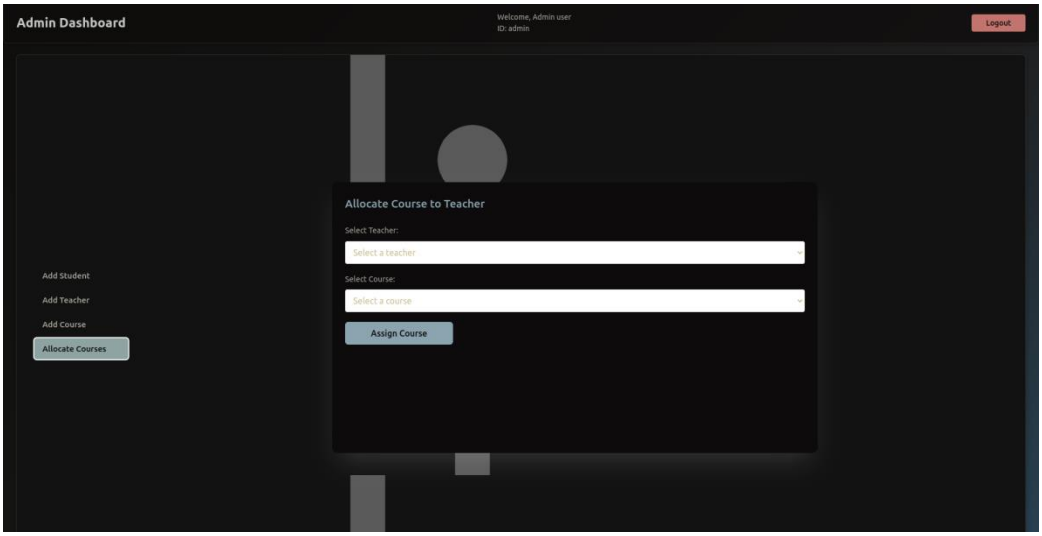


31133	9106	PES2UG22CS014	absent	2025-04-15 10:35:42
31134	9106	PES2UG22CS020	absent	2025-04-15 10:35:42
31135	9106	PES2UG22CS042	present	2025-04-20 20:08:20
31136	9106	PES2UG22CS044	absent	2025-04-15 10:35:42
31137	9106	PES2UG22CS083	absent	2025-04-15 10:35:42
31138	9106	PES2UG22CS101	absent	2025-04-15 10:35:42
31139	9106	PES2UG22CS103	absent	2025-04-15 10:35:42
31140	9106	PES2UG22CS121	present	2025-04-20 20:08:21
31141	9106	PES2UG22CS134	absent	2025-04-15 10:35:42
31142	9106	PES2UG22CS159	absent	2025-04-15 10:35:42
31143	9106	PES2UG22CS204	absent	2025-04-15 10:35:42
31144	9106	PES2UG22CS219	absent	2025-04-15 10:35:42
31145	9106	PES2UG22CS246	absent	2025-04-15 10:35:42
31146	9106	PES2UG22CS286	present	2025-04-20 20:08:19
31147	9106	PES2UG22CS351	absent	2025-04-15 10:35:42
31148	9106	PES2UG22CS368	absent	2025-04-15 10:35:42
31149	9106	PES2UG22CS376	absent	2025-04-15 10:35:42
31150	9106	PES2UG22CS398	present	2025-04-20 20:08:02
31151	9106	PES2UG22CS470	absent	2025-04-15 10:35:42
31152	9106	PES2UG22CS521	absent	2025-04-15 10:35:42
31153	9106	PES2UG22CS609	absent	2025-04-15 10:35:42
31154	9106	PES2UG22CS619	absent	2025-04-15 10:35:42
31155	9106	PES2UG22CS667	present	2025-04-20 20:08:00
31156	9106	PES2UG22CS690	absent	2025-04-15 10:35:42
31157	9106	PES2UG22CS719	absent	2025-04-15 10:35:42
31158	9106	PES2UG22CS730	absent	2025-04-15 10:35:42
31159	9106	PES2UG22CS757	absent	2025-04-15 10:35:42
31160	9106	PES2UG22CS768	absent	2025-04-15 10:35:42
31161	9106	PES2UG22CS774	absent	2025-04-15 10:35:42
31162	9106	PES2UG22CS812	absent	2025-04-15 10:35:42
31163	9106	PES2UG22CS820	absent	2025-04-15 10:35:42
31164	9106	PES2UG22CS832	absent	2025-04-15 10:35:42
31165	9106	PES2UG22CS859	absent	2025-04-15 10:35:42
31166	9106	PES2UG22CS895	absent	2025-04-15 10:35:42
31167	9106	PES2UG22CS922	absent	2025-04-15 10:35:42
31168	9106	PES2UG22CS924	absent	2025-04-15 10:35:42
31169	9106	PES2UG22CS961	present	2025-04-18 12:49:58

Admin Dashboard



Allocate Course



Individual Contributions:

Motamarri Sai Sathvik : PES2UG22CS321	***Worked on displaying attendance statistics for the enrolled courses and updating attendance for previous sessions.***
Narayan Sreekumar : PES2UG22CS339	***Worked on creating new teacher , studemt and course objects and displaying previous sessions a course.***
Nikhil Srivatsa : PES2UG22CS357	***Worked on assigning teachers to the newly created courses and worked on implementing authentication for different users***
Nitheesh Pugazhanthi : PES2UG22CS371	***Worked on scheduling a new session for a course and marking attendance and worked on displaying the enrolled courses.***