

Introduction:

Our University Management System is designed to serve as a comprehensive solution for managing various aspects of academic administration. The system encompasses a range of functionalities to streamline student, course, and enrollment processes, enhancing the overall efficiency of university operations. The primary functionalities include:

Functionalities:

Student Management:

- Create Student: Facilitates the seamless addition of new students to the system by capturing essential details such as name and student type. This empowers the university to efficiently expand its student base.
- Get Students: Provides a quick and comprehensive overview of all registered students, aiding academic advisors and administrators in their roles.
- Update Student: Allows for the modification of student information, ensuring that the database reflects the most accurate and up-to-date student profiles.
- Delete Student: Permits the removal of a student from the system, ensuring data integrity and reflecting changes in student status.

Course Management:

- Create Course: Streamlines the creation of new courses by specifying a course name, contributing to the diversification and expansion of the academic curriculum.
- Register Student in Course: Simplifies the process of associating students with specific courses, facilitating efficient enrollment management.
- Get Courses: Offers a comprehensive list of all courses offered by the university, supporting academic planning and curriculum assessment.

Enrollment Management:

- Get Enrollments: Provides an insightful overview of all enrollments, detailing the association between students and courses. This feature supports strategic decision-making in academic planning.
- Get Students for Course ID: Retrieves a specific list of students enrolled in a particular course, offering targeted information for effective classroom management and academic analysis.

Log Management:

- Get Logs: Presents a detailed log of system activities, offering transparency and aiding in system monitoring and diagnostics. This feature is crucial for maintaining accountability and system integrity.

Design Patterns Developed:

Factory Pattern:

- **Application:** Employed to standardize the instantiation process of student and course objects, enhancing maintainability and reducing code duplication.
- **Purpose:** Ensures a consistent and organized approach to creating complex objects within the system, fostering a clear and scalable object creation mechanism.

Singleton Pattern:

- **Application:** Utilized to guarantee a single instance of DB transaction classes, optimizing resource utilization.
- **Purpose:** Ensures that crucial classes related to database transactions exist as a singular entity throughout the application, preventing unnecessary duplication and promoting a unified point of access.

Repository Pattern:

- **Application:** Applied to segregate data access and storage operations, fostering a structured approach to data management.
- **Purpose:** Separates concerns related to data operations from the business logic, promoting modularity and easing future changes to the data layer without affecting overall system functionality.

Observer Pattern:

- **Potential Application:** Could be employed for efficient logging and updates across different components in response to system changes.
- **Purpose:** Establishes a mechanism for notifying and updating multiple components, enhancing system flexibility and responsiveness to changes.

Decorator Pattern:

- **Potential Application:** Offers flexibility in enhancing the behavior of student classes dynamically.
- **Purpose:** Provides a means to augment the behavior of individual classes without modifying their core structure, promoting adaptability and extensibility in the system.

Service Pattern:

- **Application:** Encapsulates business logic in service classes (StudentService, CourseService, EnrollmentService, and LogService), fostering modular and maintainable code architecture.
- **Purpose:** Modularizes and organizes business logic into dedicated service classes, enhancing code organization, readability, and maintainability. Each service class is responsible for specific operations, promoting a clear separation of concerns.

Explanation of Models *models.py*:

Student:

Attributes:

- `name`: Represents the name of the student.
- `student_id`: Serves as a unique identifier for each student.
- `student_type`: Denotes the type or category of the student (e.g., undergraduate, graduate).

Importance: The Student model encapsulates essential information about a student, providing a structured representation of individual students within the university system. It facilitates the organization and retrieval of student data.

Class:

Attributes:

- `class_id`: Acts as a unique identifier for each class.
- `class_name`: Represents the name or designation of the class.

Importance: The Class model defines characteristics of classes within the system. It allows for the identification and management of different classes, contributing to the overall organization and structure of the academic environment.

Course:

Attributes:

- `course_id`: Serves as a unique identifier for each course.
- `course_name`: Represents the name or title of the course.

Importance: The Course model encapsulates information about individual courses offered by the university. It facilitates the tracking and management of courses, supporting academic planning and curriculum development.

Enrollment:

Attributes:

- `data`: Represents enrollment data, which could include information about students enrolled in specific courses or classes.

Methods:

`to_dict()`: Converts the enrollment data into a dictionary format for easy serialization and representation.

Importance: The Enrollment model serves as a container for enrollment data, allowing for the representation of relationships between students, classes, and courses. It facilitates the efficient retrieval and presentation of enrollment information within the system.

Importance of Using Models:

Structural Organization:

Models provide a structural organization for data entities within the system, enhancing code clarity and maintainability. Each model encapsulates specific attributes related to a particular concept (e.g., student, class), promoting a clear representation of the system's entities.

Data Integrity:

By defining models, the system can enforce data integrity and consistency. Models serve as blueprints for creating instances of entities, ensuring that specific attributes are present and properly structured.

Readability and Documentation:

Models serve as a form of documentation, making it easier for developers to understand the structure and relationships between different components within the system. They contribute to code readability and act as a reference for future development.

Abstraction and Encapsulation:

Models encapsulate the essential attributes and behaviors of system entities, providing a level of abstraction. This abstraction allows developers to interact with entities at a higher level, promoting a separation of concerns and facilitating modular development.

Ease of Maintenance:

Models contribute to the ease of system maintenance. Changes to the structure or attributes of an entity can be made within the corresponding model, affecting the entire system consistently. This modular approach simplifies updates and modifications.

Explanation of Factories:

FactoryInterface:

Attributes/Methods:

*create(*args, **kwargs)*: Abstract method defining the creation of objects.

get_last_id_from_service(): Static abstract method to retrieve the last ID from the corresponding service.

Design Pattern: Factory Pattern

Role: Defines a common interface for factories, enforcing the creation of objects and retrieval of the last ID.

StudentFactory:

Attributes/Methods:

create(name, student_type, id=None): Creates a decorated student using a decorator and returns the decorated student object.

get_last_id_from_service(): Retrieves the last student ID from the StudentService.

Design Pattern: Factory Pattern

Role: Creates instances of the Student model, incorporating a decorator pattern (StudentDecorator) to enhance student objects with additional functionalities. Retrieves the last student ID from the service.

CourseFactory:

Attributes/Methods:

create(course_name, course_id=None): Creates a course object, generating a new course ID if not provided.

get_last_id_from_service(): Retrieves the last course ID from the CourseService.

Design Pattern: Factory Pattern

Role: Creates instances of the Course model, generating a new course ID if not provided. Retrieves the last course ID from the service.

EnrollmentFactory:

Attributes/Methods:

create(data): Creates an enrollment object using provided data.

update_enrollment(enrollment, student_id, course_id): Updates the enrollment data with the association of a student to a course.

Design Pattern: Factory Pattern

Role: Creates instances of the Enrollment model. The update_enrollment method allows the dynamic updating of enrollment data, associating students with specific courses.

Importance of Factories:

Encapsulation and Abstraction:

Factories encapsulate the instantiation process of objects, providing a higher level of abstraction. They shield the client code from the complexities of object creation and allow for a unified interface.

Consistent Object Creation:

Factories ensure consistent and standardized object creation. The creation methods within each factory follow a predefined pattern, promoting uniformity in the instantiation process.

Flexibility and Decorator Pattern:

The use of factories, especially in the StudentFactory, allows for the incorporation of the decorator pattern (StudentDecorator). This enhances the behavior of student objects dynamically without modifying their core structure, promoting flexibility in the application.

Service Interaction:

Factories interact with corresponding service classes (StudentService, CourseService) to retrieve the last IDs. This interaction ensures that the application maintains a coherent and up-to-date understanding of the existing data.

Code Modularity:

Factories contribute to code modularity by encapsulating the creation logic within dedicated classes. This separation of concerns enhances code organization and makes it easier to manage and maintain.

Explanation of Decorators:

StudentDecorator:

Methods:

- `create_decorated_student(name, student_type, student_id=None)`: Creates a decorated student object with a name, student type, and an optional student ID.
- `get_last_student_id_from_service()`: Retrieves the last student ID from the StudentService.

Design Pattern: Decorator Pattern

Role: Enhances the behavior of student objects dynamically by adding functionalities during instantiation.

Importance:

Dynamic Object Enhancement: The `create_decorated_student` method allows the dynamic creation of students with additional functionalities. This aligns with the decorator pattern, where functionalities can be added to objects without altering their core structure.

Last ID Retrieval: The `get_last_student_id_from_service` method retrieves the last student ID from the StudentService, ensuring that new student IDs are generated sequentially and avoid conflicts.

Error Handling: The decorator handles exceptions that may occur during ID retrieval, providing robustness to the system.

Importance of Decorators:

Dynamic Functionality Enhancement:

The decorator pattern allows the dynamic enhancement of student objects with additional functionalities. This is crucial for introducing features or behaviors without modifying the base Student class, promoting flexibility and extensibility.

Last ID Retrieval:

The decorator ensures that each new student is assigned a unique and sequentially generated ID. By retrieving the last student ID from the StudentService, it helps in avoiding ID conflicts and maintaining the integrity of the student ID sequence.

Error Handling:

The decorator includes error handling mechanisms, such as exception catching during the retrieval of the last student ID. This ensures that the system remains robust and can gracefully handle service errors or other exceptions, preventing potential disruptions.

Encapsulation of ID Generation Logic:

The decorator encapsulates the logic for generating student IDs within the StudentDecorator class. This encapsulation promotes a cleaner separation of concerns and ensures that the ID generation process is centralized and easily maintainable.

Service Interaction:

The decorator interacts with the StudentService to obtain the last student ID. This interaction ensures that the application is aware of the existing student data, contributing to a coherent understanding of the system state.

Maintainable and Readable Code:

By encapsulating ID generation and student creation logic within the decorator, the code becomes more modular and maintainable. The decorator pattern promotes a clear structure and enhances the readability of the codebase.

Explanation of Services:

StudentService:

Methods:

`create_student(student)`: Adds a student to the database.

`get_students()`: Retrieves the list of students from the database.

`update_student(student)`: Updates the information of an existing student in the database.

`delete_student(student_id)`: Deletes a student from the database.

Design Pattern: Service Pattern

Role: Encapsulates business logic related to students, interacts with the database for CRUD operations.

Importance: The StudentService provides a high-level interface for managing student-related operations. It encapsulates the logic for interacting with the database, promoting a separation of concerns and modularizing the code.

ClassService:

Methods:

`create_class(_class)`: Adds a class to the database.

`get_classes()`: Retrieves the list of classes from the database.

Similar methods for update and delete.

Design Pattern: Service Pattern

Role: Encapsulates business logic related to classes, interacts with the database for CRUD operations.

Importance: Similar to StudentService, the ClassService manages class-related operations and interacts with the database. It adheres to the service pattern, promoting modular and maintainable code.

CourseService:

Methods:

`create_course(course)`: Adds a course to the database.

`get_courses()`: Retrieves the list of courses from the database.

Similar methods for update and delete.

Design Pattern: Service Pattern

Role: Encapsulates business logic related to courses, interacts with the database for CRUD operations.

Importance: The CourseService provides a centralized interface for managing course-related operations. It follows the service pattern, ensuring a modular and organized approach to code architecture.

EnrollmentService:

Methods:

register_student_in_course(student_id, course_id): Updates enrollment data to register a student in a course.

get_enrollment(): Retrieves all enrollment data from the database.

get_enrollment_for_course(courseid): Retrieves enrollment data for a specific course.

Design Pattern: Service Pattern

Role: Manages the registration of students in courses, interacts with the database for enrollment data.

Importance: The EnrollmentService encapsulates the logic for updating and retrieving enrollment data. It plays a crucial role in managing student-course associations and adheres to the service pattern.

LogService:

Methods:

get_logs(): Retrieves log data from the database.

Design Pattern: Service Pattern

Role: Manages the retrieval of log data, interacts with the database.

Importance: The LogService provides a centralized interface for retrieving log data. It adheres to the service pattern, ensuring a consistent and organized approach to log data management.

Explanation of Services:

StudentService:

Methods:

create_student(student): Adds a student to the database.

get_students(): Retrieves the list of students from the database.

update_student(student): Updates the information of an existing student in the database.

delete_student(student_id): Deletes a student from the database.

Design Pattern: Service Pattern

Role: Encapsulates business logic related to students, interacts with the database for CRUD operations.

Importance: The StudentService provides a high-level interface for managing student-related operations. It encapsulates the logic for interacting with the database, promoting a separation of concerns and modularizing the code.

ClassService:

Methods:

create_class(_class): Adds a class to the database.

get_classes(): Retrieves the list of classes from the database.

Similar methods for update and delete.

Design Pattern: Service Pattern

Role: Encapsulates business logic related to classes, interacts with the database for CRUD operations.

Importance: Similar to StudentService, the ClassService manages class-related operations and interacts with the database. It adheres to the service pattern, promoting modular and maintainable code.

CourseService:

Methods:

create_course(course): Adds a course to the database.

get_courses(): Retrieves the list of courses from the database.

Similar methods for update and delete.

Design Pattern: Service Pattern

Role: Encapsulates business logic related to courses, interacts with the database for CRUD operations.

Importance: The CourseService provides a centralized interface for managing course-related operations. It follows the service pattern, ensuring a modular and organized approach to code architecture.

EnrollmentService:

Methods:

register_student_in_course(student_id, course_id): Updates enrollment data to register a student in a course.

get_enrollment(): Retrieves all enrollment data from the database.

get_enrollment_for_course(courseid): Retrieves enrollment data for a specific course.

Design Pattern: Service Pattern

Role: Manages the registration of students in courses, interacts with the database for enrollment data.

Importance: The EnrollmentService encapsulates the logic for updating and retrieving enrollment data. It plays a crucial role in managing student-course associations and adheres to the service pattern.

LogService:

Methods:

`get_logs()`: Retrieves log data from the database.

Design Pattern: Service Pattern

Role: Manages the retrieval of log data, interacts with the database.

Importance: The LogService provides a centralized interface for retrieving log data. It adheres to the service pattern, ensuring a consistent and organized approach to log data management.

Importance of Services:

Encapsulation of Business Logic:

Each service class encapsulates business logic related to its corresponding entity (students, classes, courses, enrollment, logs). This encapsulation promotes a modular and organized code structure, enhancing maintainability.

Separation of Concerns:

The services separate the concerns of business logic and data access. This separation adheres to the service pattern, making the codebase more modular and readable.

Consistent Interface:

The services provide a consistent interface for performing CRUD operations on different entities. This consistency simplifies the overall architecture and promotes a unified approach to data management.

Error Handling:

Services include error handling mechanisms (try-except blocks) to handle exceptions that may occur during database interactions. This ensures the robustness of the system by gracefully handling potential errors.

Explanation of Repository:

Singleton:

Design Pattern: Singleton Pattern

Role: Ensures that there is only one instance of the JsonDatabase class, promoting resource efficiency and centralizing database access.

JsonDatabase:

Design Patterns:

Repository Pattern

Observer Pattern (Implemented by DatabaseObserver)

Role: Manages the data storage and retrieval operations for various entities (students, classes, courses, enrollment) in the University Management System. Observes changes in the database and logs transactions using the DatabaseObserver.

Importance:

Repository Pattern: Encapsulates the logic for data access, abstracting the details of data storage (JSON files) from the rest of the application. Each method in JsonDatabase corresponds to CRUD operations for different entities, promoting a modular and organized approach.

Observer Pattern: The JsonDatabase class observes changes using the DatabaseObserver. When data operations (e.g., adding a student) occur, the observer logs these operations in a log file (log.json), providing a record of database transactions.

Repository Pattern:

Abstraction of Data Access:

The JsonDatabase class abstracts away the details of data storage, providing a uniform interface for accessing data related to students, classes, courses, enrollment, and logs.

Modularity:

Each method in JsonDatabase corresponds to a specific entity, promoting a modular and maintainable code structure.

Centralized Data Operations:

Data operations for different entities are centralized within the JsonDatabase class, ensuring a single source of truth for data management.

Observer Pattern:

Logging Database Transactions:

The DatabaseObserver is notified whenever a method in JsonDatabase is called. It logs the method name and data in the log.json file, providing a history of database transactions.

Decoupling Logging Logic:

The observer pattern decouples the logging logic from the data access logic. This separation of concerns ensures that changes to the logging mechanism do not impact the core data access functionality.

Importance of Design Patterns:

Singleton Pattern:

Ensures that there is only one instance of the JsonDatabase class, preventing unnecessary resource consumption and maintaining a single point of control for database access.

Repository Pattern:

Encapsulates data access logic, promoting a modular and organized approach. It abstracts away the details of data storage, making the system more maintainable and extensible.

Observer Pattern:

Facilitates logging of database transactions without tightly coupling the logging logic with the data access logic. Enhances flexibility and maintainability.

JsonDatabase Methods:

Data Retrieval:

get_students, get_classes, get_courses, get_enrollment: Retrieve data for students, classes, courses, and enrollment.

Data Addition:

add_student, add_class, add_course: Add new data for students, classes, and courses.

Data Update:

update_student: Update existing student data.

Data Deletion:

delete_student: Delete a student record.

Enrollment Operations:

get_enrollment, update_enrollment: Retrieve and update enrollment data.

Log Retrieval:

get_logs: Retrieve log data.

Note: The JsonDatabase class dynamically intercepts method calls using the `__getattr__` method to implement the observer pattern. It notifies the observer (DatabaseObserver) about data operations, and the observer logs these operations.

React.js Frontend for University Management System:

The frontend of the University Management System is developed using React.js, a popular JavaScript library for building user interfaces. The React.js frontend provides a graphical user interface (GUI) for users to interact with the various APIs exposed by the Django backend. The design focuses on creating a user-friendly and intuitive experience for managing students, classes, courses, enrollments, and logs.

Key Features:

Student Management:

Create dedicated views for managing students, allowing users to add, update, and delete student records. Include forms for entering student details, and provide validation to ensure data integrity.

Class and Course Management:

Implement views for managing classes and courses. Users can add new classes, update existing ones, and delete obsolete entries. Course management includes features for adding courses, updating details, and managing course enrollments.

Enrollment Management:

Design a user-friendly interface for managing student enrollments in different courses. Users can register or unregister students from courses, and the frontend reflects these changes in real-time.

Logging and History:

Display a log history section that shows a record of transactions and system activities. Users can review logs to track changes, identify issues, and maintain a comprehensive history of system operations.

Error Handling and Notifications:

Implement robust error handling to gracefully manage unexpected scenarios. Display notifications to users for successful operations, warnings, or errors, ensuring a smooth user experience.

Responsive Design:

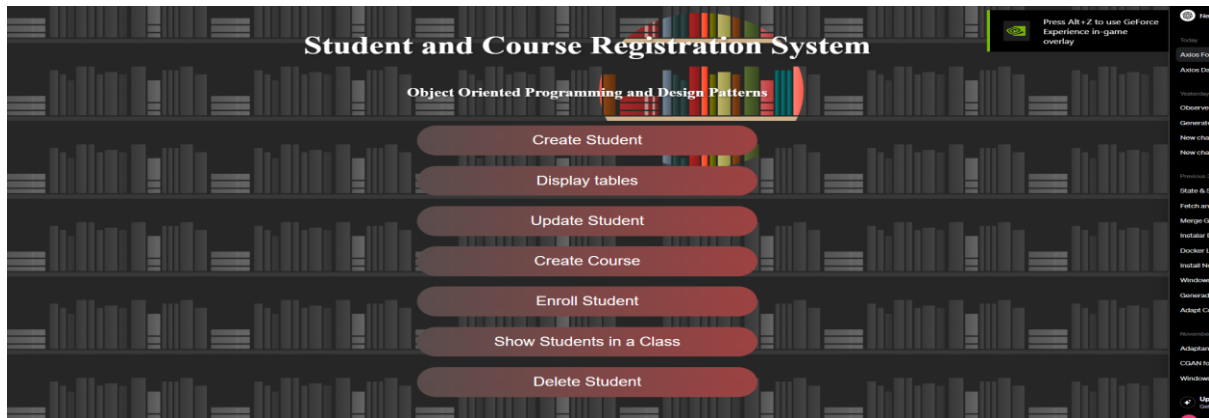
Ensure the frontend is responsive and accessible on various devices, including desktops, tablets, and mobile phones. Responsive design enhances usability and accommodates users with different devices.

Integration with Django Backend:

The React.js frontend integrates seamlessly with the Django backend, communicating with the various APIs to perform CRUD operations. The frontend sends HTTP requests to the Django server to retrieve data, create new records, update existing information, and manage enrollments. The modular and organized structure of the React.js components aligns with the backend API endpoints, facilitating smooth communication.

Screenshots

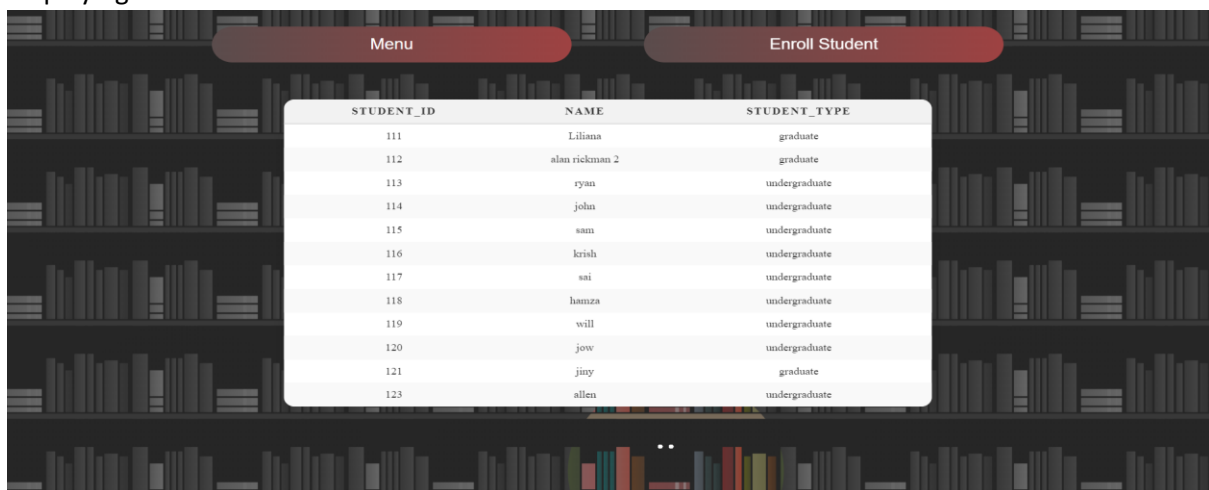
Home Screen to navigate to all the functionalities of the project:



Display Tables: to show all the existing tables and data



Displaying Student Table:



Displaying Course Table:

Menu

Enroll Student

COURSE ID	COURSE NAME
111	AI
119	AI DS
120	AI ML
123	DS
125	OOPs

Displaying Logs Table:

Menu

Enroll Student

TIME	OPERATION PERFORMED
2023-12-04 22:55:30.264191	get_enrollment
2023-12-04 22:55:30.266184	update_enrollment
2023-12-04 22:55:55.779630	get_enrollment
2023-12-05 03:09:17.293000	get_students
2023-12-05 03:09:22.548728	get_students
2023-12-05 03:09:22.556728	get_students
2023-12-05 03:09:22.563727	add_student
2023-12-05 03:15:16.193868	get_students
2023-12-05 03:20:06.270057	get_students
2023-12-05 03:22:43.193864	get_students
2023-12-05 03:23:28.749146	get_students
2023-12-05 03:23:54.800421	get_students
2023-12-05 03:24:20.801290	get_students
2023-12-05 03:24:32.873661	get_students
2023-12-05 03:25:31.695418	get_students
2023-12-05 03:25:54.947848	get_students

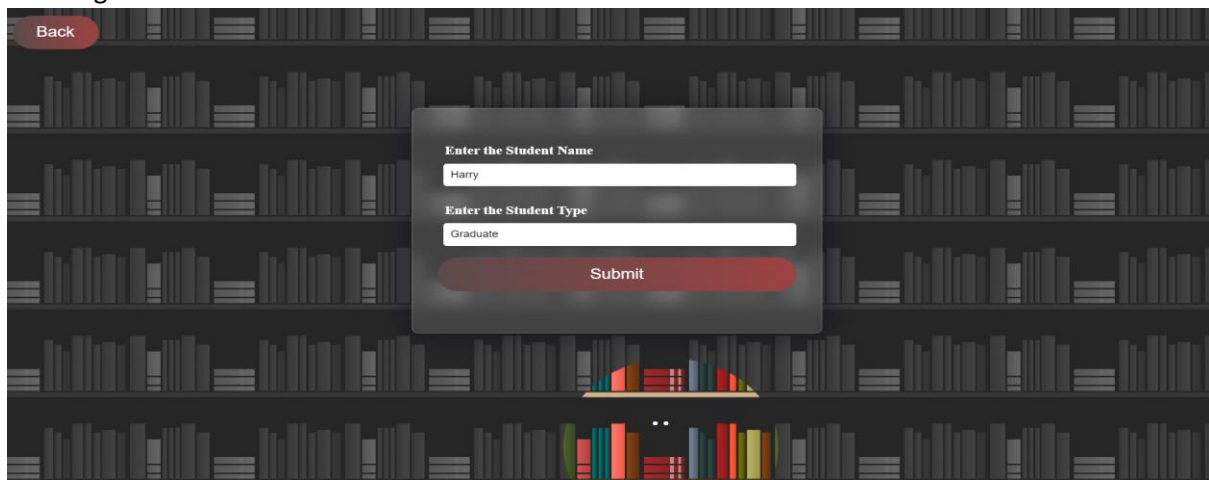
Displaying Enrollment Table:

Menu

Enroll Student

Course IDs		Student IDs
111	•	111
116	•	123
120	•	114
	•	114
	•	122

Creating New Student:



A screenshot of a web application interface for creating a new student. The background is a dark library with bookshelves. A modal form is centered, featuring a 'Back' button in the top left. The form has two input fields: 'Enter the Student Name' with the value 'Harry' and 'Enter the Student Type' with the value 'Graduate'. A red 'Submit' button is at the bottom of the form.

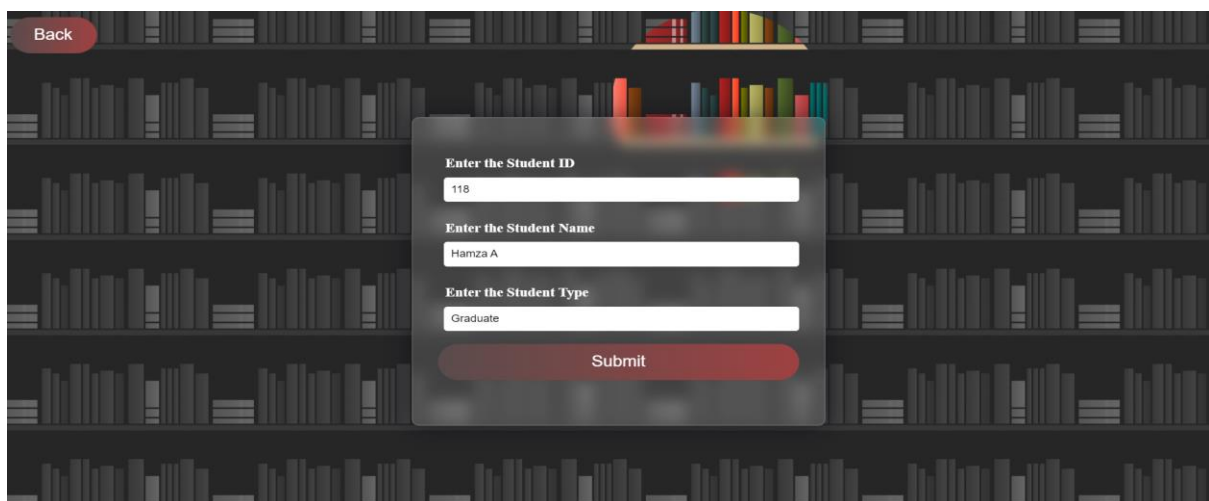
Back

Enter the Student Name
Harry

Enter the Student Type
Graduate

Submit

Update Student Details:



A screenshot of a web application interface for updating student details. The background is a dark library with bookshelves. A modal form is centered, featuring a 'Back' button in the top left. The form has three input fields: 'Enter the Student ID' with the value '118', 'Enter the Student Name' with the value 'Hamza A', and 'Enter the Student Type' with the value 'Graduate'. A red 'Submit' button is at the bottom of the form.

Back

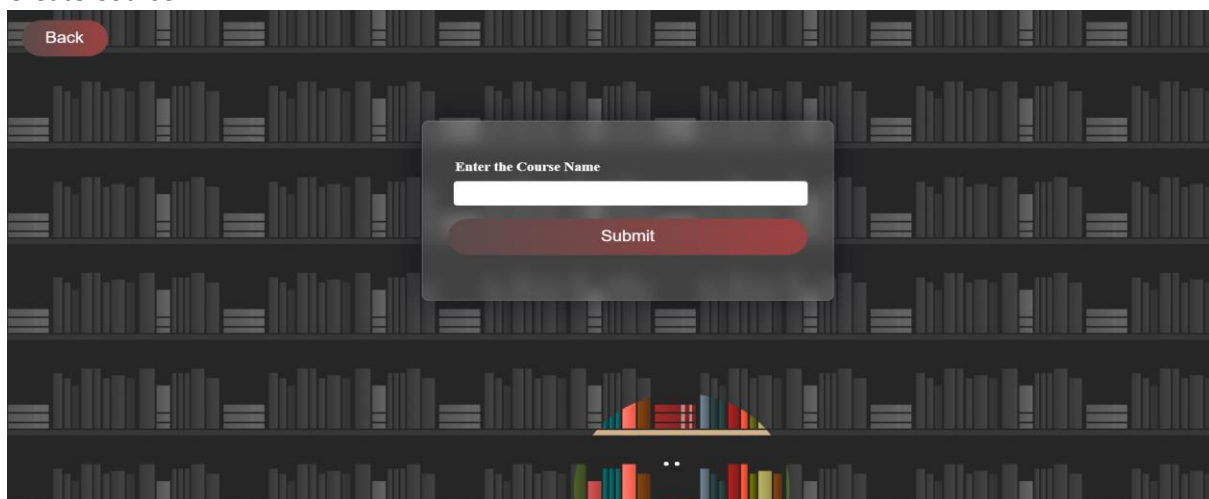
Enter the Student ID
118

Enter the Student Name
Hamza A

Enter the Student Type
Graduate

Submit

Create Course:



A screenshot of a web application interface for creating a new course. The background is a dark library with bookshelves. A modal form is centered, featuring a 'Back' button in the top left. The form has one input field: 'Enter the Course Name'. A red 'Submit' button is at the bottom of the form.

Back

Enter the Course Name

Submit

Enrolling Student in a course:

Back

Enter the Course ID

112

Enter the Student ID

121

Submit

Show All students id enrolled for a class based on course id:

Back

Enter the Class ID

120

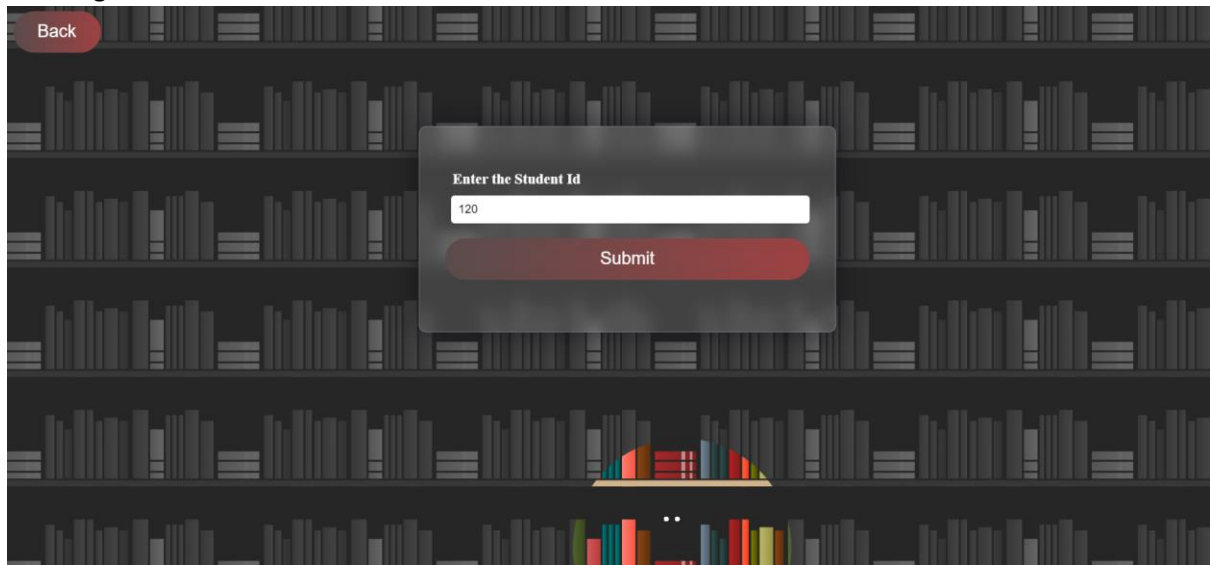
Submit

Menu

Enroll Student

B#
114
114
122

Deleting a student:



Back

Enter the Student Id

Submit