Final Project Report - Unified Smart Campus Hub

Team Members: Abdelrahman Abdelal, Wadoud Saleh, Yahya Eldemardash **Course**: CSE 460

Table of Contents

Phase 0: Proposal & Scope Definition

1. Problem Statement

Modern university campuses often rely on fragmented legacy systems to manage academic records, facility bookings, and student services. Currently, a student might use one portal for course registration, a separate physical log for lab equipment, and a third-party app for cafeteria payments. This fragmentation leads to data silos, administrative overhead, and a poor user experience. Faculty members lack real-time visibility into resource availability, and administrators struggle to generate holistic reports on campus utilization.

2. The Solution

The Unified Smart Campus Hub (USCH) is a centralized platform designed to integrate academic and operational workflows. The system replaces manual processes with automated transactions, such as intelligent course prerequisite verification and real-time equipment scheduling.

3. Stakeholders

7. **Students**: Access courses, book resources, pay for services.
8. **Faculty**: Manage courses, view resource availability.
9. **Department Admins**: Manage curriculum and users.
10. **Facility Managers**: Oversee lab and room usage.

4. Feature List

MVP (Minimum Viable Product)

11. **User Authentication**: Secure login for different roles (RBAC).
12. **Academic Services**:
    - View course catalog.
    - Add/Drop courses (Student).
    - View Class Roster (Faculty).
13. **Facility Management**:
    - View available labs/rooms.
    - Book a resource (with conflict detection).
14. **Profile Management**: Update user specific details.

Nice-to-Have

15. **Digital Wallet**: Integration for on-campus payments.
16. **Notification System**: Alerts for booking confirmations or class changes.
17. **AI/ML Analytics**: Predictive analytics for course demand.

Phase I: Requirements Analysis

1. Functional Requirements

User Management

- **FR-01**: As a user, I want to log in using my university credentials so that I can securely access the system.
- **FR-02**: As a system administrator, I want the system to support multiple user roles (Student, Faculty, Admin) so that access and permissions are properly controlled.

Academic Services

- **FR-03**: As a student, I want to view a list of available courses so that I can plan my academic schedule.
- **FR-04**: As a student, I want to add a course if prerequisites are met and seats are available so that I can enroll successfully.
- **FR-05**: As a faculty member, I want to view the list of students enrolled in my assigned courses so that I can manage my classes.

Facility Management

- **FR-07**: As a user, I want to view available labs or rooms so that I can choose an appropriate resource.
- **FR-08**: As a user, I want to book a lab or room so that I can reserve resources without scheduling conflict.

Admin Management

- **FR-10**: As an administrator, I want to create, read, update, and delete users (Students, Faculty, Admins), including National ID and phone number information, so that user records are properly maintained.
- **FR-11**: As an administrator, I want to manage department structures so that academic units are organized correctly.
- **FR-12**: As an administrator, I want to manage courses and define prerequisites so that academic requirements are enforced.
- **FR-13**: As an administrator, I want to create course sections by assigning instructors and rooms while automatically checking for schedule conflicts so that clashes are avoided.

- **FR-14**: As an administrator, I want to manage resources such as labs, rooms, and halls so that facilities are properly controlled.

2. Non-Functional Requirements

- **NFR-01 Security**: As a system owner, I want passwords to be securely hashed and access to be restricted by user role so that sensitive data is protected.
- **NFR-02 Performance**: As a user, I want the system to respond in under 2 seconds during peak times so that I can complete tasks efficiently.
- **NFR-03 Availability**: As a user, I want the system to be available 99.9% of the time during academic terms so that I can rely on it when needed.
- **NFR-04 Usability**: As a user, I want the application to be responsive and usable on mobile devices so that I can access it anywhere.

3. Use Case Model

3.1 Use Case Diagram

Actors: Student, Faculty, Admin.

Use Cases:

UC-01: Login

Use case name: Login

Description: Allows a user to authenticate into the system using valid credentials and access system features based on their assigned role (Student, Instructor, or Admin).

Primary Actor: Student / Instructor / Admin

Basic Flow:

1. The actor opens the system and selects the **Login** option.
2. The system prompts the actor to enter login credentials.
3. The actor enters username and password (or university SSO).
4. The system validates the entered credentials.
5. The system identifies the actor's role.
6. The system grants access and redirects the actor to the appropriate dashboard.

UC-02: View Courses

Use case name: View Courses

Description: Allows users to view and browse the list of available courses offered by the institution.

Primary Actor: Student

Basic Flow:

1. The actor selects **View Courses** from the menu.
2. The system verifies that the actor is logged in.
3. The system displays a list of available courses.
4. The actor may filter or search for courses.
5. The actor selects a course to view its details.
6. The system displays detailed information about the selected course.

UC-03: Register for Course

Use case name: Register for Course

Description: Allows a student to register for a selected course if prerequisites are met and seats are available.

Primary Actor: Student

Basic Flow:

1. The student selects a course and clicks **Register**.
2. The system ensures the student is logged in.
3. The system verifies course prerequisites for the student.
4. The system checks seat availability for the course.
5. If conditions are satisfied, the system registers the student in the course.
6. The system updates the student's schedule.
7. The system notifies the student of successful registration.

UC-04: View Schedule

Use case name: View Schedule

Description: Allows a student to view their registered courses and timetable.

Primary Actor: Student

Basic Flow:

1. The student selects **View Schedule**.
2. The system verifies that the student is logged in.
3. The system retrieves the student's registered courses.
4. The system displays the schedule in a structured format.
5. The student reviews schedule details.

UC-05: Book Resource

Use case name: Book Resource

Description: Allows users to book a lab or room while preventing double-booking of resources.

Primary Actor: Student / Instructor

Basic Flow:

1. The actor selects **Book Resource**.
2. The system verifies that the actor is logged in.
3. The system displays available resources and time slots.
4. The actor selects a resource and preferred time.
5. The system checks resource availability.
6. If available, the system confirms the booking.
7. The system notifies the actor of the successful reservation.

UC-06: Manage Users

Use case name: Manage Users

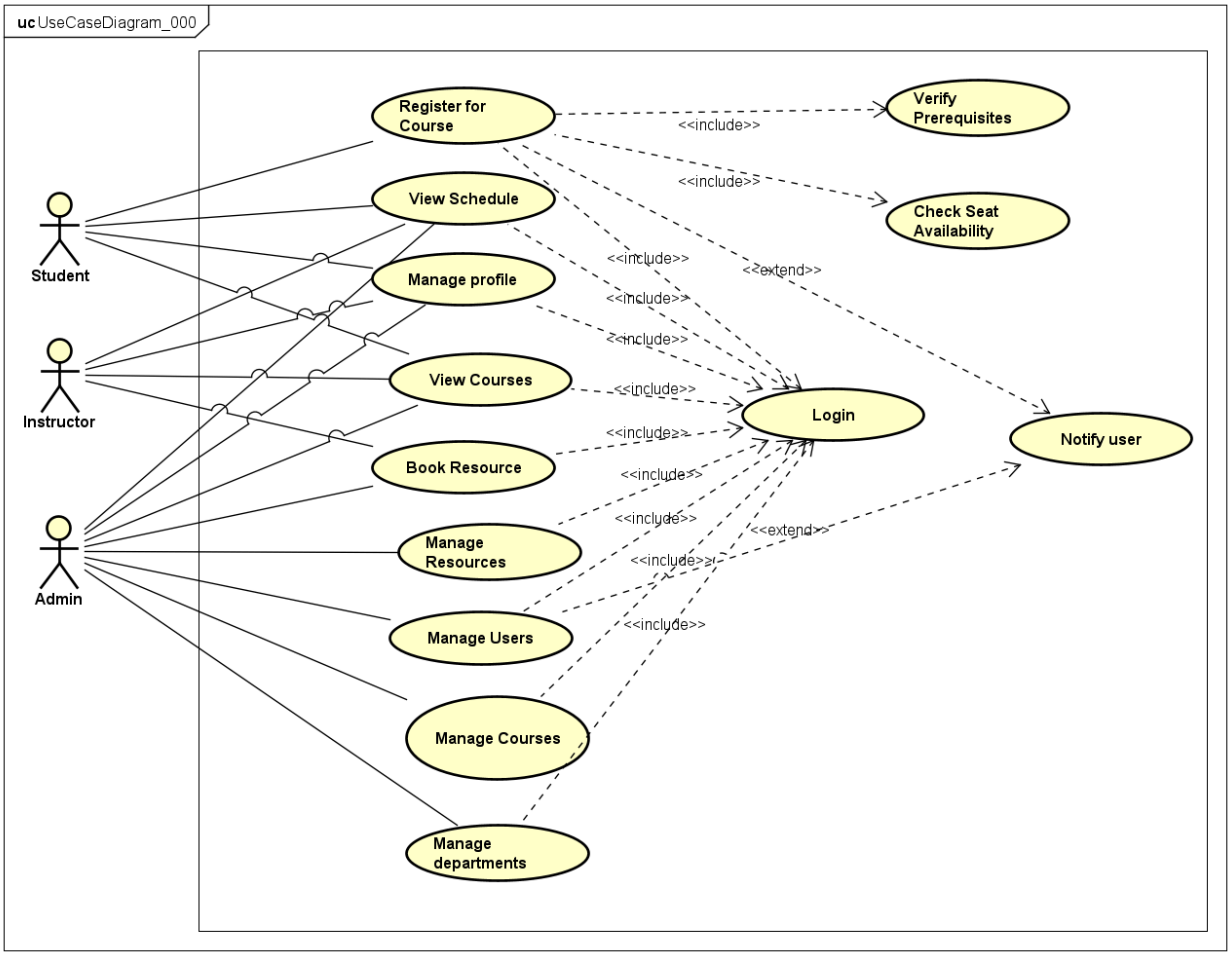
Description: Allows the Admin to create, update, view, and delete user accounts in the system.

Primary Actor: Admin

Basic Flow:

1. The admin selects **Manage Users**.
2. The system verifies admin authentication and authorization.
3. The system displays a list of users.
4. The admin selects an action (create, update, delete, or view user).
5. The admin enters or modifies user information.
6. The system validates the data.
7. The system saves changes to the database.

8. The system confirms the operation.



---

4. Domain Analysis

4.1 CRC Cards (Key Classes)

Class: Student

- *Attributes*: Name, Email, National ID, Phone Number, GPA.
- *Responsibilities*: Register for Course Sections, View Schedule.
- *Collaborators*: CourseSection, Booking.

Class: Instructor

- *Attributes*: Name, Email, National ID, Rank (Faculty/Doctor), Specialization.
- *Responsibilities*: Teach Sections, View Roster.

- *Collaborators*: CourseSection.

Class: Course

- *Responsibilities*: Define curriculum, Maintain Prerequisites.
- *Collaborators*: CoursePrerequisite, Department.

Class: CourseSection

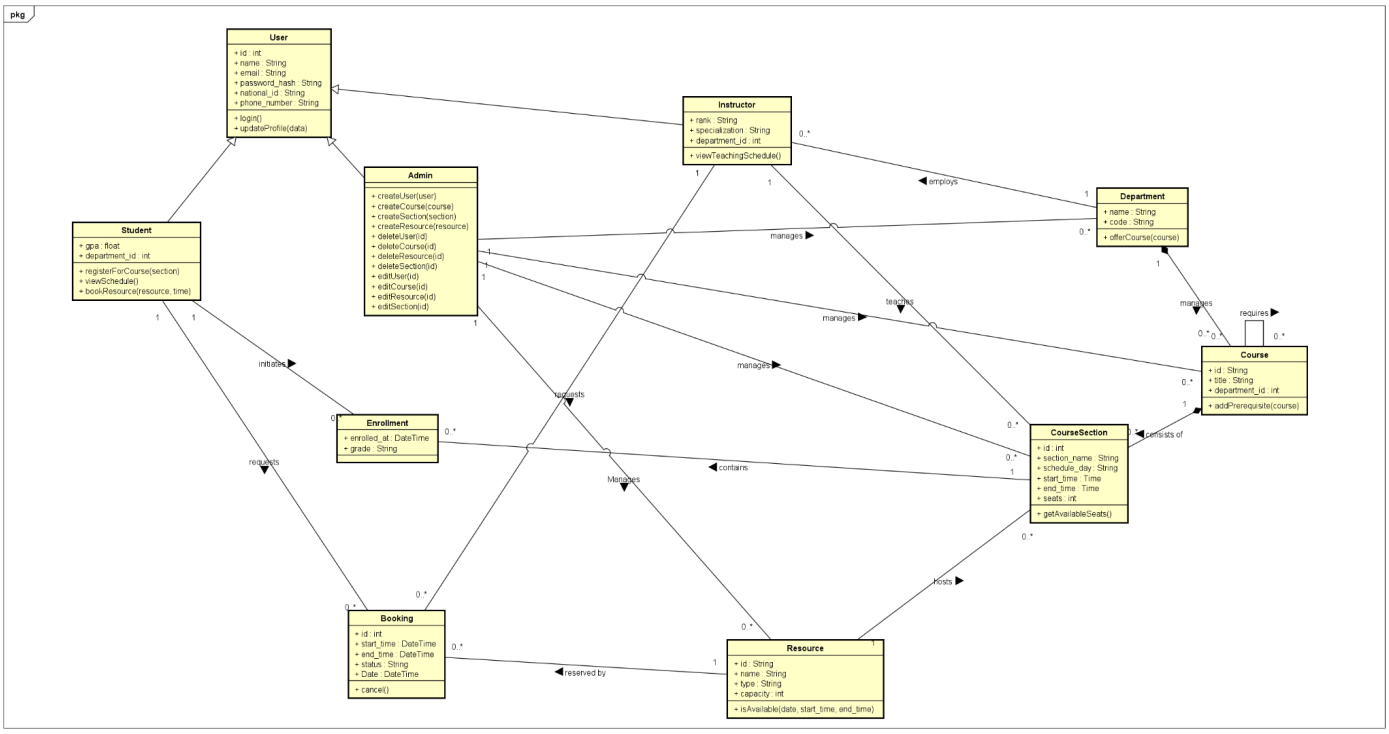
- *Responsibilities*: Schedule specific instance of a course (Day/Time), Track Seats/Capacity.
- *Collaborators*: Instructor, Room, Student (Enrollment).

4.2 Conceptual Domain Model

(Imagine a diagram here connecting these entities) `Student` --(enrolls in)--> `Course` `Faculty` --(teaches)--> `Course` `User` --(makes)--> `Booking` `Booking` --(reserves)--> `Resource`

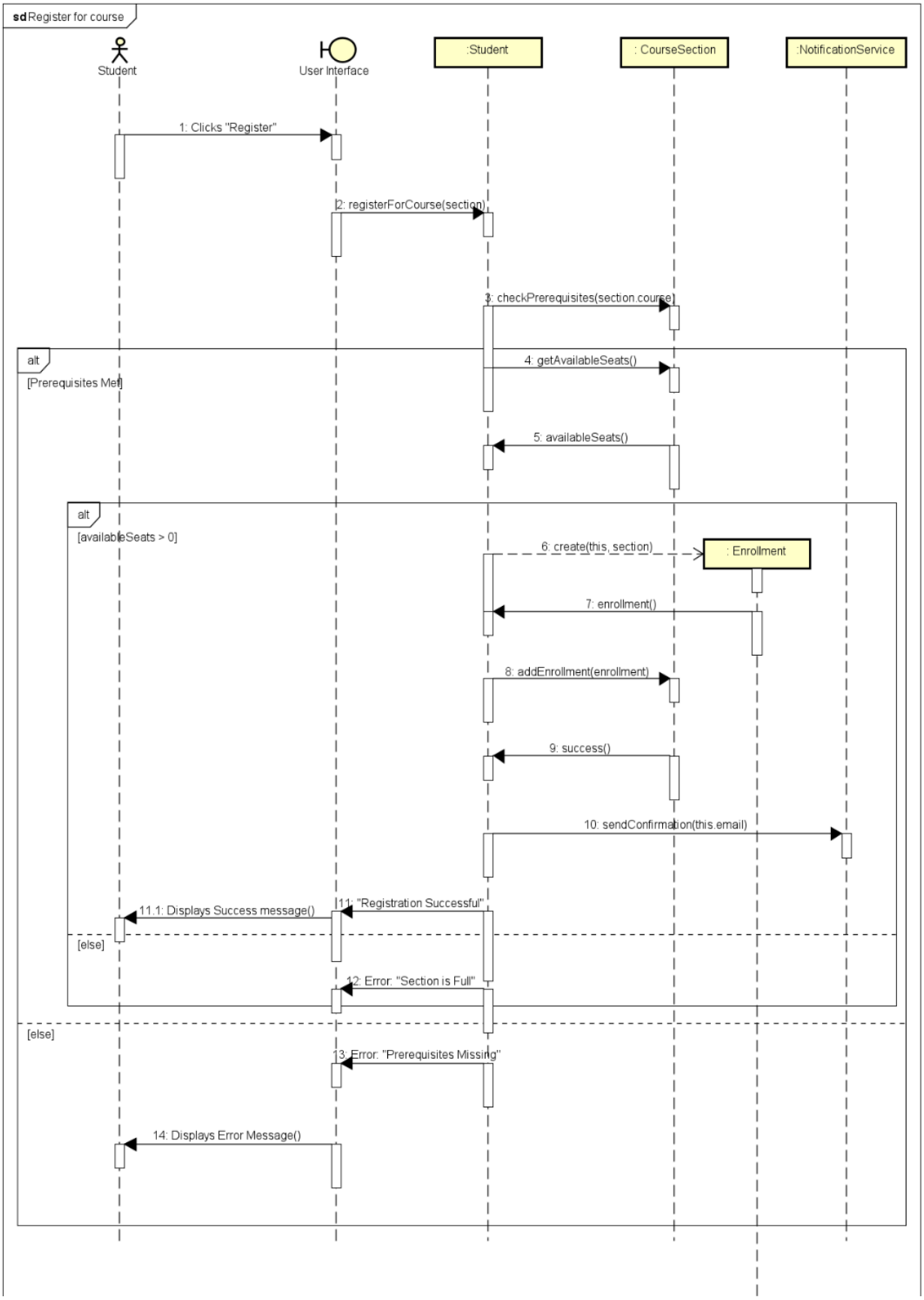
ConPhase II: Object-Oriented Design

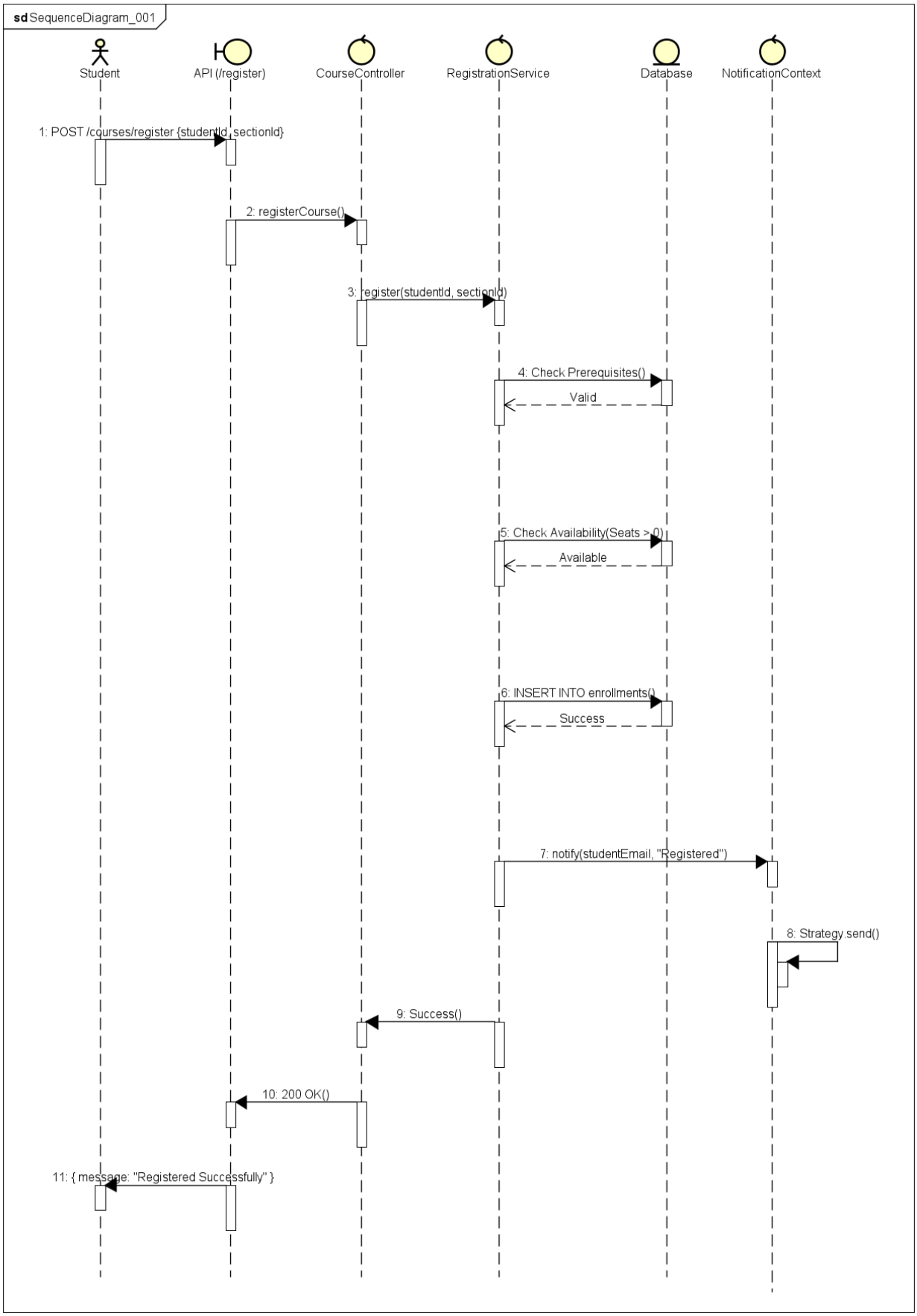
1. Detailed Class Diagram



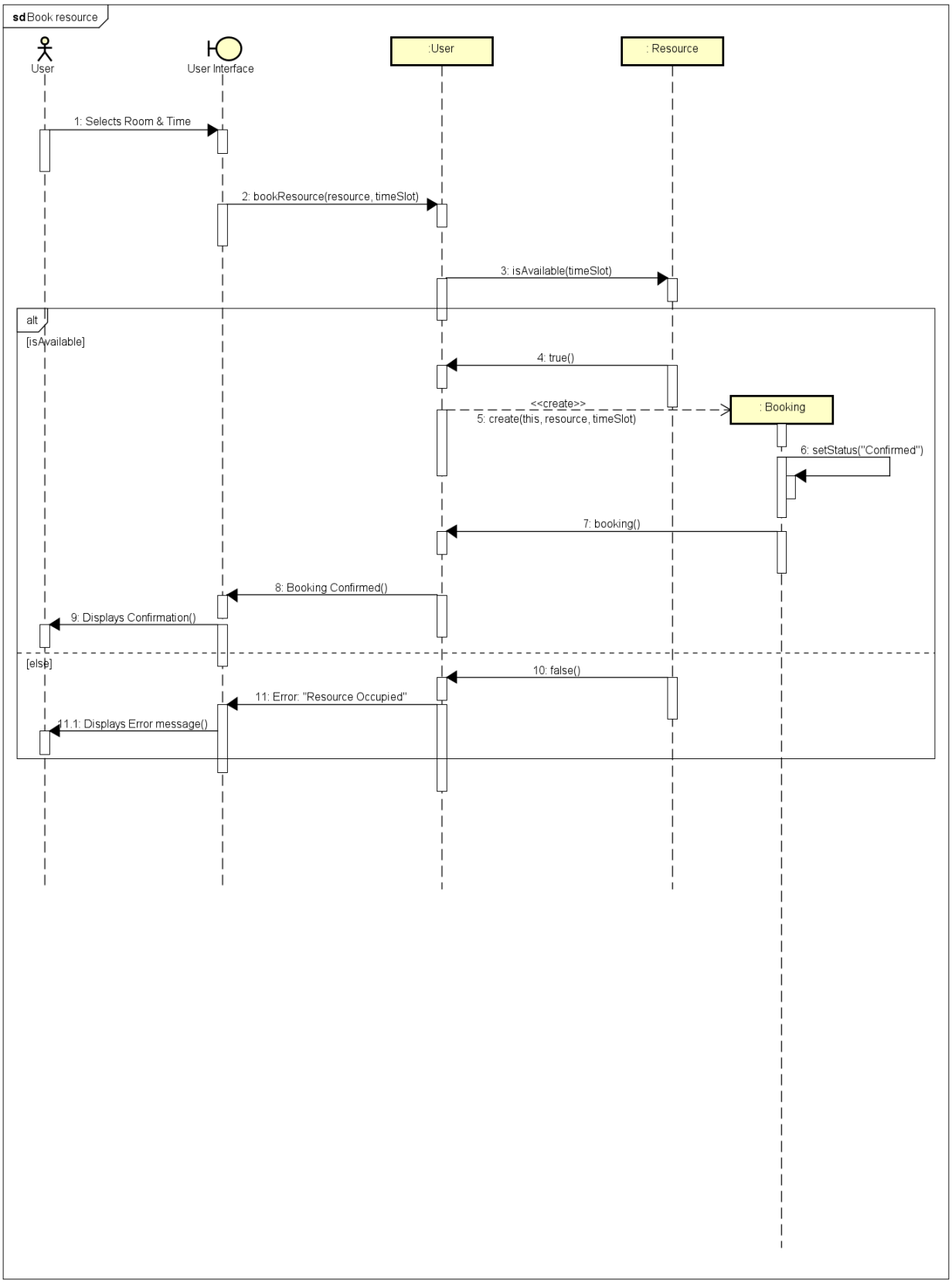
2. Sequence Diagrams

Scenario 1: Register for Course (Student)

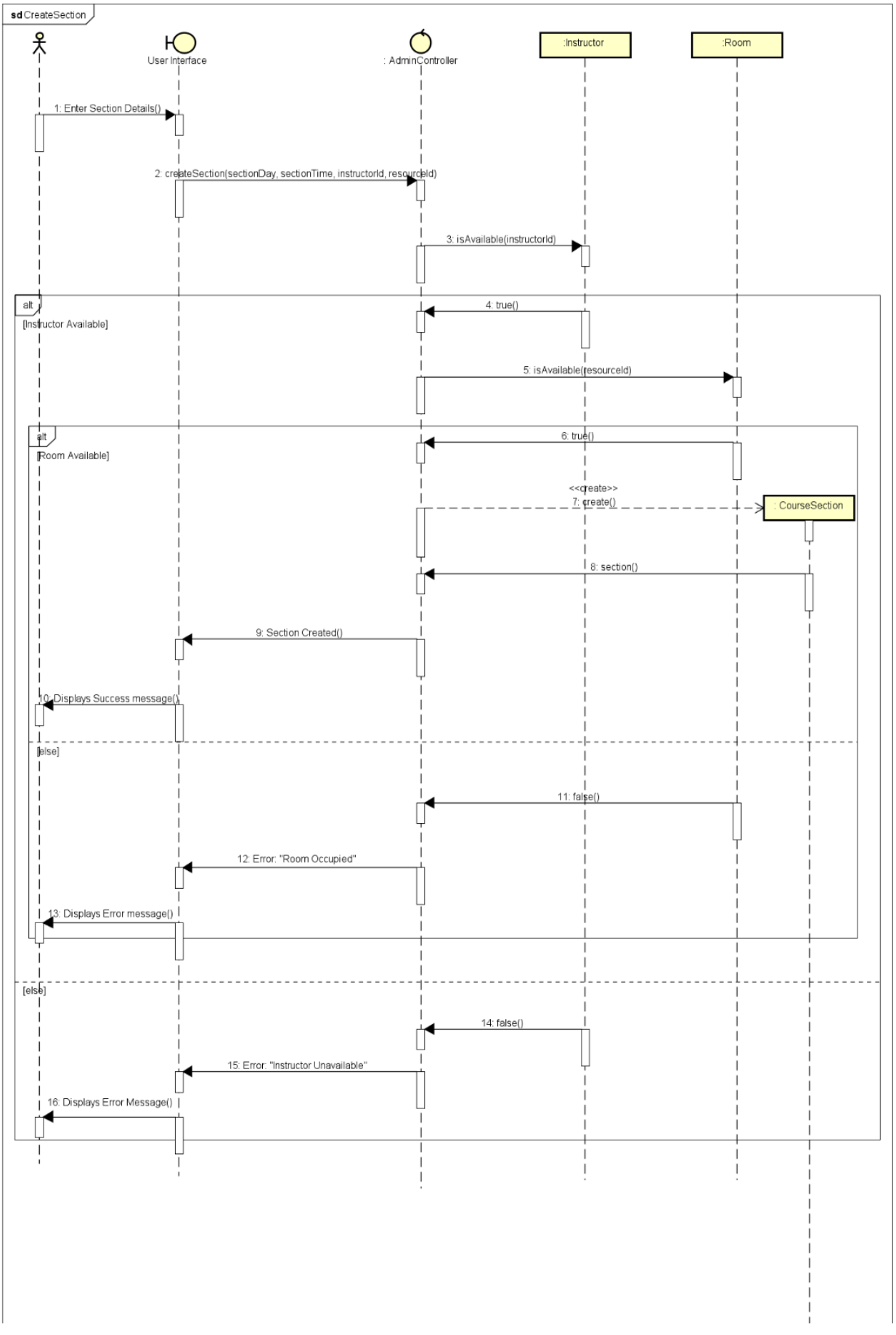

sd Register for course
Student
User Interface
:Student
: CourseSection
:NotificationService
1: Clicks "Register"
2: registerForCourse(section)
3: checkPrerequisites(section.course)
alt
[Prerequisites Met]
4: getAvailableSeats()
5: availableSeats()
alt
[availableSeats > 0]
6: create(this, section)
: Enrollment
7: enrollment()
8: addEnrollment(enrollment)
9: success()
10: sendConfirmation(this.email)
11.1: Displays Success message()
11: "Registration Successful"
[else]
12: Error: "Section is Full"
[else]
13: Error: "Prerequisites Missing"
14: Displays Error Message()


sd SequenceDiagram_001
Student
API (/register)
CourseController
RegistrationService
Database
NotificationContext
1: POST /courses/register {studentId, sectionId}
2: registerCourse()
3: register(studentId, sectionId)
4: Check Prerequisites()
Valid
5: Check Availability(Seats > 0)
Available
6: INSERT INTO enrollments()
Success
7: notify(studentEmail, "Registered")
8: Strategy.send()
9: Success()
10: 200 OK()
11: { message: "Registered Successfully" }

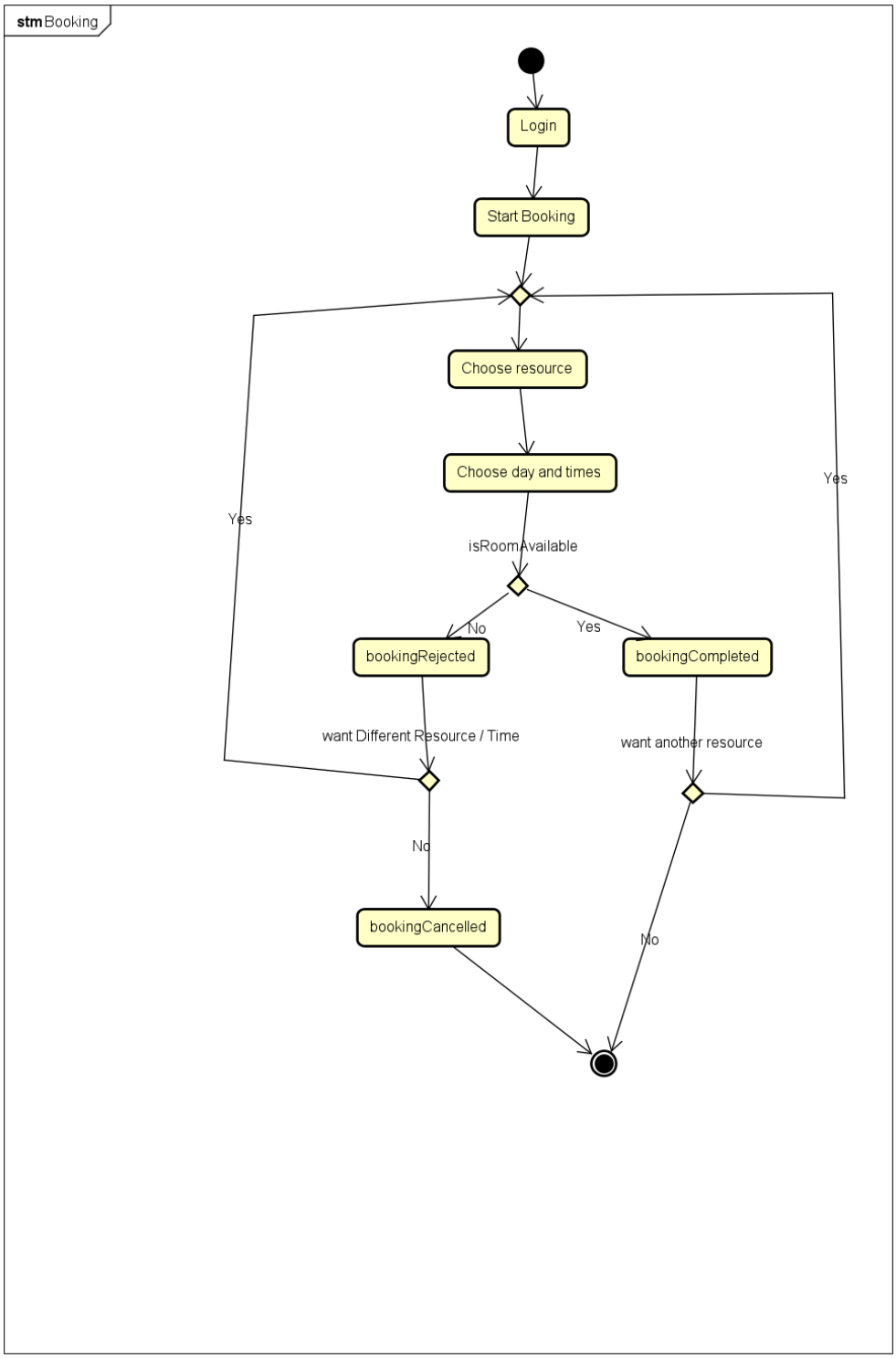
Scenario 2: Book a Resource (Faculty/Admin)


sd Book resource
User
User Interface
:User
: Resource
1: Selects Room & Time
2: bookResource(resource, timeSlot)
3: isAvailable(timeSlot)
alt
[isAvailable]
4: true()
<<create>>
5: create(this, resource, timeSlot)
: Booking
6: setStatus("Confirmed")
7: booking()
8: Booking Confirmed()
9: Displays Confirmation()
[else]
10: false()
11: Error: "Resource Occupied"
11.1: Displays Error message()

Scenario 3: Create Course Section (Admin)


sd CreateSection
User Interface
: AdminController
:Instructor
:Room
1: Enter Section Details()
2: createSection(sectionDay, sectionTime, instructorId, resourceId)
3: isAvailable(instructorId)
alt
[Instructor Available]
4: true()
5: isAvailable(resourceId)
alt
[Room Available]
6: true()
<<create>>
7: create()
: CourseSection
8: section()
9: Section Created()
10: Displays Success message()
[else]
11: false()
12: Error: "Room Occupied"
13: Displays Error message()
[else]
14: false()
15: Error: "Instructor Unavailable"
16: Displays Error Message()

3. State Diagram (Booking Lifecycle)



Phase III: Software Architecture & C4 Model

1. Architecture Pattern

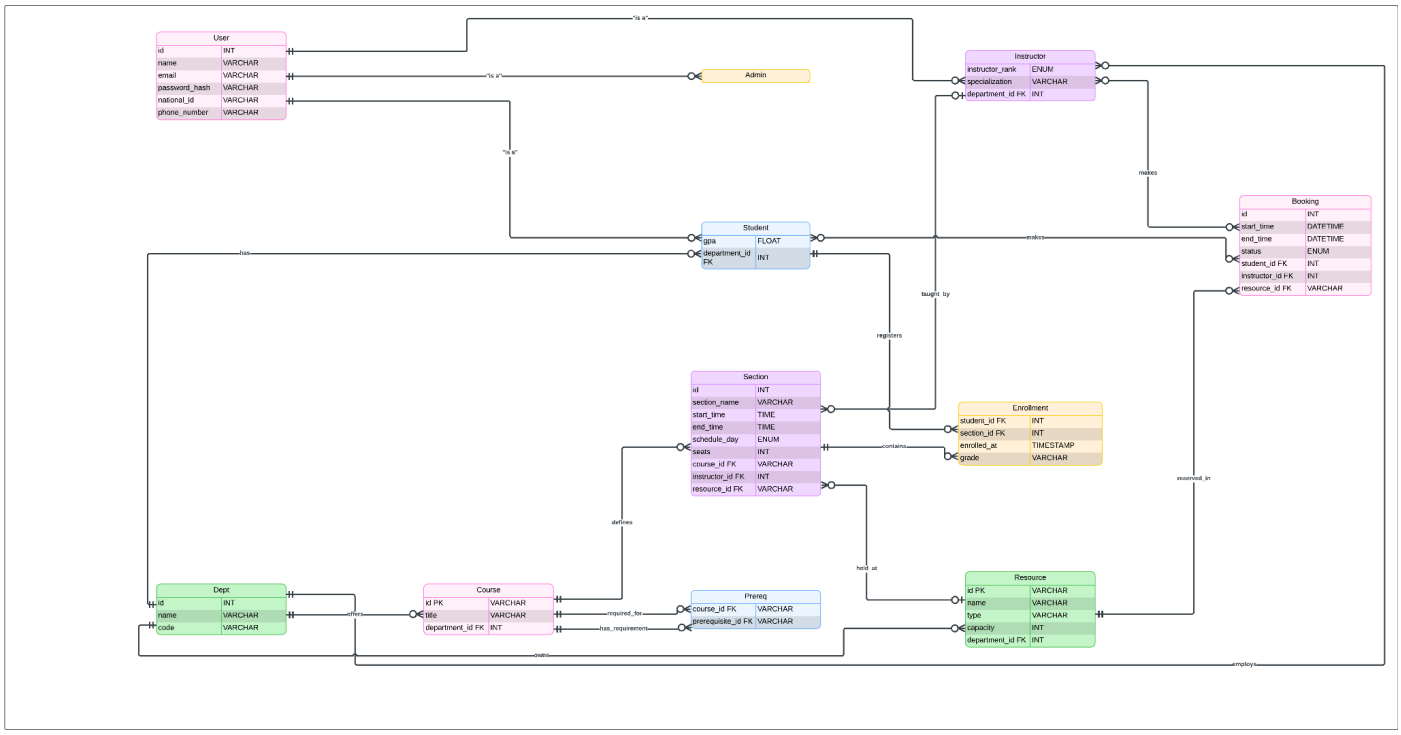
Style: **Layered Client-Server Architecture (Monolithic)** *Justification*:

- **Client-Server**: The requirements specify a Web System App and a Mobile App (conceptual), which necessitates a distinct separation between Frontend and Backend via REST API.
- **Layered**: A standard MVC layer structure (Controller, Service, Repository) ensures separation of concerns, making the system testable and maintainable. This also prepares the system for easier distinct extraction into Microservices in Phase VI (e.g., slicing vertical layers).

Evolution Plan:

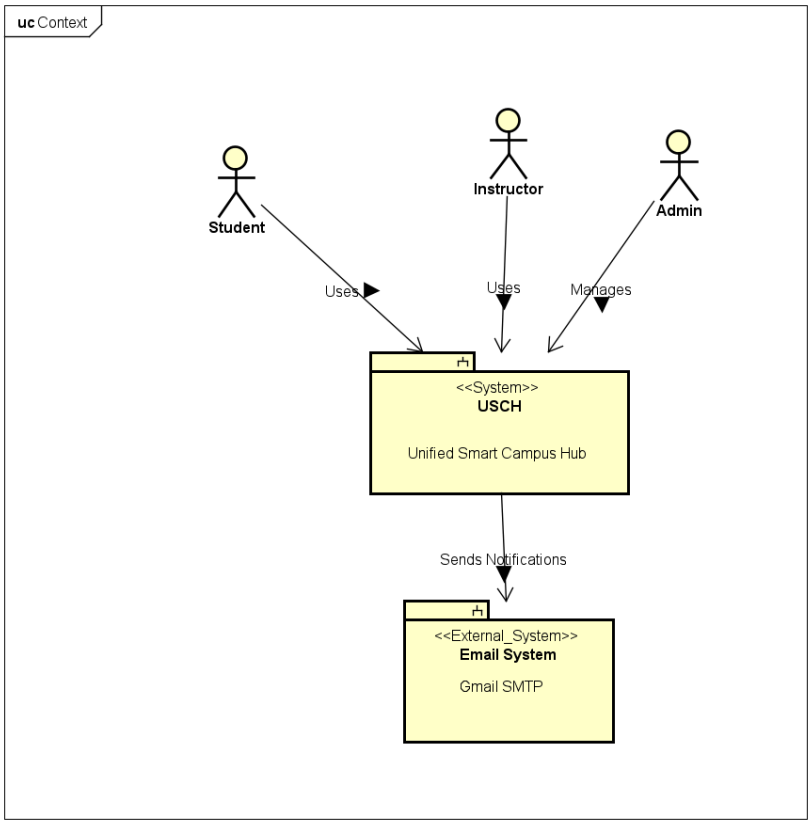
- Phase V: Monolithic MVC (Node.js API + React Client).
- Phase VI: Strangler Fig Pattern to migrate to Microservices.

ERD

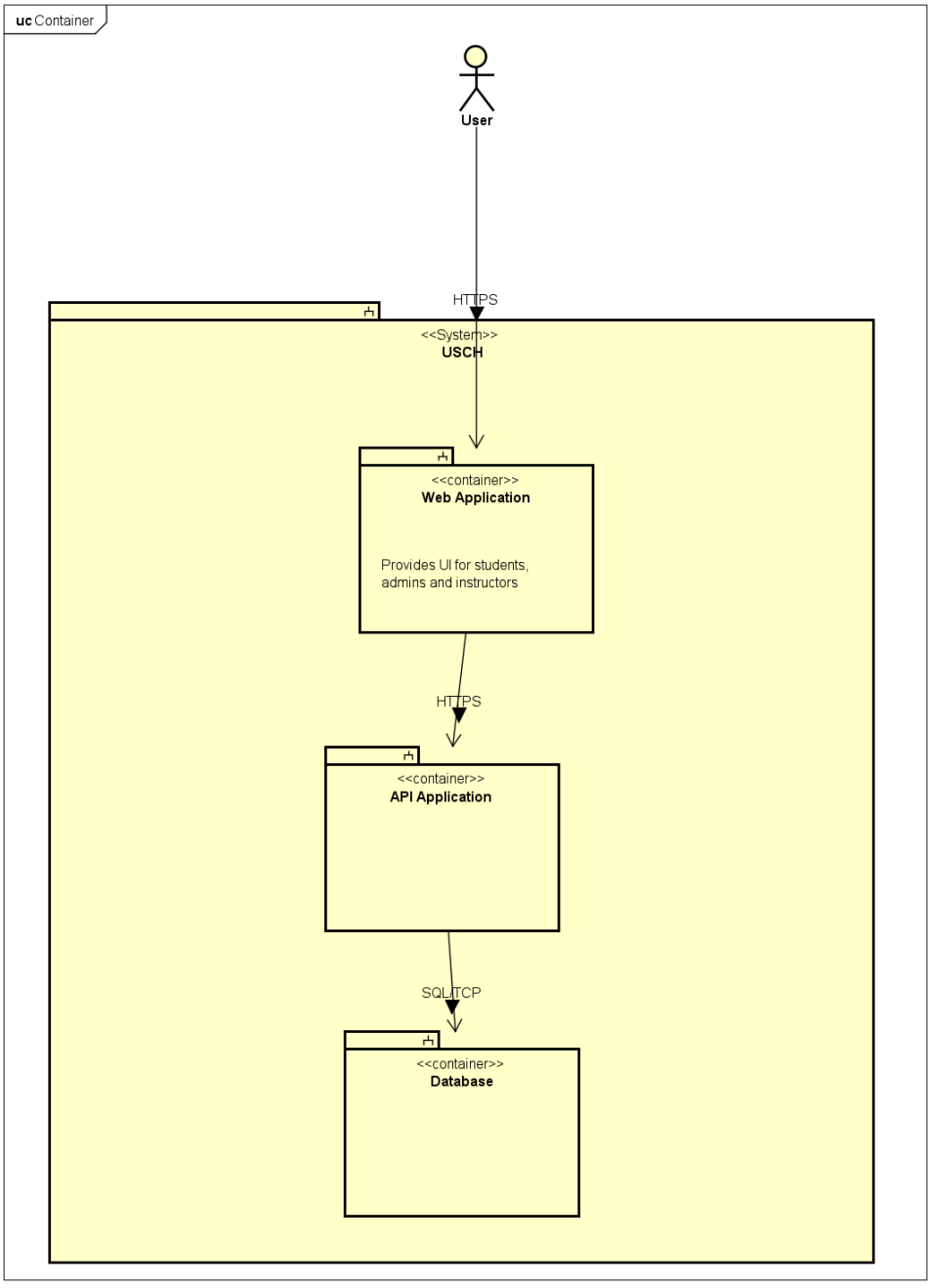


2. C4 Model Diagrams:

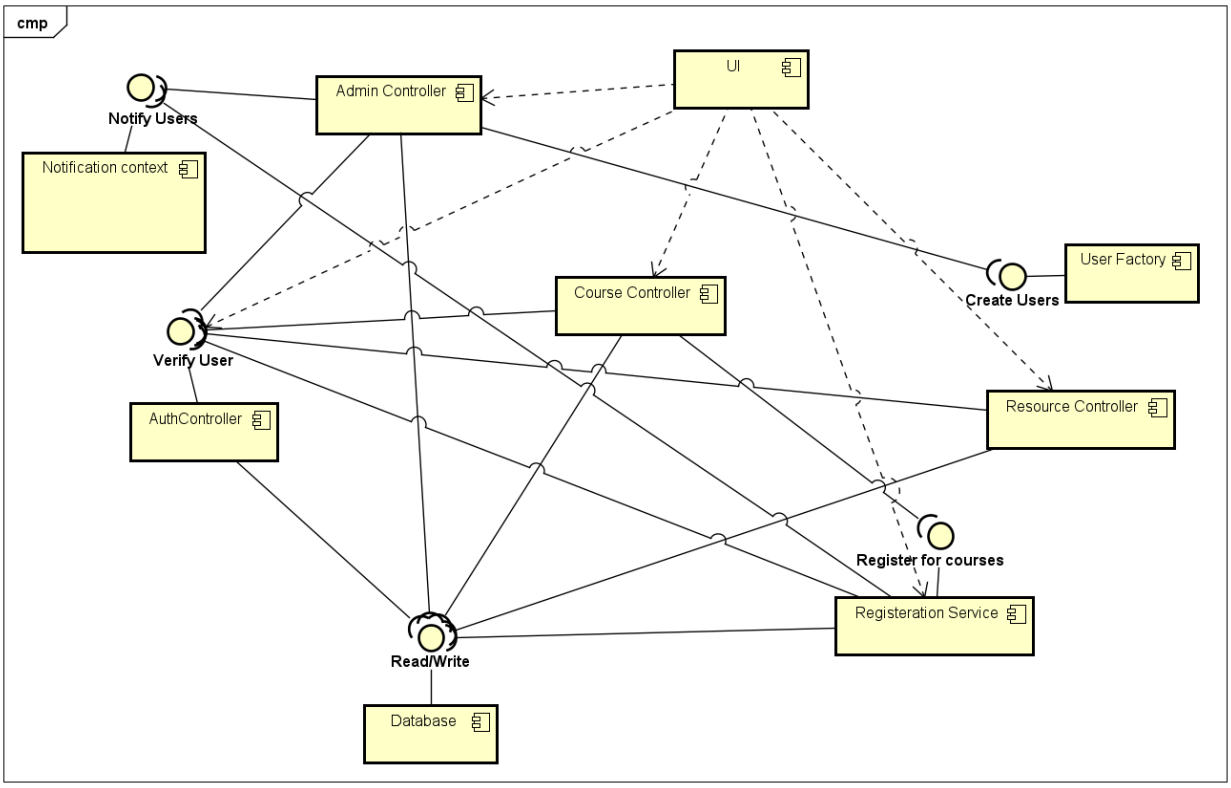
1. Context Diagram (Level 1):



2. Container Diagram (Level 2):



3. Component Diagram (Level 3):



3. Architecture Evaluation (Mini-ATAM)

Quality Attribute Scenarios

1. **Availability**: "If the Booking Database fails, the Academic Services must continue to function."
2. **Performance**: "During registration (Add/Drop), the system must handle 500 concurrent requests with < 2s latency."
3. **Modifiability**: "Adding a new user role (e.g., 'Guest') should not require changing existing core logic."

Risks & Trade-offs

Risk/Sensitivity	**Trade-off**	**Mitigation Strategy**
Risk: Monolithic Database is a single point of failure.	**Simplicity vs. Resilience**: We chose a single DB for easier development (consistency), trading off some resilience.	Implement Read Replicas and regular backups.

Risk/Sensitivity	Trade-off	Mitigation Strategy
Sensitivity: High concurrency during registration might lock the DB rows.	**Consistency vs. Performance**: Strict ACID transactions ensure no over-booking but allow fewer concurrent writes.	Use row-level locking or optimistic concurrency control.
Trade-off: Javascript (Node.js) used for Backend.	**Speed of Dev vs. CPU Intensity**: JS is fast for I/O but bad for CPU heavy tasks.	Offload complex calculations (e.g., AI scheduling) to a separate worker service later.

Phase IV: Design Patterns & Refactoring

1. Design Patterns Implementation

A. Singleton Pattern

Requirement: The system needs a single shared database connection pool to avoid overhead and connection limits. **Snippet**:

```
const mysql = require('mysql2/promise');

class Database {

    constructor() {

        if (!Database.instance) {

            this.pool = mysql.createPool({ ... });

            // Test connection

            this.pool.getConnection().then(...)

            Database.instance = this;

        }

        return Database.instance;
```

```
    }

    query(sql, params) { return this.pool.query(sql, params); }

}

const db = new Database();

module.exports = db;
```

B. Factory Pattern

Requirement: We have multiple user types (Student, Faculty, Admin) that need to be created with different initial properties and permissions during registration. **Snippet**:

```
class UserFactory {

    static async createUser(role, data) {

        let query = '';

        let params = [];

        const hash = await bcrypt.hash(data.password, 10);

        switch (role) {

            case 'student':

                query = 'INSERT INTO students ...';

                break;

            case 'admin':

                query = 'INSERT INTO admins ...';

                break;

            case 'faculty':

                query = 'INSERT INTO instructors ...';

                break;
```

```
            default:
                throw new Error('Invalid user role');
        }
        return { query, params };
    }
}

// Usage in Controller
const { query, params } = await UserFactory.createUser(role, req.body);
await pool.query(query, params);
```

C. Service Layer Pattern

Requirement: Encapsulate complex business logic. **Snippet**: (See RegistrationService.js)

D. Strategy Pattern

Requirement: Allow notification methods (Email, Console, SMS) to be swapped interchangeably. **Snippet**:

```
// Strategy Interface (Implicit)
class NotificationStrategy { async send(to, msg) {} }

// Concrete Strategy: Real Email via Nodemailer
class EmailNotification extends NotificationStrategy {
    constructor() {
        this.transporter = nodemailer.createTransport({ service: 'gmail', ... });
    }
    async send(to, msg) {
        await this.transporter.sendMail({ from: '"USCH Admin" <...>', to, text: msg });
```

```
        console.log(`[EmailService] Sent to ${to}`);
    }
}

// Context
class NotificationContext {
    constructor(strategy) { this.strategy = strategy; }
    async notify(to, msg) { return this.strategy.send(to, msg); }
}
```

2. Refactoring Scenario

Before Refactoring (Smell: Long Method / God Object)

The `register` function does validation, database checking, logic, and email sending all in one place.

```
// Bad Practice
function registerCourse(studentId, courseId) {
    const student = db.findStudent(studentId);
    if (!student) return error;

    const course = db.findCourse(courseId);
    if (course.seats <= 0) return error;
    if (student.gpa < 2.0) return error;
    db.saveEnrollment(studentId, courseId);
    emailService.send(student.email, "Registered!");
}
```

After Refactoring (Technique: Extract Method & Service Layer)

We extract logic into dedicated methods and services.

```
// Improved

// Improved: Using RegistrationService

class RegistrationService {

    async register(studentId, sectionId) {

        if (!studentId || !sectionId) throw new Error('Missing fields');

        const section = await this.getSection(sectionId);

        await this.validatePrerequisites(studentId, section.course_id);

        await this.checkAvailability(studentId, sectionId, section);

        await pool.query('INSERT INTO enrollments ...');

        return { message: 'Success' };

    }

    async validatePrerequisites(studentId, courseId) {

        // Complex query to check passed courses

    }

    async checkAvailability(studentId, sectionId, section) {

        // Check seats and existing enrollments

    }

}
```

Benefits:

- **Testability**: Can test validation logic separately.
- **Readability**: The main flow is clear.
- **Reusability**: `checkAvailability` can be used by other features.

Phase V: Implementation Details

1. Project Structure

- `src/backend`: Node.js + Express API Server.
- `src/frontend`: React + Vite Client Application.

2. API Documentation

Auth

- **POST** `/api/auth/login`
  - Body: `{ "email": "...", "password": "..." }`
  - Response: `{ "message": "...", "user": { ... } }`
- **GET** `/api/auth/profile/:role/:id`
  - Returns user profile details (sanitized).
- **PUT** `/api/auth/profile/:role/:id`
  - Body: `{ "email": "...", "phone_number": "...", "currentPassword": "...", "newPassword": "..." }`
  - Updates contact info or password (after verification).

Academic Services

- **GET** `/api/courses`
  - Returns list of courses with available seats.
- **POST** `/api/courses/register`
  - Body: `{ "studentId": 1, "courseId": "CSE460" }`
  - Response: `{ "message": "Success", "course": { ... } }`

Facility Management

- **GET** `/api/resources`
  - Returns list of labs and equipment.
- **POST** `/api/resources/book`
  - Body: `{ "userId": 1, "resourceId": "LAB-101", "startTime": "...", "endTime": "..." }`

    - Response: `{ "message": "Booking confirmed", "booking": { ... } }`

Admin Management

- **GET** `/api/admin/users`, `/api/admin/courses`, `/api/admin/sections`, `/api/admin/departments`
- **POST** `/api/admin/courses` (Create Course with Prerequisites)
- **POST** `/api/admin/sections` (Create Section with Conflict Check)
- **PUT** `/api/admin/students/:id` (Update Student info including NID/Phone)
- **PUT** `/api/admin/instructors/:id` (Update Instructor info)

3. Technology Stack

- **Backend**: Express.js (MVC Pattern: Controllers, Models, Routes).
- **Email Service**: Nodemailer (Gmail SMTP).
- **Frontend**: React.js (Client-side MVC components).
- **Styling**: Vanilla CSS (Premium Dark Mode).

---

conclusion

The USCH system has been designed with scalability in mind, transitioning from a layered monolith to a microservices architecture.