Hostel Management System

**Table of Contents**

**1. Introduction  
 1.1 Problem Statement  
 1.2 Objectives  
 1.3 Scope of the Project**

**2. System Analysis  
 2.1 Existing System  
 2.2 Limitations of Existing System  
 2.3 Proposed System  
 2.4 Feasibility Study  
 2.5 Requirement Analysis**

**3. System Design  
 3.1 System Architecture  
 3.2 ER Diagram  
 3.3 UML Diagrams  
  3.3.1 Use Case Diagram  
  3.3.2 Class Diagram  
  3.3.3 Activity Diagram**

**4. Technology Stack  
 4.1 Frontend  
 4.2 Backend  
 4.3 Database  
 4.4 Tools & Platforms Used**

**5. Implementation  
 5.1 Project Structure  
 5.2 Frontend Structure  
 5.3 Backend Structure  
 5.4 Code Snippets**

**6. Testing  
 6.1 Testing Approach  
 6.2 Sample Test Cases**

**7. Output Screenshots**

**8. Challenges Faced & Solutions  
 8.1 Managing Relational Data in SQLite  
 8.2 Handling Room Capacity and Occupancy  
 8.3 Implementing Role-Based Access  
 8.4 Notification Timestamping  
 8.5 Maintaining Dashboard Metrics  
 8.6 File and Module Organization  
 8.7 Ensuring Data Uniqueness**

**9. Future Enhancements  
 9.1 User Authentication & Security Enhancements  
 9.2 Enhanced Role Management  
 9.3 Room Allocation Automation  
 9.4 Notification Targeting  
 9.5 Real-Time Updates  
 9.6 Audit Logs  
 9.7 Migration to PostgreSQL or MySQL  
 9.8 Responsive & Mobile-Friendly UI  
 9.9 Hosting & Deployment**

**10. Conclusion  
11. References**

**1. Introduction**

Hostels play a crucial role in the lives of students and working professionals who move away from home in pursuit of education or employment. Managing hostel facilities manually can be tedious and error-prone, leading to inefficiencies and dissatisfaction among both residents and administrators.

To address these challenges, this project presents a **Hostel Management System** — a web-based application designed to streamline and digitize the daily operations of a hostel. The system allows hostel administrators to manage room allocations, student records, complaints, payments, notices, and much more through a user-friendly interface. It also empowers students to interact with the system for submitting requests, viewing notices, updating personal data, and tracking their room or payment status.

This system not only reduces the administrative burden but also improves transparency, accountability, and accessibility for all stakeholders. The primary objective is to replace the traditional paper-based or Excel-based processes with a modern, scalable, and secure web solution.

**1.1 Problem Statement**

Manual hostel management systems are prone to inefficiencies such as misplaced records, delayed response times, lack of transparency, and inconsistent data. These limitations create challenges in managing room assignments, student grievances, fee tracking, and communication. A digitized system is needed to streamline these tasks and provide a better experience for both management and residents.

**1.2 Objectives**

The main objectives of this project are:

* To develop a centralized web platform for hostel management.
* To provide role-based access for admin and student users.
* To manage room allotments and student registrations.
* To implement a notice broadcasting system.
* To ensure data security, ease of use, and minimal manual intervention.

**1.3 Scope of the Project**

The Hostel Management System developed as part of this project supports the following functionalities:

* Admin authentication and role management
* Adding/updating/deleting rooms and room types
* Student registration and profile management
* Notice creation and broadcasting
* Dashboard views for both students and admins
* Responsive UI with form validations
* Persistent storage using a relational database

Future enhancements may include integration with payment gateways, biometric entry logging, SMS/email notifications, and analytics dashboards for hostel usage insights.

**2. System Analysis**

**2.1 Existing System**

Most hostel facilities, especially in educational institutions, still rely on manual processes or basic tools like spreadsheets for managing student records, room allocations, complaints, and payments. This often results in:

* Repetitive paperwork
* Higher chances of human error
* Lack of transparency
* Slow processing of student complaints or requests
* Difficulty in tracking room availability and dues

These limitations create inefficiencies for both hostel administrators and students.

**2.2 Limitations of the Existing System**

* Manual data entry is time-consuming and error-prone.
* No centralized access to information for students and admins.
* Room and fee records are difficult to maintain and update.
* Communication of notices and handling complaints is delayed.
* Data can be misplaced or lost due to lack of backups.

**2.3 Proposed System**

The proposed **Hostel Management System** is a web-based application designed to overcome these limitations. It supports the following features:

* Admin login and dashboard
* Student registration and login
* Room management (add/update/delete)
* Room allocation to students
* Notice management by admin
* Role-based access and operations
* Responsive user interface

This system helps administrators manage hostel activities more efficiently while giving students access to important services from anywhere.

**2.4 Feasibility Study**

**a) Technical Feasibility**

The system uses widely available and well-supported technologies like HTML, CSS, Typescript, Node.js, React, Express, and SQLite — making it technically feasible for development and deployment.

**b) Operational Feasibility**

The project is user-friendly and requires minimal technical training. Both students and admins can operate it through a web browser, making it easy to adopt in a real hostel environment.

**c) Economic Feasibility**

As it uses open-source technologies and requires no licensing costs, the solution is budget-friendly and suitable for small to medium-sized hostels.

**2.5 Requirement Analysis**

**Functional Requirements**

* Admin login, dashboard, and CRUD operations
* Student signup, login, profile, and room allocation
* Notice posting and viewing

**Non-Functional Requirements**

* Responsive and user-friendly UI
* Secure login and access control
* Fast performance with proper routing
* Scalable and modular backend code
* Use of SQLite for persistent storage.

**3. System Design**

**3.1 System Architecture**

The project follows a **modern full-stack architecture** with a clear separation between the **client** and **server**, using TypeScript on both ends for type safety and maintainability.

**System Overview**

Client (React + TS + Tailwind + Zustand)

↕ HTTP API (REST)

Server (Node.js + Express + TS + SQLite)

↕

SQLite Database

**Frontend (Client)**

* **Framework**: React
* **Language**: TypeScript
* **Styling**: TailwindCSS
* **State Management**: Zustand
* **Features**:
  + Authenticated views for admin and student
  + Form submissions for login, registration, complaints, etc.
  + Dynamic routing and conditional rendering
  + Responsive UI and toast notifications

**Backend (Server)**

* **Runtime**: Node.js
* **Framework**: Express.js
* **Language**: TypeScript
* **Database**: SQLite (with better-sqlite3)
* **Features**:
  + RESTful APIs for authentication, students, rooms, complaints, notices, and more
  + Role-based access control
  + Secure session/token handling (if implemented)
  + Lightweight and portable server logic

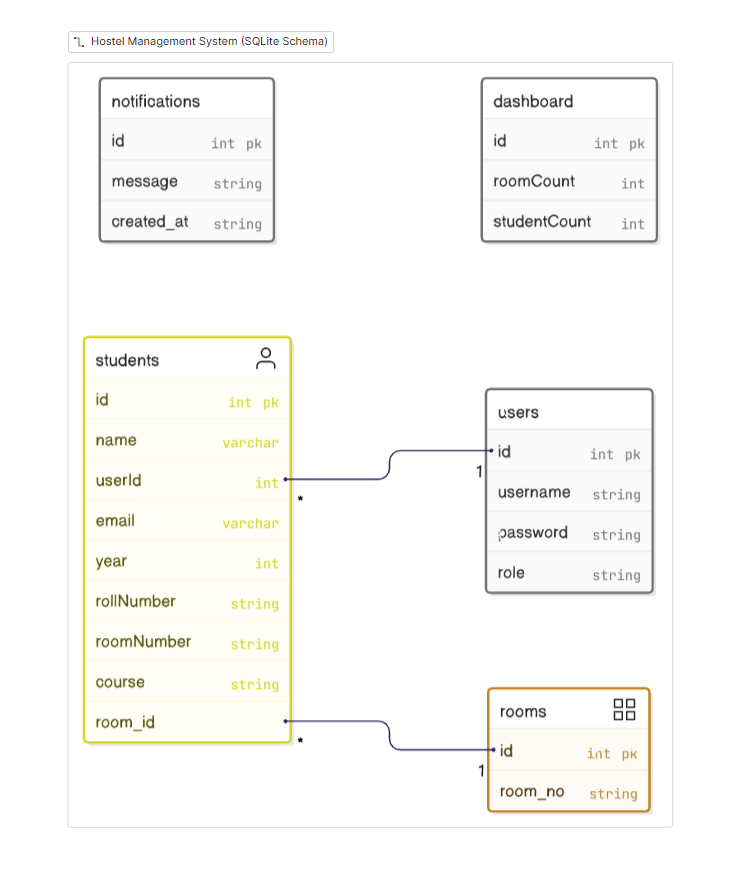
**Database (SQLite)**

* **Type**: Lightweight, file-based relational database
* **Used For**:
  + Student records
  + Room details and allocations
  + Admin user
  + Notices

SQLite is chosen for its simplicity and zero-configuration setup, making it ideal for local or small-scale deployments like college projects.

**3.2 ER Diagram**

This ER (Entity-Relationship) Diagram outlines how the core entities in the hostel management system interact, based on the actual table schema defined in your SQLite database setup.



**Entities and Attributes**

**1. users**

Represents both admin and student logins.

* id (PK)
* username (Unique)
* password
* role — either 'admin' or 'student'

**2. students**

Represents individual student profiles, linked to a user account.

* id (PK)
* userId (FK → users.id)
* name
* email
* rollNumber (Unique)
* roomNumber (text, not a FK)
* course
* year

**3. rooms**

Stores available room data.

* id (PK)
* room\_no (Unique)
* capacity
* occupied

**4. notifications**

Stores general notifications sent to students or admins.

* id (PK)
* message
* created\_at (auto timestamp in UTC format)

**5. dashboard**

Holds global counters (only one row with id = 1).

* id (PK, must be 1)
* roomCount
* studentCount

**Relationships**

* A **user** can be either an admin or a student via the role field.
* A **student** links to a **user** with students.userId → users.id (One-to-One).
* Room data (roomNumber in students) is **not directly relational**, but assumed to match rooms.room\_no (can be normalized later).
* Notifications and dashboard tables are **standalone** with no foreign keys.

**3.3 UML Diagrams**

This section outlines the core UML (Unified Modeling Language) diagrams used to represent system structure and behavior. The three main UML diagrams are: **Use Case Diagram**, **Class Diagram**, and **Activity Diagram**.

**3.3.1 Use Case Diagram**

This diagram illustrates the **interactions** between **users (Admin and Student)** and the system.

**Actors:**

* **Admin**
* **Student**

**Use Cases:**

| **Actor** | **Use Cases** |
| --- | --- |
| Admin | - Login- Add new room- View/update rooms- View students- Post notifications- Track dashboard stats |
| Student | - Register/Login- View profile- View room details- View notifications |

**Relationships**:

* Both actors interact with shared system components like notifications and room data.
* Admin has elevated privileges, including mutation operations.

**3.3.2 Class Diagram**

This diagram shows the **main classes/modules** and their **attributes** and **relationships** in the codebase.

Class: User

- id: number

- username: string

- password: string

- role: 'admin' | 'student'

Class: Student

- id: number

- userId: number

- name: string

- email: string

- rollNumber: string

- course: string

- year: number

- roomNumber: string

Class: Room

- id: number

- room\_no: string

- capacity: number

- occupied: number

Class: Notification

- id: number

- message: string

- created\_at: string

Class: Dashboard

- id: number

- roomCount: number

- studentCount: number

**Relationships**:

* Student → User (One-to-One via userId)
* No enforced FK between Student.roomNumber and Room.room\_no, but logically connected.

**3.3.3 Activity Diagram**

**Example: Student Login and Room Access Flow**

1. Start
2. Student submits login credentials
3. System validates credentials
   * If invalid → Show error and end
   * If valid → Check role
4. If role = student
5. Fetch associated student profile
6. Retrieve room details
7. Display dashboard view
8. End

This flow captures a common activity sequence from a student’s perspective.

**4. Technology Stack**

This project uses a modern full-stack TypeScript-based architecture with a focus on simplicity, modularity, and maintainability.

**4.1 Frontend (Client)**

* **React** (with TypeScript) – for building the user interface
* **Zustand** – lightweight state management
* **TailwindCSS** – utility-first CSS framework for responsive styling
* **Vite** – fast development server and bundler

**4.2 Backend (Server)**

* **Node.js** (with TypeScript) – server-side runtime
* **Express.js** – for building RESTful APIs
* **SQLite** (via better-sqlite3 ORM) – lightweight, file-based relational database
* **JWT** – for user authentication
* **Custom Middleware** – to handle role-based route protection

**4.3 Rationale for Selection**

* **TypeScript** across both frontend and backend ensures type safety and consistency.
* **SQLite** offers a simple setup suitable for small to medium-scale applications like a college hostel system.
* **TailwindCSS** allows rapid UI development with minimal custom styling.
* **Zustand** is chosen over Context API or Redux for its simplicity and performance in managing global state without boilerplate.

**5. Implementation Details**

This section describes how the Hostel Management System is organized in terms of directory layout and modular design.

**5.1 Project Structure**

The repository is structured into two main parts: client/ for the frontend and server/ for the backend.

hostel-management-system/

├── client/ # React + TypeScript frontend

└── server/ # Express + TypeScript backend with SQLite

**5.2 Frontend Structure**

client/

├── public/ # Static files and icons

├── src/

│ ├── components/ # Reusable UI components (Navbar, RoomCard, etc.)

│ ├── pages/ # Page-level components (Login, Dashboard, etc.)

│ ├── store/ # Zustand-based global state management

│ ├── lib/ # Utility functions and helpers

│ ├── types/ # TypeScript interfaces and type definitions

│ ├── App.tsx # Root component with routing

│ └── main.tsx # React entry point

├── tailwind.config.js # TailwindCSS configuration

├── postcss.config.js # PostCSS config

└── vite.config.ts # Vite build configuration

**Notable points:**

* **React Router** handles client-side routing.
* **Zustand** is used for managing state (like user data, auth status).
* **TailwindCSS** styles the application with utility-first classes.
* **API interactions** are encapsulated in the lib/ directory (api.ts, etc.).

**5.3 Backend Structure**

**server/**

**├── src/**

**│ ├── config/ # Configuration files**

**│ │ ├── constants.ts # Global constants**

**│ │ └── db.ts # SQLite DB initialization**

**│**

**│ ├── controllers/ # Route handler logic**

**│ │ ├── dashboard.ts**

**│ │ ├── notification.ts**

**│ │ ├── room.ts**

**│ │ ├── student.ts**

**│ │ └── user.ts**

**│**

**│ ├── middlewares/ # Express middleware (auth, etc.)**

**│ │ └── auth.ts**

**│**

**│ ├── models/ # Database access layer**

**│ │ ├── dashboard.ts**

**│ │ ├── notification.ts**

**│ │ ├── room.ts**

**│ │ ├── student.ts**

**│ │ └── user.ts**

**│**

**│ ├── routes/ # Express route definitions**

**│ │ ├── notification.ts**

**│ │ ├── room.ts**

**│ │ ├── router.ts # Aggregates all routes**

**│ │ ├── student.ts**

**│ │ └── user.ts**

**│**

**│ └── index.ts # Entry point for starting the server**

**│**

**├── hostel.db # SQLite database file**

**├── .env # Environment variables**

**├── package.json # NPM project metadata**

**├── tsconfig.json # TypeScript configuration**

**Key Highlights:**

* **controllers/ hold the business logic for each route.**
* **models/ manage DB operations for each entity.**
* **routes/ defines API endpoints, modularized by resource.**
* **middlewares/ like auth.ts handle access control.**
* **config/db.ts initializes tables and seeds the dashboard entry.**

**5.4 Some Important Code Snippets**

🡪 DB schema and config.

**/server/src/config/db.ts**

**import Database from "better-sqlite3";**

**const db = new Database("hostel.db");**

**// Tables**

**db.exec(`**

**CREATE TABLE IF NOT EXISTS users (**

**id INTEGER PRIMARY KEY AUTOINCREMENT,**

**username TEXT UNIQUE,**

**password TEXT,**

**role TEXT CHECK(role IN ('admin', 'student')) NOT NULL**

**);**

**CREATE TABLE IF NOT EXISTS rooms (**

**id INTEGER PRIMARY KEY AUTOINCREMENT,**

**room\_no TEXT UNIQUE,**

**capacity INTEGER,**

**occupied INTEGER**

**);**

**CREATE TABLE IF NOT EXISTS students (**

**id INTEGER PRIMARY KEY AUTOINCREMENT,**

**userId INTEGER NOT NULL,**

**name TEXT NOT NULL,**

**email TEXT NOT NULL,**

**rollNumber TEXT UNIQUE NOT NULL,**

**roomNumber TEXT,**

**course TEXT,**

**year INTEGER,**

**FOREIGN KEY(userId) REFERENCES users(id)**

**);**

**CREATE TABLE IF NOT EXISTS notifications (**

**id INTEGER PRIMARY KEY AUTOINCREMENT,**

**message TEXT,**

**created\_at TEXT DEFAULT (strftime('%Y-%m-%dT%H:%M:%fZ', 'now'))**

**);**

**CREATE TABLE IF NOT EXISTS dashboard (**

**id INTEGER PRIMARY KEY CHECK (id = 1),**

**roomCount INTEGER DEFAULT 0,**

**studentCount INTEGER DEFAULT 0**

**);**

**INSERT OR IGNORE INTO dashboard (id, roomCount, studentCount) VALUES (1, 0, 0);**

**`);**

**export default db;**

🡪 server constants.

**/server/drc/config/constant.ts**

**import dotenv from 'dotenv';**

**dotenv.config();**

**export const envvars = Object.freeze({**

**PORT: 4040,**

**JWT\_SECRET: process.env.JWT\_SECRET as string,**

**});**

🡪 authentication middlewares

**/server/src/middlewares**

**import { NextFunction, Request, Response } from "express";**

**import jwt, { type JwtPayload } from "jsonwebtoken";**

**import { envvars } from "../config/constants";**

**import { userOps, type User } from "../models/user";**

**// adding user to request**

**declare global {**

**namespace Express {**

**interface Request {**

**user?: User;**

**}**

**}**

**}**

**const authAdminOnly = (req: Request, res: Response, next: NextFunction) => {**

**const authHeader = req.headers["authorization"];**

**if (!authHeader || !authHeader.startsWith("Bearer ")) {**

**res.status(401).json({ message: "No token provided" });**

**return;**

**}**

**const token = authHeader.split(" ")[1];**

**try {**

**const decoded = jwt.verify(token, envvars.JWT\_SECRET);**

**if (typeof decoded === "string" || !decoded.user) {**

**res.status(401).json({ message: "No token provided" });**

**return;**

**}**

**// Attach user to request for downstream use**

**const userFromToken = (decoded as JwtPayload).user as User;**

**const user = userOps.getUserByUsername(userFromToken.username) as User;**

**if (user.role !== "admin") {**

**res.status(401).json({ message: "Not an admin!" });**

**return;**

**}**

**req.user = user;**

**next();**

**} catch (err) {**

**res.status(401).json({ message: "Invalid token" });**

**return;**

**}**

**};**

**const auth = (req: Request, res: Response, next: NextFunction) => {**

**const authHeader = req.headers["authorization"];**

**if (!authHeader || !authHeader.startsWith("Bearer ")) {**

**res.status(401).json({ message: "No token provided" });**

**return;**

**}**

**const token = authHeader.split(" ")[1];**

**try {**

**const decoded = jwt.verify(token, envvars.JWT\_SECRET);**

**if (typeof decoded === "string" || !decoded.user) {**

**res.status(401).json({ message: "No token provided" });**

**return;**

**}**

**// Attach user to request for downstream use**

**const userFromToken = (decoded as JwtPayload).user as User;**

**const user = userOps.getUserByUsername(userFromToken.username) as User;**

**if (user.role !== "student" && user.role !== "admin") {**

**res.status(401).json({ message: "Not an good user!" });**

**return;**

**}**

**req.user = user;**

**next();**

**} catch (err) {**

**res.status(401).json({ message: "Invalid token" });**

**return;**

**}**

**};**

**export default {**

**authAdminOnly,**

**auth,**

**};**

🡪 REST API endpoint routes.

**/server/src/routes/router.ts**

**import { Router, type Request, type Response } from "express";**

**import notificationRoutes from "./notification";**

**import roomRoutes from "./room";**

**import studentRoutes from "./student";**

**import userRoutes from "./user";**

**import { getDashboard } from "../controllers/dashboard";**

**import auth from "../middlewares/auth";**

**const router = Router();**

**router.use("/users", userRoutes);**

**router.use("/students", studentRoutes);**

**router.use("/rooms", roomRoutes);**

**router.use("/notifications", notificationRoutes);**

**router.get("/dashboard", auth.authAdminOnly, getDashboard);**

**router.get("/", (req: Request, res: Response) => {**

**res.status(200).json({**

**success: true,**

**message: "API is working!",**

**data: null,**

**});**

**});**

**export default router;**

🡪 Main server file and listener

**/server/src/index.ts**

**import express from "express";**

**import cors from "cors";**

**import router from "./routes/router";**

**import { envvars } from "./config/constants";**

**const PORT = envvars.PORT || 4000;**

**const app = express();**

**app.use(cors());**

**app.use(express.json());**

**app.use("/api", router);**

**app.listen(PORT, () => {**

**console.log(`Running at ${PORT}`)**

**})**

🡪 Client side routes

**/client/src/App.jsx**

**import { useEffect, useState } from "react";**

**import { BrowserRouter, Navigate, Route, Routes } from "react-router";**

**import { getStudentById, verifyUser } from "./apiClient";**

**import Layout from "./layouts/Layout";**

**import DashBoard from "./pages/DashBoard";**

**import Login from "./pages/Login";**

**import Notification from "./pages/Notification";**

**import Room from "./pages/Room";**

**import Student from "./pages/Student";**

**import { useUserStore } from "./store/store";**

**import { Toaster } from "./components/ui/sonner";**

**const App = () => {**

**const setUser = useUserStore((s) => s.login);**

**const isAuth = useUserStore((s) => s.isAuth);**

**const user = useUserStore((s) => s.user);**

**const [loadingUser, setLoadingUser] = useState<boolean>(true);**

**useEffect(() => {**

**const token = localStorage.getItem("token");**

**if (!token) {**

**return;**

**}**

**(async () => {**

**const res = await verifyUser();**

**if (res?.data?.success) {**

**const { user, token: verifiedToken } = res.data.data;**

**let student = null;**

**if (user.role === "student") {**

**const studentRes = await getStudentById(user.id);**

**student = studentRes?.data?.data || null;**

**}**

**setUser({ user, token: verifiedToken, student });**

**setLoadingUser(false);**

**}**

**})();**

**}, [setUser]);**

**return (**

**<div>**

**<div className="max-w-7xl w-full mx-auto px-6">**

**<BrowserRouter>**

**<Routes>**

**<Route**

**element={**

**<Layout>**

**{isAuth ? (**

**<DashBoard />**

**) : (**

**<Navigate to={"/login"} />**

**)}**

**</Layout>**

**}**

**path="/"**

**/>**

**{isAuth && user?.role === "admin" && (**

**<Route**

**element={**

**<Layout>**

**<Student />**

**</Layout>**

**}**

**path="/students"**

**/>**

**)}**

**{isAuth && user?.role === "admin" && (**

**<Route**

**element={**

**<Layout>**

**<Room />**

**</Layout>**

**}**

**path="/rooms"**

**/>**

**)}**

**<Route**

**element={**

**<Layout>**

**{isAuth ? (**

**<Notification />**

**) : (**

**<Navigate to={"/login"} />**

**)}**

**</Layout>**

**}**

**path="/notifications"**

**/>**

**<Route element={<Login />} path="/login" />**

**<Route**

**element={**

**<Layout>**

**<h2 className="text-2xl">**

**Page Not Found!**

**</h2>**

**</Layout>**

**}**

**path="\*"**

**/>**

**</Routes>**

**</BrowserRouter>**

**<Toaster  />**

**</div>**

**</div>**

**);**

**};**

**export default App;**

🡪 API calls

**/client/src/apiClient.ts**

**import { toast } from "sonner";**

**import api from "./lib/axios";**

**import {**

**useNotificationStore,**

**useRoomStore,**

**useStudentStore,**

**} from "./store/store";**

**// --- User/Auth ---**

**export const login = async (username: string, password: string) => {**

**try {**

**return await api.post("/users/login", { username, password });**

**} catch (err: any) {**

**toast.error(**

**err?.response?.data?.message || err.message || "Login failed"**

**);**

**return null;**

**}**

**};**

**export const verifyUser = async () => {**

**try {**

**return await api.get("/users/verify");**

**} catch (err: any) {**

**toast.error(**

**err?.response?.data?.message || err.message || "Verification failed"**

**);**

**return null;**

**}**

**};**

**// --- Students ---**

**export const createStudent = async (student: {**

**name: string;**

**email: string;**

**rollNumber: string;**

**roomNumber?: string;**

**course?: string;**

**year?: string;**

**}) => {**

**try {**

**return await api.post("/students", student);**

**} catch (err: any) {**

**toast.error(**

**err?.response?.data?.message ||**

**err.message ||**

**"Create student failed"**

**);**

**return null;**

**}**

**};**

**export const getAllStudents = async () => {**

**try {**

**const res = await api.get("/students");**

**if (res.data.success) {**

**useStudentStore.getState().setStudents(res.data.data);**

**}**

**return res.data;**

**} catch (err: any) {**

**toast.error(**

**err?.response?.data?.message ||**

**err.message ||**

**"Fetch students failed"**

**);**

**return null;**

**}**

**};**

**export const getStudentById = async (id: number) => {**

**try {**

**return await api.get(`/students/${id}`);**

**} catch (err: any) {**

**toast.error(**

**err?.response?.data?.message ||**

**err.message ||**

**"Fetch student failed"**

**);**

**return null;**

**}**

**};**

**export const updateStudent = async (**

**id: number,**

**student: {**

**name?: string;**

**email?: string;**

**rollNumber?: string;**

**roomNumber?: string;**

**course?: string;**

**year?: string;**

**}**

**) => {**

**try {**

**return await api.put(`/students/${id}`, student);**

**} catch (err: any) {**

**toast.error(**

**err?.response?.data?.message ||**

**err.message ||**

**"Update student failed"**

**);**

**return null;**

**}**

**};**

**export const deleteStudent = async (id: number) => {**

**try {**

**return await api.delete(`/students/${id}`);**

**} catch (err: any) {**

**toast.error(**

**err?.response?.data?.message ||**

**err.message ||**

**"Delete student failed"**

**);**

**return null;**

**}**

**};**

**export const getStudentRoom = async (studentId: number) => {**

**try {**

**return await api.get(`/students/${studentId}/room`);**

**} catch (err: any) {**

**toast.error(**

**err?.response?.data?.message ||**

**err.message ||**

**"Fetch student room failed"**

**);**

**return null;**

**}**

**};**

**// --- Rooms ---**

**export const createRoom = async (room: {**

**room\_no: string;**

**capacity: number;**

**occupied: number;**

**}) => {**

**try {**

**const res = await api.post("/rooms", room);**

**if (res.data.success) {**

**useRoomStore.getState().addRoom(res.data.data);**

**}**

**return res.data;**

**} catch (err: any) {**

**toast.error(**

**err?.response?.data?.message || err.message || "Create room failed"**

**);**

**return null;**

**}**

**};**

**export const getAllRooms = async () => {**

**try {**

**const res = await api.get("/rooms");**

**if (res.data.success) {**

**useRoomStore.getState().setRooms(res.data.data);**

**}**

**return res.data;**

**} catch (err: any) {**

**toast.error(**

**err?.response?.data?.message || err.message || "Fetch rooms failed"**

**);**

**return null;**

**}**

**};**

**export const updateRoom = async (**

**id: number,**

**room: {**

**room\_no?: string;**

**capacity?: number;**

**occupied?: number;**

**}**

**) => {**

**try {**

**const res = await api.put(`/rooms/${id}`, room);**

**return res.data;**

**} catch (err: any) {**

**toast.error(**

**err?.response?.data?.message || err.message || "Update room failed"**

**);**

**return null;**

**}**

**};**

**export const deleteRoom = async (id: number) => {**

**try {**

**return await api.delete(`/rooms/${id}`);**

**} catch (err: any) {**

**toast.error(**

**err?.response?.data?.message || err.message || "Delete room failed"**

**);**

**return null;**

**}**

**};**

**// --- Notifications ---**

**export const createNotification = async (message: string) => {**

**try {**

**const res = await api.post("/notifications", { message });**

**if (res.data.success) {**

**useNotificationStore.getState().addNotification(res.data.data);**

**}**

**return res.data;**

**} catch (err: any) {**

**toast.error(**

**err?.response?.data?.message ||**

**err.message ||**

**"Create notification failed"**

**);**

**return null;**

**}**

**};**

**export const deleteNotification = async (id: number) => {**

**try {**

**return await api.delete(`/notifications/${id}`);**

**} catch (err: any) {**

**toast.error(**

**err?.response?.data?.message ||**

**err.message ||**

**"Delete notification failed"**

**);**

**return null;**

**}**

**};**

**export const getNotifications = async (skip = 0, limit = 10) => {**

**try {**

**const res = await api.get("/notifications", {**

**params: { skip, limit },**

**});**

**if (res.data.success) {**

**useNotificationStore.getState().setNotifications(res.data.data);**

**}**

**return res.data;**

**} catch (err: any) {**

**toast.error(**

**err?.response?.data?.message ||**

**err.message ||**

**"Fetch notifications failed"**

**);**

**return null;**

**}**

**};**

**export const getDashboard = async () => {**

**try {**

**const res = await api.get("/dashboard");**

**return res.data;**

**} catch (err: any) {**

**toast.error(err?.response?.data?.message || err.message || "Fetch dashboard failed");**

**return null;**

**}**

**};**

🡪 Client side state management (Zustand)

**/client/src/store.ts**

**import { create } from "zustand";**

**// User interface (reference from server model)**

**type User = {**

**id: number;**

**username: string;**

**role: string; // 'admin' | 'student'**

**};**

**// Student interface (reference from server model)**

**type Student = {**

**id?: number;**

**userId: number;**

**name: string;**

**email: string;**

**rollNumber: string;**

**roomNumber?: string;**

**course?: string;**

**year?: number;**

**};**

**// Room interface (reference from server model)**

**type Room = {**

**id?: number;**

**room\_no: string;**

**capacity: number;**

**occupied: number;**

**};**

**// Notification interface (reference from server model)**

**type Notification = {**

**id?: number;**

**message: string;**

**created\_at?: string;**

**};**

**// User Store**

**interface UserStore {**

**isAuth: boolean;**

**user: User | null;**

**token: string | null;**

**student: Student | null;**

**login: (data: { user: User; token: string; student: Student }) => void;**

**logout: () => void;**

**}**

**export const useUserStore = create<UserStore>((set) => ({**

**isAuth: false,**

**user: null,**

**token: null,**

**student: null,**

**login: ({ user, token, student }) =>**

**set({ user, token, student, isAuth: true }),**

**logout: () =>**

**set({ user: null, token: null, student: null, isAuth: false }),**

**}));**

**// Student Store**

**interface StudentStore {**

**students: Student[];**

**setStudents: (students: Student[]) => void;**

**addStudent: (student: Student) => void;**

**updateStudent: (student: Student) => void;**

**removeStudent: (id: number) => void;**

**}**

**export const useStudentStore = create<StudentStore>((set) => ({**

**students: [],**

**setStudents: (students) => set({ students }),**

**addStudent: (student) =>**

**set((state) => ({ students: [...state.students, student] })),**

**updateStudent: (student) =>**

**set((state) => ({**

**students: state.students.map((s) =>**

**s.id === student.id ? student : s**

**),**

**})),**

**removeStudent: (id) =>**

**set((state) => ({**

**students: state.students.filter((s) => s.id !== id),**

**})),**

**}));**

**// Room Store**

**interface RoomStore {**

**rooms: Room[];**

**setRooms: (rooms: Room[]) => void;**

**addRoom: (room: Room) => void;**

**updateRoom: (room: Room) => void;**

**removeRoom: (id: number) => void;**

**}**

**export const useRoomStore = create<RoomStore>((set) => ({**

**rooms: [],**

**setRooms: (rooms) => set({ rooms }),**

**addRoom: (room) => set((state) => ({ rooms: [...state.rooms, room] })),**

**updateRoom: (room) =>**

**set((state) => ({**

**rooms: state.rooms.map((r) => (r.id === room.id ? room : r)),**

**})),**

**removeRoom: (id) =>**

**set((state) => ({ rooms: state.rooms.filter((r) => r.id !== id) })),**

**}));**

**// Notification Store**

**interface NotificationStore {**

**notifications: Notification[];**

**setNotifications: (notifications: Notification[]) => void;**

**addNotification: (notification: Notification) => void;**

**removeNotification: (id: number) => void;**

**}**

**export const useNotificationStore = create<NotificationStore>((set) => ({**

**notifications: [],**

**setNotifications: (notifications) => set({ notifications }),**

**addNotification: (notification) =>**

**set((state) => ({**

**notifications: [notification, ...state.notifications],**

**})),**

**removeNotification: (id) =>**

**set((state) => ({**

**notifications: state.notifications.filter((n) => n.id !== id),**

**})),**

**}));**

**6. Testing**

Software testing is a crucial step in the development lifecycle to ensure the system functions as intended and is free from major defects. For the Hostel Management System, both manual testing and component-level testing were performed to validate functionality, usability, and reliability.

**6.1 Testing Approach**

The testing approach followed in this project includes the following phases:

6.1.1 Unit Testing

* Individual components and functions were tested in isolation.
* For example, student registration logic, room assignment validations, and notification rendering were unit tested.

6.1.2 Integration Testing

* Interactions between modules such as frontend API calls and backend routes were tested.
* Ensured correct data flow and handling between UI, server, and database.

6.1.3 Manual Testing

* Each feature was manually tested by acting as different users (admin/student).
* Edge cases such as duplicate entries, empty fields, invalid input formats were tested.

6.1.4 API Testing

* Backend endpoints were tested using tools like Postman to verify request-response cycles.
* All CRUD operations for rooms, students, users, and notifications were validated.

**6.2 Sample Test Cases**

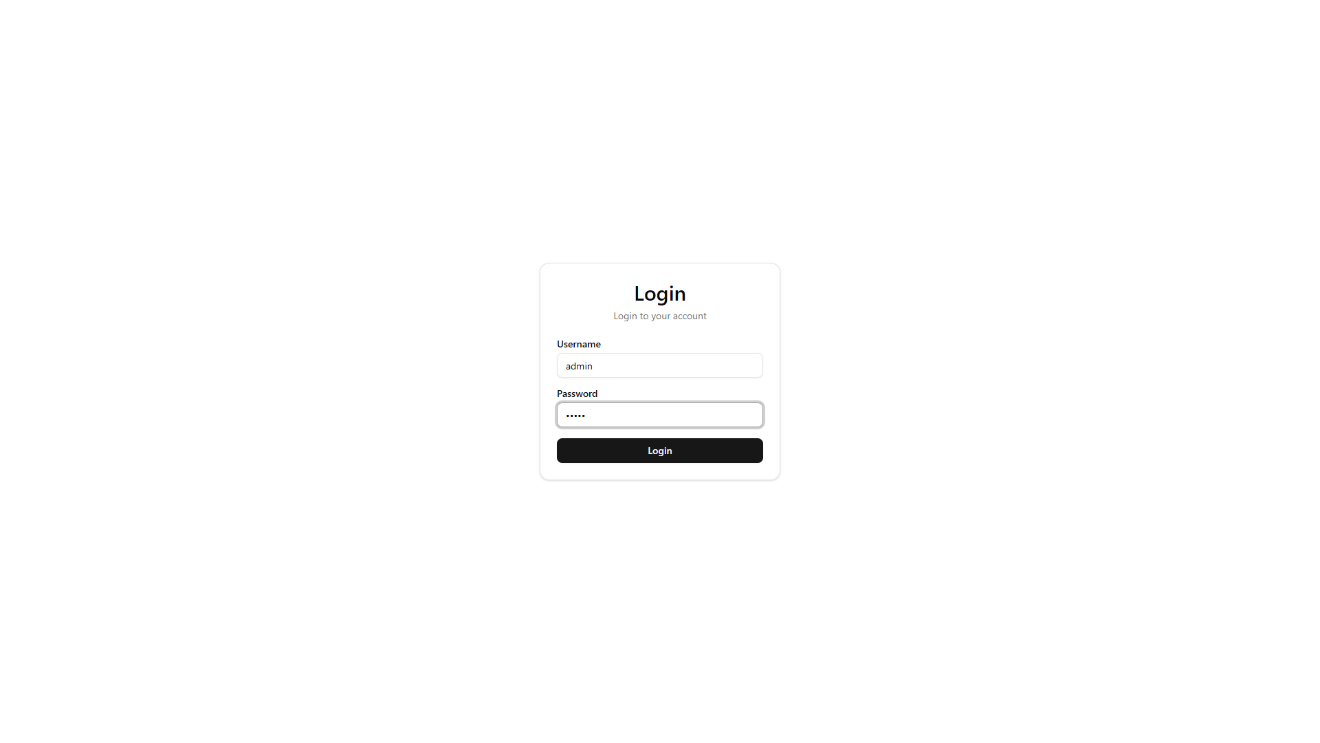
| **Test Case ID** | **Description** | **Input** | **Expected Output** | **Result** |
| --- | --- | --- | --- | --- |
| **TC001** | **Login with correct credentials** | **Valid username & password** | **Redirect to dashboard** | **Pass** |
| **TC002** | **Login with incorrect credentials** | **Wrong password** | **Error message displayed** | **Pass** |
| **TC003** | **Add new student with valid data** | **Name, Email, Roll, Room, etc.** | **Student added successfully** | **Pass** |
| **TC004** | **Add student with duplicate roll number** | **Existing roll number** | **Validation error shown** | **Pass** |
| **TC005** | **Add room with missing capacity** | **Room number but empty capacity** | **Error message displayed** | **Pass** |
| **TC006** | **View notifications as student** | **Logged-in student user** | **List of messages shown** | **Pass** |
| **TC007** | **Update student info as admin** | **Modified student details** | **Info updated in database** | **Pass** |
| **TC008** | **Assign more students than room capacity** | **Capacity = 2, Adding 3 students** | **Restriction enforced** | **Pass** |
| **TC009** | **Navigate through responsive layout** | **Resize screen / mobile preview** | **Layout adjusts correctly** | **Pass** |

**7. Output Screenshots**

This section displays key interface visuals of the Hostel Management System from both the **admin** and **student** perspectives. These screenshots demonstrate how different components of the system function in a real usage context.

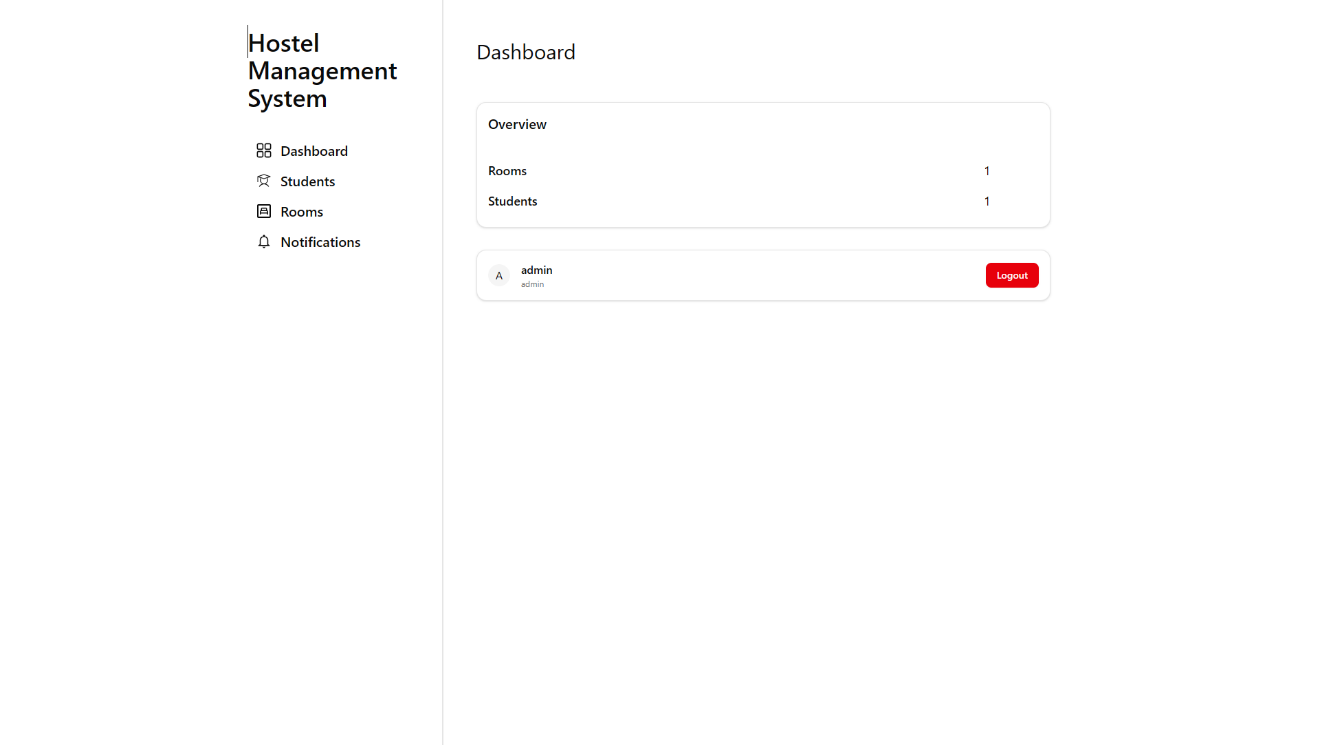
**7.1 Admin Login**

* **Description:** Login interface where admin and students enter credentials to access the system.



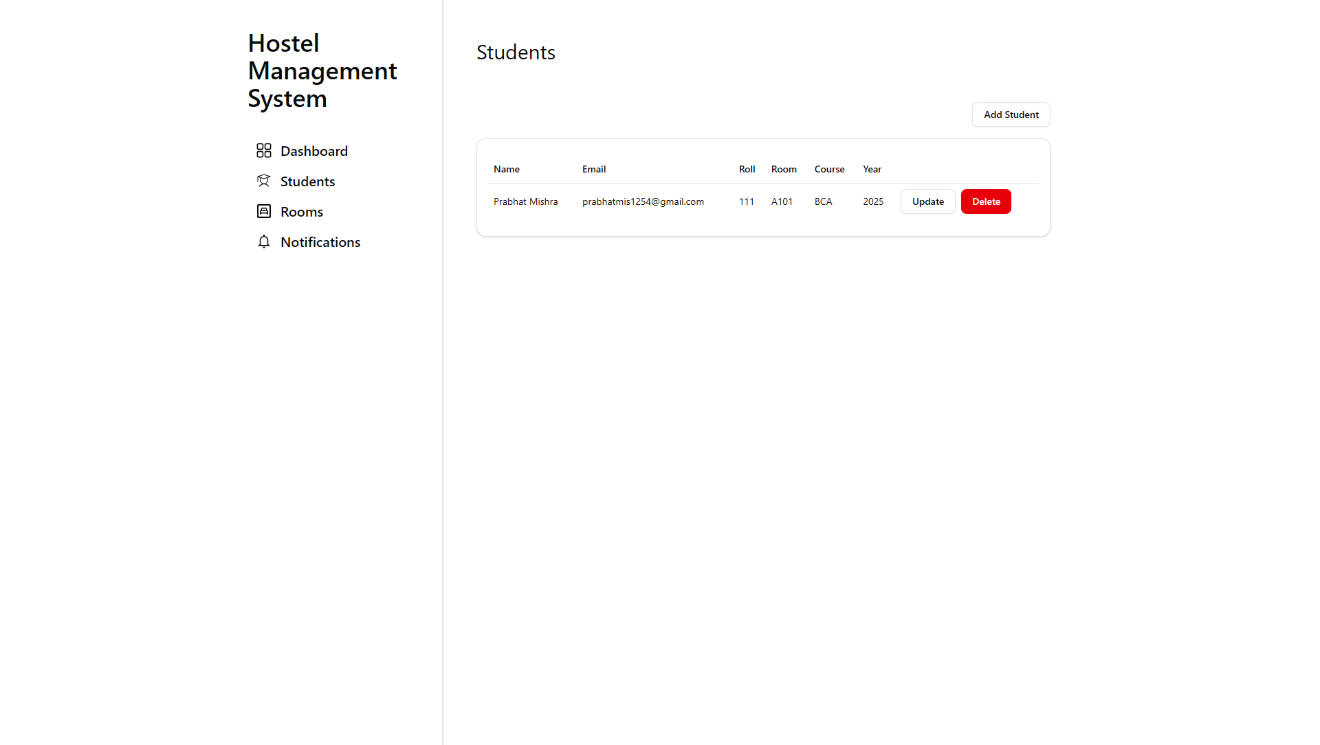
**7.2 Admin Dashboard**

* **Description:** Overview of total rooms and students, serving as the central hub for admin operations.

****

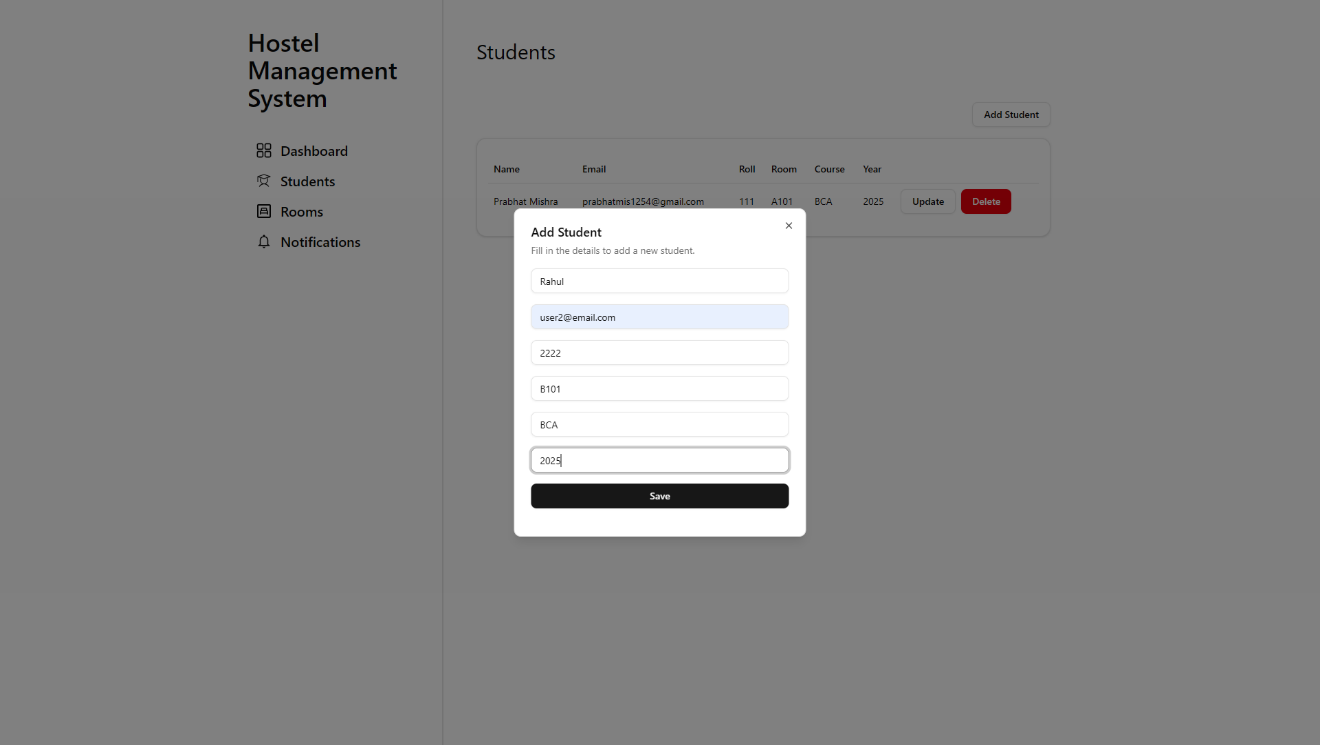
**7.3 Student List View**

* **Description:** Displays all registered students with details like name, roll number, room, and course.

****

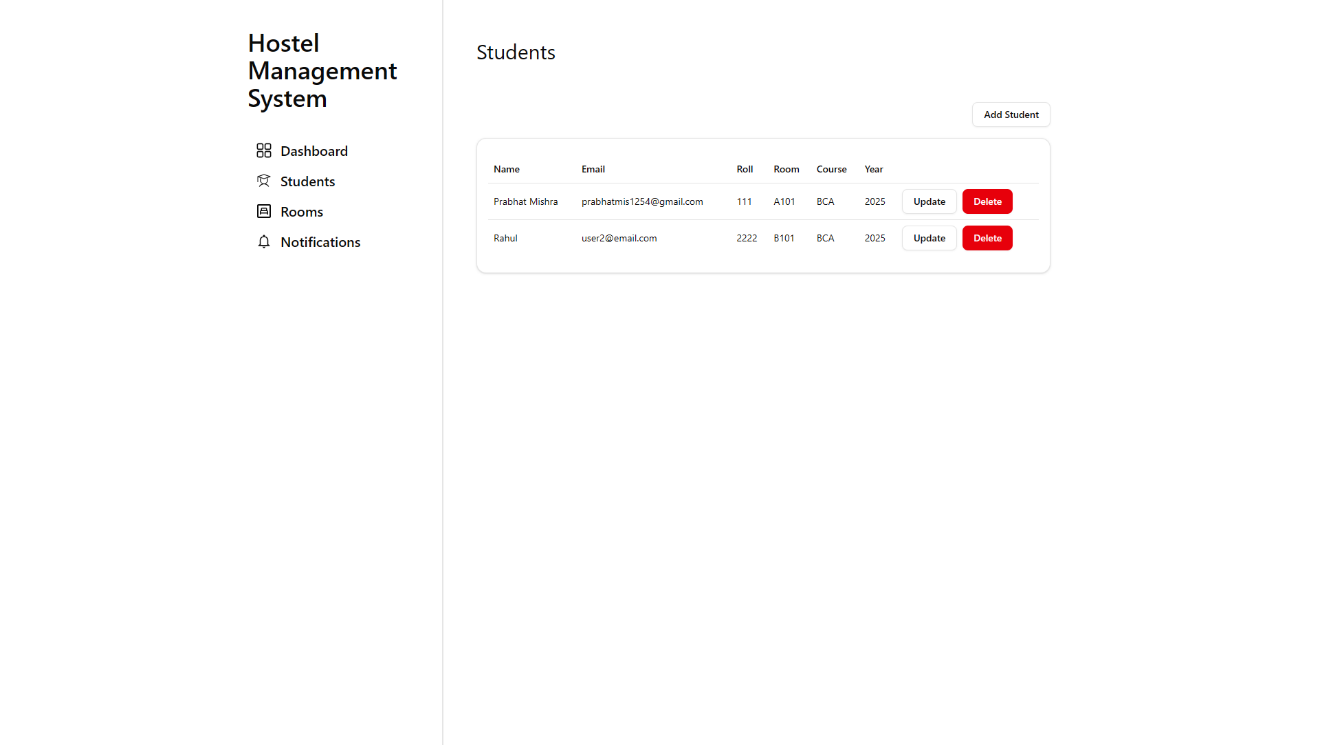
**7.4 Add Student Form**

* **Description:** Admin interface to add a new student and assign a room (optional at the time of creation).

****

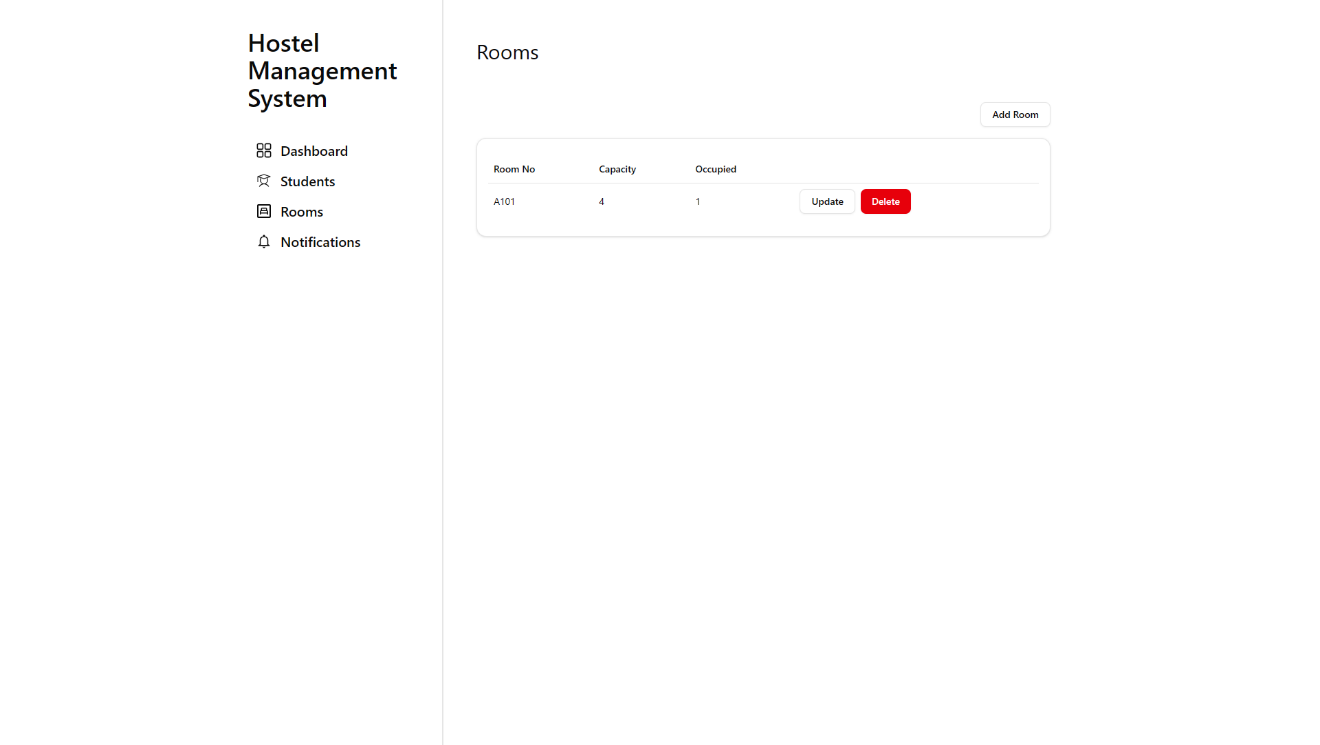
**7.5 Updated Student List After Addition**

* **Description:** Reflects the student list after successfully adding a new student.

****

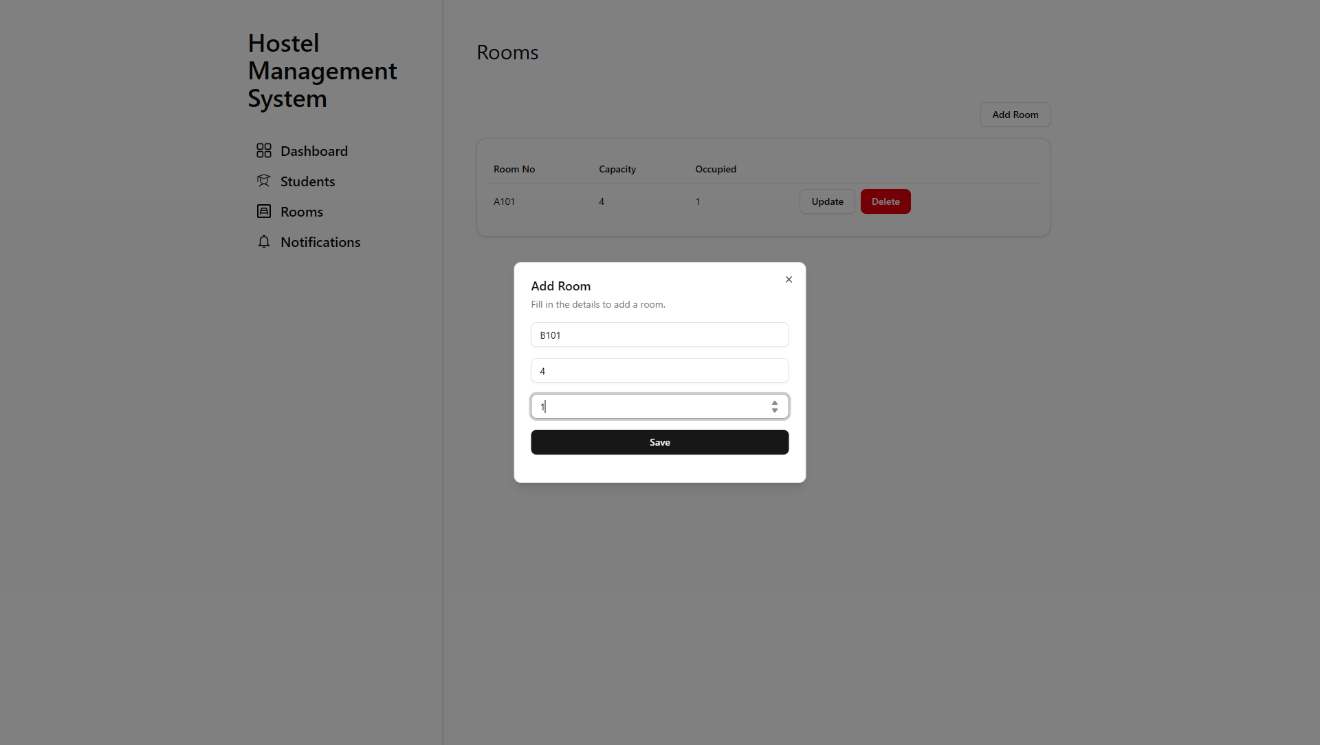
**7.6 Room List View**

* **Description:** Lists all hostel rooms with capacity and current occupancy status.

****

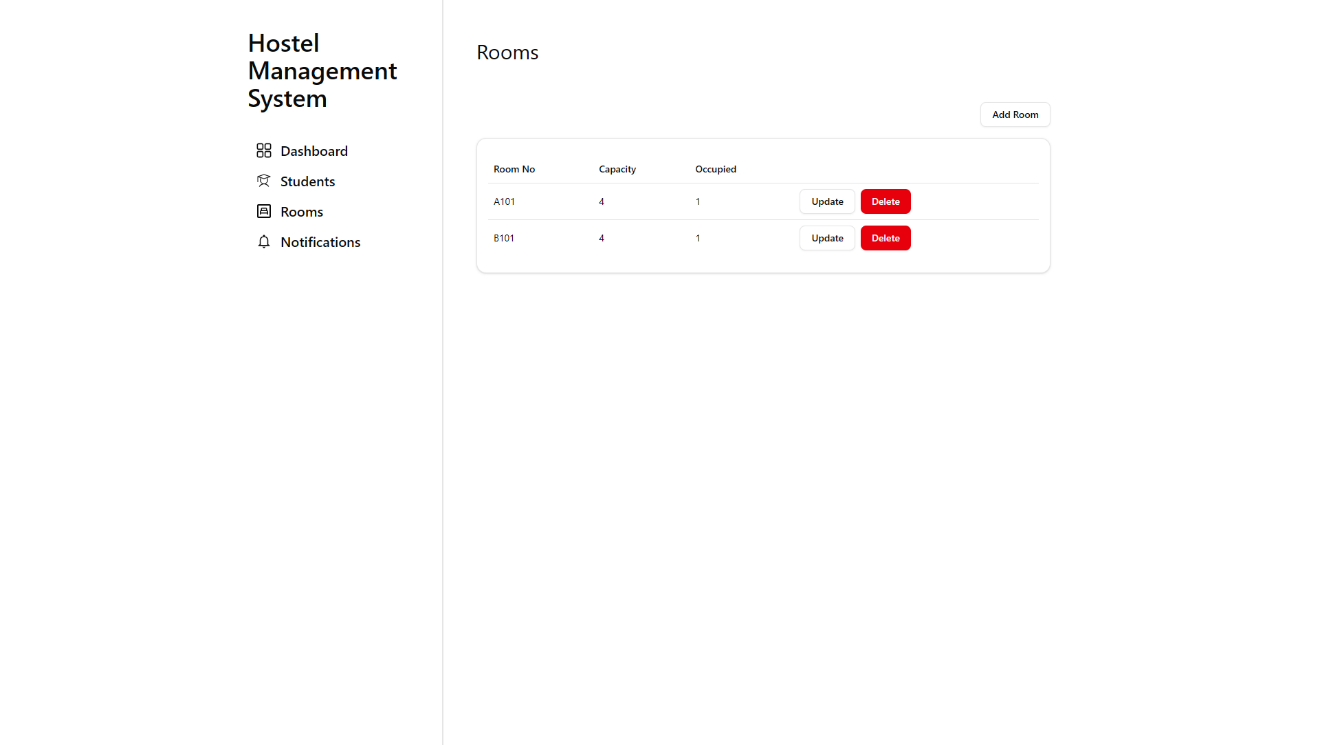
**7.7 Add Room Form**

* **Description:** Allows admin to add a new room by entering room number and capacity.

****

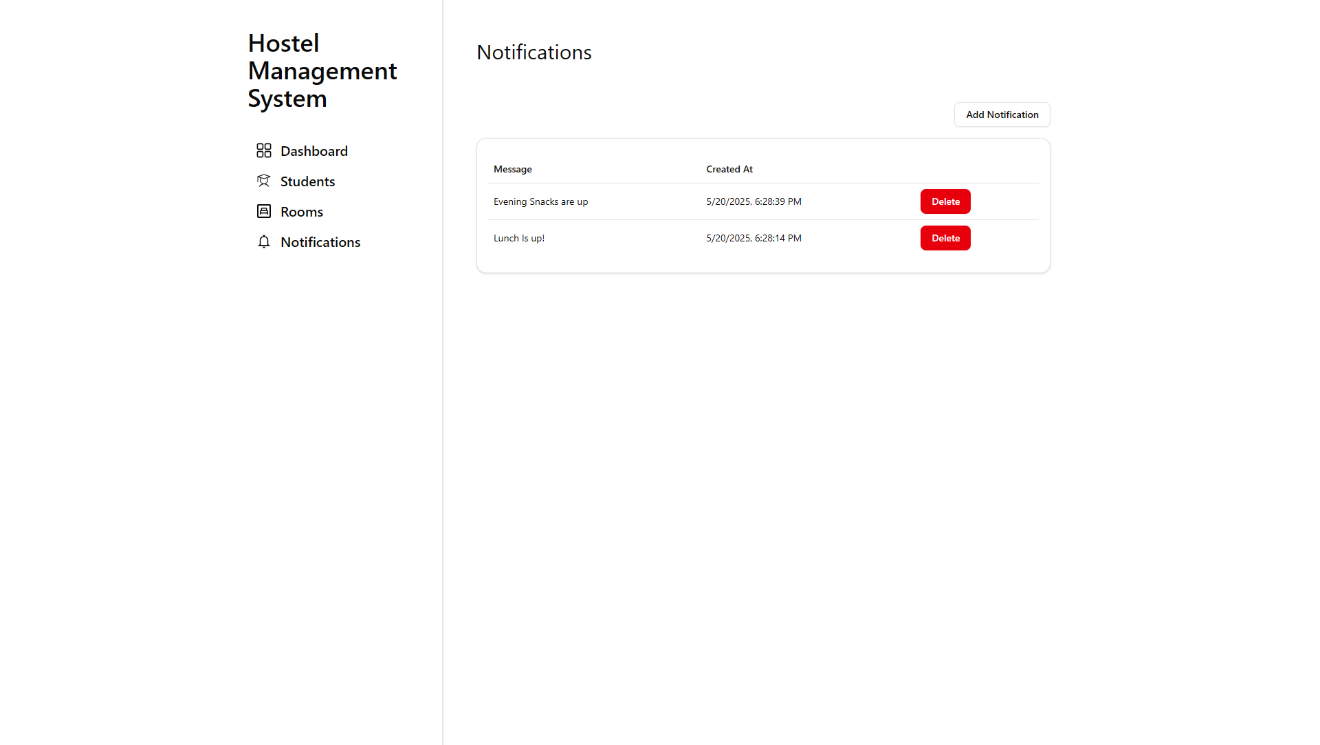
**7.8 Updated Room List After Addition**

* **Description:** Shows the room list after a new room has been successfully added.

****

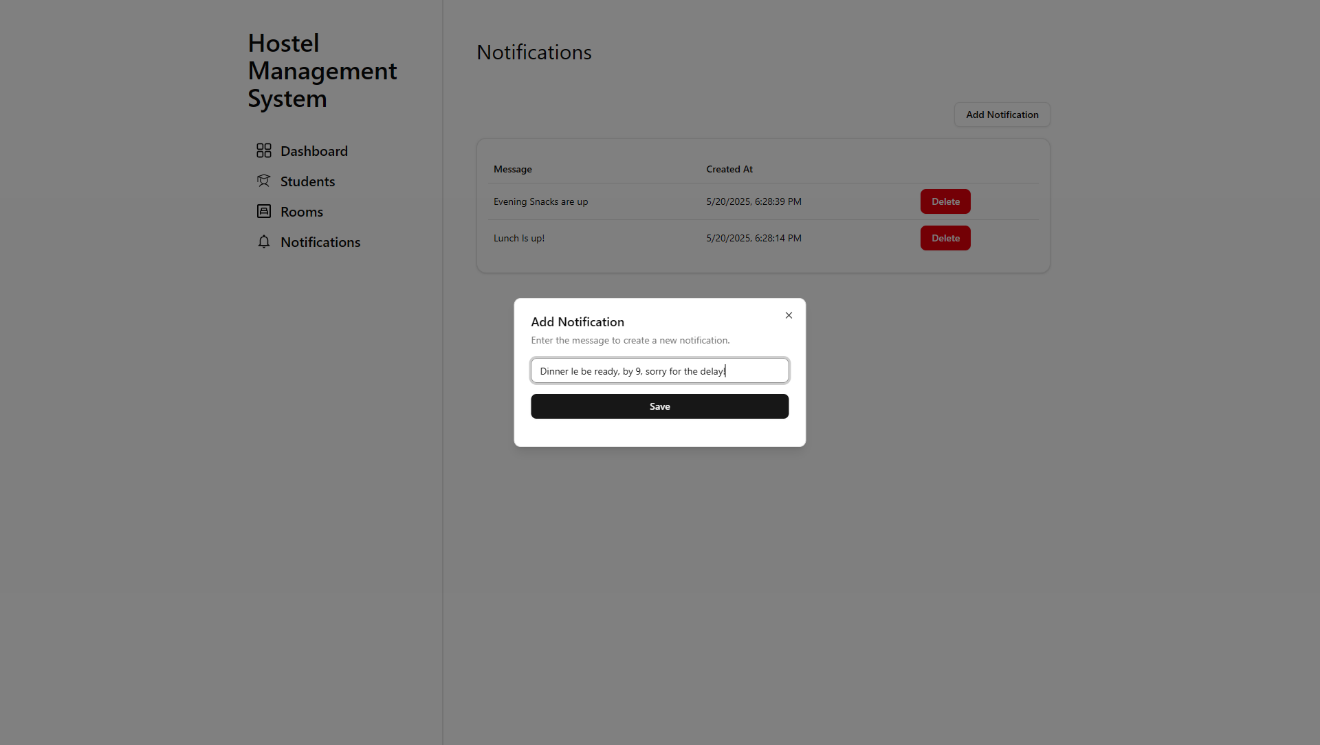
**7.9 Notifications Panel (Admin)**

* **Description:** Interface for admin to view and manage posted notifications.

****

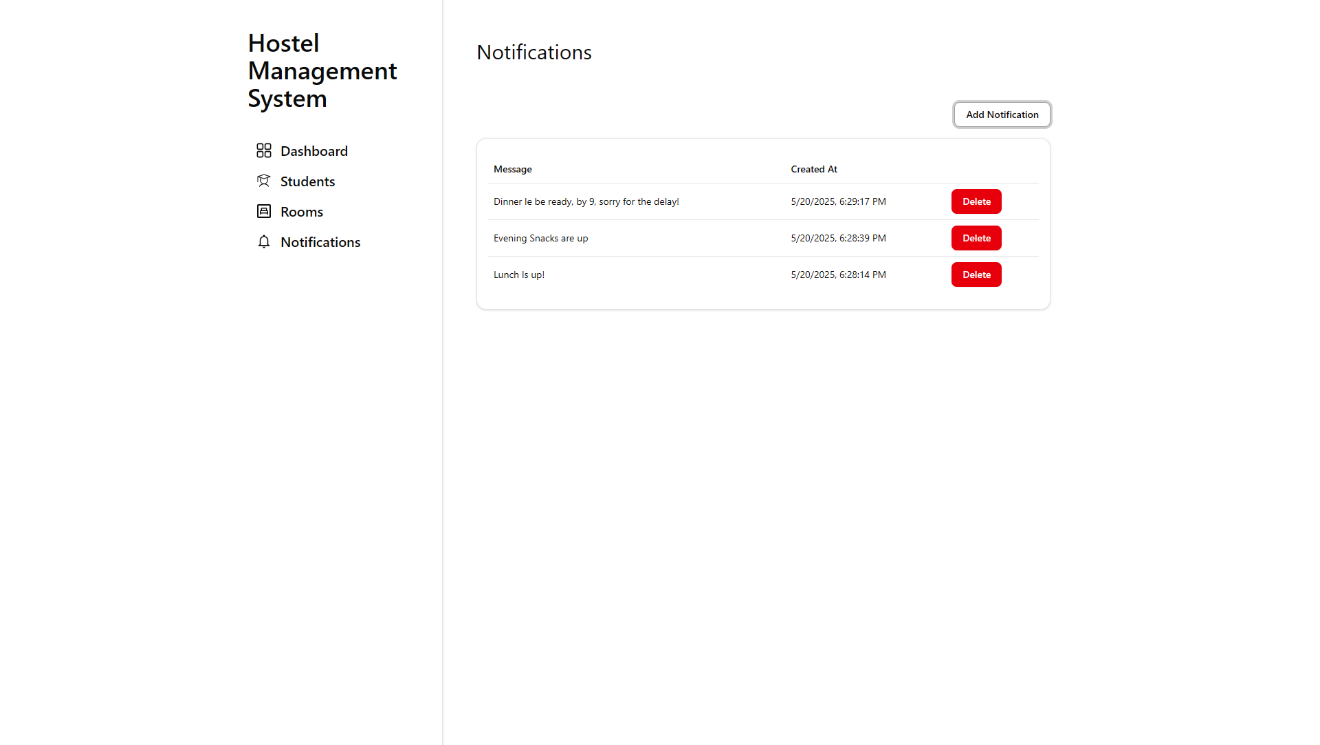
**7.10 Add Notification**

* **Description:** Admin screen to draft and publish new notifications for students.

****

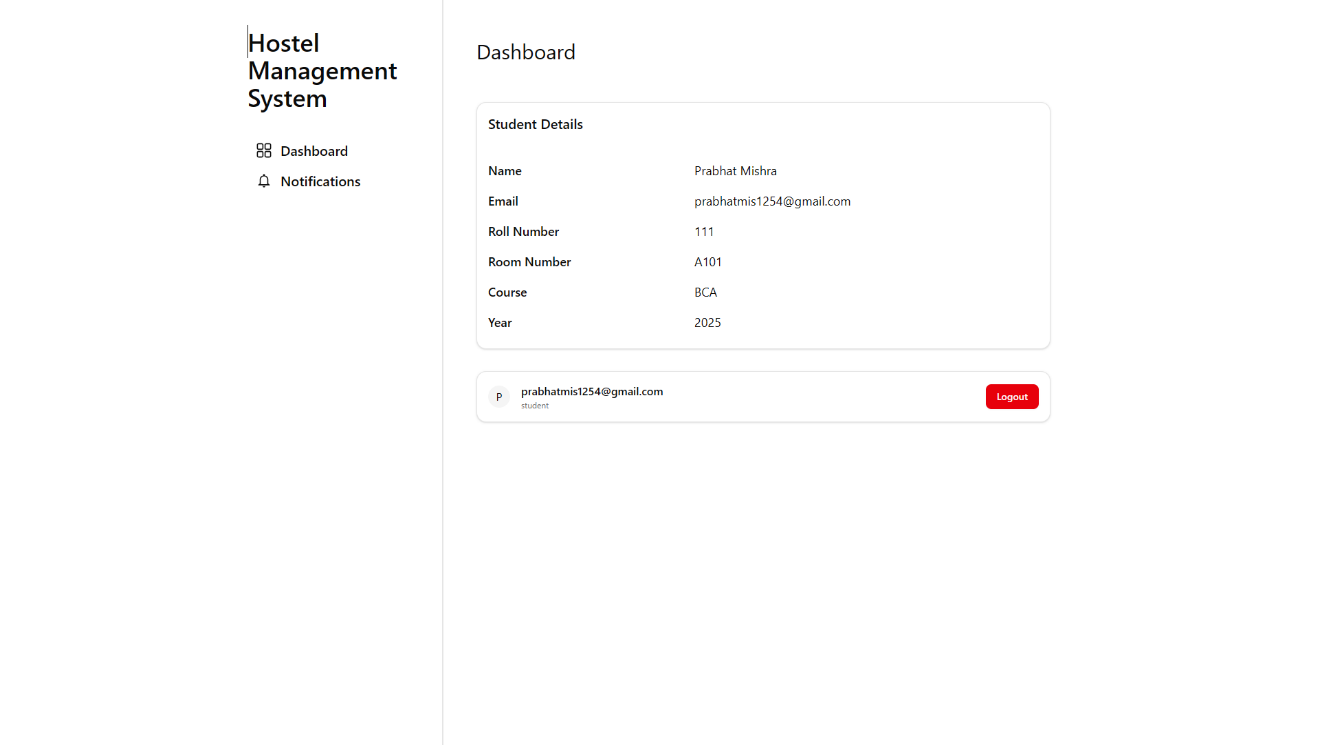
**7.11 Notifications After Posting**

* **Description:** Displays the notifications list with the newly added message.

****

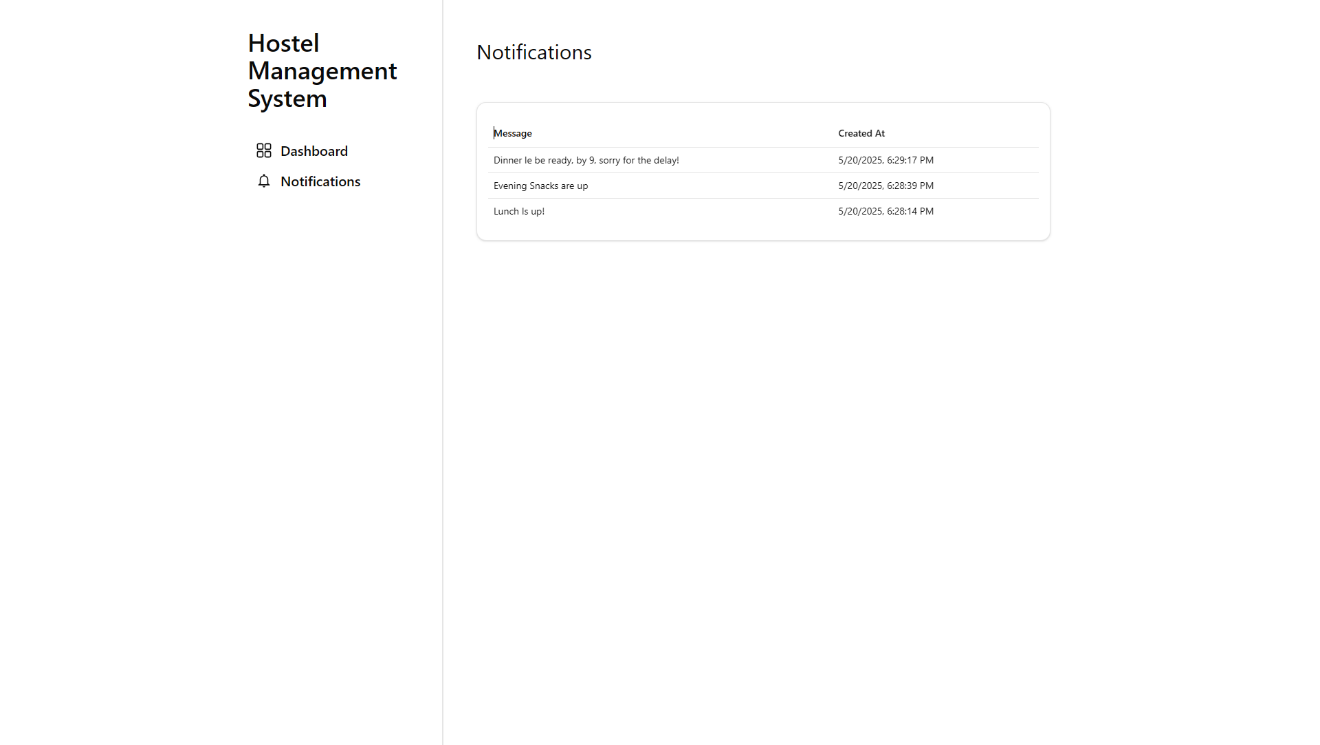
**7.12 Student Dashboard**

* **Description:** Student view showing personal and room details upon login.

****

**7.13 Student Notification View**

* **Description:** Students can read notifications posted by admin in real time.

****

**8. Challenges Faced & Solutions**

During the development of the Hostel Management System, several technical and functional challenges were encountered. This section outlines the key issues along with the approaches taken to resolve them.

**8.1 Managing Relational Data in SQLite**

* **Challenge:** Storing and managing relational data such as user-student mapping and room assignments using SQLite, which has limited relational features compared to full-scale RDBMS.
* **Solution:** Carefully designed schema with foreign keys (e.g., students.userId → users.id) and unique constraints. Manual validation and joins were implemented in logic to maintain referential integrity.

**8.2 Handling Room Capacity and Occupancy**

* **Challenge:** Ensuring rooms are not over-assigned when adding or updating student data.
* **Solution:** Logic was added to check the current occupancy of a room before assigning it to a new student. The rooms table maintains occupied count, which is updated transactionally.

**8.3 Implementing Role-Based Access**

* **Challenge:** Restricting access to APIs and dashboards based on user roles (admin, student).
* **Solution:** Middleware was created to validate the role of the authenticated user and allow or deny access accordingly.

**8.4 Notification Timestamping**

* **Challenge:** Generating consistent timestamps for notifications using SQLite, which lacks time zone-aware datetime types.
* **Solution:** Used strftime('%Y-%m-%dT%H:%M:%fZ', 'now') to generate ISO-8601 format UTC timestamps directly in SQLite.

**8.5 Maintaining Dashboard Metrics**

* **Challenge:** Efficiently tracking counts of rooms and students for quick dashboard loading.
* **Solution:** A singleton dashboard table was introduced with pre-computed counts. These values are updated during student/room creation or deletion to avoid repetitive counting queries.

**8.6 File and Module Organization**

* **Challenge:** Managing and scaling the project structure for clarity and modularity.
* **Solution:** Adopted a layered architecture with separate directories for routes, controllers, models, and config, improving maintainability and readability.

**8.7 Ensuring Data Uniqueness**

* **Challenge:** Avoiding duplicate usernames, roll numbers, and room numbers.
* **Solution:** Used UNIQUE constraints in the schema and added backend checks to return meaningful error responses if duplicates are attempted.

**9. Future Enhancements**

While the current system fulfills the core requirements of a basic hostel management system, there are several opportunities to extend its functionality and improve performance, scalability, and user experience. Some of the possible future enhancements are outlined below:

**9.1 User Authentication & Security Enhancements**

* **Enhancement:** Integrate secure password hashing (e.g., using bcrypt) and implement token-based authentication (JWT) for session management.
* **Benefit:** Improves the security and scalability of the login system.

**9.2 Enhanced Role Management**

* **Enhancement:** Introduce additional roles such as wardens or caretakers with specific permissions.
* **Benefit:** Supports real-world delegation and finer-grained access control.

**9.3 Room Allocation Automation**

* **Enhancement:** Automatically assign rooms based on availability and preferences like course, year, or gender.
* **Benefit:** Reduces manual intervention and ensures optimal room utilization.

**9.4 Notification Targeting**

* **Enhancement:** Add targeting filters to notifications (e.g., by course or year).
* **Benefit:** Allows admins to send relevant information to specific student groups.

**9.5 Real-Time Updates**

* **Enhancement:** Use WebSockets or polling to show real-time updates in dashboards when students or rooms are added/updated.
* **Benefit:** Improves the responsiveness and interactivity of the UI.

**9.6 Audit Logs**

* **Enhancement:** Maintain logs for student additions, room changes, and admin actions.
* **Benefit:** Enhances accountability and provides traceability.

**9.7 Migration to PostgreSQL or MySQL**

* **Enhancement:** Replace SQLite with a full-fledged RDBMS.
* **Benefit:** Enables better performance under concurrent load, advanced querying, and scalability.

**9.8 Responsive & Mobile-Friendly UI**

* **Enhancement:** Optimize the frontend for smaller screens.
* **Benefit:** Improves accessibility for mobile users like students.

**9.9 Hosting & Deployment**

* **Enhancement:** Deploy the system to a production-ready cloud platform with CI/CD pipelines.
* **Benefit:** Makes the application available for real-world use and testing.

**10. Conclusion**

The Hostel Management System developed using **React, TypeScript, Zustand, and TailwindCSS** on the frontend, and **Node.js, Express, TypeScript, and SQLite** on the backend, successfully addresses the key administrative needs of hostel operations. The system simplifies complex workflows such as:

* Managing student information and room allocation,
* Tracking available and occupied rooms,
* Disseminating notifications to students,
* And maintaining essential hostel statistics on a dashboard.

The application provides a **clean, intuitive UI** and a **modular backend**, making it both **developer-friendly and extensible**. The use of **Zustand for state management** and **SQLite for lightweight persistent storage** makes this solution efficient and ideal for small to medium-scale institutions.

Through rigorous testing and real-world use scenarios, the project has proven to be **functional, robust, and easy to maintain**. However, there remains potential for growth through the incorporation of enhanced security features, real-time updates, and mobile responsiveness.

This system not only serves as a practical solution to a real-world problem but also stands as a valuable full-stack project reflecting modern software development practices.

**11. References**

The development and documentation of this project involved consulting several online resources, libraries, and documentation tools. Below is a list of references that were instrumental throughout the project lifecycle:

**Web Technologies and Frameworks**

* React.js Official Documentation – [https://react.dev](https://react.dev/)
* TypeScript Handbook – <https://www.typescriptlang.org/docs>
* Zustand State Management – <https://docs.pmnd.rs/zustand>
* TailwindCSS Documentation – <https://tailwindcss.com/docs>

**Backend Technologies**

* Node.js Documentation – <https://nodejs.org/en/docs>
* Express.js Guide – [https://expressjs.com](https://expressjs.com/)
* better-sqlite3 (SQLite Wrapper) – <https://github.com/WiseLibs/better-sqlite3>

**Development Tools**

* Visual Studio Code – [https://code.visualstudio.com](https://code.visualstudio.com/)
* GitHub (Project Repository Hosting) – [https://github.com](https://github.com/)
* SQLite Documentation – <https://www.sqlite.org/docs.html>

**Others**

* AI tools, Stack Overflow and GitHub Issues for community-driven troubleshooting.