A PROJECT REPORT ON

"WEB BASED HOSTEL MANAGEMENT APPLICATION"

USING PHP

Submitted in fulfillment of the requirement for the degree of

BACHELOR OF COMPUTER APPLICATION (BCA)

Under the guidance of

Ms. C. Zorinsangi (Faculty, NIELIT Aizawl Center)

NATIONAL INSTITUTE OF ELECTRONICS AND INFORMATION TECHNOLOGY

(Deemed University)

AIZAWL – 796017, INDIA

JUNE, 2025

Submitted by:

T. LALBIAKHLUA	Reg No: 2202135
BENJAMIN LALRUATMAWIA	Reg No: 2202085

CERTIFICATE

This is to certify that this major project report entitled **“Web based Hostel Management Application”** is a bona-fide work done by the following students in partial fulfilment for award of **Bachelor of Computer Application (BCA)**, during the period **January - June, 2025** as prescribed by National Institute of Electronics and Information Technology (NIELIT), Aizawl Centre.

This report or similar report on the topic has not been submitted for any other examination and does not form part of any other course undergone by the candidates.

Submitted By:

T Lalbiakhlua (2224BCA054)

Benjamin Lalruatmawia (2224BCA004)

____________________	____________________
Project Coordinator/Supervisor	BCA Coordinator
(Ms. C.Zorinsangi)	(Dr Vanlalhruaia)
Faculty	Senior Technical Officer
NIELIT, Aizawl Center	NIELIT, Aizawl Center

External Examiners

1.( ) ______________________________

2.( ) ______________________________

DECLARATION

We hereby declare that the work which is presented in the BCA Proposed Project entitled "**Web Based Hostel Management Application**", in partial fulfillment of the requirements for the award of the bachelor of Computer Application degree and is submitted to Mizoram University. It is an authentic record of our own work carried out during the year 2025 (6th Semester) under the supervision of Ms. C.Zorinsangi (Faculty, NIELIT Aizawl Center).

The matter presented in this Proposed Project Report has not been submitted by anyone prior for the award of any degree or presented anywhere else

T. LALBIAKHLUA	BENJAMIN LALRUATMAWIA
(2224BCA054)	(2224BCA004)

ABSTRACT

The project titled **"Web-Based Hostel Management Application"** is developed to simplify and streamline the management of hostel facilities within an educational institution. The application provides a centralized platform for both students and administrators to handle various aspects of hostel room management digitally, reducing manual effort and paperwork.

The primary objective of this project is to create a user-friendly system that allows students to view room details, check payment status, file complaints, and access other hostel-related services. On the administrator side, the system enables efficient management of room allocations, tracking of maintenance requests, monitoring of payments, and updating notifications.

This application is developed using **HTML, CSS, JavaScript** for the front-end, and **PHP** for the server-side scripting. **MySQL** is used as the database for storing user and hostel-related information, and **XAMPP** is used as the local development server. The project was built and managed using **Visual Studio Code** as the code editor.

By transitioning from manual processes to a digital platform, the system ensures better accessibility, accuracy, and ease of communication between students and hostel administration. The result is a more organized and efficient hostel management process

List of Figures

List of Tables

Nomenclature & Abbreviations

Term/ Abbreviations	**Full Form/ Explanation**
UI	User Interface
UX	User Experience
PHP	Hypertext Preprocessor
CSS	Cascading Style Sheet
XAMPP	Cross-Platform Apache, MySQL, PHP, Perl
DB	Database
CRUD	Create, Read, Update, Delete
SQL	Structured Query Language
NF (1NF, 2NF)	Normal Forms
ERD	Entity Relationship DIagram
DFD	Data Flow Diagram

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Introduction to the Project

In today's digital age, managing hostel facilities through traditional paperwork and manual processes is becoming increasingly outdated. To address this, our project—**"Web-Based Hostel Room Management System"**—was developed as part of the Bachelor of Computer Applications (BCA) major project. The goal is simple: to make hostel room and resident management easier, faster, and more efficient for everyone involved.

This system is built to serve three main types of users: **Hostellers, Wardens, and Admins**. Each user type has access to features that match their specific needs and responsibilities. For instance, **Hostellers** can log in to check their room details, request checkouts, make payments, and report any issues or complaints. **Wardens** can oversee hosteller information, respond to complaints, approve checkouts, and manage announcements. **Admins** have full access—they can add new hostellers, assign rooms, update payment statuses, handle maintenance updates, view all reports, and publish important notices.

We've used common web development tools—**HTML, CSS, JavaScript, PHP, and MySQL**—to build the system, and tested it on a **local server using XAMPP**. Choosing a web-based platform makes the system accessible, centralized, and much less dependent on physical records.

Overall, this project aims to solve the common problems of traditional hostel management—like slow communication, lost paperwork, and a lack of transparency—by offering a modern, reliable, and user-friendly alternative that improves the experience for both residents and staff.

1.2 Objectives

The main goal of the **Web-Based Hostel Room Management System** is to create a smart, user-friendly digital solution that simplifies and automates key tasks involved in hostel administration. By reducing paperwork and manual work, the system helps improve efficiency and fosters better communication and transparency between hostellers and management.

Here’s a breakdown of the specific objectives:

- **Centralized Platform**: Offer a single, unified system where hostellers, wardens, and administrators can manage and access hostel-related information based on their roles and permissions.
- **For Hostellers**:
    - View their personal, room, and payment details.
    - Submit checkout requests and receive real-time status updates.
    - File complaints or report issues related to hostel facilities.
    - Make payments online and keep track of payment status.
- **For Wardens**:
    - Monitor hosteller information and room assignments.
    - Review and respond to checkout requests.
    - Manage hostel notifications and announcements.
    - View complaints and take necessary actions.
- **For Administrators**:
    - Add, update, or remove hosteller records.
    - Assign or reassign rooms as needed.
    - Update payment statuses and room condition records.
    - Access and manage all submitted complaints and reports.
- **User-Friendly Interface**: Build a secure, responsive web interface that allows easy interaction between all users, ensuring smooth data flow and usability across devices.
- **Improved Efficiency**: Automate repetitive tasks and enable real-time updates to enhance the accuracy, reliability, and speed of hostel operations.
- **Reduced Manual Work**: Minimize the risk of errors, delays, and miscommunication by shifting from manual to digital processes.

1.3 Scope and Purpose

The **Web-Based Hostel Room Management System** was created in response to the growing need for a more organized, efficient, and digital approach to managing hostel operations in educational institutions. It aims to simplify communication, room allocation, complaint handling, and data management for everyone involved—hostellers, wardens, and administrators.

Purpose

The main purpose of this project is to **digitize and automate the hostel management process**, replacing outdated manual systems that are often slow, error-prone, and inefficient. With a web-based platform accessible to all user types, the system improves coordination, speeds up decision-making, and ensures greater transparency.

This system offers a complete solution by:

- Enhancing the **user experience** for hostellers through online access to key services.
- Helping hostel staff easily **track hosteller records, room availability**, and **maintenance issues**.
- Equipping the admin team with tools to **manage all hostel operations from a single, unified dashboard**.

Scope

The system's features cover a wide range of hostel management tasks, including:

- A **role-based login system** for Hostellers, Wardens, and Admins.
- Viewing of **personal, room, and payment details** by hostellers.
- Online **checkout requests** and tracking.
- A **complaint/report submission system** with real-time status updates.
- **Room assignment** and occupancy management by the admin.
- **Digital payment tracking**, with manual updates as needed by the admin.
- A built-in **notification system** for updates and announcements.
- A **centralized, secure database** for storing all hostel-related data.

While it's currently tailored for a single institution, the system can be scaled and adapted to support multiple hostels or campuses with some additional development.

2. SYSTEM ANALYSIS

2.1 Need for System

In traditional hostel management, key tasks like room allotment, fee tracking, complaint handling, and record-keeping are usually done manually. This not only takes a lot of time but also increases the risk of errors, data loss, and delays in communication.

The **Web-Based Hostel Room Management System** was developed to overcome these challenges by bringing automation and digital tools into the hostel administration process. It provides a centralized platform where hostellers, wardens, and administrators can easily interact, access information, and carry out their respective tasks efficiently.

Why This System is Needed

Some of the major issues with manual hostel management systems include:

- No centralized system for accessing hosteller data and room details.
- Slow and inefficient handling of requests like room changes, complaints, or checkouts.
- Difficulty maintaining accurate, up-to-date records for payments, room conditions, and resident status.
- Limited communication between hostellers and hostel staff.
- Increased chances of mismanagement and delayed decisions due to a lack of real-time information.

By switching to this digital system, institutions can **streamline their hostel operations**, **reduce administrative workload**, and **create a more transparent, reliable, and user-friendly experience** for everyone involved.

2.2 Feasibility Study

A feasibility study is conducted to determine the practicality, viability, and overall potential for successful implementation of the Web-Based Hostel Room Management

System. This analysis helps in evaluating whether the proposed system can be developed with available resources and within acceptable limits of cost, time, and technology. The feasibility of the system is assessed under the following categories:

1. Technical Feasibility:

The project utilizes widely used and well-supported web technologies such as HTML, CSS, JavaScript, PHP, and MySQL, which are compatible with most platforms and require no specialized hardware. The development and deployment are carried out using XAMPP as a local server, ensuring ease of setup and testing, the system is technically feasible and achievable with the available tools and knowledge.

2. Operational Feasibility:

The system is designed with a user-friendly interface, ensuring that all users — hostellers, wardens, and admins — can interact with the application without requiring advanced technical skills. The role-based access model provides clear segregation of responsibilities, ensuring smooth workflow and reducing the possibility of errors, making it highly feasible from an operational standpoint.

3. Economic Feasibility:

The cost involved in developing the system is minimal, as it is built using open-source technologies and hosted locally using XAMPP. No additional investment in software licenses or hardware infrastructure is required. As a result, the project is economically viable, especially for educational institutions with limited budgets.

4. Schedule Feasibility:

Given the defined scope and the availability of necessary resources, the project is well within the timeline constraints of the academic curriculum. The development tasks are broken down into manageable modules, allowing the system to be completed within the required timeframe for the BCA major project.

2.3 Hardware Requirements

While the **Web-Based Hostel Room Management System** is primarily software-driven and built using open-source web technologies, it's still important to outline the basic hardware needed for smooth development, testing, and use. Since the system is lightweight and runs on a local server using **XAMPP**, it doesn't require high-end hardware—standard computing devices are more than sufficient.

Minimum Hardware Requirements

For Development and Local Hosting:

- **Processor**: Intel Core i3 or equivalent (or better)
- **RAM**: 4 GB or higher
- **Storage**: At least 250 GB hard disk (with a minimum of 100 MB free for the project and XAMPP)
- **Display**: Standard monitor with a resolution of 1024x768 or higher
- **Input Devices**: Keyboard and mouse
- **Network**: Localhost access or LAN setup for testing multi-user functionality

For End Users (Hostellers, Wardens, Admins):

- **Device**: Desktop, laptop, or tablet with a modern web browser
- **Processor**: Any modern processor capable of running a browser
- **RAM**: 2 GB or more
- **Network**: Stable LAN or Wi-Fi connection for system access
- **Display**: Responsive design ensures compatibility across various screen sizes

Since this is a **web-based, responsive application**, it is **platform-independent** and can be accessed from any device with a browser and an internet or network connection. This makes it highly **flexible, accessible, and hardware-efficient**, even in resource-constrained environments.

2.4 Software Requirement Specifications

The **Software Requirements Specification (SRS)** outlines both the functional and non-functional requirements for the **Web-Based Hostel Room Management System**. It acts as a guiding document throughout the development process, ensuring a clear understanding of the system's scope, features, and constraints.

---

1. Functional Requirements

These define the core operations the system must support to meet user needs:

User Authentication & Role-Based Access

- Secure login system for **Hostellers**, **Wardens**, and **Admins**
- Role-based redirection and access to specific functionalities

Hosteller Features

- View personal, room, and payment information
- Submit checkout requests
- File complaints or reports
- Make fee payments and track payment status
- View notifications and announcements

Warden Features

- Access hosteller and room information
- Approve or reject checkout requests
- View and manage complaints
- Post and update notifications

Admin Features

- Add, update, or delete hosteller records
- Assign or reassign rooms
- Update payment statuses and room condition data
- View submitted reports and manage notifications

Database Management

- Store and retrieve data related to users, rooms, complaints, payments, and notices
- Maintain data accuracy, consistency, and integrity across all modules

---

2. Non-Functional Requirements

These requirements ensure the system is reliable, user-friendly, and scalable:

- **Usability**: Interface should be clean, intuitive, and accessible for all user types
- **Security**: Role-based access control and basic data validation to prevent misuse

- **Scalability**: Ability to expand system capabilities or extend to multiple hostels
- **Maintainability**: Modular code structure with clear documentation for easy updates and debugging
- **Portability**: Runs smoothly on any device with a modern browser and internet/local server access
- **Performance**: Fast load times and real-time data updates for a smooth user experience

---

3. Software & Tools Used

- **Frontend**: HTML, CSS, JavaScript
- **Backend**: PHP
- **Database**: MySQL
- **Local Server Environment**: XAMPP (includes Apache, PHP, and MySQL)
- **Browser Compatibility**: Supports major browsers like **Google Chrome**, **Mozilla Firefox**, **Microsoft Edge**, etc.

2.5 System Flow Chart

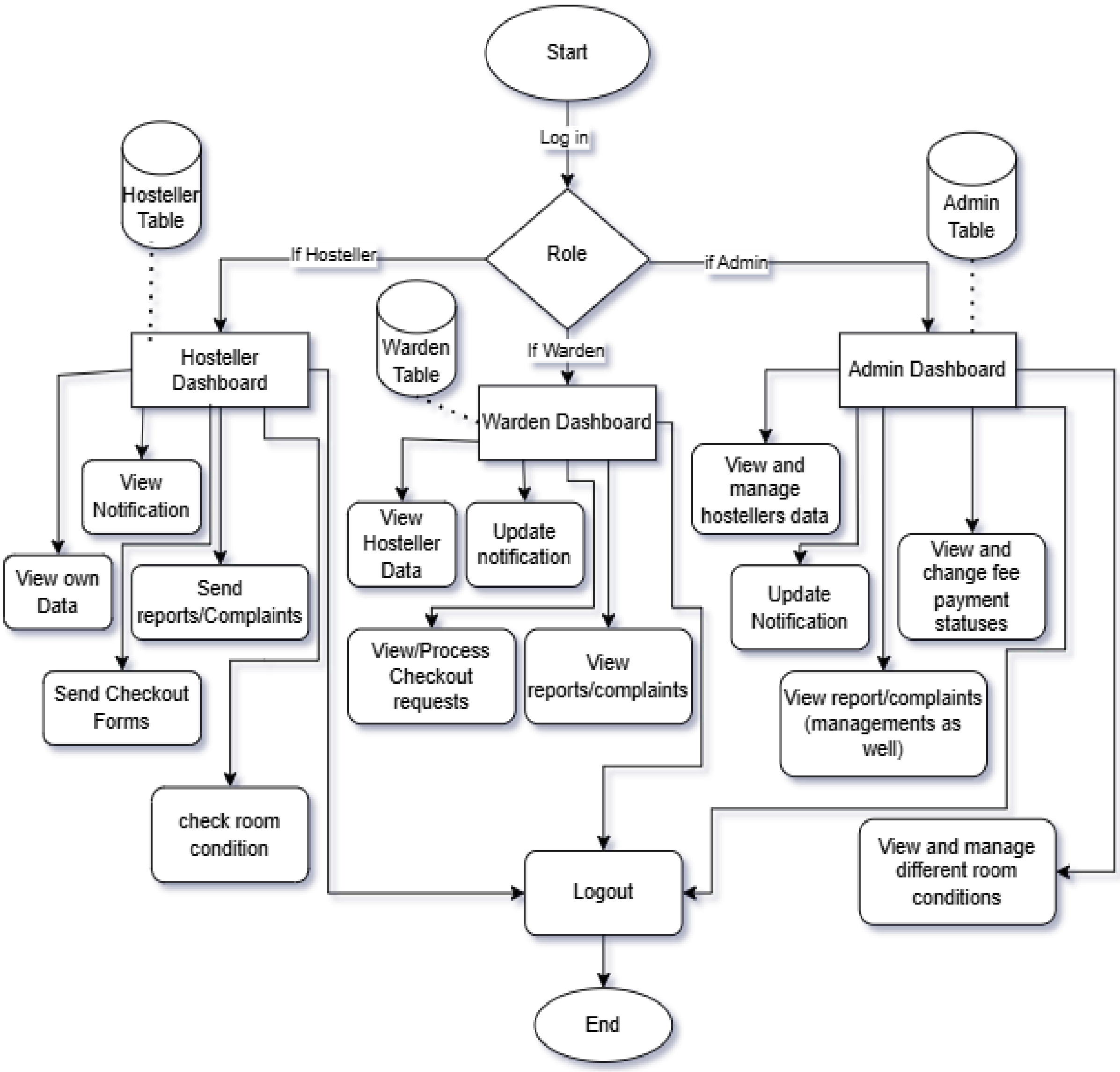


Fig 1.1: System Flow chart

2.6 Entity Relationship Data

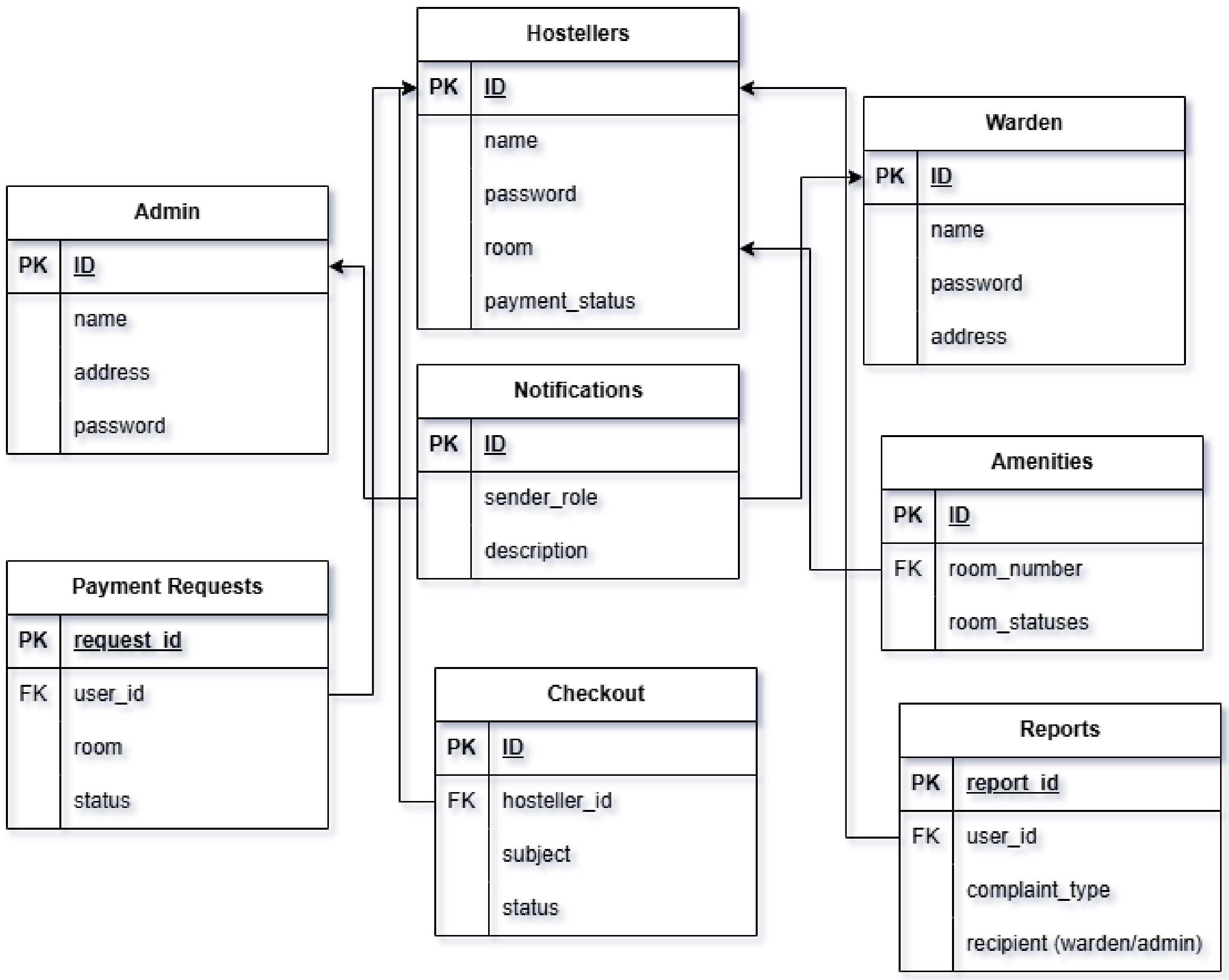


Fig 1.2: ERD for the system

2.7 Data Flow Diagram (DFD)

DFD Level 0

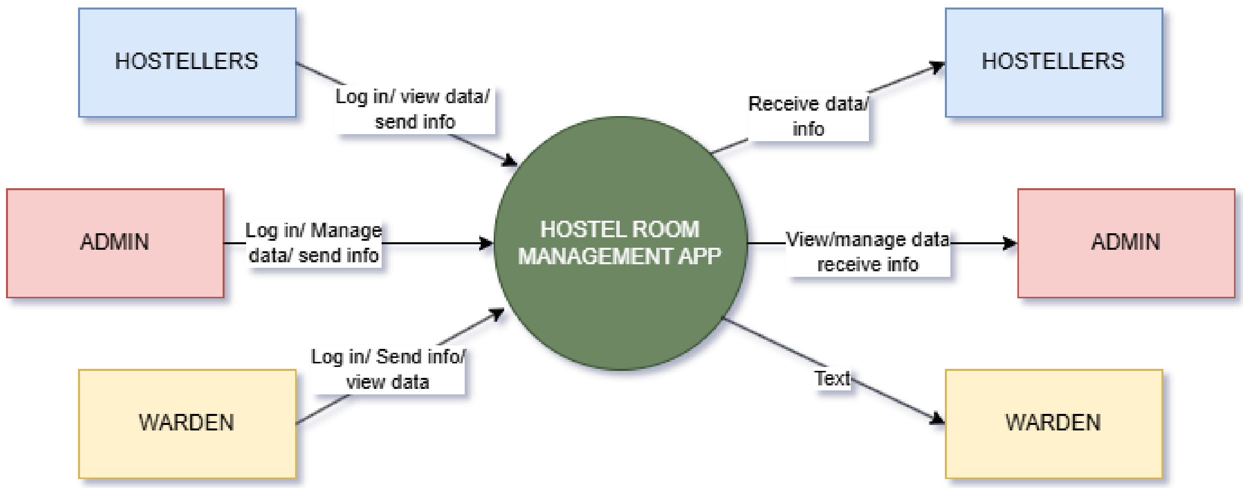

HOSTELLERS
Log in/ view data/ send info
ADMIN
Log in/ Manage data/ send info
WARDEN
Log in/ Send info/ view data
HOSTEL ROOM MANAGEMENT APP
Receive data/ info
HOSTELLERS
View/manage data receive info
ADMIN
Text
WARDEN

Fig 1.3: DFD level-0 for the system

DFD Level 1

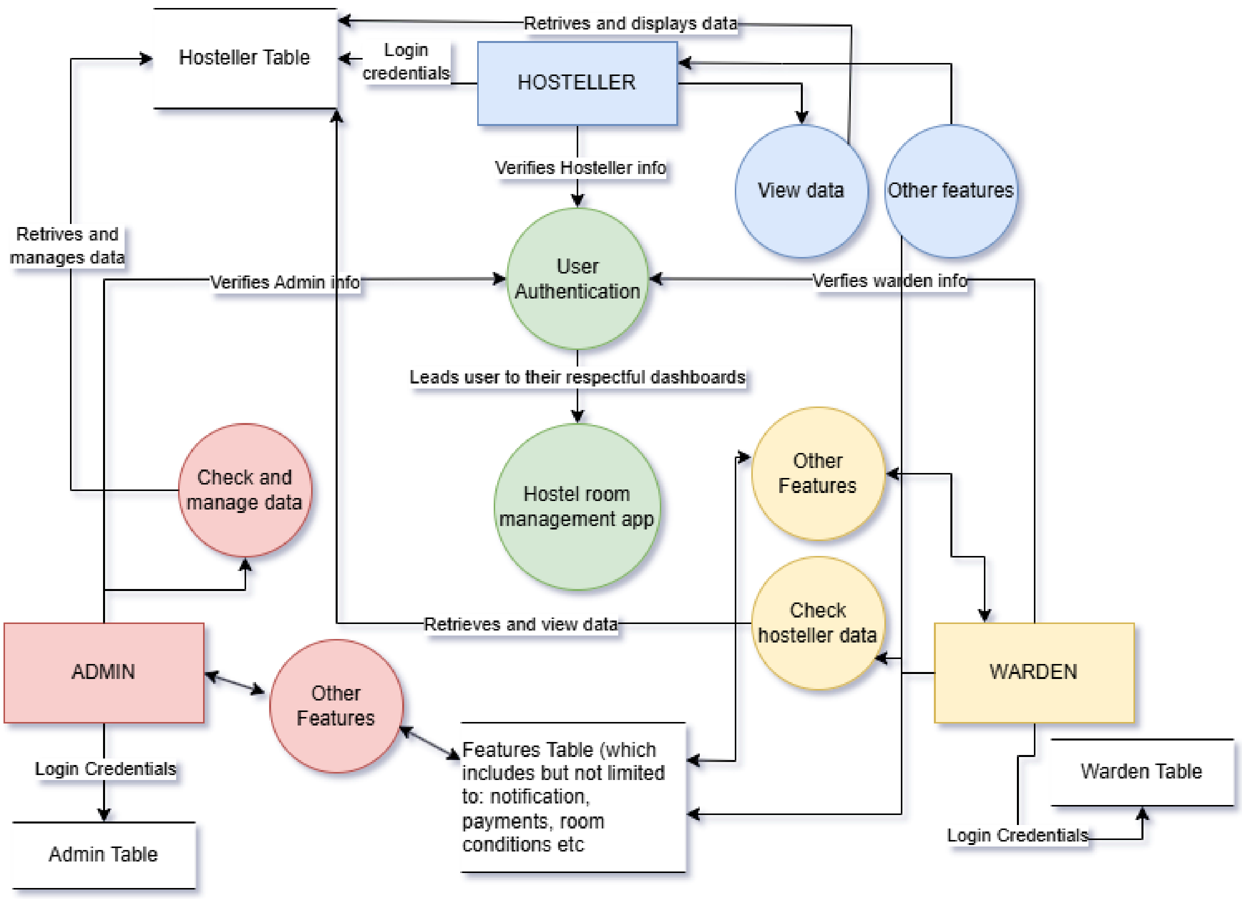


Fig 1.4: DFD level-1 for the system

3.SYSTEM DESIGN

3.1 Program Structure

The **Web-Based Hostel Room Management System** is built using a layered architecture to ensure modularity, scalability, and ease of maintenance. The application follows a standard **three-tier structure**, separating responsibilities across the **Presentation Layer**, **Application Layer**, and **Data Layer**. Additionally, the system is organized into **role-based modules** for Hostellers, Wardens, and Admins, allowing smooth and secure interaction based on user privileges.

---

1. Presentation Layer (Frontend)

This is the user-facing layer, responsible for all interactions with the system. It includes web pages and interfaces built using **HTML, CSS, and JavaScript**. The design is responsive and role-specific, meaning each user type—Hosteller, Warden, or Admin—sees a tailored interface suited to their tasks. The focus is on ease of navigation and usability.

---

2. Application Layer (Backend Logic)

The backend, developed in **PHP**, handles the core logic of the application. It processes user inputs, communicates with the database, and controls the system's responses. This layer is divided into functional scripts for different actions, including:

- User login and authentication
- Loading dashboards for each user type
- Handling complaint submissions and responses
- Managing payments and tracking payment status
- Posting and updating notifications
- Assigning and updating room information
- Generating and processing reports

---

3. Data Layer (Database)

The data layer is powered by **MySQL**, managed through **phpMyAdmin** in the XAMPP environment. It stores all persistent information and ensures data integrity through well-structured tables and relationships. Key tables include:

- hostellers
- rooms
- payments
- complaints
- notifications
- admins and wardens

All tables are **normalized** and linked with **foreign key constraints** where necessary to maintain consistency and avoid data duplication.

---

4. User Role Modules

The system is divided into modules based on user roles, each with access to relevant features:

Hosteller Module

- View personal details, room info, and payment status
- Submit complaints and checkout requests
- Make payments and receive notifications

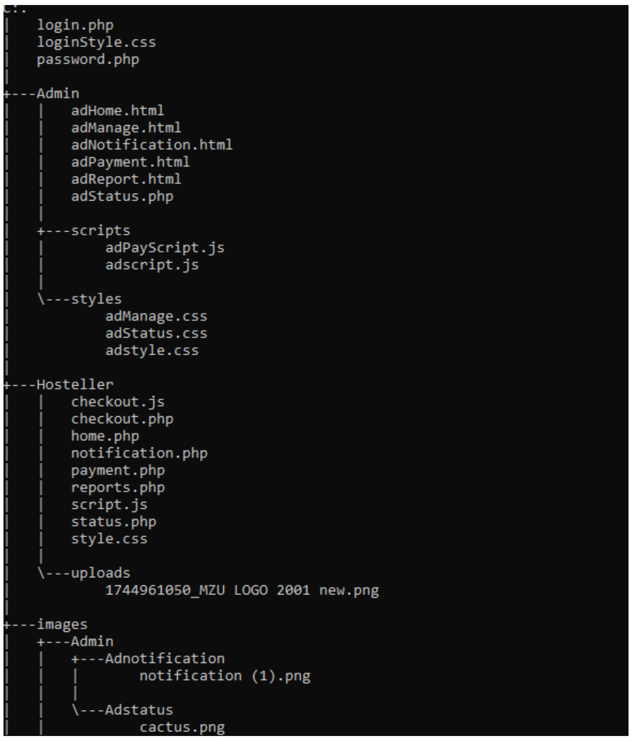
Warden Module

- View and manage hosteller complaints
- Approve or reject checkout requests
- Post and update hostel-wide notifications

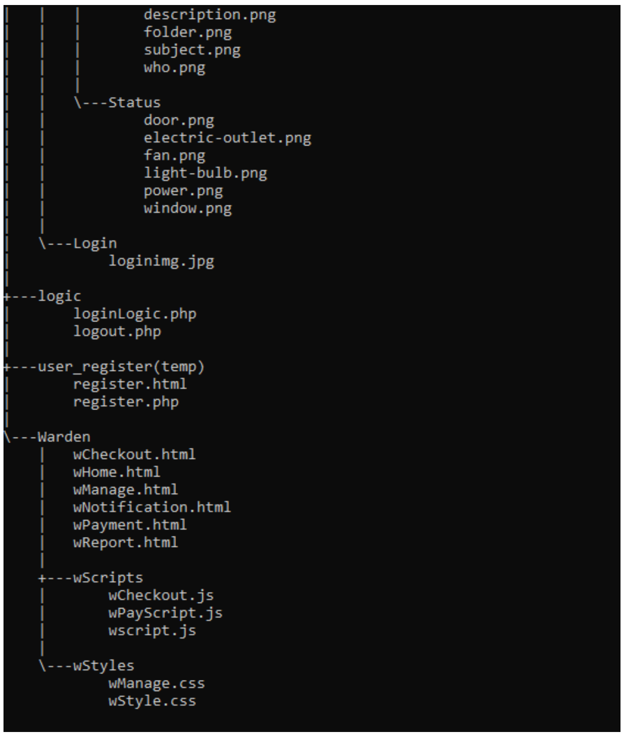
Admin Module

- Full access to hosteller records and room assignments
- Update payment and room status
- Manage all complaints, reports, and notifications across the system

This architecture promotes clean separation of concerns, making the system **easy to understand, modify, and scale** in the future.



```
C:.
|   login.php
|   loginStyle.css
|   password.php
|
+---Admin
|   |   adHome.html
|   |   adManage.html
|   |   adNotification.html
|   |   adPayment.html
|   |   adReport.html
|   |   adStatus.php
|   |
|   +---scripts
|   |       adPayScript.js
|   |       adscript.js
|   |
|   \---styles
|           adManage.css
|           adStatus.css
|           adstyle.css
|
+---Hosteller
|   |   checkout.js
|   |   checkout.php
|   |   home.php
|   |   notification.php
|   |   payment.php
|   |   reports.php
|   |   script.js
|   |   status.php
|   |   style.css
|   |
|   \---uploads
|           1744961050_MZU LOGO 2001 new.png
|
+---images
|   +---Admin
|   |   +---Adnotification
|   |   |       notification (1).png
|   |   |
|   |   \---Adstatus
|   |           cactus.png
```



```
|   |   |       description.png
|   |   |       folder.png
|   |   |       subject.png
|   |   |       who.png
|   |   |
|   |   \---Status
|   |           door.png
|   |           electric-outlet.png
|   |           fan.png
|   |           light-bulb.png
|   |           power.png
|   |           window.png
|   |
|   \---Login
|           loginimg.jpg
|
+---logic
|       loginLogic.php
|       logout.php
|
+---user_register(temp)
|       register.html
|       register.php
|
\---Warden
    |   wCheckout.html
    |   wHome.html
    |   wManage.html
    |   wNotification.html
    |   wPayment.html
    |   wReport.html
    |
    +---wScripts
    |       wCheckout.js
    |       wPayScript.js
    |       wscript.js
    |
    \---wStyles
            wManage.css
            wStyle.css
```

Fig 1.5 – Program structure in tree view

3.2 Modularization Details

The **Web-Based Hostel Room Management System** is developed using a **modular architecture**, meaning the application is divided into smaller, independent components (modules). Each module is focused on a specific set of tasks or responsibilities, which improves **code reusability**, simplifies **debugging**, and makes the system **easier to maintain and expand**.

This approach also allows developers to work on and test each module independently, leading to a more efficient and organized development process. The key modules are grouped by user roles and system functionalities as outlined below:

---

1. Authentication Module

- Manages login and logout functionality for all user types (Hosteller, Warden, Admin)
- Validates user credentials securely
- Controls user sessions and restricts access based on role privileges

---

2. Dashboard Module

- Loads the appropriate dashboard depending on the logged-in user
- Provides quick access to key features like complaints, payments, and notifications
- Displays summary information (e.g., total complaints, pending checkouts, latest announcements)

---

3. Hosteller Module

- Lets hostellers view their personal and room details
- Submit complaints or issue reports to the warden or admin
- Make fee payments (e.g., through QR code) and view payment status
- Request checkout from the hostel

---

4. Warden Module

- View hosteller profiles and submitted complaints
- Review and approve or deny checkout requests
- Post or update notifications that are visible to hostellers

---

5. Admin Module

- Has full administrative access to all data and system functions
- Add, edit, or remove hostellers and wardens from the system
- Manage room assignments and update room condition details
- Update and monitor payment statuses
- Oversee all complaints and control system-wide notifications

---

6. Database Interaction Module

- Connects the application to the **MySQL** database using **PHP**
- Handles all database operations (Create, Read, Update, Delete)
- Ensures data is accessed in a secure and organized manner by all other modules

---

7. Notification & Reporting Module

- Manages the creation and delivery of system notifications
- Enables both wardens and admins to post announcements or updates
- Ensures hostellers are promptly informed of important changes or alerts

---

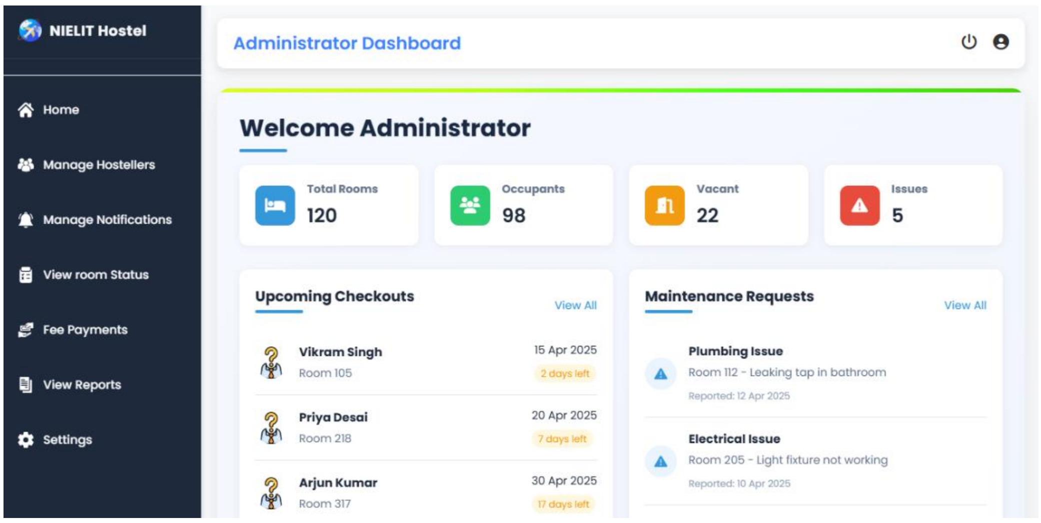
This modular structure ensures that each part of the system works independently yet cohesively, making future enhancements or maintenance tasks far more manageable.

3.3 User Interface Design

3.3.1 Menu Explanation

The user interface is designed with a focus on simplicity, clarity, and accessibility for all user roles — Hostellers, Wardens, and Admins. The layout follows a clean and consistent structure across all pages, ensuring that users can navigate the system with minimal effort.

Key functions are organized into sidebar or top navigation menus depending on the role, with clearly labeled buttons and sections. The use of soft color schemes, proper spacing, and uniform typography enhances readability and creates a professional appearance.



Welcome Administrator

Total Rooms 120
Occupants 98
Vacant 22
Issues 5

Upcoming Checkouts

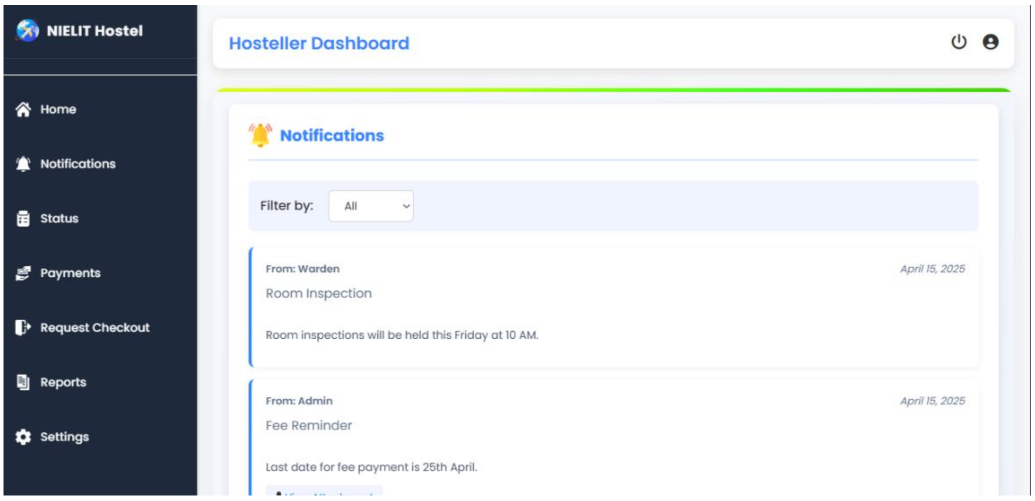
Vikram Singh, Room 105 — 15 Apr 2025, 2 days left
Priya Desai, Room 218 — 20 Apr 2025, 7 days left
Arjun Kumar, Room 317 — 30 Apr 2025, 17 days left

Maintenance Requests

Plumbing Issue — Room 112 - Leaking tap in bathroom — Reported: 12 Apr 2025
Electrical Issue — Room 205 - Light fixture not working — Reported: 10 Apr 2025

Fig 1.6: Admin Dashboard UI

Smooth transitions and minimal animations have been implemented using JavaScript and CSS to enhance user experience without causing delays or distractions. Elements such as dropdown menus, modals, and status updates use subtle motion effects to make interactions feel responsive and dynamic. The UI is designed to be intuitive and lightweight, maintaining a balance between aesthetic appeal and performance, especially for users on standard hardware or slower connections.

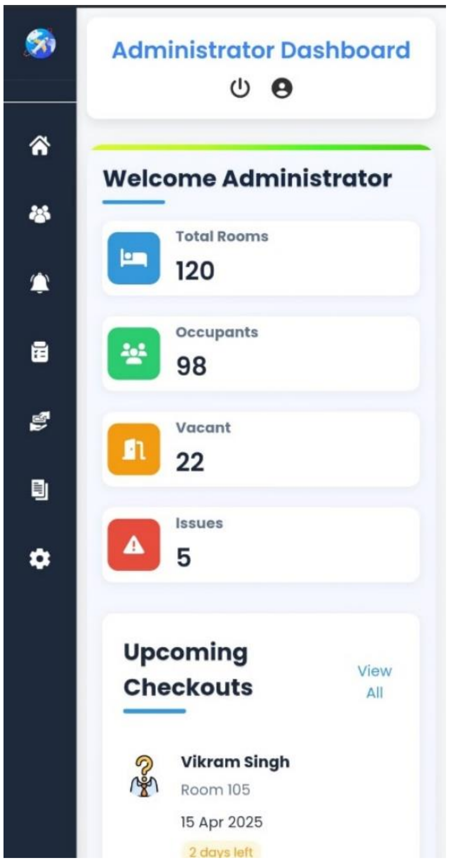


From: Warden
April 15, 2025
Room Inspection
Room inspections will be held this Friday at 10 AM.

From: Admin
April 15, 2025
Fee Reminder
Last date for fee payment is 25th April.

Fig: Hosteller notification view UI

This web application is developed to have responsive view for different screen sizes, although not designed for smartphones, it is perfectly usable on a smartphone.



Administrator Dashboard

Welcome Administrator

Total Rooms 120

Occupants 98

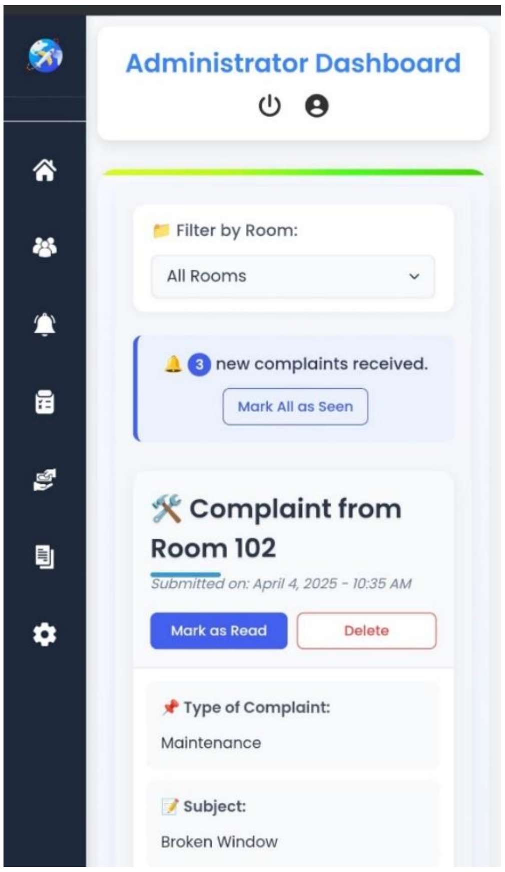
Vacant 22

Issues 5

Upcoming Checkouts

View All

Vikram Singh
Room 105
15 Apr 2025
2 days left



Administrator Dashboard

📁 Filter by Room:
All Rooms

🔔 3 new complaints received.
Mark All as Seen

🛠 Complaint from Room 102

Submitted on: April 4, 2025 – 10:35 AM

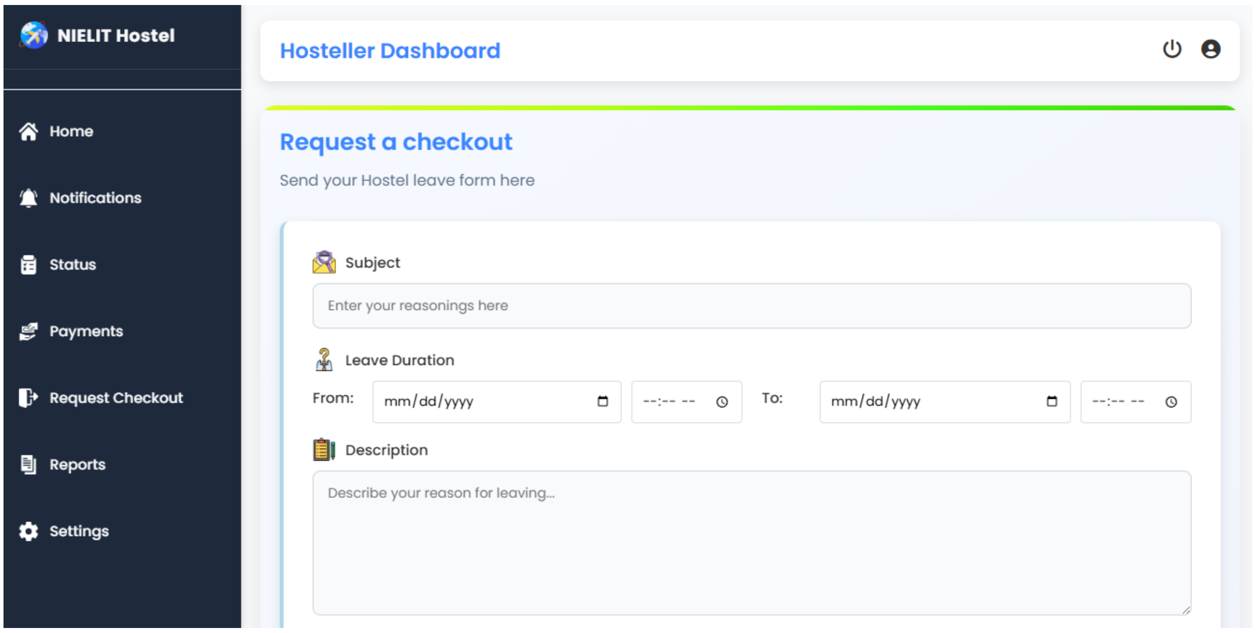
Mark as Read | Delete

📌 Type of Complaint:
Maintenance

📝 Subject:
Broken Window

Fig 1.8: Smartphone view UI

3.3.2 Design of input screens (showing only hosteller side for this documentation to save space)


NIELIT Hostel

Home
Notifications
Status
Payments
Request Checkout
Reports
Settings

Hosteller Dashboard

Request a checkout

Send your Hostel leave form here

Subject

Enter your reasonings here

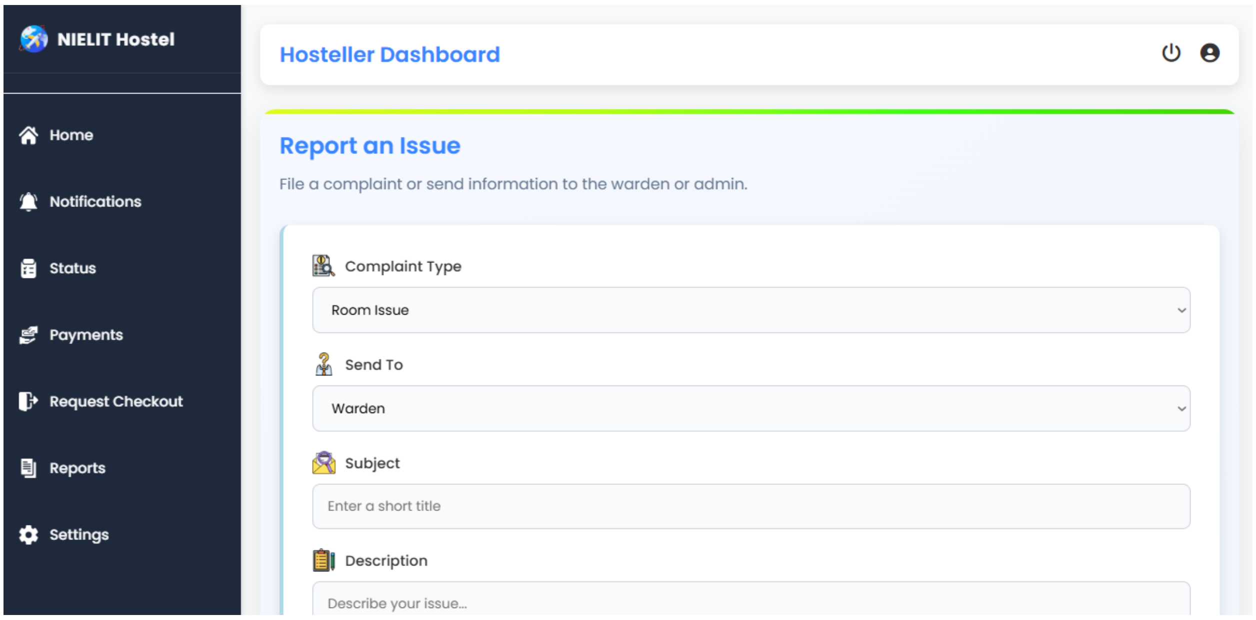
Leave Duration

From: mm/dd/yyyy --:-- -- To: mm/dd/yyyy --:-- --

Description

Describe your reason for leaving...

Fig 1.9: Checkout form request UI


NIELIT Hostel

Home
Notifications
Status
Payments
Request Checkout
Reports
Settings

Hosteller Dashboard

Report an Issue

File a complaint or send information to the warden or admin.

Complaint Type

Room Issue

Send To

Warden

Subject

Enter a short title

Description

Describe your issue...

Fig 1.10: Report/complaint submission UI

3.4 Database Design

3.4.1 Schema Design and normalization

The **Web-Based Hostel Room Management System** uses a well-structured database schema designed to efficiently store, retrieve, and manage all relevant data while avoiding duplication and maintaining consistency.

The system is built around multiple clearly defined tables—such as hostellers, rooms, payments, complaints, notifications, and others—each with its own specific purpose. Relationships between these tables are managed using **foreign keys**, which help preserve **referential integrity** and allow for seamless connections between related data.

Although a formal step-by-step normalization process (like 1NF, 2NF, 3NF) wasn't strictly followed, the schema naturally aligns with core normalization principles:

- Each table contains **atomic values** (single, indivisible data points)
- **Redundancy is minimized** by separating data into logically related entities
- **Data consistency** is ensured through well-defined relationships

For instance, instead of storing full room details in every payment or complaint entry, those tables reference the rooms table using unique IDs. This not only keeps the database lean and organized but also makes querying and updating more efficient.

This modular design is particularly helpful in cases where **data overlaps between roles**, such as wardens and admins accessing hosteller or complaint information. By structuring the database around shared but well-linked tables, the system ensures **clean data handling**, easier maintenance, and better scalability.

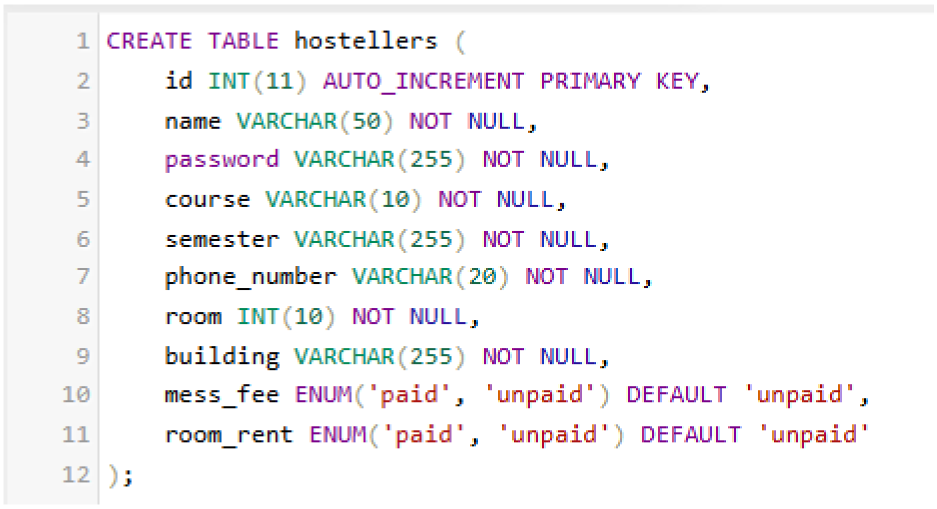
3.4.2 Data integrity and Constraints

To maintain consistent, accurate, and reliable information across the **Web-Based Hostel Room Management System**, a variety of **data integrity rules and constraints** have been incorporated into the database design. These rules play a key role in preventing data anomalies, duplication, and inconsistencies—whether the data is entered by users or generated by the system.

For example:

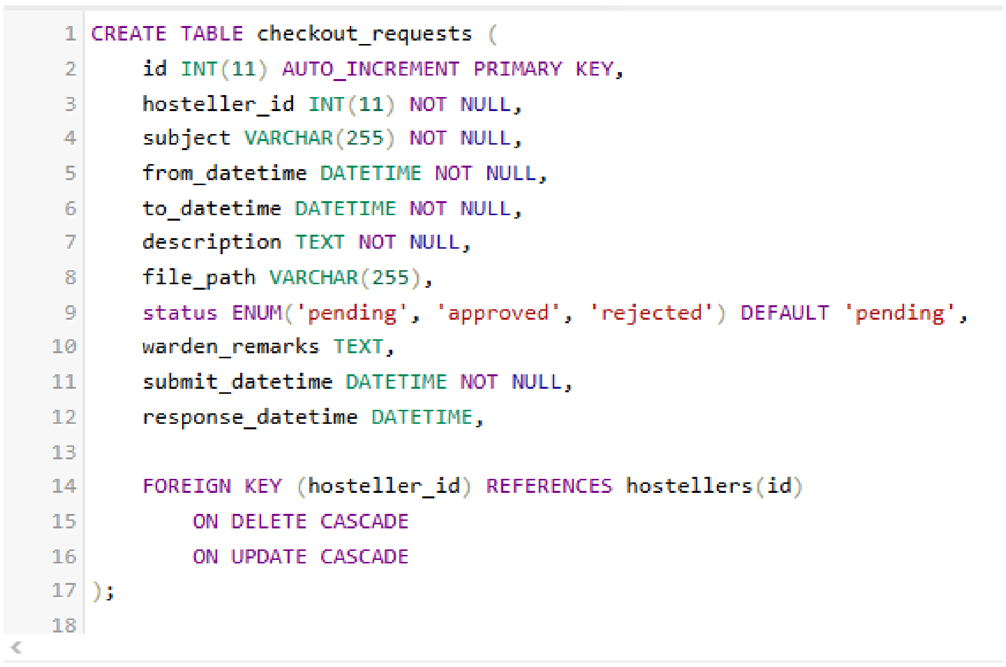
- In the **hostellers** table:
  - The id field is defined as a **primary key** with AUTO_INCREMENT, ensuring each record is unique and easily identifiable.
  - Essential fields like name, password, course, semester, and phone_number are marked as **NOT NULL**, meaning no record can be saved without this critical information.
  - Payment-related fields such as mess_fee and room_rent use **ENUM constraints** with values like 'paid' and 'unpaid', enforcing clarity and consistency in tracking fee status.
- In the **checkout_requests** table:
  - The id is also a **primary key with AUTO_INCREMENT**.
  - The hosteller_id acts as a **foreign key**, linking back to the hostellers table to establish a reliable connection between hostellers and their checkout records.
  - The status field is restricted by an **ENUM** ('pending', 'approved', 'rejected') with a default value of 'pending', supporting a clear and predictable approval workflow.
  - Key fields like subject, from_datetime, and to_datetime are **NOT NULL**, ensuring all submissions are complete and actionable.

Together, these constraints help enforce data accuracy, streamline operations, and simplify database management. They also support the system's goal of offering a trustworthy and efficient experience to its users—whether it's a hosteller checking out, a warden managing reports, or an admin updating payment records.



```sql
CREATE TABLE hostellers (
    id INT(11) AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    password VARCHAR(255) NOT NULL,
    course VARCHAR(10) NOT NULL,
    semester VARCHAR(255) NOT NULL,
    phone_number VARCHAR(20) NOT NULL,
    room INT(10) NOT NULL,
    building VARCHAR(255) NOT NULL,
    mess_fee ENUM('paid', 'unpaid') DEFAULT 'unpaid',
    room_rent ENUM('paid', 'unpaid') DEFAULT 'unpaid'
);
```

Fig 1.11: SQL code for the creation of hostellers table



```sql
CREATE TABLE checkout_requests (
    id INT(11) AUTO_INCREMENT PRIMARY KEY,
    hosteller_id INT(11) NOT NULL,
    subject VARCHAR(255) NOT NULL,
    from_datetime DATETIME NOT NULL,
    to_datetime DATETIME NOT NULL,
    description TEXT NOT NULL,
    file_path VARCHAR(255),
    status ENUM('pending', 'approved', 'rejected') DEFAULT 'pending',
    warden_remarks TEXT,
    submit_datetime DATETIME NOT NULL,
    response_datetime DATETIME,

    FOREIGN KEY (hosteller_id) REFERENCES hostellers(id)
        ON DELETE CASCADE
        ON UPDATE CASCADE
);

```

Fig 1.12: SQL for the creation of checkout_request table

3.4.3 Data Dictionary

The Data Dictionary provides a detailed description of the structure, attributes, and data types of the tables used in the Web-Based Hostel Room Management System. It serves as a reference for understanding how data is stored and managed within the system's database. Below is an outline of two key tables used in the system.

Below provided is a data dictionary based on already mentioned Hosteller table and Checkout_request table (for reference go to page number 27, 28)

Table name: Hostellers

Field name	**Data Type**	**Description**	**Constraints**
Id	INT(11)	Unique identifier for every hostellers	Primary Key, auto_increment
Name	VARCHAR(50)	Name of the hostellers	Not null
Password	VARCHAR(255)	Password(hashed) for secure login	Not null
Course	VARCHAR(10)	Course enrolled by the hosteller	Not null
Semester	VARCHAR(255)	Semester of the hosteller	Not null
Phone_number	VARCHAR(20)	Contact number of the hosteller	Not null
Room	INT(10)	Alotted room number	Not null
Building	VARCHAR(255)	Building/block name	Not null

Mess_fee	ENUM	Status of mess fee payment	Default: unpaid
Room_rent	ENUM	“ “ room rent	Default: Unpaid

Table 1.1

Table name: Checkout_requests

Field name	**Data Type**	**Description**	**Constraints**
Id	INT(11)	Unique identifier for every requests	Primary Key, auto_increment
Hosteller_id	INT(11)	Referencing to requesting hostellers	Foreign key -> hostellers(id)
subject	VARCHAR(255)	Reason for checkout	Not null
From_datetime	VARCHAR(10)	Starting date and time of requested checkout	Not null
To_datetime	VARCHAR(255)	Ending date and time of requesting checkout	Not null
description	VARCHAR(20)	Detailed explanation of reason for checkout	Not null
File_path	INT(10)	Path to any supported document attachments	nullable
status	VARCHAR(255)	Request status	Default:pending
Warden_remarks	ENUM	Remarks added by the warden	nullable

Submit_datetime	ENUM	Timestamp when the request was submitted	Not null
Response_datetime	DATETIME	Timestamp to when the request was reponded to	Nullable

Table 1.2

Table name: payment_requests

Field name	**Data Type**	**Description**	**Constraints**
request_id	INT(20)	Unique identifier for every requests	Primary Key, auto_increment
User_id	INT(50)	Unique Identifier for hostellers	Foreign Key -> hostellers(id)
room	VARCHAR(10)	Room of the hostellers	Not null
building	VARCHAR(50)	Building name	Not null
occupant	VARCHAR(100)	Name of the occupant	Not null
Fee_type	ENUM('room_rent', 'mess_fee')	Type of fee concerned	Not null
Requested_change	VARCHAR(20)	Requested status change	Not null
Request_date	TIMESTAMP	Time when the request was made	Current_timestamp
status	ENUM('pending', 'approved', 'rejected')	Current status of the request	Default: pending

Table 1.3

Table name: notifications

Field name	Data Type	Description	Constraints
id	INT(20)	Unique Notification id	Primary Key, auto_increment
Sender_role	ENUM(''warden','admin')	Role of the senders	Not null
title	VARCHAR(255)	Title of the notification	Not null
description	TEXT	Body text of the notification	Not null
Attachment	VARCHAR(255)	Path to attached file (if any)	Nullable
Date_sent	DATETIME	Time the notification is sent	Current_timestamp

Table 1.4

Table 5: Table name: reports

Field name	Data Type	Description	Constraints
Report_id	INT(20)	Unique report id	Primary Key, auto_increment
User_id	ENUM(‘’warden’,’admin’)	Sender id	Foreign Key -> hostellers(id)
Complaint_type	VARCHAR(255)	Type of complaint	Not null
recipient	TEXT	Receiver’s role	Not null
subject	VARCHAR(255)	Title of the report	Nullable
description	DATETIME	Details of the complaint	Current_timestamp
Attachment_path	VARCHAR(255)	Path to attachment(if any)	Nullable
status	ENUM(‘read’,’unread’)	Read status of the report	Default: unread
Created_at	TIMESTAMP	Time the report was created	Current_timestamp

Table 1.5

4. CODING

4.1 Important code from the project

Provided below are some important codes that run some of the main features of the project (not all of them are shown, only a select few):

Backend Logic for Login:

```
<?php
session_start();
$conn = new mysqli("localhost", "root", "", "hostel");

if ($conn->connect_error) {
   die("Connection failed: " . $conn->connect_error);
}

$role = $_POST['role'];
$username = $_POST['username'];
$password = $_POST['password'];

if (empty($role) || empty($username) || empty($password)) {
   echo "All fields are required.";
   exit;
}

$tableMap = [
   "hosteller" => "hostellers",
   "warden"    => "warden",  // Assuming future table
   "admin"     => "admin"    // Assuming future table
];

if (!array_key_exists($role, $tableMap)) {
   echo "Invalid role selected.";
   exit;
}
```

```
$table = $tableMap[$role];

// Check if user exists
$sql = "SELECT * FROM $table WHERE name = ?";
$stmt = $conn->prepare($sql);
$stmt->bind_param("s", $username);
$stmt->execute();
$result = $stmt->get_result();

if ($result->num_rows === 1) {
    $user = $result->fetch_assoc();

    if (password_verify($password, $user['password'])) {
        // Successful login
        $_SESSION['user_id'] = $user['id'];
        $_SESSION['username'] = $user['name'];
        $_SESSION['role'] = $role;

        // Redirect based on role
        switch ($role) {
            case "hosteller":
                header("Location: ../Hosteller/home.php");
                break;
            case "warden":
                header("Location: ../Warden/wHome.php");
                break;
            case "admin":
                header("Location: ../Admin/adHome.php");
                break;
        }
        exit;
    } else {
        echo "Invalid password.";
    }
} else {
    echo "User not found.";
}

$stmt->close();
$conn->close();
```

```
?>
```

This php code handles the login details entered by the users on the login form using our special role detection logic where we make use of a radio button to choose between the three roles to start different login sessions for each users.

Backend Logic for retrieval of data (for Admin)

```
<?php
// Start session for authentication
session_start();

// Database connection parameters
$servername = "localhost";
$username = "root";
$password = "";
$dbname = "hostel";

// Create connection
$conn = null;

try {
    $conn = new mysqli($servername, $username, $password, $dbname);

    // Check connection
    if ($conn->connect_error) {
        throw new Exception("Connection failed: " . $conn->connect_error);
    }
} catch (Exception $e) {
    echo json_encode(['status' => 'error', 'message' => $e->getMessage()]);
    exit;
}

// Function to sanitize input data
function sanitize_input($data) {
    $data = trim($data);
    $data = stripslashes($data);
    $data = htmlspecialchars($data);
    return $data;
}
```

```
// Handle AJAX requests
if ($_SERVER["REQUEST_METHOD"] == "POST" && isset($_POST['action'])) {
    $action = $_POST['action'];

    switch ($action) {
        case 'add_hosteller':
            add_hosteller();
            break;
        case 'update_hosteller':
            update_hosteller();
            break;
        case 'delete_hosteller':
            delete_hosteller();
            break;
        case 'get_hosteller':
            get_hosteller();
            break;
        case 'get_hostellers':
            get_hostellers();
            break;
        case 'add_warden':
            add_warden();
            break;
        case 'update_warden':
            update_warden();
            break;
        case 'delete_warden':
            delete_warden();
            break;
        case 'get_warden':
            get_warden();
            break;
        case 'get_wardens':
            get_wardens();
            break;
        default:
            echo json_encode(['status' => 'error', 'message' => 'Invalid action']);
            break;
    }
    exit;
}

// Function to add new hosteller
function add_hosteller() {
    global $conn;

    try {
        // Get form data
        $name = sanitize_input($_POST['hosteller-name']);
        $course = sanitize_input($_POST['hosteller-course']);
        $semester = sanitize_input($_POST['hosteller-semester']);
        $room = sanitize_input($_POST['hosteller-room']);
        $contact = sanitize_input($_POST['hosteller-contact']);
        $password = sanitize_input($_POST['hosteller-password']);

        // Determine building based on room number (you can modify this logic)
        $building = (substr($room, 0, 1) == 'B') ? 'Boys' : 'Girls';

        // Default fees (modify as needed)
        $mess_fee = 0;
        $room_rent = 0;

        // Hash password
        $hashed_password = password_hash($password, PASSWORD_DEFAULT);

        // Prepare and execute SQL statement
        $stmt = $conn->prepare("INSERT INTO hostellers (name, password, course, semester,
phone_number, room, building, mess_fee, room_rent) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)");
        $stmt->bind_param("sssssssdd", $name, $hashed_password, $course, $semester,
$contact, $room, $building, $mess_fee, $room_rent);

        if ($stmt->execute()) {
            echo json_encode(['status' => 'success', 'message' => 'Hosteller added successfully']);
        } else {
            throw new Exception("Error: " . $stmt->error);
        }

        $stmt->close();
    } catch (Exception $e) {
        echo json_encode(['status' => 'error', 'message' => $e->getMessage()]);
    }
}

// Function to update existing hosteller
function update_hosteller() {
    global $conn;

    try {
        // Get form data
        $id = sanitize_input($_POST['hosteller-id']);
        $name = sanitize_input($_POST['hosteller-name']);
        $course = sanitize_input($_POST['hosteller-course']);
        $semester = sanitize_input($_POST['hosteller-semester']);
        $room = sanitize_input($_POST['hosteller-room']);
        $contact = sanitize_input($_POST['hosteller-contact']);
        $password = sanitize_input($_POST['hosteller-password']);

        // Determine building based on room number
        $building = (substr($room, 0, 1) == 'B') ? 'Boys' : 'Girls';

        // Check if password is provided
        if (!empty($password)) {
            // Hash new password
            $hashed_password = password_hash($password, PASSWORD_DEFAULT);

            // Update with new password
            $stmt = $conn->prepare("UPDATE hostellers SET name=?, password=?, course=?,
semester=?, phone_number=?, room=?, building=? WHERE id=?");
            $stmt->bind_param("sssssssi", $name, $hashed_password, $course, $semester,
$contact, $room, $building, $id);
        } else {
            // Update without changing password
            $stmt = $conn->prepare("UPDATE hostellers SET name=?, course=?, semester=?,
phone_number=?, room=?, building=? WHERE id=?");
            $stmt->bind_param("ssssssi", $name, $course, $semester, $contact, $room,
$building, $id);
        }
        if ($stmt->execute()) {
            echo json_encode(['status' => 'success', 'message' => 'Hosteller updated 
successfully']);
        } else {
            throw new Exception("Error: " . $stmt->error);
        }

        $stmt->close();
    } catch (Exception $e) {
        echo json_encode(['status' => 'error', 'message' => $e->getMessage()]);
    }
}

// Function to delete hosteller
function delete_hosteller() {
    global $conn;

    try {
        $id = sanitize_input($_POST['id']);

        $stmt = $conn->prepare("DELETE FROM hostellers WHERE id=?");
        $stmt->bind_param("i", $id);

        if ($stmt->execute()) {
            echo json_encode(['status' => 'success', 'message' => 'Hosteller deleted 
successfully']);
        } else {
            throw new Exception("Error: " . $stmt->error);
        }
             /*Rest of the code here, removed to save space*/

// Close connection
if ($conn) {
    $conn->close();
}
?>
```

This code’s main purpose is to retrieve hostellers and warden’s data and display them using javascript code to an admin management page, it also handles fetching data counts as well as the addition of new hostellers and wardens. This code is chosen to be displayed because most of the pages on our project that uses CRUD uses similar logic to here, this one is the single most largest one to represent them.

Javascript code to handle the functionalities of the admin management page part 1:

```
// Hosteller & Warden Management JavaScript
document.addEventListener('DOMContentLoaded', function() {
  // Section toggle functionality
  const toggleButtons = document.querySelectorAll('.toggle-btn');
  const contentSections = document.querySelectorAll('.content-section');

  toggleButtons.forEach(button => {
   button.addEventListener('click', function() {
    // Remove active class from all buttons
    toggleButtons.forEach(btn => btn.classList.remove('active'));

    // Add active class to clicked button
    this.classList.add('active');

    // Hide all content sections
    contentSections.forEach(section => section.classList.remove('active'));

    // Show target content section
    const targetSection = document.getElementById(this.dataset.target);
    if (targetSection) {
     targetSection.classList.add('active');

     // Load data for the active section
     if (this.dataset.target === 'hosteller-section') {
      loadHostellers();
     } else if (this.dataset.target === 'warden-section') {
      loadWardens();
     }
    }
   });
  });

  // Filter panel toggle
  const toggleFiltersBtn = document.getElementById('toggle-filters');
const filterContent = document.querySelector('.filter-content');
const filterActions = document.querySelector('.filter-actions');

if (toggleFiltersBtn) {
  toggleFiltersBtn.addEventListener('click', function() {
    filterContent.classList.toggle('hidden');
    filterActions.classList.toggle('hidden');

    // Toggle icon
    const icon = this.querySelector('i');
    icon.classList.toggle('fa-chevron-up');
    icon.classList.toggle('fa-chevron-down');
  });
}

// Filter functionality
const applyFiltersBtn = document.getElementById('apply-filters');
const clearFiltersBtn = document.getElementById('clear-filters');

if (applyFiltersBtn) {
  applyFiltersBtn.addEventListener('click', function() {
    loadHostellers(1); // Load with page 1
  });
}

if (clearFiltersBtn) {
  clearFiltersBtn.addEventListener('click', function() {
    // Clear all filter inputs
    document.getElementById('semester-filter').value = '';
    document.getElementById('course-filter').value = '';
    document.getElementById('room-search').value = '';
    document.getElementById('name-search').value = '';
    //document.getElementById('address-search').value = '';

    // Load hostellers without filters
    loadHostellers(1);
  });
}

// Warden search functionality
```

```
const searchWardenBtn = document.getElementById('search-warden-btn');
const clearWardenSearchBtn = document.getElementById('clear-warden-search');

if (searchWardenBtn) {
  searchWardenBtn.addEventListener('click', function() {
    loadWardens(1);
  });
}

if (clearWardenSearchBtn) {
  clearWardenSearchBtn.addEventListener('click', function() {
    document.getElementById('warden-name-search').value = '';
    loadWardens(1);
  });
}

// Form submissions
const addHostellerForm = document.getElementById('add-hosteller-form');
const addWardenForm = document.getElementById('add-warden-form');

if (addHostellerForm) {
  addHostellerForm.addEventListener('submit', function(e) {
    e.preventDefault();

    const hostellerId = document.getElementById('hosteller-id').value;
    const action = hostellerId ? 'update_hosteller' : 'add_hosteller';

    const formData = new FormData(this);
    formData.append('action', action);

    fetch('', {
      method: 'POST',
      body: formData
    })
    .then(response => response.json())
    .then(data => {
      if (data.status === 'success') {
        document.getElementById('hosteller-modal').style.display = 'none';
        showMessage(data.message, 'success');
        loadHostellers(); // Reload hostellers
      } else {
        showMessage(data.message, 'error');
      }
    })
    .catch(error => {
      showMessage('Error: ' + error, 'error');
    });
  });
}

if (addWardenForm) {
  addWardenForm.addEventListener('submit', function(e) {
    e.preventDefault();

    const wardenId = document.getElementById('warden-id').value;
    const action = wardenId ? 'update_warden' : 'add_warden';

    const formData = new FormData(this);
    formData.append('action', action);

    fetch('', {
      method: 'POST',
      body: formData
    })
    .then(response => response.json())
    .then(data => {
      if (data.status === 'success') {
        document.getElementById('warden-modal').style.display = 'none';
        showMessage(data.message, 'success');
        loadWardens(); // Reload wardens
      } else {
        showMessage(data.message, 'error');
      }
    })
    .catch(error => {
      showMessage('Error: ' + error, 'error');
    });
  });
}
// Load hostellers on initial page load
loadHostellers();

// Initial toggle state
filterContent.classList.remove('hidden');
filterActions.classList.remove('hidden');
});

// Function to show message
function showMessage(message, type) {
const messageContainer = document.getElementById('message-container');
const alertClass = type === 'success' ? 'alert-success' : 'alert-danger';

messageContainer.innerHTML = `
  <div class="alert ${alertClass}">
    ${message}
  </div>
`;

// Auto-hide message after 5 seconds
setTimeout(() => {
  messageContainer.innerHTML = '';
}, 5000);
}

// Function to load hostellers with filters and pagination
function loadHostellers(page = 1) {
const formData = new FormData();
formData.append('action', 'get_hostellers');
formData.append('page', page);

// Add filters if they have values
const semester = document.getElementById('semester-filter').value;
const course = document.getElementById('course-filter').value;
const room = document.getElementById('room-search').value;
const name = document.getElementById('name-search').value;
//const address = document.getElementById('address-search').value;

if (semester) formData.append('semester', semester);
if (course) formData.append('course', course);
if (room) formData.append('room', room);
if (name) formData.append('name', name);
//if (address) formData.append('address', address);

fetch('', {
  method: 'POST',
  body: formData
})
.then(response => response.json())
.then(data => {
  if (data.status === 'success') {
    renderHostellerTable(data.data);
    renderPagination(data.pagination, 'hosteller-pagination', loadHostellers);
    document.getElementById('results-count').textContent = data.pagination.total_items;
  } else {
    showMessage(data.message, 'error');
  }
})
.catch(error => {
  showMessage('Error loading hostellers: ' + error, 'error');
});
}

// Function to load wardens with filters and pagination
function loadWardens(page = 1) {
const formData = new FormData();
formData.append('action', 'get_wardens');
formData.append('page', page);

// Add filter if it has a value
const nameSearch = document.getElementById('warden-name-search').value;
if (nameSearch) formData.append('name', nameSearch);

fetch('', {
  method: 'POST',
  body: formData
})
.then(response => response.json())
.then(data => {
  if (data.status === 'success') {
      renderWardenTable(data.data);
      renderPagination(data.pagination, 'warden-pagination', loadWardens);
      document.getElementById('warden-results-count').textContent =
data.pagination.total_items;
    } else {
      showMessage(data.message, 'error');
    }
  })
  .catch(error => {
    showMessage('Error loading wardens: ' + error, 'error');
  });
}

// Function to render hosteller table
function renderHostellerTable(hostellers) {
  const tableBody = document.getElementById('hosteller-table-body');
  tableBody.innerHTML = '';

  if (hostellers.length === 0) {
    tableBody.innerHTML = `
      <tr>
        <td colspan="7" class="no-results">No hostellers found</td>
      </tr>
    `;
    return;
  }

  hostellers.forEach(hosteller => {
    const row = document.createElement('tr');
    row.innerHTML = `
      <td>${hosteller.id}</td>
      <td>${hosteller.name}</td>
      <td>${hosteller.room}</td>
      <td>${hosteller.course}</td>
      <td>${hosteller.semester}</td>
      <td>${hosteller.phone_number}</td>
      <td>
        <button class="action-btn edit-btn" data-id="${hosteller.id}">
          <i class="fas fa-edit"></i> Edit
        </button>
        <button class="action-btn delete-btn" data-id="${hosteller.id}">
          <i class="fas fa-trash"></i> Delete
        </button>
      </td>
    `;

    tableBody.appendChild(row);
  });

  // Add event listeners to buttons
  addActionButtonEventListeners('hosteller');
}

// Function to render warden table
function renderWardenTable(wardens) {
  const tableBody = document.getElementById('warden-table-body');
  tableBody.innerHTML = '';

  if (wardens.length === 0) {
    tableBody.innerHTML = `
      <tr>
        <td colspan="6" class="no-results">No wardens found</td>
      </tr>
    `;
    return;
  }

  wardens.forEach(warden => {
    const row = document.createElement('tr');
    row.innerHTML = `
      <td>${warden.id}</td>
      <td>${warden.name}</td>
      <td>${warden.building}</td>
      <td>${warden.phone_number}</td>
      <td><span class="status-badge
${warden.status.toLowerCase()}">${warden.status}</span></td>
      <td>
        <button class="action-btn edit-btn" data-id="${warden.id}">
          <i class="fas fa-edit"></i> Edit
        </button>
        <button class="action-btn delete-btn" data-id="${warden.id}">
          <i class="fas fa-trash"></i> Delete
        </button>
      </td>
    `;

    tableBody.appendChild(row);
  });

  // Add event listeners to buttons
  addActionButtonEventListeners('warden');
}

// Function to add event listeners to action buttons
function addActionButtonEventListeners(type) {
  // Edit buttons
  const editButtons = document.querySelectorAll(`#${type}-table-body .edit-btn`);
  editButtons.forEach(button => {
    button.addEventListener('click', function() {
      const id = this.dataset.id;
      if (type === 'hosteller') {
        window.editHosteller(id);
      } else {
        window.editWarden(id);
      }
    });
  });

  // Delete buttons
  const deleteButtons = document.querySelectorAll(`#${type}-table-body .delete-btn`);
  deleteButtons.forEach(button => {
    button.addEventListener('click', function() {
      const id = this.dataset.id;
      if (confirm(`Are you sure you want to delete this ${type}?`)) {
        deleteRecord(type, id);
      }
    });
  });
}
```

```
// Function to delete a record
function deleteRecord(type, id) {
  const formData = new FormData();
  formData.append('action', `delete_${type}`);
  formData.append('id', id);

  fetch('', {
    method: 'POST',
    body: formData
  })
  .then(response => response.json())
  .then(data => {
    if (data.status === 'success') {
      showMessage(data.message, 'success');
      if (type === 'hosteller') {
        loadHostellers();
      } else {
        loadWardens();
      }
    } else {
      showMessage(data.message, 'error');
    }
  })
  .catch(error => {
    showMessage(`Error deleting ${type}: ` + error, 'error');
  });
}

// Function to render pagination
function renderPagination(pagination, containerId, loadFunction) {
  const container = document.getElementById(containerId);
  if (!container) return;

  container.innerHTML = '';

  if (pagination.total_pages <= 1) {
    return;
  }

  // Create pagination controls
```

```
const paginationNav = document.createElement('div');
paginationNav.className = 'pagination-nav';

// Previous button
const prevBtn = document.createElement('button');
prevBtn.className = 'pagination-btn';
prevBtn.innerHTML = '<i class="fas fa-chevron-left"></i> Previous';
prevBtn.disabled = pagination.current_page === 1;
prevBtn.addEventListener('click', () => loadFunction(pagination.current_page - 1));
paginationNav.appendChild(prevBtn);

// Page buttons
const startPage = Math.max(1, pagination.current_page - 2);
const endPage = Math.min(pagination.total_pages, pagination.current_page + 2);

// First page
if (startPage > 1) {
  const firstBtn = document.createElement('button');
  firstBtn.className = 'pagination-btn';
  firstBtn.textContent = '1';
  firstBtn.addEventListener('click', () => loadFunction(1));
  paginationNav.appendChild(firstBtn);

  if (startPage > 2) {
    const ellipsis = document.createElement('span');
    ellipsis.className = 'pagination-ellipsis';
    ellipsis.textContent = '...';
    paginationNav.appendChild(ellipsis);
  }
}

// Page numbers
for (let i = startPage; i <= endPage; i++) {
  const pageBtn = document.createElement('button');
  pageBtn.className = `pagination-btn ${i === pagination.current_page ? 'active' : ''}`;
  pageBtn.textContent = i;
  pageBtn.addEventListener('click', () => loadFunction(i));
  paginationNav.appendChild(pageBtn);
}
```

```
    // Last page
    if (endPage < pagination.total_pages) {
      if (endPage < pagination.total_pages - 1) {
        const ellipsis = document.createElement('span');
        ellipsis.className = 'pagination-ellipsis';
        ellipsis.textContent = '...';
        paginationNav.appendChild(ellipsis);
      }

      const lastBtn = document.createElement('button');
      lastBtn.className = 'pagination-btn';
      lastBtn.textContent = pagination.total_pages;
      lastBtn.addEventListener('click', () => loadFunction(pagination.total_pages));
      paginationNav.appendChild(lastBtn);
    }

    // Next button
    const nextBtn = document.createElement('button');
    nextBtn.className = 'pagination-btn';
    nextBtn.innerHTML = 'Next <i class="fas fa-chevron-right"></i>';
    nextBtn.disabled = pagination.current_page === pagination.total_pages;
    nextBtn.addEventListener('click', () => loadFunction(pagination.current_page + 1));
    paginationNav.appendChild(nextBtn);

    container.appendChild(paginationNav);
  }
```

The above javascript codes handle the displaying and filter functionalities of the Admin Management page, Part 1 of the code handles the previously mentioned functionalities while Part 2 of the code handles the new hosteller and warden addition using modal forms which are simple and optimal way of adding more functionalities without cluttering the dashboard.

4.2 Comments and Descriptions of Coding Segments

Code Documentation Practices

During the development of the **Hostel Room Management App**, clear and informative comments were added throughout the codebase to enhance readability and ensure easy maintenance. The purpose of these comments was to explain complex logic, describe the functions and methods used, and highlight important sections of the code for future developers or reviewers.

Here’s how the code was documented:

- **Function Headers**: Each major function or class is preceded by a short comment explaining its purpose, the parameters it accepts, and the output it returns. This helps developers quickly understand the function’s role in the system.
- **Inline Comments**: Within functions, brief inline comments were used to explain crucial steps, such as database queries, condition checks, or data processing. These comments help clarify the logic behind specific operations.
- **Section Breaks**: Larger files were divided into logical sections with comment headers, making it easier to navigate through different parts of the application (e.g., authentication, dashboard management, or payment processing).
- **Clarifying Complex Logic**: In areas where complex conditions or algorithms were used—like checking room availability or handling different user types (e.g., hostellers vs. admins)—detailed comments were added to explain the thought process behind the code.

By following these documentation practices, the codebase remains accessible and understandable to anyone working on or reviewing the project in the future. This approach reduces confusion, improves collaboration, and makes future updates easier to implement.

An example of this can be seen in the previous pages on the backend code for editing and handling user password:

```
// Check if password is provided
    if (!empty($password)) {
      // Hash new password
      $hashed_password = password_hash($password, PASSWORD_DEFAULT);

      // Update with new password
```

```
            $stmt = $conn->prepare("UPDATE hostellers SET name=?, password=?, course=?,
semester=?, phone_number=?, room=?, building=? WHERE id=?");
            $stmt->bind_param("sssssssi", $name, $hashed_password, $course, $semester,
$contact, $room, $building, $id);
        } else {
            // Update without changing password
            $stmt = $conn->prepare("UPDATE hostellers SET name=?, course=?, semester=?,
phone_number=?, room=?, building=? WHERE id=?");
            $stmt->bind_param("ssssssi", $name, $course, $semester, $contact, $room,
$building, $id);
        }
```

This code is used to update users passwords, hashing and storing them if it is doesn't exists, keeping it blank to keep the old password etc.

4.3 Standardization of the code

To ensure clarity, consistency, and professionalism in the **Hostel Room Management App**, several standardization practices were implemented throughout the development process.

Key techniques used include:

- **Consistent Naming Conventions**: Variables, functions, and database fields were named using the **camelCase** format. This made the code more readable and maintainable, helping developers quickly understand the purpose of each element.
- **Structured File Organization**: The project was divided into logically structured folders, including **frontend**, **backend**, and **database scripts**. This ensured that files with similar functionalities were grouped together, improving navigation and collaboration.
- **Uniform Formatting**: Consistent formatting practices—such as proper indentation, regular use of semicolons (where needed), and clean HTML structure—were followed to improve readability and make the code easier to follow.
- **Reusability**: Common functions, such as those for form validation, data fetching, and database operations, were written in a reusable way to avoid duplication. This ensured that similar tasks didn't require redundant code, promoting efficiency and maintainability.

- **Error Handling Standardization**: All user inputs were validated before being processed. Errors were handled consistently by displaying clear, informative messages to users, helping them understand and resolve issues quickly.

By applying these standardization practices, the codebase remains well-organized, easy to read, and future-proof for updates or scaling.

Tools Used for Standardization

To further enhance consistency and maintain a clean coding style, the following tools and extensions were employed during development:

- **Prettier**: An automatic code formatter that ensured consistent styling across all files, maintaining proper indentation, spacing, and bracket placement.
- **IntelliSense (VS Code)**: Provided intelligent code completions, parameter information, and quick documentation, making it easier to maintain consistent function usage and syntax.
- **VS Code Built-in Formatting**: The default formatting tools in Visual Studio Code were also used to quickly format documents and maintain a uniform code style across the project.

Using these tools significantly improved the overall quality, readability, and maintainability of the code, while reducing the risk of formatting errors and enhancing developer productivity.

4.4 Error Handling Mechanism

To ensure the **Hostel Room Management App** remains reliable and stable, an effective error handling mechanism was implemented across both the frontend and backend of the system.

Key approaches to error handling include:

- **Form Validation**: On the frontend, all major input forms (e.g., login, complaints, payments) include **client-side validation**. This prevents invalid or incomplete data from being submitted. For instance, users are alerted if required fields are empty or if data formats (such as email or numeric fields) are incorrect.

- **Server-side Validation**: In addition to frontend checks, **backend validation** was implemented to ensure data integrity and security. This step helps prevent malicious or incorrect data from impacting the database or server logic.
- **Error Messages for Users**: **User-friendly error messages** are displayed whenever a form submission fails, a login attempt is unsuccessful, or issues arise during data retrieval. These messages guide users in correcting their input or retrying the action, ensuring a smoother experience.
- **Try-Catch Blocks (Backend)**: For backend functions interacting with the database or handling files, **try-catch blocks** are used to gracefully catch and respond to unexpected errors. This approach helps manage issues like database connectivity problems or failed queries without crashing the system.
- **Default/Fallback Handling**: In certain components, **fallback behaviors** were introduced. For example, an empty state or a "No data found" message is displayed when data is unavailable, rather than breaking the page or leaving it blank.
- **Logging for Debugging**: During development, **error messages** and exceptions were logged using tools like console.log and console.error in JavaScript, or equivalent logging methods in PHP. This made it easier to trace and debug issues as they occurred.

These error handling practices contribute to a smoother user experience, improve system robustness, and ensure the app is easier to maintain.

Below are some examples from the project code:

```
fetch(apiUrl)
     .then(response => {
      return response.text();
       })
       .then(text => {
         console.log('Raw server response:', text);
         try {
           const data = JSON.parse(text);
           console.log('Parsed data:', data);
           return data;
         } catch(e) {
           console.error('JSON parse error:', e);
           throw new Error('Invalid JSON response from server');
         }
       })
     .then(data => {
            // Clear existing content if not appending
            if (!append) {
               requestsList.innerHTML = '';
            }

            if (data.data.length === 0 && !append) {
               requestsList.innerHTML = '<div class="no-requests">No checkout requests 
found.</div>';
               loadMoreBtn.style.display = 'none';
               return;
            }

            if (!data || !data.data) {
              console.error('Unexpected response format:', data);
              requestsList.innerHTML = '<div class="error">Server returned unexpected data 
format.</div>';
              return;
           }

            // Render each request
            data.data.forEach(request => {
               const requestCard = createRequestCard(request);
               requestsList.appendChild(requestCard);
            });

            // Update load more button visibility
            loadMoreBtn.style.display = data.pagination.has_more ? 'block' : 'none';
         })
         .catch(error => {
           console.error('Error loading checkout requests:', error);
           requestsList.innerHTML = '<div class="error">Failed to load requests. Please try 
again.</div>';
        });
```

This example shows two type of error handling technique, mainly try catch method and using console.log to print out errors

4.5 Validation Checks

Validation checks were an essential part of the Hostel Room Management App since there are a lot of features making use of input forms. These checks were applied on both the frontend and backend to provide a reliable and secure user experience.

Key validation strategies used:

Client-Side Validation (Frontend):

- Required fields such as name, email, room number, and complaint description were enforced using HTML5 attributes and JavaScript.
- Input types were restricted (e.g., using type="number" for numeric fields, type="email" for email addresses).
- Simple JavaScript checks were used to ensure values met minimum requirements (e.g., non-empty fields, valid email formats, password length).

Example (one html post form from the project code):

```
<form class="report-form" id="checkout-form" method="post" enctype="multipart/form-
data">
          <label for="subject"><img src="../images/Hosteller/Reports/subject.png"
class="reportpage-icon">Subject</label>
          <input type="text" id="subject" name="subject" placeholder="Enter your
reasonings here" required>

          <label><img src="../images/Hosteller/Reports/who.png" class="reportpage-
icon">Leave Duration</label>
          <div class="time-inputs">
            <div class="date-range">
              <label for="from-date">From:</label>
              <input type="date" id="from-date" name="from_date" required>
              <input type="time" id="from-time" name="from_time" required>
            </div>
            <div class="date-range">
              <label for="to-date">To:</label>
              <input type="date" id="to-date" name="to_date" required>
              <input type="time" id="to-time" name="to_time" required>
            </div>
```

```
                </div>
                <input type="hidden" id="time" name="time" value="">

                <label for="description"><img src="../images/Hosteller/Reports/description.png"
class="reportpage-icon">Description</label>
                <textarea id="description" name="description" placeholder="Describe your
reason for leaving..." required></textarea>

                <label for="file"><img src="../images/Hosteller/Reports/folder.png"
class="reportpage-icon">Attach File (Optional)</label>
                <input type="file" id="file" name="file">

                <button type="submit">Submit Request</button>
 </form>
```

This example shows the use of 'type' for inputting different information

Server-Side Validation (Backend):

- Additional checks were performed in the backend to verify that submitted data was not only valid but also safe.
- Fields were sanitized and trimmed before being used in queries to prevent issues such as SQL injection or data corruption.

Validation and sanitizing input example from the project code:

```
// Validate recipient
  $valid_recipients = ['Warden', 'Admin'];
  $recipient = $_POST['recipient'];
  if (!in_array($recipient, $valid_recipients)) {
    $error_message = "Invalid recipient";
    // Handle error appropriately
  }

  // Sanitize text inputs
  $subject = htmlspecialchars(trim($_POST['subject']));
  $description = htmlspecialchars(trim($_POST['description']));
```

Duplicate and Conflict Prevention:

- Validation was implemented to prevent duplicate entries, such as trying to assign a room that's already booked or submitting the same complaint multiple times.

- For payment updates, validation ensured that the same fee wasn’t marked as "paid" twice.

Feedback for Users:

- Clear error messages were displayed next to invalid fields, guiding users to correct input mistakes.
- On successful validation, appropriate success messages were shown to confirm correct submission.

By implementing robust validation checks, the system prevents incorrect data from being entered and ensures the smooth functioning of critical features like room booking, login, payments, and complaint submissions.

Summary

The coding stage of this project was where everything really came to life. I focused on writing clean, organized, and reusable code that could support the core features of the hostel management system — from room bookings and user logins to payment tracking and complaint handling. I used consistent naming, proper file structuring, and tools like Prettier and IntelliSense to keep everything neat and standardized.

I also made sure to add clear comments and validation checks so that both the users and the system stay on track — avoiding errors and handling unexpected inputs smoothly. Overall, this phase was all about turning ideas into working features while keeping the code maintainable and ready for future improvements.

5. TESTING

5.1 Testing techniques and technique strategies

Although no formal testing frameworks were used, a practical and hands-on approach was employed throughout the development of the **Hostel Room Management App**. The testing process primarily consisted of **manual testing** techniques, with each module tested in real-time during and after development.

Key strategies included:

1. **Incremental Testing**: Features were developed and tested incrementally. After completing each major function (e.g., login, room allocation, payment updates), it was manually tested to confirm it worked as intended.
2. **Bug Fixing Through Iteration**: Errors and unexpected behaviors encountered during testing were addressed immediately. The code was reviewed and adjusted in short feedback loops, allowing for quick resolution of issues.
3. **User Flow Simulation**: Common user actions were simulated, including signing up, logging in, requesting rooms, submitting complaints, and checking payment statuses. This ensured the user experience was smooth and error-free.
4. **Cross-Role Testing**: Features were tested under different user roles (admin, hosteller) to ensure that access permissions, data visibility, and role-specific actions worked correctly for each user type.
5. **Data Validation Testing**: Forms were filled with incorrect, incomplete, or unexpected input to evaluate how well the validation and error handling mechanisms worked. This helped ensure the system was resilient against invalid data.

While this approach was informal, it enabled the efficient identification and resolution of issues during development. All core features were verified manually, and the system was refined based on real-world usage and observation.

5.2 Test Case

To verify the reliability and functionality of the Hostel Room Management App, several key test cases were manually executed. These focused on validating core features, handling edge cases, and ensuring the system responded correctly to different user actions.

Below are some of the key test cases that were carried out:

TEST CASE	DESCRIPTION	EXPECTED RESULT	STATUS
LOGIN WITH VALID CREDENTIALS	Hosteller/Admin logs in with correct email and password	User is redirected to their respective dashboard	✓ Passed
LOGIN WITH INVALID CREDENTIALS	Incorrect login details submitted	Error message is shown, login is denied	✓ Passed
USER DATA FILTERING	Admin filters hostellers based on criteria like room number or payment status	Table updates to show only matching records	✓ Passed
ONLINE PAYMENT WITH INVALID/MISSING INFO	Hosteller submits payment form with missing or incorrect details	Payment fails, and an error message is displayed	✓ Passed
PAYMENT STATUS UPDATE BY ADMIN	Admin updates rent/mess fee status for a hosteller	Status changes are saved and reflected in the hosteller's view	✓ Passed
COMPLAINT SUBMISSION	Hosteller submits a complaint	Complaint is saved to the database and visible to admin/warden	✓ Passed
FORM SUBMISSION WITH MISSING FIELDS	Form (e.g., complaint or signup) submitted with empty required fields	Error message is shown, submission is blocked	✓ Passed
UNAUTHORIZED ACCESS TO ADMIN PAGE	Hosteller tries to manually access admin URL	Access is denied or redirected	✓ Passed

Table 1.6: Test Cases

These test cases were executed manually during development, and helped ensure that all major components of the system performed as intended under various real-world scenarios.

5.3 Test Reports

The testing phase of the Hostel Room Management App confirmed that all core functionalities were working as intended. Manual testing was carried out on key modules including login, data filtering, form submissions, complaint handling, and admin operations.

All identified test cases were executed successfully, with expected outcomes being met in each scenario. Minor issues and bugs discovered during testing were fixed promptly. Special attention was given to edge cases, such as invalid form inputs or unauthorized access, to ensure the system responded securely and gracefully.

No critical failures were observed during testing, and the app demonstrated stable performance across all user roles and key interactions.

Summary

The testing process for the Hostel Room Management App was informal but thorough, focusing on real-world usage and core functionality. Manual testing techniques and practical test cases helped identify and resolve bugs early during development. All critical features, from user authentication to data filtering and admin controls, were tested for correctness, usability, and reliability. The results showed that the system performs well under expected conditions, providing a smooth and secure experience for both hostellers and administrators.

6. DRAWBACKS AND LIMITATIONS

6.1 Conclusion

While the **Hostel Room Management App** successfully delivers core functionalities for hostel administration — including room allocation, complaint tracking, payment management, and user access control — several limitations and areas for improvement have become apparent during development and testing.

Current Limitations

1. **Lack of Real-Time Notifications**: Currently, users and admins must manually check for updates (e.g., complaint responses or room assignments). Integrating email or in-app notifications would significantly improve communication and responsiveness within the system.
2. **Security Concerns**: Although basic access control and validation are in place, the application lacks robust security mechanisms. For instance, sensitive user data is not encrypted, and session timeout features are absent. Future enhancements, such as hashed passwords, login rate limiting, and enforcing HTTPS, would be critical for production-level deployment.
3. **Lack of Mobile Optimization**: While the app is usable on smaller screens, the layout is not fully responsive in all areas, impacting user experience for mobile and tablet users.
4. **Absence of Automated Testing**: Since no formal automated testing was performed, there is a potential for edge-case bugs or usability issues, particularly in less common workflows. Introducing a proper QA/testing framework would help catch such issues early.
5. **Manual Administrative Tasks**: All updates — such as approving complaints or changing payment status — must be performed manually by the admin. Automating some processes or introducing batch operation tools could save time and reduce administrative workload.

Future Improvements

Despite these drawbacks, the app lays a solid foundation for digital hostel management. Here are some potential upgrades for future versions:

1. **Real-Time Notifications**: Adding real-time notifications (e.g., email or in-app alerts) for key events like room assignments, payment status changes, or complaint responses would enhance communication between hostellers and admins.
2. **Enhanced Security**: Strengthening security features, such as encrypting sensitive data (e.g., passwords, payment details) and introducing session timeouts, would improve the platform's overall security. Enforcing HTTPS would also provide a secure browsing experience.
3. **Improved Mobile Responsiveness**: A fully responsive design would optimize the app for mobile and tablet users, providing a smooth user experience across all device types.
4. **Automated Testing and Quality Assurance**: Implementing a formal automated testing framework (e.g., unit tests, integration tests) would help identify issues more efficiently and ensure stability as new features are added. Regular QA testing would help catch edge cases missed during manual testing.
5. **User Role Flexibility and Permissions**: Future versions could introduce more granular user roles with custom permissions (e.g., wardens or support staff), allowing for more specialized access control and management of responsibilities.
6. **Automated Admin Tasks**: Automating administrative tasks — such as updating room assignments, approving complaints, or sending payment reminders — would save time for administrators and improve operational efficiency.
7. **Integration with Payment Gateways**: Integrating online payment gateways (e.g., PayPal, Stripe) would enable hostellers to make payments directly through the app, eliminating the need for manual intervention and enhancing the overall user experience.

By addressing these limitations and implementing the suggested improvements, the **Hostel Room Management App** could become more robust, secure, and user-friendly, further enhancing its functionality for both hostellers and administrators.

6.2 Bibliography

1. **SuperSimpleDev (YouTube Channel)** - Tutorial channel for learning HTML, CSS, and other web development fundamentals.
   URL: https://www.youtube.com/c/SuperSimpleDev
2. **Stack Overflow** - Community-driven Q&A platform for developers, providing solutions and coding insights.
   URL: https://stackoverflow.com
3. **FrontendPractice.com** - Resource for frontend development challenges and practice exercises.
   URL: https://frontendpractice.com
4. **ChatGPT by OpenAI** - AI-powered assistant for coding guidance, especially in using javascript and complex backend logic
   URL: https://chat.openai.com
5. **Flaticon** - Source for custom icons used in the project.
   URL: https://www.flaticon.com
6. **Unsplash** - Free stock images used in the project.
   URL: https://unsplash.com
7. **Dribbble** - Platform for frontend design inspiration and creative resources.
   URL: https://dribbble.com
8. **Draw.io** - Online tool used for creating all project diagrams and flowcharts.
   URL: https://app.diagrams.net