DATABASE SYSTEMS PROJECT

IBA ROOM **BOOKING** SYSTEM

Team Members:

Muskan 28394

Abdullah 28084

Kashish 29105

Business Scenario

1. Introduction

The **IBA Room Booking System** is designed to digitize and streamline the room booking process at the Institute of Business Administration (IBA), Karachi. Currently, room bookings for classrooms, breakout rooms, labs, and conference spaces are managed manually through email requests and shared schedules maintained in spreadsheets. This process is time-consuming, error-prone, and lacks transparency, often resulting in scheduling conflicts, inefficient utilization of rooms, and delayed approvals for events and tutorials.

The goal of this system is to provide **a centralized online platform where students, building incharges, and the program office can efficiently manage and track room bookings in real-time.** The system ensures that rooms are allocated fairly, conflicts are minimized, and approval workflows are automated, while also allowing administrators to maintain oversight and flexibility.

2. Summary of Interview with Program Office Head (Sir Shahmuneer Khan)

2.1 Types of Bookings

Sir Shahmuneer identified the following room booking categories:

- Classrooms (regular lectures and labs)
- Makeup sessions and tutorials
- Seminars, conferences, workshops, and trainings
- Society meetings
- Third-party rent-outs

2.2 Responsibilities

- **Program Office (PO):** Manages bookings for academic activities, classrooms, and labs
- **Building Incharge:** Manages breakout rooms and meeting/conference rooms
- **Administration:** Handles sports and other facility bookings

2.3 Current Booking Process

- Society events are first approved by OSA and then forwarded to Baig sb for final approval.
- Regular lectures, classrooms, and labs are booked by the PO.
- Tutorials are requested via email to Baig sb, who checks availability manually.

- Availability is verified using two spreadsheets: the class schedule sheet and the venue booking sheet.
- Once approved, an entry is made manually in the venue sheet.

2.4 Cancellation and Modifications

- Students may cancel bookings by emailing the PO, ideally one day before the booking.
- The system should allow automatic approval for available slots but must also allow cancellations for high-priority events.

2.5 Room Types and Allocation Preferences

- **Faculty lounges, conference rooms, breakout rooms, specialized labs, and classrooms** exist on campus.
- Certain rooms are preferentially allocated based on department needs (e.g., Tabba's blackboards are prioritized for CS or Math departments).

2.6 Rules for Students

- Approved bookings store user details, date, time, venue, and purpose.
- Recurring bookings are allowed.
- Students can book multiple slots as long as rooms are available.
- Full-day occupancy is generally not possible.

2.7 Campus-wide Events

- During events such as Enigma or exams, the entire campus may be booked, pausing regular bookings.

2.8 No-Show Policy

- There are no penalties for booking a room and not showing up.

3. Analysis of Similar Applications

To design a robust system, three applications were analyzed:

Application	Purpose	Key Features	Relevance to IBMS
BookMe.pk	Seat & venue booking in Pakistan	Real-time availability, online booking, OTP/email verification, confirmation notifications	Demonstrates real-time booking and automated confirmations
DePaul University Library Portal (LibCal)	Room & resource booking in academic institutions	Calendar-based scheduling, recurring bookings, user authentication, resource allocation	Provides an academic context similar to classrooms and labs
Calendly	Online scheduling & event booking	Time-slot selection, automatic conflict checking, recurring events, notifications	Shows automated approval workflow and slot-based booking logic

Key Insights:

- All three apps provide automated availability checks, user authentication, and notifications.
- Academic portals (like LibCal) handle recurring bookings, which is critical for classroom reservations.
- Calendly's conflict checking informs how to implement slot validation in IBMS.
- BookMe.pk emphasizes local user experience and seat bookings using clear UI and visuals.

4. Justification for IBA Room Booking System

The proposed IRBS system addresses the limitations of the current manual process:

- **Real-time availability:** Automatically prevents double-booking and checks against class schedules.
- **Automated notifications:** Users are informed when bookings are approved, rejected or cancelled in the "Notifications" feature.
- **Department-aware allocation:** Ensures fair allocation of rooms with departmental priorities.

- **Role-based access:** PO, building incharges, and students have defined permissions, maintaining oversight.
- **Recurring booking support:** Students and faculty can book recurring slots while maintaining availability integrity.

By integrating these features, IRBS will increase efficiency, transparency, and accountability in the IBA room booking process.

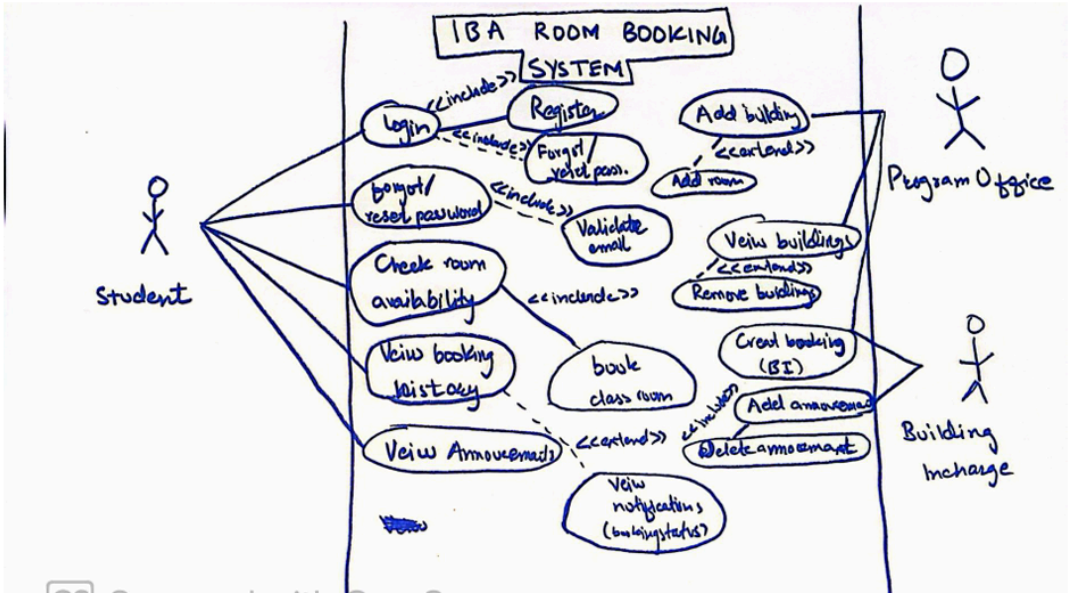
BUSINESS RULES:

- Each room of type classroom has a class schedule (room_id, class_id) that it follows. A classroom booking is approved if no class is happening at that time on that day and the room is not approved for booking by any other user. An approved booking can be cancelled by the user and rejected by the PO before the slot on the reserved day begins.
- Each booking must have booking_id(pk), user_id(fk), room_id(fk), booking_date, start_time, end_time, purpose, and status (Approved, Cancelled, Rejected). A booking is approved by default because user picks a slot and room from available rooms only. The end_time of the booking has to be greater than the start_time of the booking slot. Each booking slot (start_time-end_time) is of 1 hour 15 minutes. A classroom booking is approved if no class is happening at that time on that day and the room is not approved for booking by any other user. A Breakout room booking is approved if it wasn't already approved by somebody else for the same slot and day. An approved booking can be cancelled by the user and rejected by the PO before the slot on the reserved day begins. Whenever a booking is approved, rejected, or cancelled, user receives a notification.
- Each announcement must have announcement_id(pk), title, date_posted, and created_date. Optionally, it can also have description of the announcement. ONE BUILDING INCHARGE CAN POST zero or MANY ANNOUNCEMENTS AND ONE ANNOUNCEMENT is posted by one building incharge. Each Building Incharge can only delete announcements posted by them.
- Each lecture must have schedule_id (pk), room_id(fk), day_of_week (out of Monday, Tuesday, Wednesday, Thursday, Friday, and Saturday), start_time, end_time, and course_code. The start_time must be less than the end_time.
- Each User must have ERP (PK), name, email (must be unique), user_password, role (Student, Building Incharge, Program Office), and phone_number. The email must end with either '@iba.edu.pk' or '@khi.iba.edu.pk'. The length of the password must be greater than 7 and less than 17. The phone number must be of 11 digits and start with 03. One user can book zero or many rooms and one room is booked by one user at a time for the same slot and day. EACH USER CAN REQUEST TO BOOK A ROOM. HOWEVER, STUDENT CAN REQUEST FOR BOTH CLASSROOMS

AND BREAKOUTS whereas PROGRAM OFFICE CAN REQUEST A CLASSROOM ONLY AND BUILDING INCHARGE CAN REQUEST A BREAOUT ROOM ONLY.

- Each Student must have ERP(PK)(FK), program (out of BSCS, BBA, BSSS, BSEM, BSMT, BSACF, BSBA), and intake_year. The ERP must be of 5 digits and the intake_year must be of 4 digits.
- Each Incharge must have an incharge_id (PK) OF 5 digits, and unique building_id(FK).
- Each room must have room_id(PK), building_id(fk), room_name (unique within a building), and room_type (Classroom, Breakout). One classroom can have zero or many classes and one class can happen in one room. A room is available on a day and slot if no class is happening at that time and it is not approved for booking by anyone else. Only available rooms can be booked. One room can have zero or many bookings but one booking can be of one room only.
- Each building must have building_id (PK) and a unique building_name. Each building is multi-floored and each floor has many rooms (room_id). ONE ROOM CAN BE IN ONE BUILDING ONLY. Each building is assigned one BUILDING INCHARGE (INCHARGE_ID) and each BI is assigned to one and only one building.
- Only PO can add buildings and rooms within those buildings. Only PO can assign Building Incharge to Buildings.

Use Case Diagram



ENTITIES, ATTRIBUTES & MULTIPLICITY CONSTRAINTS

1. USER_ENTITY (User_Table)

Attributes:

- ERP (PK): 5-digit unique identifier (10000-99999)
- name: Full name (100 chars max)
- email: Unique IBA email (@iba.edu.pk or @khi.iba.edu.pk)
- user_password: 8-16 character password
- role: Student, BuildingIncharge, or ProgramOffice
- phone_number: 11-digit Pakistani mobile (03XXXXXXXXX)

Multiplicity Constraints:

- User to Student: **1:1** (Each user has at most one student record)
- User to Incharge: **1:1** (Each user has at most one incharge role)
- User to Booking: **1:M** (One user can make many bookings)
- User to Announcement: **1:M** (One user(role == "BuildingIncharge") can post many announcements)

2. STUDENT_ENTITY (Student Table)

Attributes:

- ERP (PK, FK): References User_Table
- program: Valid IBA program (BBA, BSACF, BSECO, BSBA, BSSS, BSCS, BSEM, BSMT)
- intake_year: 4-digit enrollment year

Multiplicity Constraints:

- Student to User: **1:1** (Each student corresponds to exactly one user)
- Student to Booking: **1:M** (One student can make many bookings)

3. BUILDING_ENTITY (Building Table)

Attributes:

- building_id (PK): Auto-generated unique identifier
- building_name: Unique building name (50 chars max)

Multiplicity Constraints:

- Building to Room: **1:M** (One building contains many rooms)
- Building to Incharge: **1:1** (Each building has exactly one incharge)

4. INCHARGE_ENTITY (Incharge Table)

Attributes:

- incharge_id (PK, FK): References User_Table (BuildingIncharge role)
- building_id (FK, UNIQUE): References Building (one per building)

Multiplicity Constraints:

- Incharge to User: **1:1** (Each incharge is exactly one user)
- Incharge to Building: **1:1** (Each incharge manages exactly one building)
- Incharge to Announcement: **1:M** (One incharge can post many announcements)

5. ROOM_ENTITY (Room Table)

Attributes:

- room_id (PK): Auto-generated unique identifier
- building_id (FK): References Building
- room_name: Room identifier (15 chars max, unique within building)
- room_type: CLASSROOM or BREAKOUT

Multiplicity Constraints:

- Room to Building: **M:1** (Many rooms belong to one building)
- Room to Schedule: **1:M** (One room can have many scheduled classes)
- Room to Booking: **1:M** (One room can have many bookings)

6. SCHEDULE_ENTITY (Schedule Table)

Attributes:

- schedule_id (PK): Auto-generated unique identifier
- room_id (FK): References Room
- day_of_week: MONDAY through SATURDAY

- start_time: Class start timestamp
- end_time: Class end timestamp (must be > start_time)
- course_code: Academic course identifier (20 chars max)

Multiplicity Constraints:

- Schedule to Room: **M:1** (Many schedules belong to one room)

7. BOOKING_ENTITY (Booking Table)

Attributes:

- booking_id (PK): Auto-generated unique identifier
- ERP (FK): References User_Table (student)
- room_id (FK): References Room
- booking_date: Date of booking (not in past)
- start_time: Booking start timestamp
- end_time: Booking end timestamp (must be > start_time, max 1hr15min)
- purpose: Booking reason (500 chars max)
- status: Approved, Rejected, or Cancelled (default: Approved)
- created_date: Auto-generated creation timestamp

Multiplicity Constraints:

- Booking to User: **M:1** (Many bookings belong to one user)
- Booking to Room: **M:1** (Many bookings belong to one room)
- Booking to User (for admin actions): **M:M** (Many bookings can be managed by many admins)

Special Constraint: Unique combination of (room_id, booking_date, start_time, status) prevents double-booking

8. ANNOUNCEMENT_ENTITY (Announcement Table)

Attributes:

- announcement_id (PK): Auto-generated unique identifier
- ERP (FK): References User_Table (BuildingIncharge)
- title: Announcement title (100 chars max)

- description: Announcement details (300 chars max)
- date_posted: Posting timestamp
- created_date: Auto-generated creation timestamp

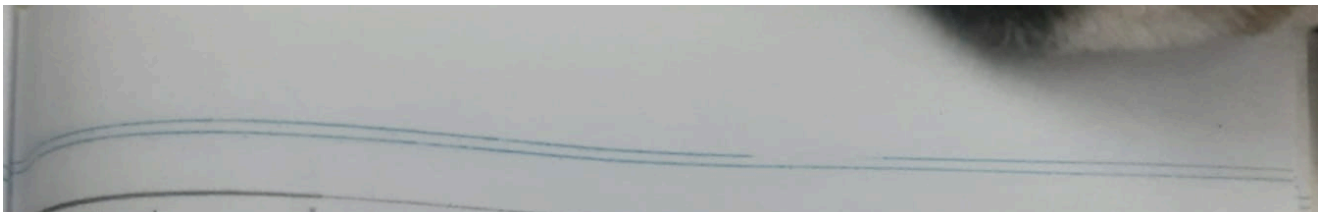
Multiplicity Constraints:

- Announcement to User: **M:1** (Many announcements posted by one user)
- Announcement to Building (indirect): **M:1** (Many announcements relate to one building via incharge)

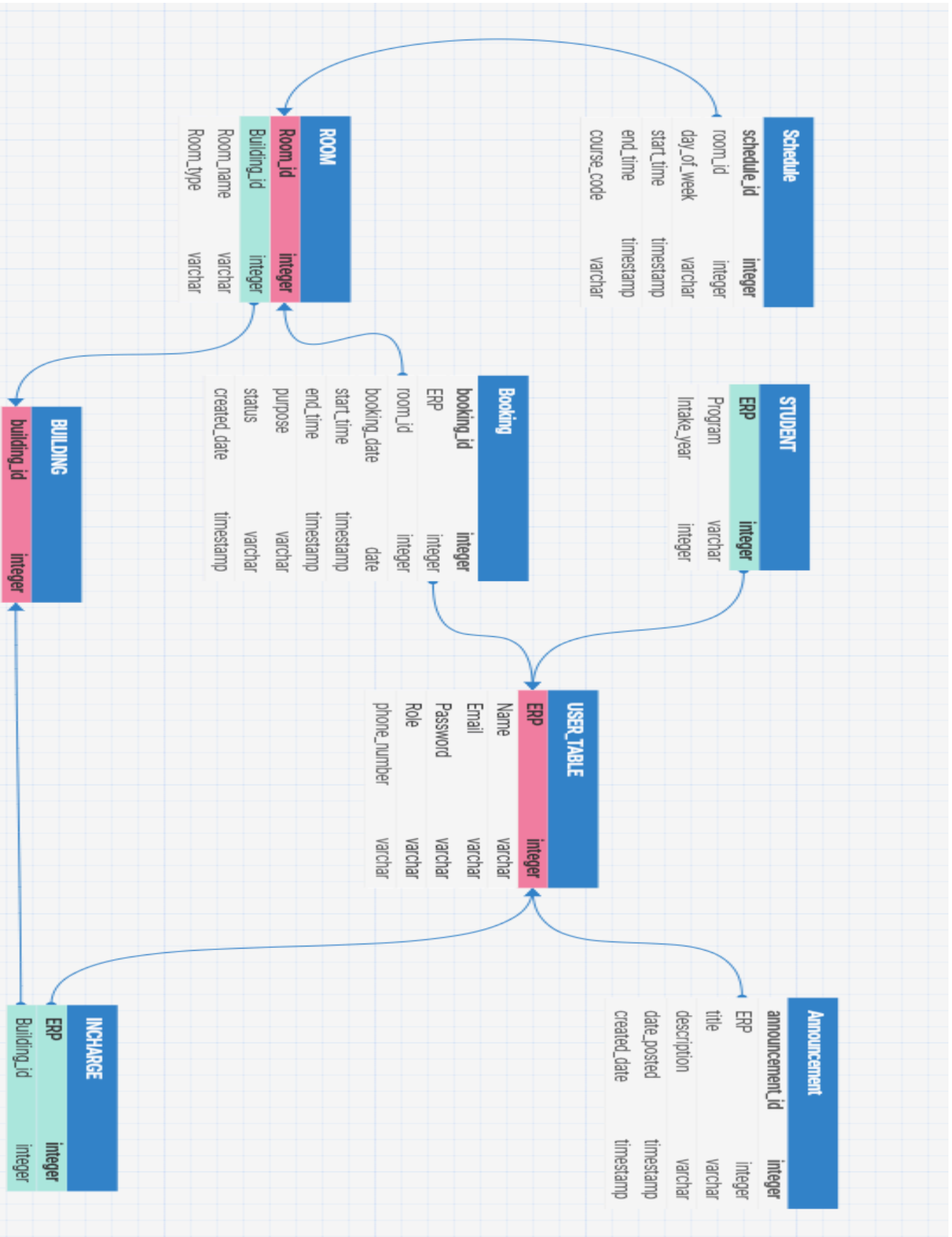
CARDINALITY CONSTRAINTS:

Relationship	Multiplicity	Business Rule
User → Student	1:1 (Partial)	A user may optionally be a student
User → Incharge	1:1 (Partial)	A user may optionally be an incharge
Incharge → Building	1:1 (Total)	Every building must have exactly one incharge
Building → Room	1:M	A building can have many rooms, a room belongs to one building
Room → Schedule	1:M	A room can have many scheduled classes
Room → Booking	1:M	A room can have many bookings
User → Booking	1:M	A user can make many bookings
User → Announcement	1:M	A user can post many announcements

ER Diagram – Crow's Foot



Database Schema



Normalized (3NF)

1NF — Unnormalized → First Normal Form

BookingSystem_1NF(
booking_id, ERP, user_name, user_email, phone_number, role,
program, intake_year,

announcement_id, announcement_title, announcement_description, announcement_date,
incharge_id, building_id, building_name,

room_id, room_name, room_type,

schedule_id, day_of_week, schedule_start_time, schedule_end_time, course_code,

booking_date, start_time, end_time, booking_purpose, booking_status,
booking_created_date
)

2NF — Remove Partial Dependencies

1. User_Table
    i. User_Table(ERP, name, email, phone_number, password, role)
2. Student Table
    i. Student(ERP, program, intake_year)
3. Building Table
    i. Building(building_id, building_name)
4. Room Table
    i. Room(room_id, building_id, room_name, room_type)
5. Schedule Table
    i. Schedule(schedule_id, room_id, day_of_week, start_time, end_time, course_code)
6. Announcements (PK: announcement_id, incharge_id)

i. Announcement(announcement_id,ncharge_id,building_id,title,description,date_posted,PRIMARY KEY(announcement_id, incharge_id))

7. Booking Table (PK: booking_id)
    a. Booking (booking_id, ERP, room_id, booking_date, start_time, end_time, purpose, status, created_date)

Functional Dependencies:

- announcement_id → title, description, date_posted
- incharge_id → building_id (due to our business rule: one building = one incharge)

3NF - **Remove transitive dependencies**

No transitive dependencies found.

1. User_Table (PK: ERP)

User_Table (ERP, name, email, phone_number, user_password, role)

2. Student (PK: ERP)

Student (ERP, program, intake_year)

3. Building (PK: building_id)

Building (building_id, building_name)

4. Incharge (PK: incharge_id)

Incharge (incharge_id, building_id)

5. Room (PK: room_id)

Room (room_id, building_id, room_name, room_type)

6. Schedule (PK: schedule_id)

Schedule (schedule_id, room_id, day_of_week, start_time, end_time, course_code)

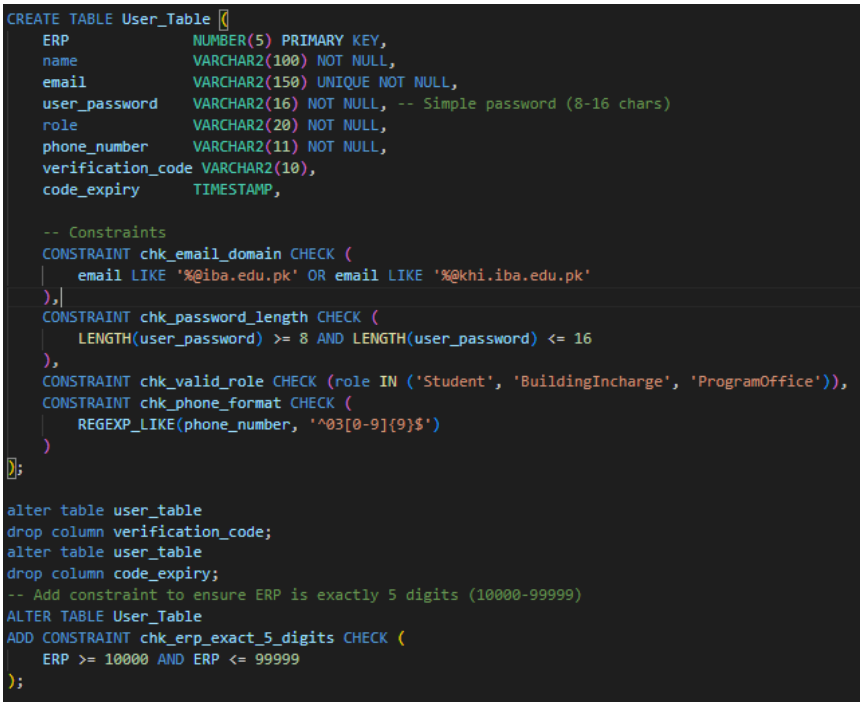
7. Announcement (PK: announcement_id)

Announcement (announcement_id, incharge_id, title, description, date_posted)

8. Booking (PK: booking_id)

Booking (booking_id, ERP, room_id, booking_date, start_time, end_time, purpose, status, created_date)

DDL Script Screenshots + Explanation:



```
CREATE TABLE User_Table (
    ERP             NUMBER(5) PRIMARY KEY,
    name            VARCHAR2(100) NOT NULL,
    email           VARCHAR2(150) UNIQUE NOT NULL,
    user_password   VARCHAR2(16) NOT NULL, -- Simple password (8-16 chars)
    role            VARCHAR2(20) NOT NULL,
    phone_number    VARCHAR2(11) NOT NULL,
    verification_code VARCHAR2(10),
    code_expiry     TIMESTAMP,

    -- Constraints
    CONSTRAINT chk_email_domain CHECK (
        email LIKE '%@iba.edu.pk' OR email LIKE '%@khi.iba.edu.pk'
    ),
    CONSTRAINT chk_password_length CHECK (
        LENGTH(user_password) >= 8 AND LENGTH(user_password) <= 16
    ),
    CONSTRAINT chk_valid_role CHECK (role IN ('Student', 'BuildingIncharge', 'ProgramOffice')),
    CONSTRAINT chk_phone_format CHECK (
        REGEXP_LIKE(phone_number, '^03[0-9]{9}$')
    )
);

alter table user_table
drop column verification_code;
alter table user_table
drop column code_expiry;
-- Add constraint to ensure ERP is exactly 5 digits (10000-99999)
ALTER TABLE User_Table
ADD CONSTRAINT chk_erp_exact_5_digits CHECK (
    ERP >= 10000 AND ERP <= 99999
);
```

User_Table Constraints Documentation

1. ERP (Employee/Student ID)

- **Primary Key Constraint**: Must be unique and not null
- **Exact 5-Digit Constraint**: Must be exactly 5 numerical digits (10000-99999)
- **Purpose**: Uniquely identifies each user in the system with a standardized ID format

2. Name

- **Not Null Constraint**: Cannot be empty
- **Purpose**: Displayed on frontend for user identification and personalization

3. Email

- **Not Null Constraint**: Cannot be empty
- **Unique Constraint**: Must be unique across all users
- **Domain Constraint**: Must end with @iba.edu.pk or @khi.iba.edu.pk
- **Purpose**:
    - Ensures only IBA-affiliated individuals can register
    - Serves as primary login identifier
    - Enforces institutional email standards

4. User_Password

- **Not Null Constraint**: Cannot be empty
- **Length Constraint**: Must be between 8-16 characters inclusive
- **Purpose**:
    - Security requirement for user authentication
    - Balanced between security (minimum 8 chars) and usability (maximum 16 chars)
    - Supports both email and phone-based login

5. Role

- **Not Null Constraint**: Cannot be empty
- **Value Constraint**: Must be one of: Student, BuildingIncharge, ProgramOffice
- **Purpose**:
    - Determines access permissions and system privileges
    - Enforces role-based access control (RBAC)
    - Ensures proper segregation of duties

6. Phone_Number

- **Not Null Constraint**: Cannot be empty
- **Length Constraint**: Exactly 11 characters
- **Format Constraint**: Must match pattern 03XXXXXXXXX where X is digit 0-9
- **Purpose**:
    - Secondary authentication method (phone-based login)
    - Ensures valid Pakistani mobile number format

```sql
-- Students
INSERT INTO User_Table (ERP, name, email, user_password, role, phone_number)
VALUES (10005, 'Abdullah Malik', 'abdullah@khi.iba.edu.pk', 'abdullah1234', 'Student', '03123456705');

-- Program Office
INSERT INTO User_Table (ERP, name, email, user_password, role, phone_number)
VALUES (20001, 'Ahsan Raza', 'ahsan@iba.edu.pk', 'Program1234', 'ProgramOffice', '03123456706');

-- Building Incharges
INSERT INTO User_Table (ERP, name, email, user_password, role, phone_number)
VALUES (30001, 'Mohsin Khan', 'mohsin@iba.edu.pk', 'Building1234', 'BuildingIncharge', '03123456707');
```

```sql
CREATE TABLE Student (
    ERP          NUMBER(5) PRIMARY KEY,
    program      VARCHAR2(10) NOT NULL,
    intake_year  NUMBER(4) NOT NULL,

    -- Foreign Key with CASCADE
    FOREIGN KEY (ERP) REFERENCES User_Table(ERP) ON DELETE CASCADE,

    -- Constraints
    CONSTRAINT chk_valid_program CHECK (program IN (
        'BBA', 'BSACF', 'BSECO', 'BSBA', 'BSSS', 'BSCS', 'BSEM', 'BSMT'
    ))
);
```

Student Table Constraints Explained

1. ERP (Student ID)

Primary Key + Foreign Key with CASCADE

- Each student must have a unique 5-digit ERP number
- Every ERP in the Student table must exist in the User_Table
- If a user is deleted from User_Table, their student record is automatically deleted

2. Program

Valid Program Check

- Only accepts approved IBA undergraduate programs
- Valid programs: BBA, BSACF, BSECO, BSBA, BSSS, BSCS, BSEM, BSMT
- Prevents invalid program names from being entered

3. Intake Year

NOT NULL + 4-Digit Format

- Every student must have an intake year
- Year must be exactly 4 digits (e.g., 2024)
- Ensures proper tracking of student enrollment year

These constraints ensure data quality and system reliability. They prevent invalid entries, maintain consistency between student and user accounts, enable automatic cleanup when users are deleted, and support accurate reporting through standardized data formats.

```sql
INSERT INTO Student (ERP, program, intake_year) VALUES (10005, 'BSECO', 2025);
```

```sql
CREATE TABLE Building (
    building_id   NUMBER GENERATED BY DEFAULT AS IDENTITY (START WITH 1 INCREMENT BY 1) PRIMARY KEY,
    building_name VARCHAR2(50) NOT NULL UNIQUE
);
```

Building Table Constraints Explained

1. Building_ID

Primary Key + Auto-generated

- Automatically creates a unique number for each new building
- Starts at 1 and increases by 1 for every new building added
- Ensures each building has a permanent, unique identifier

2. Building_Name

NOT NULL + UNIQUE

- Every building must have a name
- No two buildings can have the same name
- Based on real IBA campus structure where each building has a unique name

Why These Constraints Matter

These constraints ensure every building is uniquely identifiable through both an auto-generated ID and a unique name, preventing duplicate entries and maintaining accurate building records that reflect the actual IBA campus structure.

```sql
-- ========================================
-- 3. INSERT BUILDINGS
-- ========================================
INSERT INTO Building (building_name) VALUES ('Adamjee');
INSERT INTO Building (building_name) VALUES ('Aman CED');
INSERT INTO Building (building_name) VALUES ('Tabba');
INSERT INTO Building (building_name) VALUES ('Executive Center');
INSERT INTO Building (building_name) VALUES ('Sports Complex');
```

```sql
CREATE TABLE Incharge (
    incharge_id NUMBER(5) PRIMARY KEY,
    building_id NUMBER NOT NULL UNIQUE,

    -- Foreign Keys with CASCADE
    FOREIGN KEY (incharge_id) REFERENCES User_Table(ERP) ON DELETE CASCADE,
    FOREIGN KEY (building_id) REFERENCES Building(building_id) ON DELETE CASCADE
);
```

Incharge Table Constraints Explained

1. Incharge_ID

Primary Key + Foreign Key with CASCADE

- Must be a valid ERP from the User_Table
- Only users with 'BuildingIncharge' role can be assigned
- If the user is deleted, their incharge assignment is automatically removed

2. Building_ID

NOT NULL + UNIQUE + Foreign Key with CASCADE

- Every incharge must be assigned to a building
- Each building can have only one incharge
- Must reference an existing building in the Building table
- If a building is deleted, the incharge assignment is automatically removed

Why These Constraints Matter

These constraints ensure proper building management by enforcing a strict one-incharge-per-building policy while maintaining data integrity through automatic cleanup when related users or buildings are deleted.

```
DECLARE
    v_adamjee_id NUMBER;
    v_aman_id NUMBER;
    v_tabba_id NUMBER;
    v_exec_id NUMBER;
    v_sports_id NUMBER;
BEGIN
    -- Get building IDs again (or you could store them in a package variable)
    SELECT building_id INTO v_adamjee_id FROM Building WHERE building_name = 'Adamjee';
    SELECT building_id INTO v_aman_id FROM Building WHERE building_name = 'Aman CED';
    SELECT building_id INTO v_tabba_id FROM Building WHERE building_name = 'Tabba';
    SELECT building_id INTO v_exec_id FROM Building WHERE building_name = 'Executive Center';
    SELECT building_id INTO v_sports_id FROM Building WHERE building_name = 'Sports Complex';
    -- Zaid (30003) for Adamjee
    INSERT INTO Incharge (incharge_id, building_id) VALUES (30003, v_adamjee_id);
    -- Taimoor (30002) for Aman CED
    INSERT INTO Incharge (incharge_id, building_id) VALUES (30002, v_aman_id);
    -- Mohsin (30001) for Tabba
    INSERT INTO Incharge (incharge_id, building_id) VALUES (30001, v_tabba_id);
    -- Maheen (30005) for Executive Center
    INSERT INTO Incharge (incharge_id, building_id) VALUES (30005, v_exec_id);
    -- Samreen (30004) for Sports Complex
    INSERT INTO Incharge (incharge_id, building_id) VALUES (30004, v_sports_id);
    COMMIT;
    DBMS_OUTPUT.PUT_LINE('5 building incharges assigned.');
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('ERROR: Could not find buildings or incharges.');
        ROLLBACK;
        RAISE;
END;
/
```

```sql
CREATE TABLE Room (
    room_id      NUMBER GENERATED BY DEFAULT AS IDENTITY (START WITH 1 INCREMENT BY 1) PRIMARY KEY,
    building_id  NUMBER NOT NULL,
    room_name    VARCHAR2(15) NOT NULL,
    room_type    VARCHAR2(20) NOT NULL,

    -- Foreign Key
    FOREIGN KEY (building_id) REFERENCES Building(building_id) ON DELETE CASCADE,

    -- Constraints
    CONSTRAINT chk_room_type CHECK (room_type IN ('CLASSROOM', 'BREAKOUT')),
    CONSTRAINT uk_room_building UNIQUE (building_id, room_name)
);
```

Room Table Constraints Explained

1. Room_ID

Primary Key + Auto-generated

- Automatically creates a unique number for each new room
- Starts at 1 and increases by 1 for every new room added
- Ensures permanent, unique room identification

2. Building_ID

NOT NULL + Foreign Key with CASCADE

- Every room must belong to an existing building
- If a building is deleted, all its rooms are automatically removed

3. Room_Name

NOT NULL

- Every room must have a name (e.g., "AUDITORIUM", "BREAKOUT-1")

4. Room_Type

NOT NULL + CHECK Constraint

- Must be either 'CLASSROOM' or 'BREAKOUT'
- Restricts room categorization to valid academic space types

5. Unique Room-Building Combination

UNIQUE Constraint

- No two rooms in the same building can have the same name
- Ensures room names are unique within each building

Why These Matter:

Ensures proper room categorization, prevents duplicate room names in same building, and maintains data integrity when buildings are deleted.

```
-- Aman CED Building
INSERT INTO Room (building_id, room_name, room_type) VALUES (v_aman_id, 'MCC-9', 'CLASSROOM');
INSERT INTO Room (building_id, room_name, room_type) VALUES (v_aman_id, 'MCC-10', 'CLASSROOM');
INSERT INTO Room (building_id, room_name, room_type) VALUES (v_aman_id, 'MCC-11', 'CLASSROOM');
INSERT INTO Room (building_id, room_name, room_type) VALUES (v_aman_id, 'MCC-12', 'CLASSROOM');
INSERT INTO Room (building_id, room_name, room_type) VALUES (v_aman_id, 'MCC-13', 'CLASSROOM');
INSERT INTO Room (building_id, room_name, room_type) VALUES (v_aman_id, 'MCC-14', 'CLASSROOM');
INSERT INTO Room (building_id, room_name, room_type) VALUES (v_aman_id, 'MC-BREAKOUT-1', 'BREAKOUT');
INSERT INTO Room (building_id, room_name, room_type) VALUES (v_aman_id, 'MC-BREAKOUT-2', 'BREAKOUT');
```

```
CREATE TABLE Schedule (
    schedule_id   NUMBER GENERATED BY DEFAULT AS IDENTITY (START WITH 1 INCREMENT BY 1) PRIMARY KEY,
    room_id       NUMBER NOT NULL,
    day_of_week   VARCHAR2(10) NOT NULL,
    start_time    TIMESTAMP NOT NULL,
    end_time      TIMESTAMP NOT NULL,
    course_code   VARCHAR2(20) NOT NULL,

    -- Foreign Keys
    FOREIGN KEY (room_id) REFERENCES Room(room_id) ON DELETE CASCADE,

    -- Constraints
    CONSTRAINT chk_valid_day CHECK (UPPER(day_of_week) IN (
        'MONDAY', 'TUESDAY', 'WEDNESDAY', 'THURSDAY', 'FRIDAY', 'SATURDAY'
    )),
    CONSTRAINT chk_time_order CHECK (end_time > start_time)
);
```

Schedule Table Constraints Explained

1. Schedule_ID

Primary Key + Auto-generated

- Automatically creates a unique number for each class schedule entry

2. Room_ID

NOT NULL + Foreign Key with CASCADE

- Every class must be assigned to an existing room
- If a room is deleted, all its scheduled classes are automatically removed

3. Day_of_Week

NOT NULL + CHECK Constraint

- Must be a valid weekday: Monday, Tuesday, Wednesday, Thursday, Friday, or Saturday
- Excludes Sundays (non-academic day)

4. Start_Time & End_Time

NOT NULL + Time Order Check

- Class must have both start and end times
- End time must be later than start time (no zero or negative duration classes)

Why These Matter:

Prevents scheduling conflicts, ensures valid class timings, and maintains clean data by removing orphaned schedules automatically.

```
-- Monday classes
INSERT INTO Schedule (room_id, day_of_week, start_time, end_time, course_code)
VALUES (v_auditorium_id, 'MONDAY', TO_TIMESTAMP('1970-01-01 08:30:00', 'YYYY-MM-DD HH24:MI:SS'),
        TO_TIMESTAMP('1970-01-01 09:45:00', 'YYYY-MM-DD HH24:MI:SS'), 'CS401');
```

```sql
CREATE TABLE Booking (
    booking_id      NUMBER GENERATED BY DEFAULT AS IDENTITY (START WITH 1 INCREMENT BY 1) PRIMARY KEY,
    ERP             NUMBER(5) NOT NULL,
    room_id         NUMBER NOT NULL,
    booking_date    DATE NOT NULL,
    start_time      TIMESTAMP NOT NULL,
    end_time        TIMESTAMP NOT NULL,
    purpose         VARCHAR2(500) NOT NULL,
    status          VARCHAR2(20) DEFAULT 'Approved',
    created_date    TIMESTAMP DEFAULT SYSTIMESTAMP,

    -- Foreign Keys
    FOREIGN KEY (ERP) REFERENCES User_Table(ERP) ON DELETE CASCADE,
    FOREIGN KEY (room_id) REFERENCES Room(room_id) ON DELETE CASCADE,

    -- Constraints
    CONSTRAINT chk_booking_status CHECK (status IN ('Approved', 'Rejected', 'Cancelled')),
    CONSTRAINT chk_booking_time_order CHECK (end_time > start_time),
    CONSTRAINT chk_booking_duration CHECK (
        (end_time - start_time) <= INTERVAL '1' HOUR + INTERVAL '15' MINUTE
    ),
    CONSTRAINT uk_booking_time UNIQUE (room_id, booking_date, start_time, status)
);
```

Booking Table Constraints Explained

1. Booking_ID

Primary Key + Auto-generated

- Automatically creates a unique number for each booking

2. ERP

NOT NULL + Foreign Key with CASCADE

- Every booking must be made by an existing user
- If a user is deleted, all their bookings are automatically removed

3. Room_ID

NOT NULL + Foreign Key with CASCADE

- Every booking must be for an existing room
- If a room is deleted, all its bookings are automatically removed

4. Booking_Date

NOT NULL

- Every booking must have a date

5. Time Constraints

- **Time Order:** End time must be later than start time
- **Duration Limit:** Maximum booking length is 1 hour 15 minutes

6. Status

CHECK Constraint + DEFAULT

- Must be 'Approved', 'Rejected', or 'Cancelled'
- Defaults to 'Approved' when a booking is created

7. Unique Time Slot

UNIQUE Constraint

- Prevents double-booking the same room at the same time with the same status
- Allows different users to book the same room/time with different statuses (e.g., one Approved, one Rejected)

Why These Matter:

Prevents double bookings, enforces reasonable booking durations, maintains user accountability, and ensures automated data cleanup.

```
-- Muskan's booking (next Monday 10:00 AM)
INSERT INTO Booking (ERP, room_id, booking_date, start_time, end_time, purpose, status)
VALUES (10001, v_breakout1_id, v_next_monday,
        TO_TIMESTAMP(TO_CHAR(v_next_monday, 'YYYY-MM-DD') || ' 10:00:00', 'YYYY-MM-DD HH24:MI:SS'),
        TO_TIMESTAMP(TO_CHAR(v_next_monday, 'YYYY-MM-DD') || ' 11:00:00', 'YYYY-MM-DD HH24:MI:SS'),
        'Group study session for CS401', 'Approved');

-- Kashish's booking (next Tuesday 2:00 PM)
INSERT INTO Booking (ERP, room_id, booking_date, start_time, end_time, purpose, status)
VALUES (10002, v_mc_breakout1_id, v_next_tuesday,
        TO_TIMESTAMP(TO_CHAR(v_next_tuesday, 'YYYY-MM-DD') || ' 14:00:00', 'YYYY-MM-DD HH24:MI:SS'),
        TO_TIMESTAMP(TO_CHAR(v_next_tuesday, 'YYYY-MM-DD') || ' 15:00:00', 'YYYY-MM-DD HH24:MI:SS'),
        'Project meeting with team', 'Approved');

-- Mustafa's booking (next Wednesday 9:00 AM)
INSERT INTO Booking (ERP, room_id, booking_date, start_time, end_time, purpose, status)
VALUES (10003, v_mt_breakout1_id, v_next_wednesday,
        TO_TIMESTAMP(TO_CHAR(v_next_wednesday, 'YYYY-MM-DD') || ' 09:00:00', 'YYYY-MM-DD HH24:MI:SS'),
        TO_TIMESTAMP(TO_CHAR(v_next_wednesday, 'YYYY-MM-DD') || ' 10:00:00', 'YYYY-MM-DD HH24:MI:SS'),
        'Presentation practice', 'Approved');
```

```sql
CREATE TABLE Announcement (
    announcement_id NUMBER GENERATED BY DEFAULT AS IDENTITY (START WITH 1 INCREMENT BY 1) PRIMARY KEY,
    ERP             NUMBER(5) NOT NULL,
    title           VARCHAR2(100) NOT NULL,
    description     VARCHAR2(300),
    date_posted     TIMESTAMP NOT NULL,
    created_date    TIMESTAMP DEFAULT SYSTIMESTAMP,

    -- Foreign Key
    FOREIGN KEY (ERP) REFERENCES User_Table(ERP) ON DELETE CASCADE
);
```

Announcement Table Constraints Explained

1. Announcement_ID

Primary Key + Auto-generated

- Automatically creates a unique number for each announcement

2. ERP

NOT NULL + Foreign Key with CASCADE

- Every announcement must be posted by an existing user
- If a user is deleted, all their announcements are automatically removed

3. Title

NOT NULL

- Every announcement must have a title

4. Date_Posted

NOT NULL

- Every announcement must have a posting timestamp

5. Created_Date

DEFAULT

- Automatically records when the announcement was created in the system

Why These Matter:

Maintains clear communication records, ensures announcements are traceable to valid users, and provides audit trails for all posted information.

```sql
INSERT INTO Announcement (ERP, title, description, date_posted)
VALUES (30003, 'Adamjee Building Maintenance',
        'Please note that Adamjee building will undergo maintenance this Saturday. All rooms will be unavailable from 8 AM to 5 PM.',
        SYSTIMESTAMP);
```

```sql
CREATE OR REPLACE PROCEDURE RegisterStudent(
    p_erp         IN User_Table.ERP%TYPE,
    p_name        IN User_Table.name%TYPE,
    p_email       IN User_Table.email%TYPE,
    p_password    IN User_Table.user_password%TYPE,
    p_phonenumber IN User_Table.phone_number%TYPE,
    p_program     IN Student.program%TYPE,
    p_intake_year IN Student.intake_year%TYPE,
    p_success     OUT NUMBER,
    p_message     OUT VARCHAR2
)
AS
    v_erp_exists   NUMBER;
    v_email_exists NUMBER;
    v_phone_exists NUMBER;
BEGIN
    p_success := 0;
    p_message := '';

    -- only IBA email allowed
    IF NOT (p_email LIKE '%@khi.iba.edu.pk') THEN
        p_message := 'Only IBA student emails (@khi.iba.edu.pk) allowed';
        RETURN;
    END IF;

    -- ERP must be unique
    SELECT COUNT(*) INTO v_erp_exists
    FROM User_Table
    WHERE erp = p_erp;

    IF v_erp_exists > 0 THEN
        p_message := 'ERP number already registered';
        RETURN;
    END IF;

    -- Email must be unique
    SELECT COUNT(*) INTO v_email_exists
    FROM User_Table
    WHERE email = p_email;

    IF v_email_exists > 0 THEN
        p_message := 'Email already registered';
        RETURN;
    END IF;
```

```sql
    -- Phone number must be unique
    SELECT COUNT(*) INTO v_phone_exists
    FROM User_Table
    WHERE phone_number = p_phonenumber;

    IF v_phone_exists > 0 THEN
        p_message := 'Phone number already registered';
        RETURN;
    END IF;

    -- DIRECT REGISTRATION (NO OTP)
    INSERT INTO User_Table (
        name, email, erp, phone_number, role, user_password
    ) VALUES (
        p_name, p_email, p_erp, p_phonenumber, 'Student', p_password
    );

    INSERT INTO Student (
        erp, program, intake_year
    ) VALUES (
        p_erp,
        p_program,
        p_intake_year
    );

    COMMIT;

    p_success := 1;
    p_message := 'Student registered successfully';

EXCEPTION
    WHEN OTHERS THEN
        ROLLBACK;
        p_success := 0;
        p_message := 'Registration failed: ' || SQLERRM;
END RegisterStudent;
/
```

Procedure RegisterStudent

Purpose: This procedure handles student registration for the system. It allows new students to create accounts without requiring OTP verification (currently simplified due to SMTP complexity).

Steps:

1. Takes ERP, name, email, password, phone number, program, and intake year as inputs. Returns success status and message as outputs.
2. Validates that the email domain ends with @khi.iba.edu.pk
3. If valid, checks if a user with the same ERP already exists
4. If not, checks if a user with the same email already exists
5. If not, checks if a user with the same phone number already exists
6. If all checks pass, inserts the data:
    - Into User_Table: name, email, ERP, phone number, role='Student', password
    - Into Student table: ERP, program, intake year
7. Commits the transaction
8. Sets success = 1 and message = 'Student registered successfully'

9. If any errors occur:
    - Success = 0
    - Message = 'Registration failed: ' + error details
    - Rolls back the transaction

```sql
CREATE OR REPLACE PROCEDURE StudentLogin(
    p_identifier  IN VARCHAR2,  -- Can be email OR phone
    p_password    IN VARCHAR2,
    p_success     OUT NUMBER,
    p_erp         OUT NUMBER,
    p_name        OUT VARCHAR2,
    p_program     OUT VARCHAR2,
    p_intake_year OUT NUMBER,
    p_message     OUT VARCHAR2
)
AS
    v_is_email BOOLEAN;
BEGIN
    p_success := 0;
    p_erp := NULL;
    p_name := NULL;
    p_program := NULL;
    p_intake_year := NULL;
    p_message := '';

    v_is_email := INSTR(p_identifier, '@') > 0;

    IF v_is_email THEN
        BEGIN
            SELECT u.erp, u.name, s.program, s.intake_year
            INTO p_erp, p_name, p_program, p_intake_year
            FROM User_Table u
            JOIN Student s ON u.ERP = s.ERP
            WHERE u.email = p_identifier
              AND u.user_password = p_password
              AND u.role = 'Student';

            p_success := 1;
            p_message := 'Login successful';

        EXCEPTION
            WHEN NO_DATA_FOUND THEN
                p_message := 'Invalid email or password';
            WHEN TOO_MANY_ROWS THEN
                p_message := 'System error: Multiple accounts found';
        END;
    ELSE
        BEGIN
            SELECT u.erp, u.name, s.program, s.intake_year
            INTO p_erp, p_name, p_program, p_intake_year
            FROM User_Table u
            JOIN Student s ON u.ERP = s.ERP
            WHERE u.phone_number = p_identifier
              AND u.user_password = p_password
              AND u.role = 'Student';

            p_success := 1;
            p_message := 'Login successful';

        EXCEPTION
            WHEN NO_DATA_FOUND THEN
                p_message := 'Invalid phone number or password';
            WHEN TOO_MANY_ROWS THEN
                p_message := 'System error: Multiple accounts found';
        END;
    END IF;

END StudentLogin;
/
```

Procedure StudentLogin

Purpose: This procedure authenticates students by allowing login with either email or phone number. It verifies credentials and returns student information upon successful login.

Steps:

1. Takes identifier (email or phone) and password as inputs. Returns success status, student details (ERP, name, program, intake year), and message as outputs.
2. Determines if the identifier is an email (contains '@') or phone number
3. If identifier is email:
    - Searches User_Table for matching email, password, and 'Student' role
    - Joins with Student table to get academic details
4. If identifier is phone:
    - Searches User_Table for matching phone number, password, and 'Student' role
    - Joins with Student table to get academic details
5. If credentials are valid:
    - Success = 1, message = 'Login successful'
    - Returns student details
6. If no match found:

- Returns appropriate error message (email or phone specific)

7. If multiple accounts found (system error):
    - Returns error message indicating system issue

```sql
CREATE OR REPLACE PROCEDURE AdminLogin(
    p_identifier IN VARCHAR2,  -- Can be email OR phone
    p_password   IN User_Table.user_password%TYPE,
    p_success    OUT NUMBER,
    p_role       OUT VARCHAR2,
    p_erp        OUT NUMBER,
    p_name       OUT VARCHAR2,
    p_message    OUT VARCHAR2
)
AS
    v_is_email BOOLEAN;
BEGIN

    p_success := 0;
    p_role := NULL;
    p_erp := NULL;
    p_name := NULL;
    p_message := '';

    -- Determine identifier type
    v_is_email := INSTR(p_identifier, '@') > 0;

    IF v_is_email THEN
        -- Login with email
        BEGIN
            SELECT erp, name, role
            INTO p_erp, p_name, p_role
            FROM User_Table
            WHERE email = p_identifier
              AND user_password = p_password
              AND role IN ('ProgramOffice', 'BuildingIncharge');

            p_success := 1;
            p_message := 'Admin login successful';

        EXCEPTION
            WHEN NO_DATA_FOUND THEN
                p_message := 'Invalid admin credentials';
        END;
    ELSE
        -- Login with phone
        BEGIN
            SELECT erp, name, role
            INTO p_erp, p_name, p_role
            FROM User_Table
            WHERE phone_number = p_identifier
              AND user_password = p_password
              AND role IN ('ProgramOffice', 'BuildingIncharge');

            p_success := 1;
            p_message := 'Admin login successful';

        EXCEPTION
            WHEN NO_DATA_FOUND THEN
                p_message := 'Invalid admin credentials';
        END;
    END IF;

END AdminLogin;
/
```

Procedure AdminLogin

Purpose: This procedure authenticates administrative users (Program Office and Building Incharge) by allowing login with either email or phone number. It verifies credentials and returns admin information.

Steps:

1. Takes identifier (email or phone) and password as inputs. Returns success status, admin details (ERP, name, role), and message as outputs.
2. Determines if the identifier is an email (contains '@') or phone number
3. If identifier is email:
    - Searches User_Table for matching email, password, and admin role (ProgramOffice or BuildingIncharge)
4. If identifier is phone:
    - Searches User_Table for matching phone number, password, and admin role
5. If credentials are valid:
    - Success = 1, message = 'Admin login successful'

    - Returns admin details including specific role
6. If no match found:
    - Returns 'Invalid admin credentials' message
7. Only users with ProgramOffice or BuildingIncharge roles can login through this procedure

```
CREATE OR REPLACE PROCEDURE ShowReservationHistoryForStudent(
    p_erp    IN Booking.ERP%TYPE,
    p_result OUT SYS_REFCURSOR
)
AS
BEGIN
    OPEN p_result FOR
        SELECT
            b.booking_id,
            b.room_id,
            r.room_name,
            bld.building_name,
            b.booking_date,
            b.start_time,
            b.end_time,
            b.purpose,
            b.status,
            b.created_date
        FROM Booking b
        JOIN Room r ON b.room_id = r.room_id
        JOIN Building bld ON r.building_id = bld.building_id
        WHERE b.ERP = p_erp
        ORDER BY b.booking_date DESC, b.start_time DESC;
END;
/
```

Procedure ShowReservationHistoryForStudent

Purpose: This procedure retrieves the complete booking history for a specific student. It provides a detailed view of all past and current room reservations.

Steps:

1. Takes student ERP as input. Returns a cursor with booking details as output.
2. Queries the Booking table for all records belonging to the specified student
3. Joins with Room and Building tables to get room names and building names
4. Returns comprehensive booking information including:
    - Booking ID and room details
    - Building name and room name
    - Booking date and time slots
    - Purpose and current status
    - Creation timestamp
5. Orders results by booking date (newest first) and start time (latest first)

```sql
CREATE OR REPLACE PROCEDURE CancelBookingByStudent(
    p_booking_id IN Booking.booking_id%TYPE,
    p_erp        IN Booking.ERP%TYPE,
    p_success    OUT NUMBER,
    p_message    OUT VARCHAR2
)
AS
    v_booking_exists NUMBER;
    v_current_status Booking.status%TYPE;
    v_start_datetime TIMESTAMP;
BEGIN
    p_success := 0;
    p_message := '';
    -- First check if booking exists and belongs to student
    SELECT COUNT(*)
    INTO v_booking_exists
    FROM Booking
    WHERE booking_id = p_booking_id
      AND ERP = p_erp;

    IF v_booking_exists = 0 THEN
        p_message := 'Booking not found or does not belong to you';
        RETURN;
    END IF;

    -- Now get booking details separately
    BEGIN
        SELECT status, booking_date + (start_time - TRUNC(start_time))
        INTO v_current_status, v_start_datetime
        FROM Booking
        WHERE booking_id = p_booking_id
          AND ERP = p_erp;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            p_message := 'Booking details not found';
            RETURN;
    END;
```

```sql
    -- Check status is 'Approved'
    IF v_current_status != 'Approved' THEN
        p_message := 'Only Approved bookings can be cancelled';
        RETURN;
    END IF;

    -- Check booking hasn't started
    IF v_start_datetime <= SYSTIMESTAMP THEN
        p_message := 'Cannot cancel booking that has already started';
        RETURN;
    END IF;

    -- Cancel the booking
    UPDATE Booking
    SET status = 'Cancelled'
    WHERE booking_id = p_booking_id
      AND ERP = p_erp;

    COMMIT;
    p_success := 1;
    p_message := 'Booking cancelled successfully';

EXCEPTION
    WHEN OTHERS THEN
        ROLLBACK;
        p_success := 0;
        p_message := 'Error cancelling booking: ' || SQLERRM;
END CancelBookingByStudent;
/
```

Procedure CancelBookingByStudent

Purpose: This procedure allows students to cancel their own bookings while enforcing business rules about which bookings can be cancelled.

Steps:

1. Takes booking ID and student ERP as inputs. Returns success status and message as outputs.
2. Checks if the booking exists and belongs to the student
3. If not found, returns error: 'Booking not found or does not belong to you'
4. Retrieves the booking's current status and calculated start datetime
5. Validates cancellation rules:
    - Booking must have 'Approved' status
    - Booking must not have already started
6. If valid, updates the booking status to 'Cancelled'
7. Commits the transaction

8. Returns success = 1 and message = 'Booking cancelled successfully'
9. If any errors occur:
    - Rolls back the transaction
    - Returns success = 0 and error details message

```
CREATE OR REPLACE PROCEDURE ShowReservationHistoryForPO(
    p_result OUT SYS_REFCURSOR
)
AS
BEGIN
    OPEN p_result FOR
        SELECT
            b.booking_id,
            b.ERP,
            u.name AS student_name,
            r.room_name,
            bld.building_name,
            b.booking_date,
            b.start_time,
            b.end_time,
            b.purpose,
            b.status,
            b.created_date
        FROM Booking b
        JOIN Room r ON b.room_id = r.room_id
        JOIN Building bld ON r.building_id = bld.building_id
        JOIN User_Table u ON b.ERP = u.ERP
        WHERE r.room_type = 'CLASSROOM'
        ORDER BY b.booking_date DESC, b.start_time DESC;
END;
/
```

Procedure ShowReservationHistoryForPO

Purpose: This procedure provides the Program Office with a complete view of all classroom bookings across the entire campus for oversight and management purposes.

Steps:

1. Takes no input parameters. Returns a cursor with all classroom booking details as output.
2. Queries the Booking table for all records
3. Filters results to include only 'CLASSROOM' type rooms
4. Joins with multiple tables to get comprehensive information:
    - Room and Building tables for room and building names
    - User_Table for student names
5. Returns detailed booking information including:
    - Booking ID and student details (ERP and name)
    - Room and building information
    - Booking date, times, and purpose

- Current status and creation timestamp

6. Orders results by booking date (newest first) and start time (latest first)

```
CREATE OR REPLACE PROCEDURE ShowReservationHistoryForBI(
    p_incharge_id IN Incharge.incharge_id%TYPE,
    p_result      OUT SYS_REFCURSOR
)
AS
    v_building_id Building.building_id%TYPE;
BEGIN
    -- Get building assigned to this incharge
    SELECT building_id INTO v_building_id
    FROM Incharge
    WHERE incharge_id = p_incharge_id;

    OPEN p_result FOR
        SELECT
            b.booking_id,
            b.ERP,
            u.name AS student_name,
            r.room_name,
            bld.building_name,
            b.booking_date,
            b.start_time,
            b.end_time,
            b.purpose,
            b.status,
            b.created_date
        FROM Booking b
        JOIN Room r ON b.room_id = r.room_id
        JOIN Building bld ON r.building_id = bld.building_id
        JOIN User_Table u ON b.ERP = u.ERP
        WHERE r.room_type = 'BREAKOUT'
          AND r.building_id = v_building_id
        ORDER BY b.booking_date DESC, b.start_time DESC;

EXCEPTION
    WHEN NO_DATA_FOUND THEN
        p_result := NULL;
END;
/
```

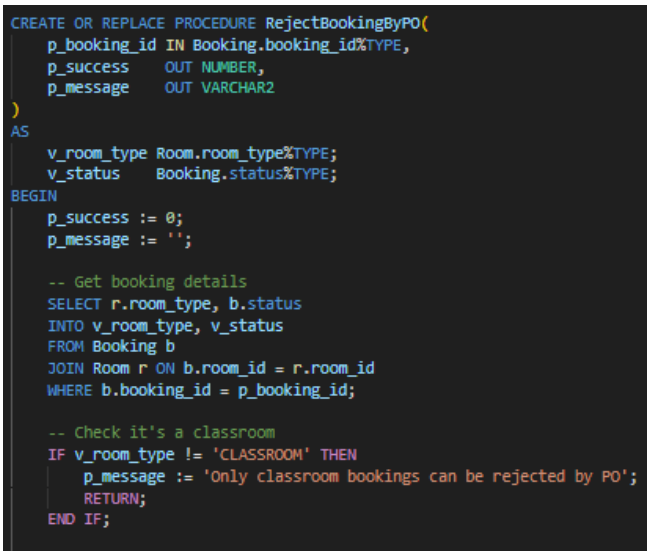
Procedure ShowReservationHistoryForBI

Purpose: This procedure allows Building Incharges to view breakout room bookings only for their assigned building, providing building-specific oversight.

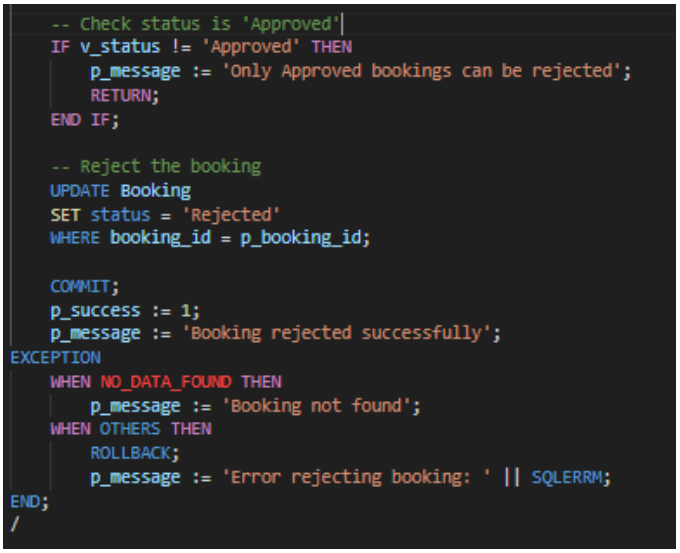
Steps:

1. Takes Building Incharge's ERP as input. Returns a cursor with breakout room booking details for their building as output.
2. Retrieves the building ID assigned to the specified incharge from the Incharge table
3. Queries the Booking table for all records

4. Filters results with two conditions:
    - Only 'BREAKOUT' type rooms
    - Only rooms in the incharge's assigned building
5. Joins with multiple tables to get comprehensive information:
    - Room and Building tables for room and building names
    - User_Table for student names
6. Returns detailed booking information including:
    - Booking ID and student details
    - Room and building information
    - Booking date, times, and purpose
    - Current status and creation timestamp
7. Orders results by booking date (newest first) and start time (latest first)
8. If incharge is not assigned to any building, returns null cursor



```
CREATE OR REPLACE PROCEDURE RejectBookingByPO(
    p_booking_id IN Booking.booking_id%TYPE,
    p_success    OUT NUMBER,
    p_message    OUT VARCHAR2
)
AS
    v_room_type Room.room_type%TYPE;
    v_status    Booking.status%TYPE;
BEGIN
    p_success := 0;
    p_message := '';

    -- Get booking details
    SELECT r.room_type, b.status
    INTO v_room_type, v_status
    FROM Booking b
    JOIN Room r ON b.room_id = r.room_id
    WHERE b.booking_id = p_booking_id;

    -- Check it's a classroom
    IF v_room_type != 'CLASSROOM' THEN
        p_message := 'Only classroom bookings can be rejected by PO';
        RETURN;
    END IF;
```



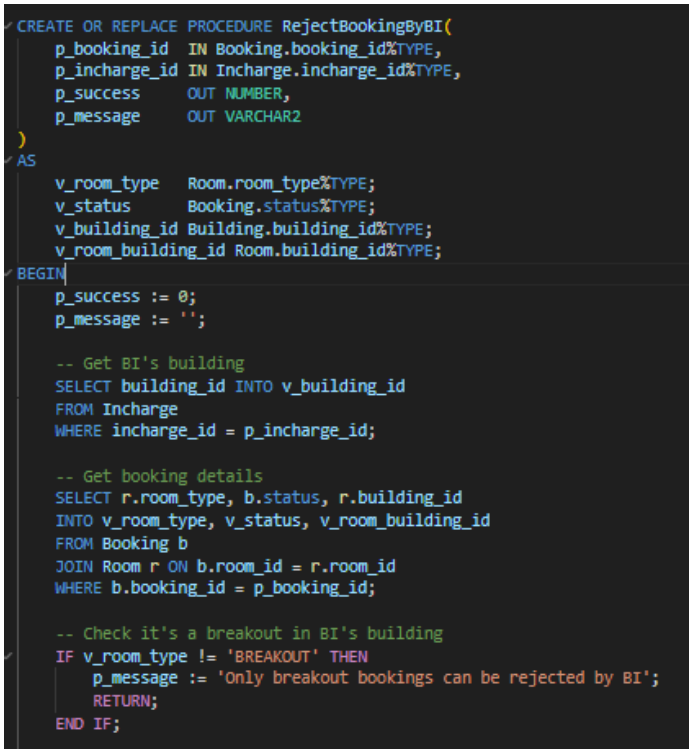
```
    -- Check status is 'Approved'
    IF v_status != 'Approved' THEN
        p_message := 'Only Approved bookings can be rejected';
        RETURN;
    END IF;

    -- Reject the booking
    UPDATE Booking
    SET status = 'Rejected'
    WHERE booking_id = p_booking_id;

    COMMIT;
    p_success := 1;
    p_message := 'Booking rejected successfully';
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        p_message := 'Booking not found';
    WHEN OTHERS THEN
        ROLLBACK;
        p_message := 'Error rejecting booking: ' || SQLERRM;
END;
/
```

Procedure RejectBookingByPO

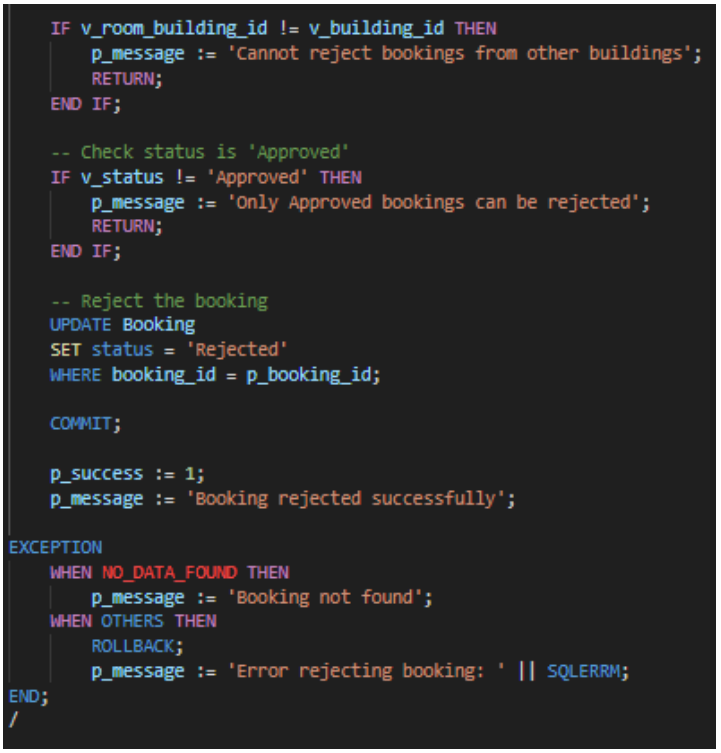
Purpose: This procedure allows the Program Office to reject classroom bookings, enforcing their oversight authority over academic spaces.

Steps:

1. Takes booking ID as input. Returns success status and message as outputs.
2. Retrieves the booking's room type and current status
3. Validates rejection rules:
    - Booking must be for a 'CLASSROOM' type room
    - Booking must have 'Approved' status
4. If valid, updates the booking status to 'Rejected'
5. Commits the transaction
6. Returns success = 1 and message = 'Booking rejected successfully'
7. If booking not found or other errors:
    - Returns appropriate error message
    - Rolls back transaction if database error occurs



```
CREATE OR REPLACE PROCEDURE RejectBookingByBI(
    p_booking_id  IN Booking.booking_id%TYPE,
    p_incharge_id IN Incharge.incharge_id%TYPE,
    p_success     OUT NUMBER,
    p_message     OUT VARCHAR2
)
AS
    v_room_type   Room.room_type%TYPE;
    v_status      Booking.status%TYPE;
    v_building_id Building.building_id%TYPE;
    v_room_building_id Room.building_id%TYPE;
BEGIN
    p_success := 0;
    p_message := '';

    -- Get BI's building
    SELECT building_id INTO v_building_id
    FROM Incharge
    WHERE incharge_id = p_incharge_id;

    -- Get booking details
    SELECT r.room_type, b.status, r.building_id
    INTO v_room_type, v_status, v_room_building_id
    FROM Booking b
    JOIN Room r ON b.room_id = r.room_id
    WHERE b.booking_id = p_booking_id;

    -- Check it's a breakout in BI's building
    IF v_room_type != 'BREAKOUT' THEN
        p_message := 'Only breakout bookings can be rejected by BI';
        RETURN;
    END IF;
```



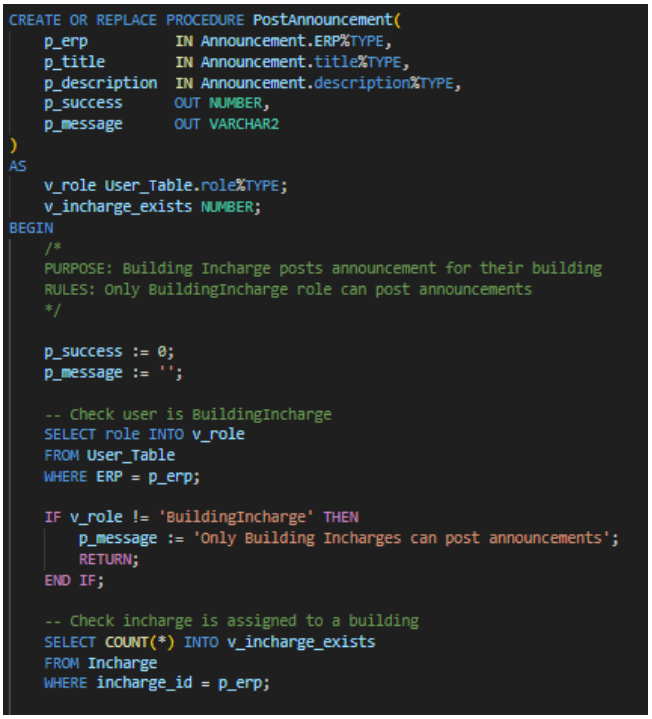
```
    IF v_room_building_id != v_building_id THEN
        p_message := 'Cannot reject bookings from other buildings';
        RETURN;
    END IF;

    -- Check status is 'Approved'
    IF v_status != 'Approved' THEN
        p_message := 'Only Approved bookings can be rejected';
        RETURN;
    END IF;

    -- Reject the booking
    UPDATE Booking
    SET status = 'Rejected'
    WHERE booking_id = p_booking_id;

    COMMIT;

    p_success := 1;
    p_message := 'Booking rejected successfully';

EXCEPTION
    WHEN NO_DATA_FOUND THEN
        p_message := 'Booking not found';
    WHEN OTHERS THEN
        ROLLBACK;
        p_message := 'Error rejecting booking: ' || SQLERRM;
END;
/
```

Procedure RejectBookingByBI

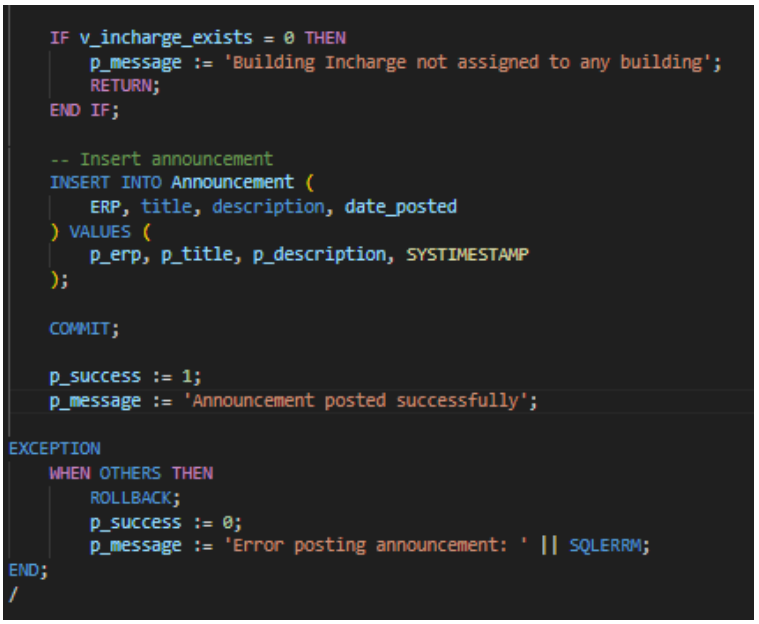
Purpose: This procedure allows Building Incharges to reject breakout room bookings, but only for their assigned building, enforcing building-level authority.

Steps:

1. Takes booking ID and incharge ERP as inputs. Returns success status and message as outputs.
2. Retrieves the building ID assigned to the specified incharge
3. Retrieves the booking's room type, current status, and building location
4. Validates rejection rules with three checks:
    - Booking must be for a 'BREAKOUT' type room
    - Room must be in the incharge's assigned building
    - Booking must have 'Approved' status
5. If all conditions met, updates the booking status to 'Rejected'
6. Commits the transaction
7. Returns success = 1 and message = 'Booking rejected successfully'
8. If booking not found, incharge not assigned, or other errors:
    - Returns appropriate error message
    - Rolls back transaction if database error occurs



```
CREATE OR REPLACE PROCEDURE PostAnnouncement(
    p_erp          IN Announcement.ERP%TYPE,
    p_title        IN Announcement.title%TYPE,
    p_description  IN Announcement.description%TYPE,
    p_success      OUT NUMBER,
    p_message      OUT VARCHAR2
)
AS
    v_role User_Table.role%TYPE;
    v_incharge_exists NUMBER;
BEGIN
    /*
    PURPOSE: Building Incharge posts announcement for their building
    RULES: Only BuildingIncharge role can post announcements
    */

    p_success := 0;
    p_message := '';

    -- Check user is BuildingIncharge
    SELECT role INTO v_role
    FROM User_Table
    WHERE ERP = p_erp;

    IF v_role != 'BuildingIncharge' THEN
        p_message := 'Only Building Incharges can post announcements';
        RETURN;
    END IF;

    -- Check incharge is assigned to a building
    SELECT COUNT(*) INTO v_incharge_exists
    FROM Incharge
    WHERE incharge_id = p_erp;
```



```
    IF v_incharge_exists = 0 THEN
        p_message := 'Building Incharge not assigned to any building';
        RETURN;
    END IF;

    -- Insert announcement
    INSERT INTO Announcement (
        ERP, title, description, date_posted
    ) VALUES (
        p_erp, p_title, p_description, SYSTIMESTAMP
    );

    COMMIT;

    p_success := 1;
    p_message := 'Announcement posted successfully';

EXCEPTION
    WHEN OTHERS THEN
        ROLLBACK;
        p_success := 0;
        p_message := 'Error posting announcement: ' || SQLERRM;
END;
/
```

Procedure PostAnnouncement

Purpose: This procedure allows Building Incharges to create and publish announcements for their assigned building, serving as a communication channel to students.

Steps:

1. Takes user ERP, title, and description as inputs. Returns success status and message as outputs.
2. Validates the posting user's authority with two checks:
    - User must have 'BuildingIncharge' role in User_Table
    - User must be assigned to a building in the Incharge table
3. If both conditions met, inserts new announcement with:
    - User's ERP, provided title and description
    - Automatic current timestamp for posting time
4. Commits the transaction
5. Returns success = 1 and message = 'Announcement posted successfully'
6. If user lacks authority or other errors:
    - Returns appropriate error message
    - Rolls back transaction if database error occurs

```sql
CREATE OR REPLACE PROCEDURE DeleteAnnouncement(
    p_announcement_id IN Announcement.announcement_id%TYPE,
    p_erp             IN Announcement.ERP%TYPE,
    p_success         OUT NUMBER,
    p_message         OUT VARCHAR2
)
AS
    v_announcement_exists NUMBER;
BEGIN
    p_success := 0;
    p_message := '';

    -- Check announcement exists and belongs to user
    SELECT COUNT(*) INTO v_announcement_exists
    FROM Announcement
    WHERE announcement_id = p_announcement_id
      AND ERP = p_erp;

    IF v_announcement_exists = 0 THEN
        p_message := 'Announcement not found or you do not have permission';
        RETURN;
    END IF;

    -- Delete announcement
    DELETE FROM Announcement
    WHERE announcement_id = p_announcement_id
      AND ERP = p_erp;

    COMMIT;

    p_success := 1;
    p_message := 'Announcement deleted successfully';

EXCEPTION
    WHEN OTHERS THEN
        ROLLBACK;
        p_success := 0;
        p_message := 'Error deleting announcement: ' || SQLERRM;
END;
/
```

Procedure DeleteAnnouncement

Purpose: This procedure allows users to delete announcements they previously posted, maintaining user control over their own communications.

Steps:

1. Takes announcement ID and user ERP as inputs. Returns success status and message as outputs.
2. Checks if the specified announcement exists and belongs to the requesting user
3. If announcement doesn't exist or belongs to another user:
    - Returns error: 'Announcement not found or you do not have permission'
4. If user owns the announcement:
    - Deletes the announcement from the Announcement table
5. Commits the transaction
6. Returns success = 1 and message = 'Announcement deleted successfully'
7. If any errors occur:
    - Rolls back the transaction

- Returns success = 0 with error details message

```
CREATE OR REPLACE PROCEDURE ShowAnnouncementsByUser(
    p_erp     IN Announcement.ERP%TYPE,
    p_result  OUT SYS_REFCURSOR
)
AS
BEGIN
    OPEN p_result FOR
        SELECT
            announcement_id,
            title,
            description,
            date_posted,
            created_date
        FROM Announcement
        WHERE ERP = p_erp
        ORDER BY date_posted DESC;
END;
/
```

Procedure ShowAnnouncementsByUser

Purpose: This procedure retrieves all announcements posted by a specific user, typically used by Building Incharges to view their own posting history.

Steps:

1. Takes user ERP as input. Returns a cursor with the user's announcements as output.
2. Queries the Announcement table for all records belonging to the specified user
3. Returns announcement details including:
    - Announcement ID, title, and description
    - Posting date and creation timestamp
4. Orders results by posting date (newest first)

```
CREATE OR REPLACE PROCEDURE ShowAllAnnouncements(
    p_result OUT SYS_REFCURSOR
)
AS
BEGIN
    OPEN p_result FOR
        SELECT
            a.announcement_id,
            a.title,
            a.description,
            a.date_posted,
            u.name AS posted_by,
            b.building_name
        FROM Announcement a
        JOIN User_Table u ON a.ERP = u.ERP
        LEFT JOIN Incharge i ON a.ERP = i.incharge_id
        LEFT JOIN Building b ON i.building_id = b.building_id
        ORDER BY a.date_posted DESC;
END;
/
```

Procedure ShowAllAnnouncements

Purpose: This procedure provides students with a complete view of all announcements from all buildings, serving as the main announcement feed.

Steps:

1. Takes no input parameters. Returns a cursor with all announcements as output.
2. Queries the Announcement table with comprehensive joins:
    - User_Table to get poster's name
    - Incharge table (left join) to link to buildings
    - Building table (left join) to get building names
3. Returns detailed announcement information including:
    - Announcement ID, title, and description
    - Posting date
    - Poster's name
    - Building name (if poster is a Building Incharge)
4. Orders results by posting date (newest first)

```
CREATE OR REPLACE PROCEDURE FilterAnnouncementsByBuilding(
    p_building_name IN Building.building_name%TYPE,
    p_result      OUT SYS_REFCURSOR
)
AS
    v_building_exists NUMBER;
BEGIN
    -- Check building exists
    SELECT COUNT(*) INTO v_building_exists
    FROM Building
    WHERE UPPER(building_name) = UPPER(p_building_name);

    IF v_building_exists = 0 THEN
        p_result := NULL;
        RETURN;
    END IF;

    OPEN p_result FOR
        SELECT
            a.announcement_id,
            a.title,
            a.description,
            a.date_posted,
            u.name AS posted_by,
            b.building_name
        FROM Announcement a
        JOIN User_Table u ON a.ERP = u.ERP
        JOIN Incharge i ON a.ERP = i.incharge_id
        JOIN Building b ON i.building_id = b.building_id
        WHERE UPPER(b.building_name) = UPPER(p_building_name)
        ORDER BY a.date_posted DESC;

END;
/
```

Procedure FilterAnnouncementsByBuilding

Purpose: This procedure allows filtering announcements by specific building, enabling targeted information retrieval for students interested in particular buildings.

Steps:

1. Takes building name as input. Returns a cursor with filtered announcements as output.
2. First checks if the specified building exists in the Building table
3. If building doesn't exist, returns a null cursor
4. If building exists, queries announcements with building-specific filtering:
    - Joins Announcement, User_Table, Incharge, and Building tables
    - Filters results to only include announcements from the specified building
5. Returns announcement details including:
    - Announcement ID, title, and description
    - Posting date
    - Poster's name

    - Building name
6. Orders results by posting date (newest first)

```sql
CREATE OR REPLACE PROCEDURE add_building(
    p_building_name IN Building.building_name%TYPE,
    p_incharge_erp IN User_Table.ERP%TYPE,
    p_incharge_name IN User_Table.name%TYPE,
    p_incharge_email IN User_Table.email%TYPE,
    p_phonenumber IN User_Table.phone_number%TYPE,
    p_result OUT VARCHAR2
)
AS
    v_building_count NUMBER;
    v_building_id Building.building_id%TYPE;
    v_user_count NUMBER;
    v_incharge_count NUMBER;
BEGIN
    -- Step 1: Check if building already exists by name
    SELECT COUNT(*) INTO v_building_count
    FROM Building
    WHERE UPPER(building_name) = UPPER(p_building_name);

    -- If building exists, return error message
    IF v_building_count > 0 THEN
        p_result := 'Building "' || p_building_name || '" already exists';
        RETURN;
    END IF;

    -- only IBA email allowed
    IF NOT (p_incharge_email LIKE '%@iba.edu.pk') THEN
        p_result := 'Only IBA incharge emails (@iba.edu.pk) allowed';
        RETURN;
    END IF;

    -- Step 2: Check if incharge ERP exists in User_Table
    SELECT COUNT(*) INTO v_user_count
    FROM User_Table
    WHERE ERP = p_incharge_erp;

    -- Step 3: Check if incharge is already assigned to another building
    SELECT COUNT(*) INTO v_incharge_count
    FROM Incharge
    WHERE incharge_id = p_incharge_erp;
```

```sql
    -- If incharge is already assigned to another building, return error
    IF v_incharge_count > 0 THEN
        p_result := 'Incharge with ERP ' || p_incharge_erp || ' is already assigned to another building';
        RETURN;
    END IF;

    -- Step 4: If incharge doesn't exist in User_Table, add them
    IF v_user_count = 0 THEN
        -- Use consistent role name from your existing code ('BuildingIncharge')
        INSERT INTO User_Table (ERP, name, email, user_password, role, phone_number)
        VALUES (p_incharge_erp, p_incharge_name, p_incharge_email, 'default_password', 'BuildingIncharge', p_phonenumber);
    END IF;

    -- Step 5: Insert the new building (ID will auto-generate)
    INSERT INTO Building (building_name)
    VALUES (p_building_name)
    RETURNING building_id INTO v_building_id;

    -- Step 6: Assign incharge to the building
    INSERT INTO Incharge (incharge_id, building_id)
    VALUES (p_incharge_erp, v_building_id);

    -- Step 7: Set success message with building ID
    p_result := 'Building "' || p_building_name || '" added successfully with ID: ' || v_building_id || '. Incharge assigned.';

    -- Commit the transaction
    COMMIT;

EXCEPTION
    WHEN OTHERS THEN
        -- Handle any exceptions and return error message
        p_result := 'Error adding building: ' || SQLERRM;
        ROLLBACK;
END add_building;
/
```

Procedure add_building

Purpose: This procedure creates a new building in the system and assigns a building incharge to manage it, handling both new and existing incharges.

Steps:

1. Takes building name, incharge details (ERP, name, email, phone) as inputs. Returns result message as output.
2. Checks if building already exists by name
    - If exists, returns error: 'Building "[name]" already exists'
3. Validates incharge email domain (@iba.edu.pk)
    - If invalid, returns error: 'Only IBA incharge emails allowed'
4. Checks if incharge ERP exists in User_Table
5. Checks if incharge is already assigned to another building
    - If assigned, returns error: 'Incharge already assigned to another building'
6. If incharge doesn't exist in User_Table, creates new user with:
    - Provided details, role='BuildingIncharge', default password
7. Inserts new building (auto-generates building_id)
8. Assigns incharge to the new building
9. Commits transaction

10. Returns success message with building ID
11. If any errors occur:
    - Rolls back transaction
    - Returns error message with details

```
CREATE OR REPLACE PROCEDURE add_room(
    p_building_name IN Building.building_name%TYPE,
    p_room_name IN Room.room_name%TYPE,
    p_room_type IN Room.room_type%TYPE,
    p_result OUT VARCHAR2
)
AS
    v_building_count NUMBER;
    v_room_count NUMBER;
    v_building_id Building.building_id%TYPE;
    v_room_id Room.room_id%TYPE;
BEGIN
    -- Step 1: Check if building exists
    BEGIN
        SELECT building_id INTO v_building_id
        FROM Building
        WHERE UPPER(building_name) = UPPER(p_building_name);

        v_building_count := 1;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            v_building_count := 0;
    END;

    -- If building doesn't exist, return error message
    IF v_building_count = 0 THEN
        p_result := 'Building "' || p_building_name || '" does not exist. Please add building first.';
        RETURN;
    END IF;
```

```
    -- Step 2: Check if room already exists in this building
    SELECT COUNT(*) INTO v_room_count
    FROM Room
    WHERE UPPER(room_name) = UPPER(p_room_name)
    AND building_id = v_building_id;

    -- If room exists, return error message
    IF v_room_count > 0 THEN
        p_result := 'Room "' || p_room_name || '" already exists in building "' || p_building_name || '"';
        RETURN;
    END IF;

    -- Step 3: Insert the new room (ID will auto-generate)
    INSERT INTO Room (building_id, room_name, room_type)
    VALUES (v_building_id, p_room_name, p_room_type)
    RETURNING room_id INTO v_room_id;

    -- Step 4: Set success message
    p_result := 'Room "' || p_room_name || '" added successfully to building "' || p_building_name || '" with ID: ' || v_room_id;

    -- Commit the transaction
    COMMIT;

EXCEPTION
    WHEN OTHERS THEN
        -- Handle any exceptions and return error message
        p_result := 'Error adding room: ' || SQLERRM;
        ROLLBACK;
END add_room;
/
```

Procedure add_room

Purpose: This procedure adds a new room to an existing building in the system, ensuring room names are unique within each building.

Steps:

1. Takes building name, room name, and room type as inputs. Returns result message as output.
2. Checks if the specified building exists
    - If building doesn't exist, returns error: 'Building "[name]" does not exist. Please add building first.'

3. Checks if a room with the same name already exists in that building
    - If room exists, returns error: 'Room "[name]" already exists in building "[name]"'
4. Inserts new room with:
    - Building ID (retrieved from building name)
    - Room name and type (CLASSROOM or BREAKOUT)
    - Auto-generated room_id
5. Commits transaction
6. Returns success message with room ID
7. If any errors occur:
    - Rolls back transaction
    - Returns error message with details

```
-- This procedure already exists as viewAllRoo
CREATE OR REPLACE PROCEDURE get_buildings(
    p_cursor OUT SYS_REFCURSOR
)
AS
BEGIN
    OPEN p_cursor FOR
    SELECT building_id, building_name
    FROM Building
    ORDER BY building_name;
END get_buildings;
/
```

Procedure get_buildings

Purpose: This procedure retrieves a list of all buildings in the system for display in dropdown menus or selection interfaces.

Steps:

1. Takes no input parameters. Returns a cursor with building information as output.
2. Queries the Building table for all records
3. Returns building ID and building name for each building
4. Orders results alphabetically by building name

```sql
CREATE OR REPLACE PROCEDURE get_available_rooms(
    p_date IN DATE,
    p_start_time IN VARCHAR2,
    p_end_time IN VARCHAR2,
    p_building_id IN NUMBER,
    p_room_type IN VARCHAR2,
    p_cursor OUT SYS_REFCURSOR
)
AS
    v_start_datetime TIMESTAMP;  -- Changed from DATE to TIMESTAMP to match your schema
    v_end_datetime TIMESTAMP;    -- Changed from DATE to TIMESTAMP to match your schema
    v_day_of_week VARCHAR2(10);  -- Changed to match your Schedule.day_of_week
BEGIN
    -- Convert time strings to datetime
    v_start_datetime := TO_TIMESTAMP(TO_CHAR(p_date, 'YYYY-MM-DD') || ' ' || p_start_time, 'YYYY-MM-DD HH24:MI');
    v_end_datetime := TO_TIMESTAMP(TO_CHAR(p_date, 'YYYY-MM-DD') || ' ' || p_end_time, 'YYYY-MM-DD HH24:MI');
    v_day_of_week := UPPER(TO_CHAR(p_date, 'DAY'));  -- Get day of week

    OPEN p_cursor FOR
    SELECT
        r.room_id,
        r.room_name,
        r.room_type,
        b.building_name,
        b.building_id
    FROM Room r
    JOIN Building b ON r.building_id = b.building_id
    WHERE b.building_id = p_building_id
    AND r.room_type = p_room_type
    AND r.room_id NOT IN (
        -- Rooms already booked for this time slot (only approved bookings block availability)
        SELECT room_id FROM Booking
        WHERE booking_date = p_date  -- Changed from date_of_booking to booking_date
        AND status = 'Approved'  -- Changed to match your schema (capitalized)
        AND NOT (
            end_time <= v_start_datetime OR
            start_time >= v_end_datetime
        )
    )
    AND r.room_id NOT IN (
        -- Rooms scheduled for classes
        SELECT room_id FROM Schedule
        WHERE day_of_week = v_day_of_week  -- Changed from day to day_of_week
        AND NOT (
            end_time <= v_start_datetime OR
            start_time >= v_end_datetime
        )
    )
    ORDER BY r.room_name;
END get_available_rooms;
/
```

Procedure get_available_rooms

Purpose: This procedure finds all rooms that are available for booking during a specified time period, considering both existing bookings and academic schedules.

Steps:

1. Takes date, start time, end time, building ID, and room type as inputs. Returns a cursor with available rooms as output.
2. Converts time strings to proper timestamp format for comparison
3. Determines the day of week from the provided date
4. Queries for rooms that meet all criteria:
    - Are in the specified building

- Match the requested room type (CLASSROOM or BREAKOUT)
- Are NOT already booked during the requested time (only checks 'Approved' bookings)
- Are NOT scheduled for classes during the requested time

5. Excludes rooms with time conflicts using overlap checking logic
6. Returns room details including room ID, name, type, building name, and building ID
7. Orders results by room name

```
CREATE OR REPLACE PROCEDURE create_booking(
    p_erp IN NUMBER,
    p_room_id IN NUMBER,
    p_date_of_booking IN DATE,
    p_start_time IN VARCHAR2,
    p_end_time IN VARCHAR2,
    p_purpose IN VARCHAR2,
    p_booking_id OUT NUMBER,
    p_success OUT NUMBER,
    p_message OUT VARCHAR2
)
AS
    v_start_datetime TIMESTAMP;  -- Changed to TIMESTAMP
    v_end_datetime TIMESTAMP;    -- Changed to TIMESTAMP
    v_conflict_count NUMBER;
    v_schedule_count NUMBER;
    v_current_date DATE := SYSDATE;
    v_day_of_week VARCHAR2(10);  -- Changed to match your schema
BEGIN
    p_success := 0;
    p_message := 'Booking failed';

    IF TRUNC(p_date_of_booking) < TRUNC(v_current_date) THEN
        p_message := 'Booking date cannot be in the past. Please select today or a future date.';
        RETURN;
    END IF;

    v_start_datetime := TO_TIMESTAMP(TO_CHAR(p_date_of_booking, 'YYYY-MM-DD') || ' ' || p_start_time, 'YYYY-MM-DD HH24:MI');
    v_end_datetime := TO_TIMESTAMP(TO_CHAR(p_date_of_booking, 'YYYY-MM-DD') || ' ' || p_end_time, 'YYYY-MM-DD HH24:MI');
    v_day_of_week := UPPER(TO_CHAR(p_date_of_booking, 'DAY'));  -- Get day of week

    -- If booking is for today, validate that start time is not in the past
    IF TRUNC(p_date_of_booking) = TRUNC(v_current_date) THEN
        IF v_start_datetime < SYSTIMESTAMP THEN  -- Changed to SYSTIMESTAMP for timestamp comparison
            p_message := 'Start time cannot be in the past for today''s booking.';
            RETURN;
        END IF;
    END IF;
```

```
    -- Validate end time is after start time
    IF v_end_datetime <= v_start_datetime THEN
        p_message := 'End time must be after start time.';
        RETURN;
    END IF;

    -- Check for booking conflicts (only check approved bookings)
    SELECT COUNT(*) INTO v_conflict_count
    FROM Booking
    WHERE room_id = p_room_id
    AND booking_date = p_date_of_booking  -- Changed from date_of_booking
    AND status = 'Approved'  -- Capitalized to match your schema
    AND NOT (
        end_time <= v_start_datetime OR
        start_time >= v_end_datetime
    );

    -- Check for schedule conflicts
    SELECT COUNT(*) INTO v_schedule_count
    FROM Schedule
    WHERE room_id = p_room_id
    AND day_of_week = v_day_of_week  -- Changed from day
    AND NOT (
        end_time <= v_start_datetime OR
        start_time >= v_end_datetime
    );

    IF v_conflict_count > 0 THEN
        p_message := 'Room already booked for this time slot';
        RETURN;
    END IF;

    IF v_schedule_count > 0 THEN
        p_message := 'Room is scheduled for classes during this time';
        RETURN;
    END IF;

    -- Insert the booking with 'Approved' status (capitalized to match your schema)
    INSERT INTO Booking (
        ERP, room_id, booking_date,  -- Changed from date_of_booking
        start_time, end_time, purpose, status
    ) VALUES (
        p_erp, p_room_id, p_date_of_booking,
        v_start_datetime, v_end_datetime, p_purpose, 'Approved'
    )
    RETURNING booking_id INTO p_booking_id;

    COMMIT;
    p_success := 1;
    p_message := 'Booking approved successfully!';

EXCEPTION
    WHEN OTHERS THEN
        ROLLBACK;
        p_message := 'Error creating booking: ' || SQLERRM;
        p_success := 0;
END create_booking;
/
```

Procedure create_booking

Purpose: This procedure creates a new room booking with comprehensive validation to prevent conflicts and ensure logical booking times.

Steps:

1. Takes user ERP, room ID, date, times, and purpose as inputs. Returns booking ID, success status, and message as outputs.
2. Validates booking date is not in the past
3. Converts time strings to proper timestamp format
4. Additional validation for today's bookings:
    - Start time cannot be in the past
5. Validates end time is after start time
6. Checks for existing booking conflicts:
    - Only considers 'Approved' bookings
    - Checks for time overlaps in the same room on same date
7. Checks for academic schedule conflicts:
    - Considers regular class schedules for that day of week
    - Checks for time overlaps with scheduled classes
8. If no conflicts found, inserts new booking with:
    - User ERP, room ID, date, times, purpose
    - Auto-set 'Approved' status
    - Auto-generated booking_id
9. Commits transaction
10. Returns success = 1, booking ID, and success message
11. If conflicts or other errors:
    - Returns specific error message about conflict type
    - Rolls back transaction if database error occurs

```
CREATE OR REPLACE PROCEDURE ViewAllRooms(
    p_result OUT SYS_REFCURSOR
)
AS
BEGIN
    OPEN p_result FOR
        SELECT
            r.room_id,
            r.room_name,
            r.room_type,
            b.building_id,
            b.building_name
        FROM Room r
        JOIN Building b ON r.building_id = b.building_id
        ORDER BY b.building_name, r.room_type, r.room_name;

END ViewAllRooms;
/
```

Procedure ViewAllRooms

Purpose: This procedure retrieves a comprehensive list of all rooms across all buildings for system-wide room management and selection.

Steps:

1. Takes no input parameters. Returns a cursor with all room details as output.
2. Queries the Room table joined with Building table
3. Returns complete room information including:
    - Room ID, name, and type
    - Building ID and building name
4. Orders results by building name, then room type, then room name

```sql
CREATE OR REPLACE PROCEDURE GetRoomsByBuilding(
    p_building_name IN Building.building_name%TYPE,
    p_result        OUT SYS_REFCURSOR
)
AS
    v_building_exists NUMBER;
BEGIN

    -- Check if building exists
    SELECT COUNT(*) INTO v_building_exists
    FROM Building
    WHERE UPPER(building_name) = UPPER(p_building_name);

    IF v_building_exists = 0 THEN
        -- Return empty cursor if building doesn't exist
        OPEN p_result FOR
            SELECT
                r.room_id,
                r.room_name,
                r.room_type,
                b.building_id,
                b.building_name
            FROM Room r
            JOIN Building b ON r.building_id = b.building_id
            WHERE 1 = 0; -- Always false
        RETURN;
    END IF;

    -- Get all rooms in the specified building
    OPEN p_result FOR
        SELECT
            r.room_id,
            r.room_name,
            r.room_type,
            b.building_id,
            b.building_name
        FROM Room r
        JOIN Building b ON r.building_id = b.building_id
        WHERE UPPER(b.building_name) = UPPER(p_building_name)
        ORDER BY r.room_type, r.room_name;

END GetRoomsByBuilding;
/
```

Procedure GetRoomsByBuilding

Purpose: This procedure retrieves all rooms within a specific building for targeted room management and selection.

Steps:

1. Takes building name as input. Returns a cursor with room details for that building as output.
2. First checks if the specified building exists

3. If building doesn't exist:
    - Returns an empty cursor (no results)
4. If building exists:
    - Queries rooms in that specific building
    - Returns room ID, name, type, and building details
    - Orders results by room type, then room name

```
CREATE OR REPLACE PROCEDURE SearchRoomsByName(
    p_search_term IN VARCHAR2,
    p_result      OUT SYS_REFCURSOR
)
AS
BEGIN
    IF p_search_term IS NULL OR LENGTH(TRIM(p_search_term)) = 0 THEN
        -- If search term is empty, return all rooms
        OPEN p_result FOR
            SELECT
                r.room_id,
                r.room_name,
                r.room_type,
                b.building_id,
                b.building_name
            FROM Room r
            JOIN Building b ON r.building_id = b.building_id
            ORDER BY b.building_name, r.room_name;
    ELSE
        -- Search for rooms with partial match in room_name
        OPEN p_result FOR
            SELECT
                r.room_id,
                r.room_name,
                r.room_type,
                b.building_id,
                b.building_name
            FROM Room r
            JOIN Building b ON r.building_id = b.building_id
            WHERE UPPER(r.room_name) LIKE '%' || UPPER(TRIM(p_search_term)) || '%'
            ORDER BY b.building_name, r.room_name;
    END IF;

END SearchRoomsByName;
/
```

Procedure SearchRoomsByName

Purpose: This procedure searches for rooms by partial name match, enabling flexible room discovery across the system.

Steps:

1. Takes search term as input. Returns a cursor with matching room details as output.
2. If search term is empty or null:
    - Returns all rooms (same as ViewAllRooms)
3. If search term provided:
    - Searches for rooms where the room name contains the search term (case-insensitive)
    - Uses partial matching with wildcards
4. Returns room ID, name, type, and building details for matching rooms
5. Orders results by building name, then room name

```sql
CREATE OR REPLACE PROCEDURE RejectBooking(
    p_booking_id  IN Booking.booking_id%TYPE,
    p_role        IN VARCHAR2,
    p_user_erp    IN NUMBER DEFAULT NULL,  -- For BI verification
    p_success     OUT NUMBER,
    p_message     OUT VARCHAR2
)
AS
    v_room_type   Room.room_type%TYPE;
    v_status      Booking.status%TYPE;
    v_building_id Building.building_id%TYPE;
    v_room_building_id Room.building_id%TYPE;
BEGIN
    p_success := 0;
    p_message := '';

    -- Get booking details
    BEGIN
        SELECT r.room_type, b.status, r.building_id
        INTO v_room_type, v_status, v_room_building_id
        FROM Booking b
        JOIN Room r ON b.room_id = r.room_id
        WHERE b.booking_id = p_booking_id;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            p_message := 'Booking not found';
            RETURN;
    END;

    -- Check status is 'Approved'
    IF v_status != 'Approved' THEN
        p_message := 'Only Approved bookings can be rejected';
        RETURN;
    END IF;

    IF p_role = 'ProgramOffice' THEN
        -- PO can only reject classrooms
        IF v_room_type != 'CLASSROOM' THEN
            p_message := 'Only classroom bookings can be rejected by Program Office';
            RETURN;
        END IF;
```

```
    ELSIF p_role = 'BuildingIncharge' THEN
        -- BI can only reject breakouts in their building
        IF v_room_type != 'BREAKOUT' THEN
            p_message := 'Only breakout room bookings can be rejected by Building Incharge';
            RETURN;
        END IF;

        -- Verify BI is incharge of this building
        IF p_user_erp IS NULL THEN
            p_message := 'Building Incharge ERP required';
            RETURN;
        END IF;

        BEGIN
            SELECT building_id INTO v_building_id
            FROM Incharge
            WHERE incharge_id = p_user_erp;

            IF v_room_building_id != v_building_id THEN
                p_message := 'Cannot reject bookings from other buildings';
                RETURN;
            END IF;
        EXCEPTION
            WHEN NO_DATA_FOUND THEN
                p_message := 'Building Incharge not found or not assigned to a building';
                RETURN;
        END;

    ELSE
        p_message := 'Invalid role. Only ProgramOffice or BuildingIncharge can reject bookings';
        RETURN;
    END IF;

    -- Reject the booking
    UPDATE Booking
    SET status = 'Rejected'
    WHERE booking_id = p_booking_id;

    COMMIT;

    p_success := 1;
    p_message := 'Booking rejected successfully';

EXCEPTION
    WHEN OTHERS THEN
        ROLLBACK;
        p_success := 0;
        p_message := 'Error rejecting booking: ' || SQLERRM;
END RejectBooking;
/
```

Procedure RejectBooking

Purpose: This procedure provides a unified method for administrators to reject bookings, with role-based permissions that differ between Program Office and Building Incharges.

Steps:

1. Takes booking ID, user role, and (for BI) user ERP as inputs. Returns success status and message as outputs.
2. Retrieves booking details: room type, current status, and building location
3. Validates booking has 'Approved' status (only approved bookings can be rejected)
4. Role-specific validation:

For Program Office (PO):

   - Can only reject 'CLASSROOM' type rooms
   - Returns error if room is not a classroom

For Building Incharge (BI):

   - Can only reject 'BREAKOUT' type rooms
   - Must provide their ERP for verification
   - Must be assigned to the building containing the room
   - Returns error if room not in their building or wrong room type

5. If all validations pass, updates booking status to 'Rejected'
6. Commits transaction
7. Returns success = 1 and success message
8. If booking not found, invalid role, or other errors:
   - Returns specific error message
   - Rolls back transaction if database error occurs

Trigger: trg_booking_future_date

```
-- Create trigger for future date validation
CREATE OR REPLACE TRIGGER trg_booking_future_date
BEFORE INSERT OR UPDATE ON Booking
FOR EACH ROW
BEGIN
    IF :NEW.booking_date < TRUNC(SYSDATE) THEN
        RAISE_APPLICATION_ERROR(-20001, 'Booking date cannot be in the past.');
    END IF;
END;
/
```

Purpose:

This trigger ensures that no booking can be inserted or updated with a past date. It enforces data integrity by validating that booking_date is always today or a future date.

Steps:

- The trigger fires **before every INSERT or UPDATE** on the Booking table.
- It checks the value of :NEW.booking_date against the system date (SYSDATE), trimmed to remove the time component.
- If the new booking date is earlier than today's date:
    - The trigger raises an application error using RAISE_APPLICATION_ERROR.
    - The error message returned is: **"Booking date cannot be in the past."**
- If the date is valid, the operation proceeds normally.

Trigger: trg_check_student_role

```
CREATE OR REPLACE TRIGGER trg_check_student_role
BEFORE INSERT ON Student
FOR EACH ROW
DECLARE
    v_user_role User_Table.role%TYPE;
BEGIN
    -- Check that the user has 'Student' role
    SELECT role INTO v_user_role
    FROM User_Table
    WHERE ERP = :NEW.ERP;

    IF v_user_role != 'Student' THEN
        RAISE_APPLICATION_ERROR(-20001,
            'Only users with Student role can have Student records');
    END IF;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RAISE_APPLICATION_ERROR(-20002, 'User not found in User_Table');
END;
/
```

Purpose:

This trigger ensures that only users with the **Student** role can have records in the Student table. It validates ERP entries during student creation and prevents role mismatch issues.

Steps:

- The trigger fires **before every INSERT** on the Student table.
- It extracts the user role from User_Table corresponding to the ERP being inserted.
- If the ERP does not exist in User_Table:
    - The trigger raises the error: **"User not found in User_Table"**
- If the ERP exists but the user's role is **not 'Student'**:
    - The trigger raises the error:
      "Only users with Student role can have Student records"
- If the role is valid:
    - The Student record is inserted normally.

Trigger: trg_check_incharge_role

```
CREATE OR REPLACE TRIGGER trg_check_incharge_role
BEFORE INSERT ON Incharge
FOR EACH ROW
DECLARE
    v_user_role User_Table.role%TYPE;
BEGIN
    -- Check that the user has 'BuildingIncharge' role
    SELECT role INTO v_user_role
    FROM User_Table
    WHERE ERP = :NEW.incharge_id;

    IF v_user_role != 'BuildingIncharge' THEN
        RAISE_APPLICATION_ERROR(-20003,
            'Only BuildingIncharge users can be assigned to buildings');
    END IF;
END;
/
```

Purpose:

This trigger ensures that only users with the **BuildingIncharge** role can be assigned as incharges for buildings. It validates ERP entries during building incharge creation and prevents role mismatches.

Steps:

- The trigger fires **before every INSERT** on the Incharge table.
- It retrieves the role of the ERP being inserted from User_Table.
- If the ERP does not have the role **'BuildingIncharge'**:
    - The trigger raises an application error using RAISE_APPLICATION_ERROR.
    - The error message returned is: **"Only BuildingIncharge users can be assigned to buildings"**.
- If the role is valid:
    - The Incharge record is inserted normally.

Trigger: trg_booking_status_change

```
CREATE OR REPLACE TRIGGER trg_booking_status_change
BEFORE UPDATE OF status ON Booking
FOR EACH ROW
BEGIN
    -- Log status changes (optional)
    IF :OLD.status != :NEW.status THEN
        DBMS_OUTPUT.PUT_LINE('Booking ' || :OLD.booking_id ||
            ' changed from ' || :OLD.status || ' to ' || :NEW.status);
    END IF;
END;
/
```

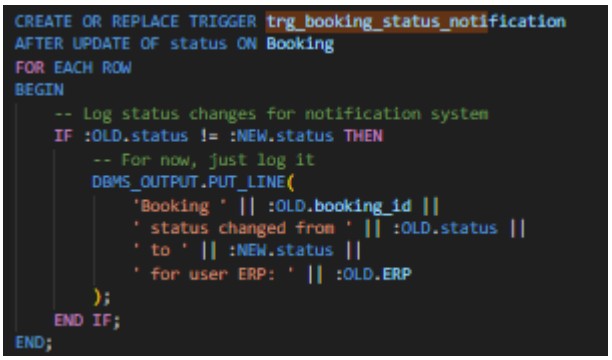
Purpose:

This trigger monitors changes to the status column of the Booking table. It allows administrators or developers to track when booking statuses are updated, such as from 'Approved' to 'Cancelled' or 'Rejected'.

Steps:

- The trigger fires **before every UPDATE** on the status column of the Booking table.
- It compares the old value (:OLD.status) with the new value (:NEW.status).
- If the status has changed:
    - The trigger outputs a message using DBMS_OUTPUT.PUT_LINE showing the booking ID and the status change.
        - Example message: **"Booking 101 changed from Approved to Cancelled"**
- If the status has not changed:
    - No action is taken.

Trigger: trg_booking_status_notification



```
CREATE OR REPLACE TRIGGER trg_booking_status_notification
AFTER UPDATE OF status ON Booking
FOR EACH ROW
BEGIN
    -- Log status changes for notification system
    IF :OLD.status != :NEW.status THEN
        -- For now, just log it
        DBMS_OUTPUT.PUT_LINE(
            'Booking ' || :OLD.booking_id ||
            ' status changed from ' || :OLD.status ||
            ' to ' || :NEW.status ||
            ' for user ERP: ' || :OLD.ERP
        );
    END IF;
END;
```

Purpose:

This trigger is designed to monitor booking status changes and act as a **foundation for a notification system**. It alerts the system whenever a booking's status changes (e.g., Approved → Cancelled).

Steps:

- The trigger fires **after every UPDATE** on the status column of the Booking table.
- It compares the old (:OLD.status) and new (:NEW.status) values.
- If the status has changed:
    - Logs a message using DBMS_OUTPUT.PUT_LINE that includes:
        - Booking ID
        - Previous status
        - New status
        - User ERP associated with the booking
    - Example log:
      "Booking 101 status changed from Approved to Cancelled for user ERP: 28084"

- If the status has not changed, no action is taken.

Trigger: trg_validate_booking

```
CREATE OR REPLACE TRIGGER trg_validate_booking
BEFORE INSERT ON Booking
FOR EACH ROW
DECLARE
    v_conflict_count NUMBER;
    v_schedule_count NUMBER;
    v_day_of_week VARCHAR2(10);
BEGIN
    v_day_of_week := UPPER(TO_CHAR(:NEW.booking_date, 'DAY'));
    -- Check schedule conflicts
    SELECT COUNT(*) INTO v_schedule_count
    FROM Schedule s
    WHERE s.room_id = :NEW.room_id
      AND s.day_of_week = v_day_of_week
      AND NOT (
          s.end_time <= :NEW.start_time OR
          s.start_time >= :NEW.end_time
      );
    IF v_schedule_count > 0 THEN
        RAISE_APPLICATION_ERROR(-20010, 'Room is scheduled for classes during this time');
    END IF;
    -- Check booking conflicts (only check 'Approved' bookings)
    SELECT COUNT(*) INTO v_conflict_count
    FROM Booking b
    WHERE b.room_id = :NEW.room_id
      AND b.booking_date = :NEW.booking_date
      AND b.status = 'Approved'
      AND NOT (
          b.end_time <= :NEW.start_time OR
          b.start_time >= :NEW.end_time
      );
    IF v_conflict_count > 0 THEN
        RAISE_APPLICATION_ERROR(-20011, 'Room already booked for this time slot');
    END IF;
    -- Auto-set status to 'Approved' (since no conflicts)
    :NEW.status := 'Approved';

END;
/
-- ========================================
```

Purpose:
This trigger ensures that **new bookings do not conflict** with existing approved bookings or scheduled classes. It automatically approves a booking if no conflicts exist.

Steps:

- Fires **before every INSERT** on the Booking table.
- Converts the booking date to the **day of the week** to check against class schedules.
- **Schedule conflict check:**
    - Checks the Schedule table for the same room and day.
    - Ensures that the new booking does **not overlap** with any class.

- If overlap exists → raises error:
  "Room is scheduled for classes during this time"

- **Booking conflict check:**
  - Checks the Booking table for approved bookings in the same room, same date, and overlapping times.
  - If overlap exists → raises error:
    "Room already booked for this time slot"

- If no conflicts are found:
  - Status is automatically set to **'Approved'**.

- No commit/rollback is required because the trigger fires **before insert**, so it validates the data prior to insertion.

Trigger: trg_booking_future_date

```
-- Create trigger for future date validation
CREATE OR REPLACE TRIGGER trg_booking_future_date
BEFORE INSERT OR UPDATE ON Booking
FOR EACH ROW
BEGIN
    IF :NEW.booking_date < TRUNC(SYSDATE) THEN
        RAISE_APPLICATION_ERROR(-20001, 'Booking date cannot be in the past.');
    END IF;
END;
/
```

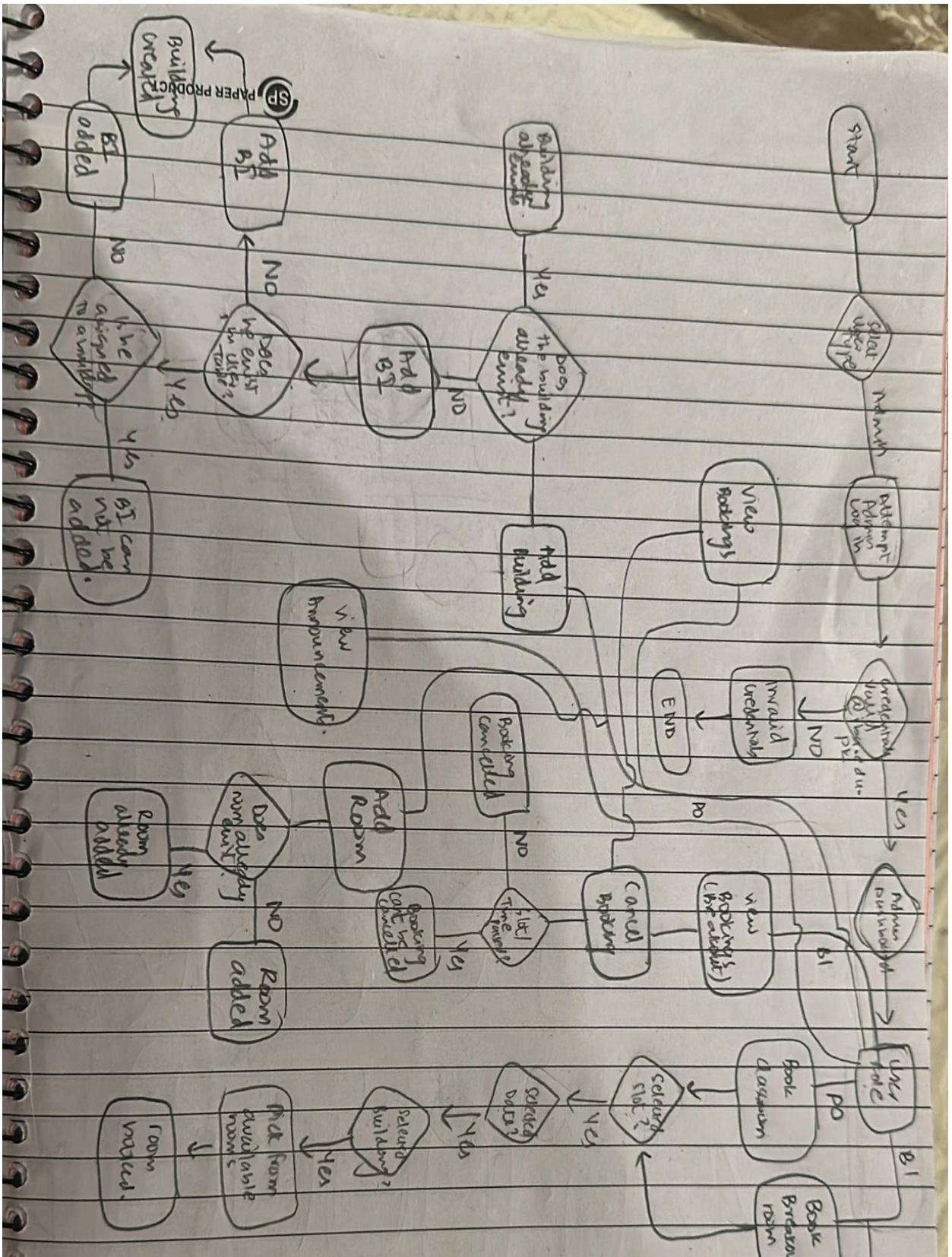
Purpose:
This trigger ensures that bookings cannot be created or updated with a **date in the past**. It enforces the business rule that all bookings must be for the current or a future date.

Steps:

- Fires **before INSERT or UPDATE** on the Booking table.
- Checks the value of :NEW.booking_date against the current date (TRUNC(SYSDATE)):
  - TRUNC(SYSDATE) removes the time component to only compare the date.

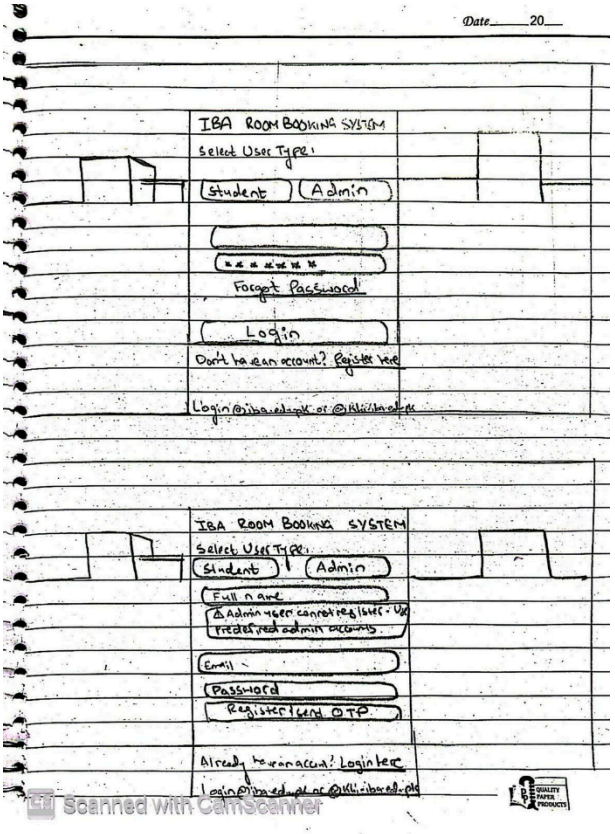
- If the booking date is **earlier than today**:
    - Raises an application error:
      "Booking date cannot be in the past."
    - Prevents the INSERT or UPDATE operation from completing.
- If the booking date is today or in the future, the trigger allows the operation to proceed.

Flow Diagram:

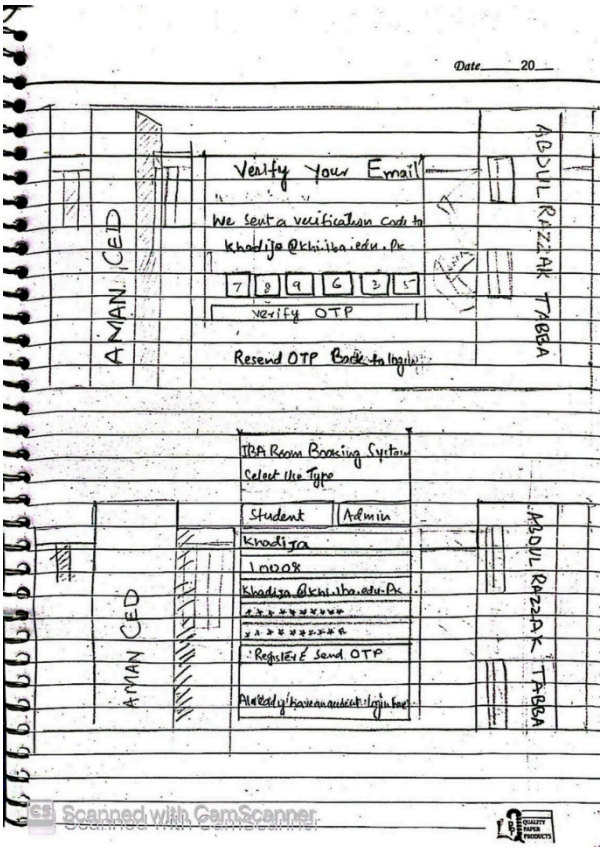


Wireframes:

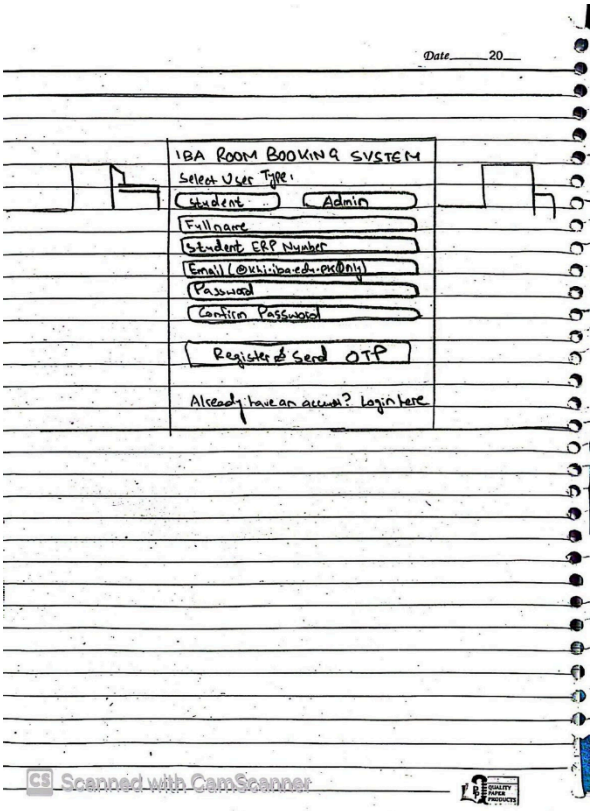
Simple logi



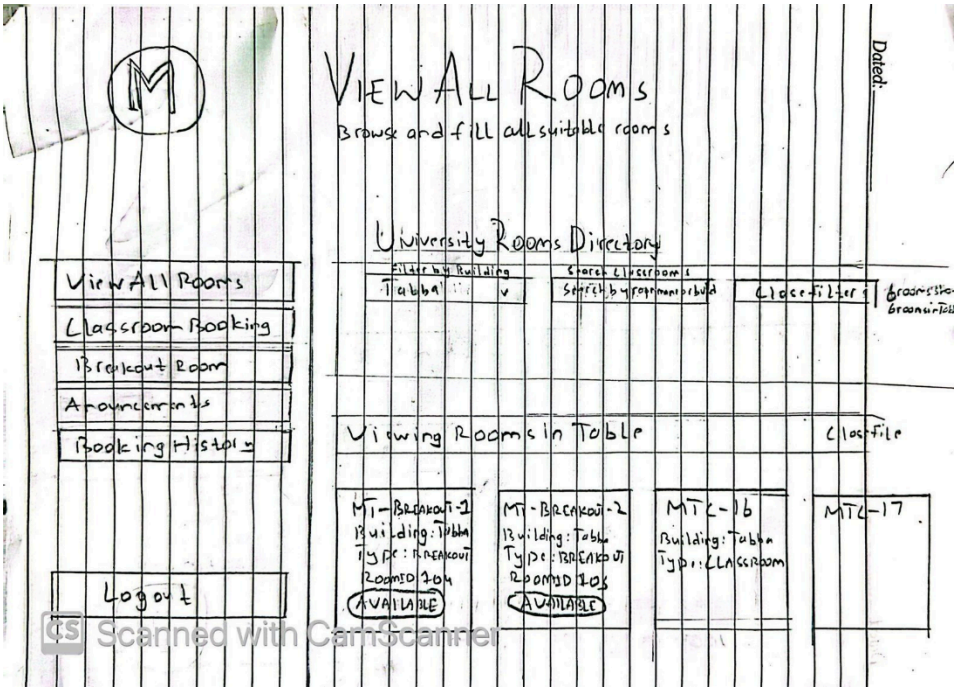
OTP after login



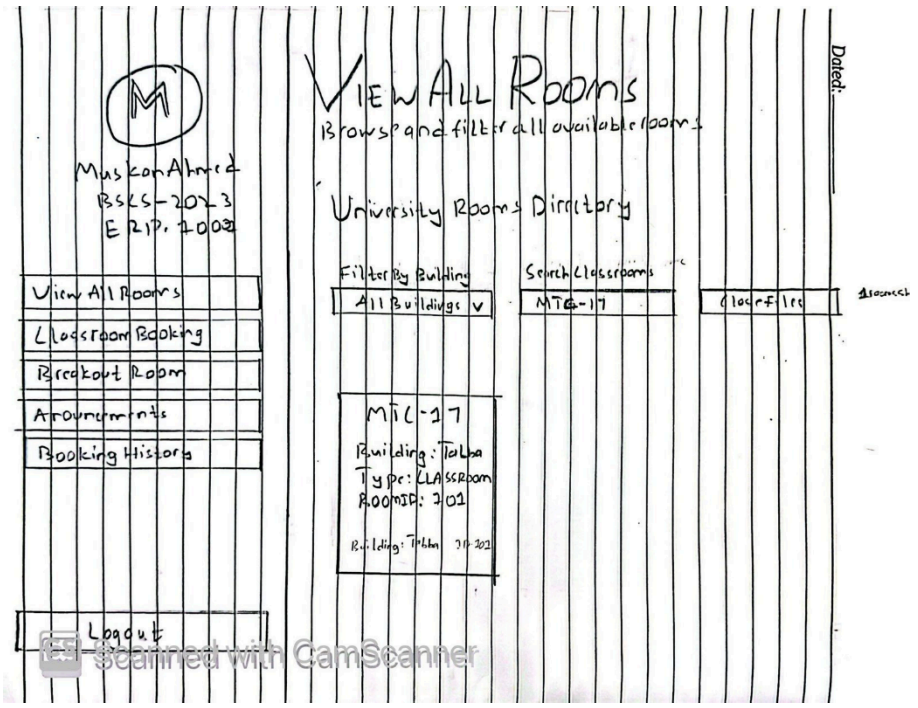
Register student



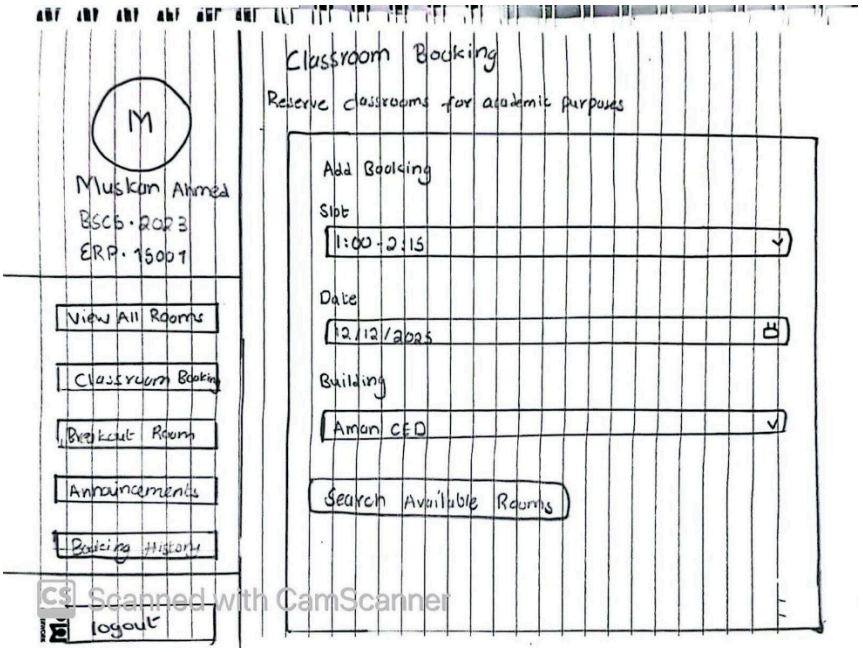
Student -> View all rooms



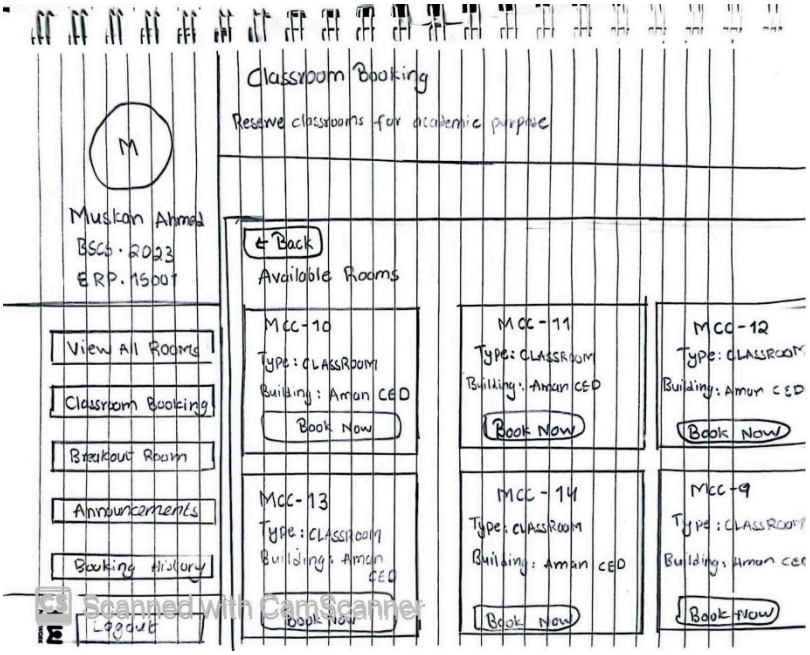
Student-> View all rooms (filters applied)



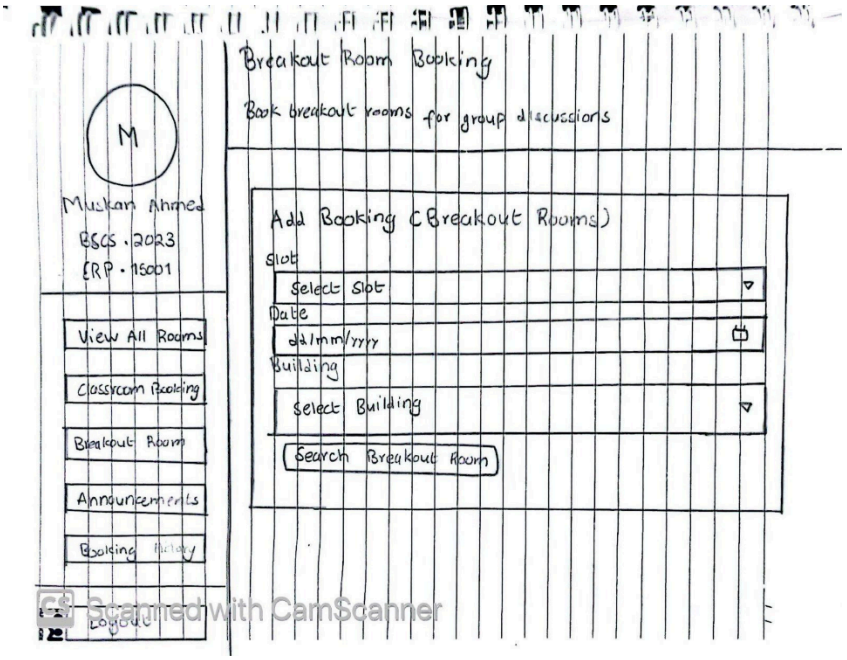
Classroom Booking main screen



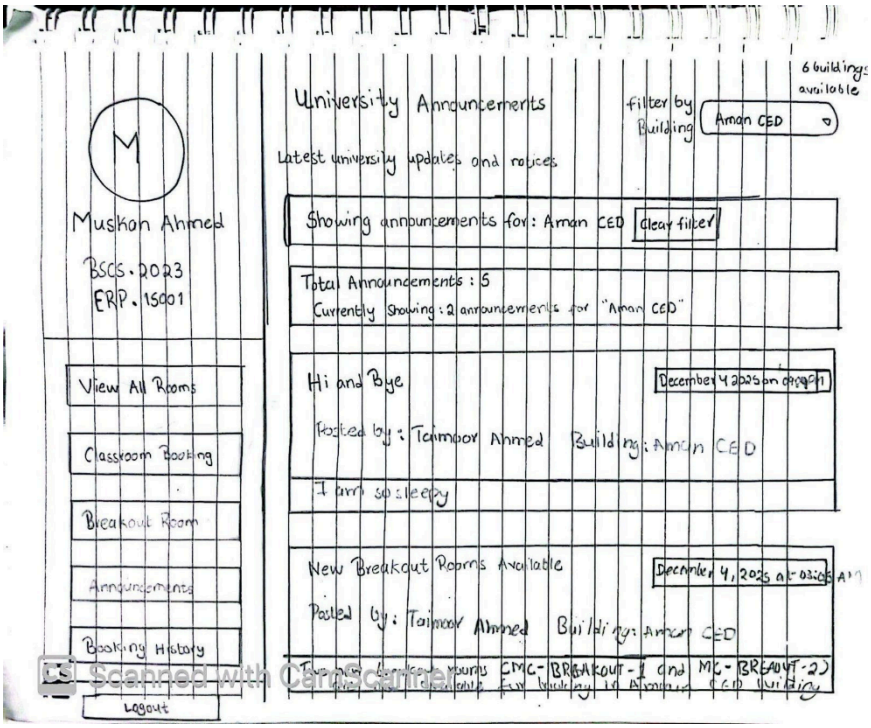
classroom booking -> available rooms



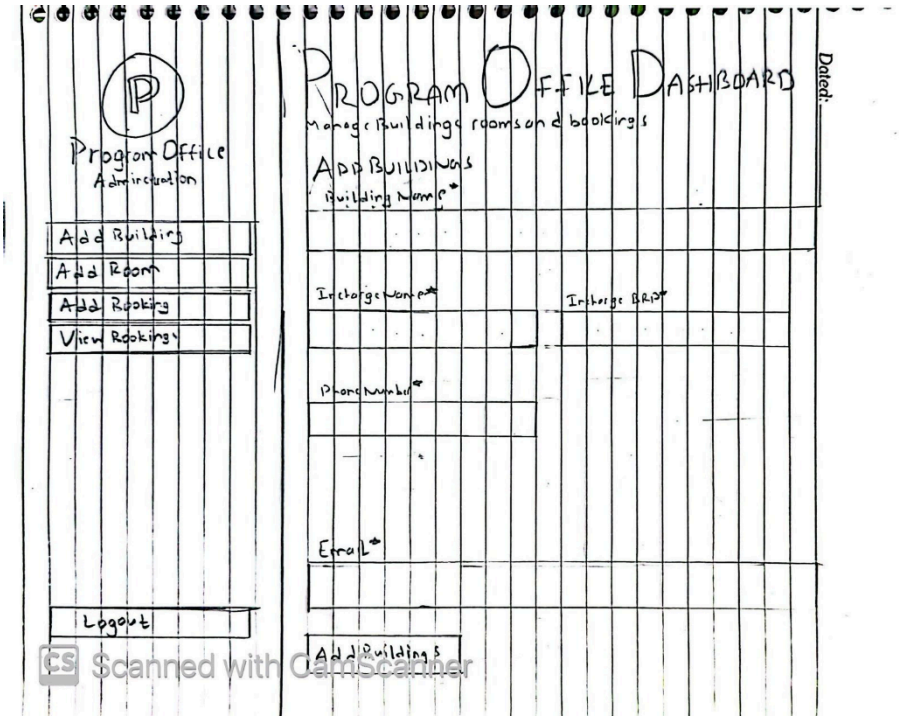
Breakout Room booking



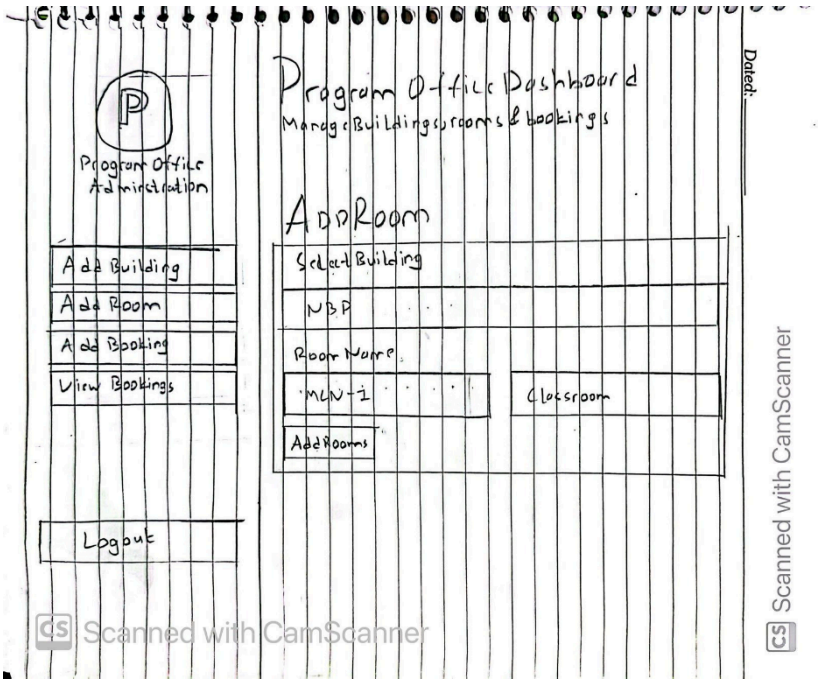
Student -> university announcements



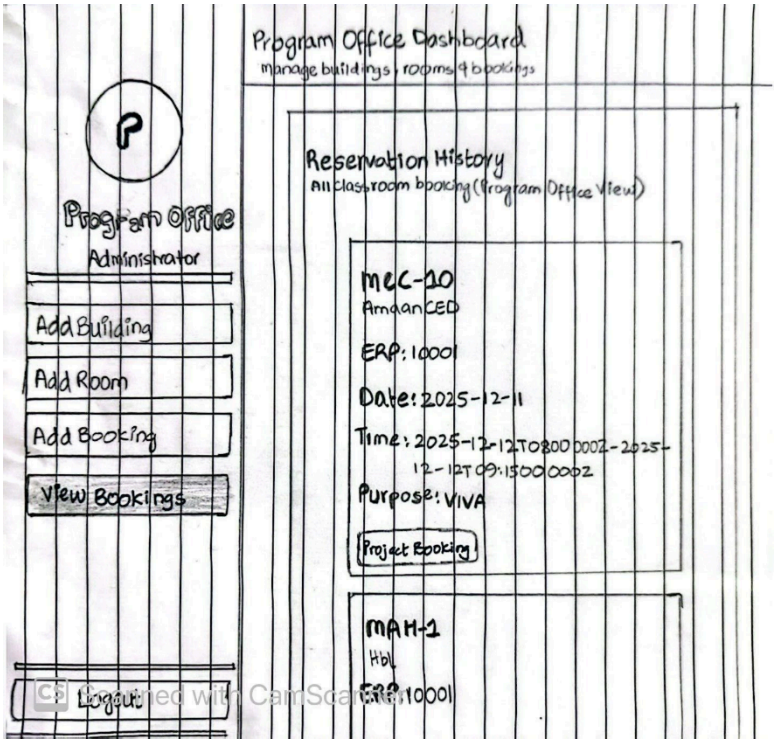
PO -> add building



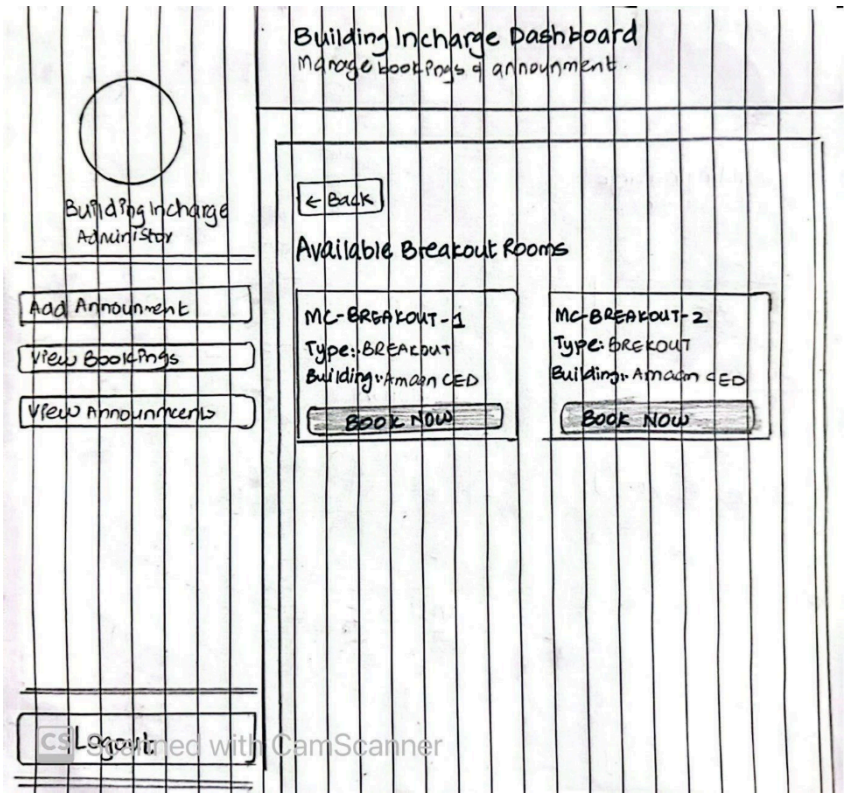
PO -> add room



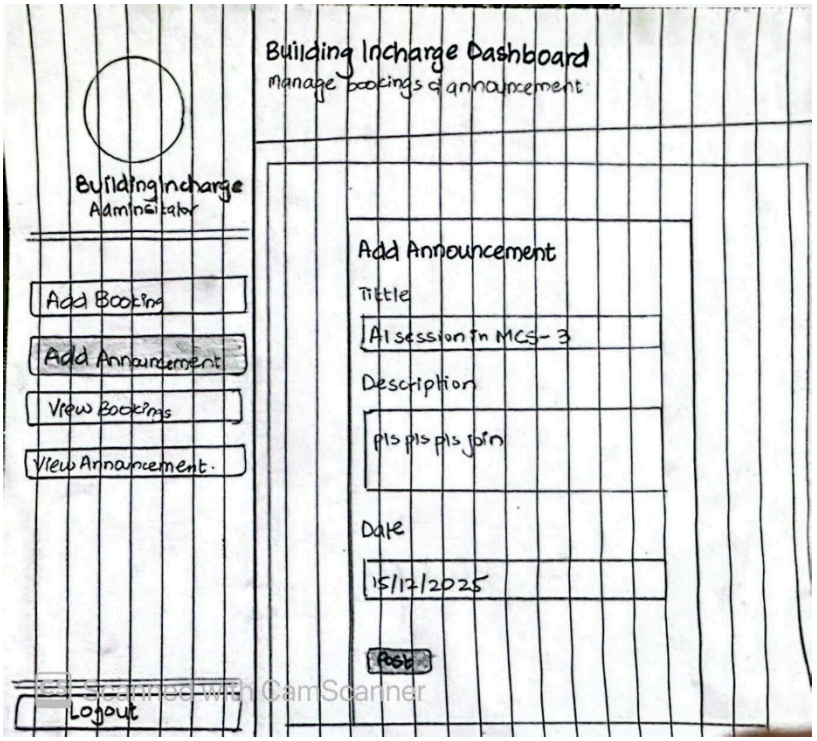
PO -> view bookings



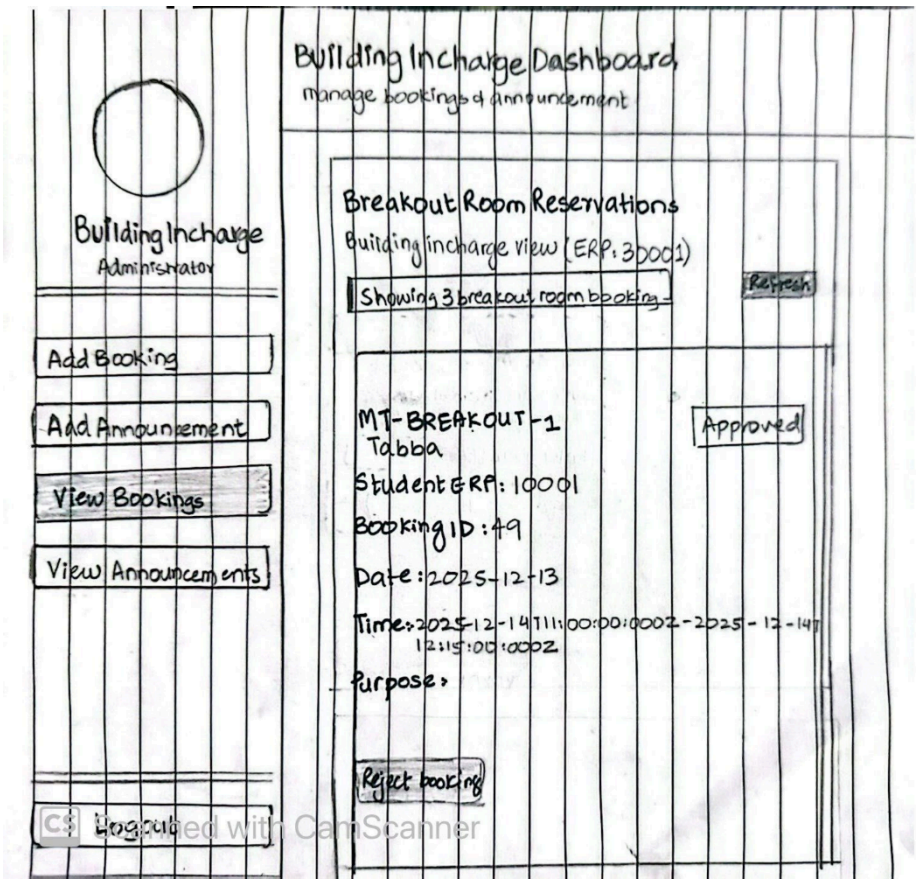
BI -> available rooms



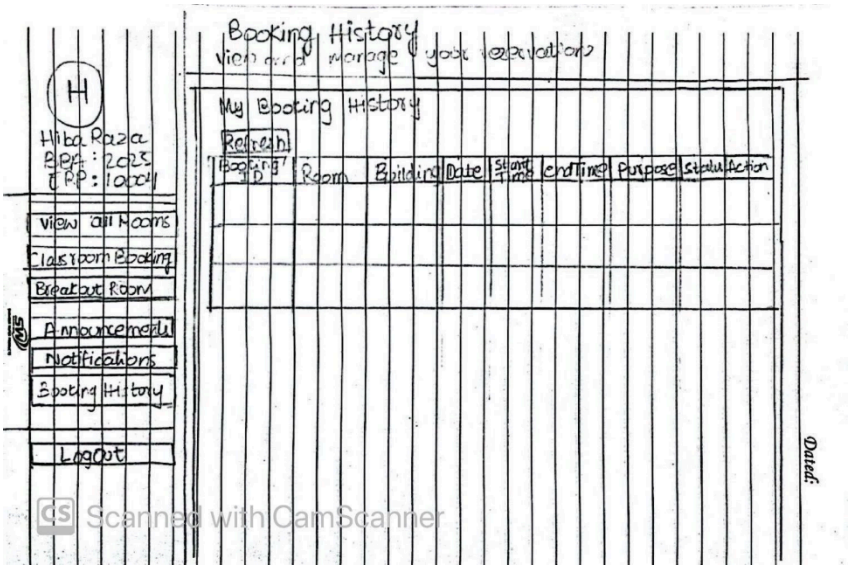
BI -> add announcements



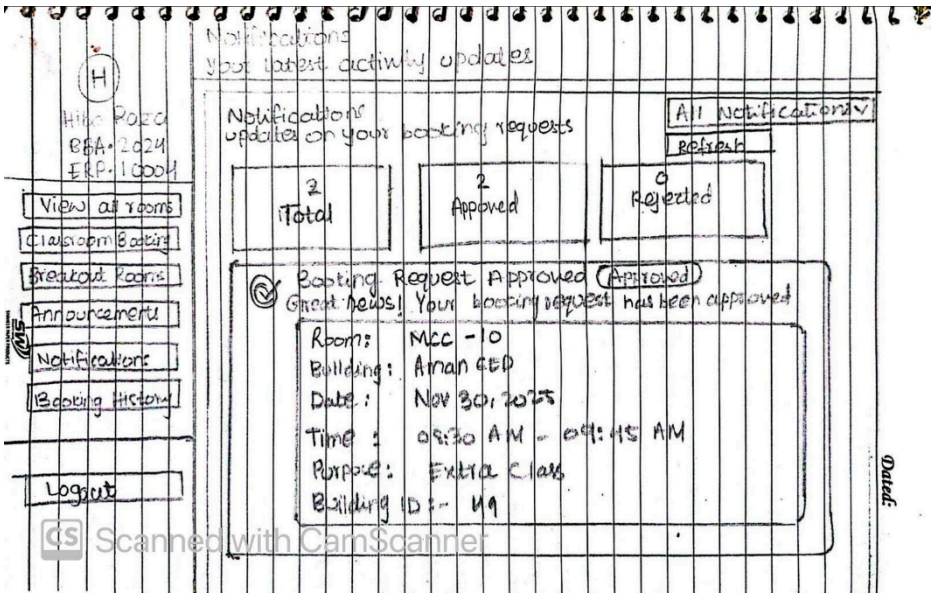
BI -> view bookings



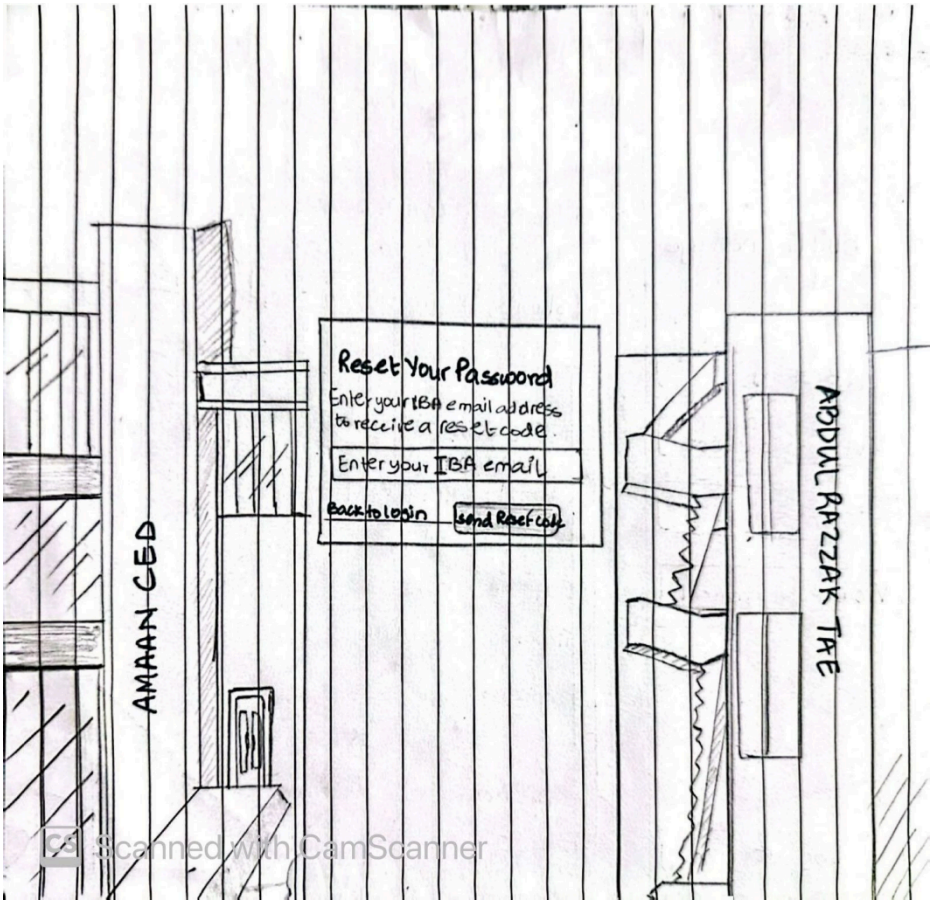
student -> booking history



student -> booking approved

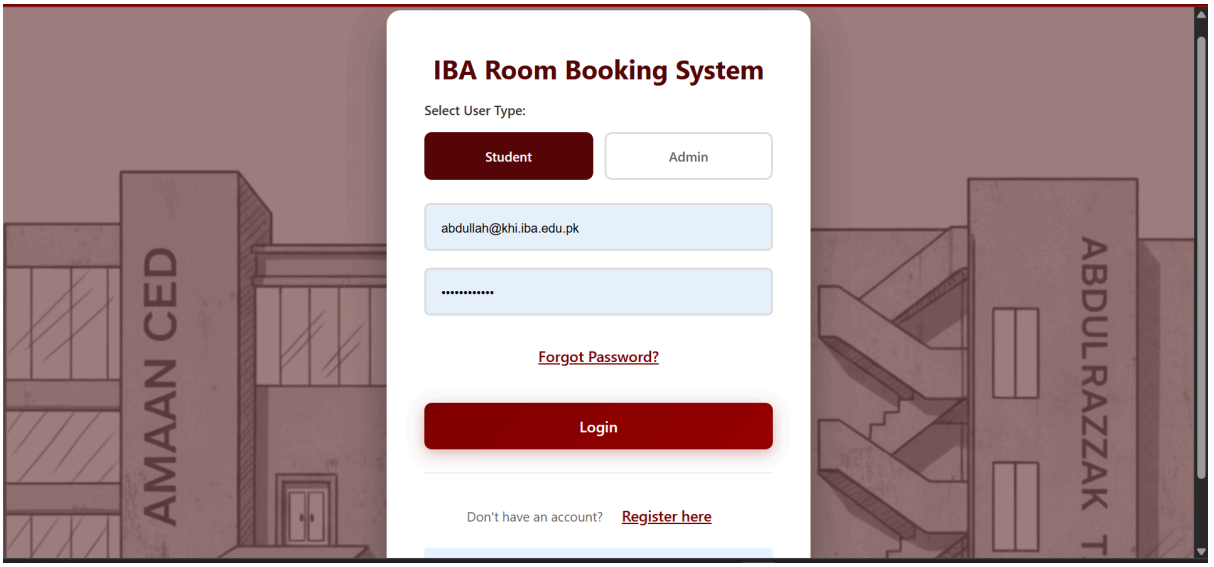


reset password

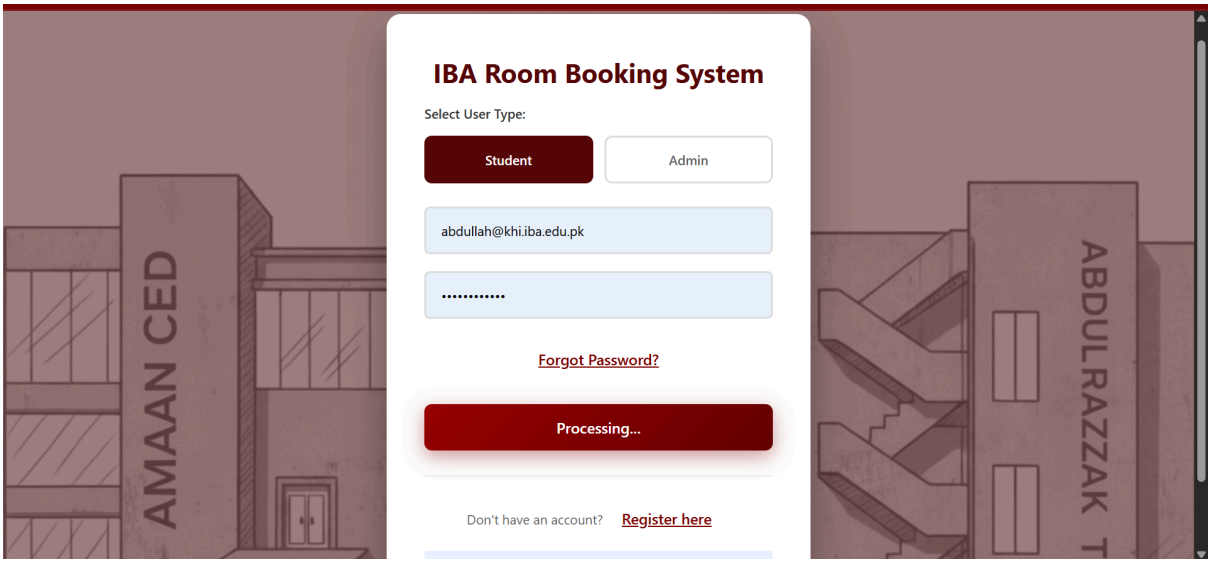


Page by Page Navigation and SQL Queries:

Login Page (for student)



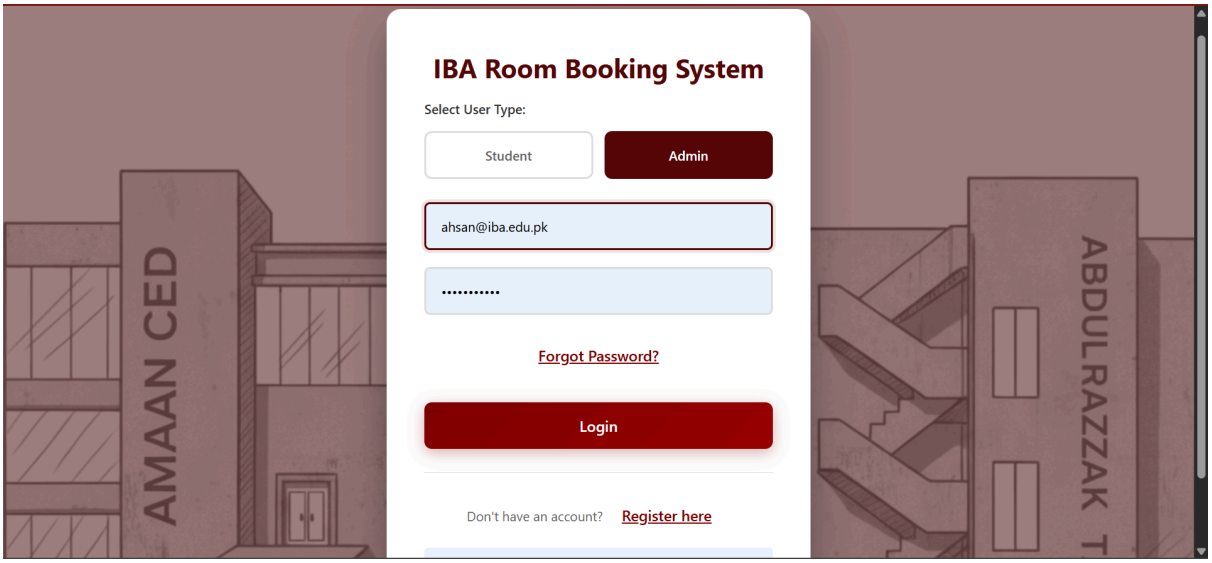
Login Page -> Login Button



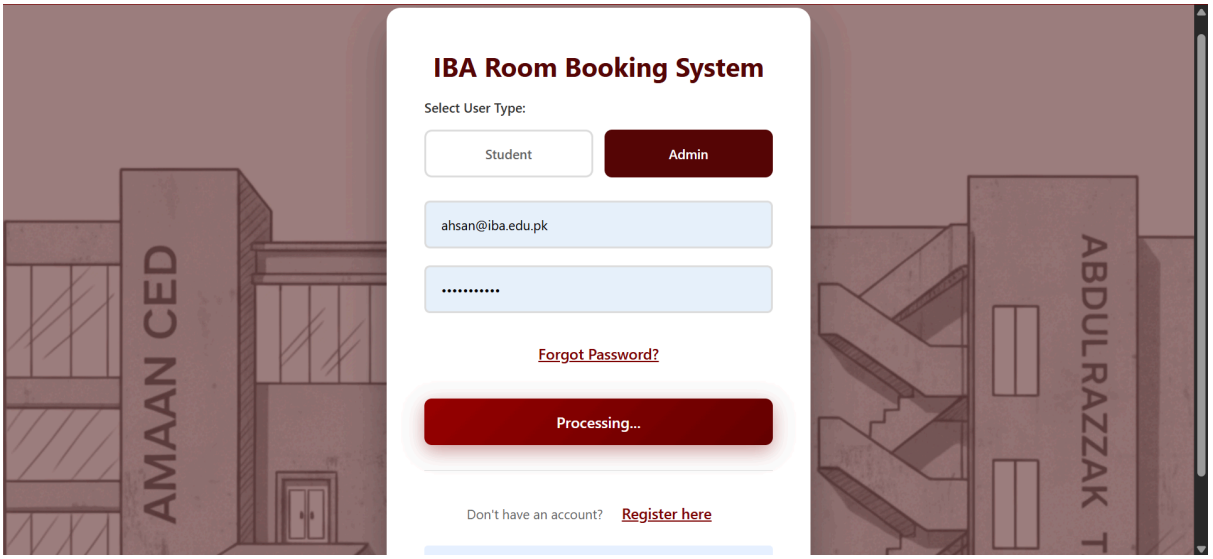

```
CREATE OR REPLACE PROCEDURE StudentLogin(
    p_identifier  IN VARCHAR2,  -- Can be email OR phone
    p_password    IN VARCHAR2,
    p_success     OUT NUMBER,
    p_erp         OUT NUMBER,
    p_name        OUT VARCHAR2,
    p_program     OUT VARCHAR2,
    p_intake_year OUT NUMBER,
    p_message     OUT VARCHAR2
)
AS
    v_is_email BOOLEAN;
BEGIN
    p_success := 0;
    p_erp := NULL;
    p_name := NULL;
    p_program := NULL;
    p_intake_year := NULL;
    p_message := '';

    v_is_email := INSTR(p_identifier, '@') > 0;

    IF v_is_email THEN
        BEGIN
            SELECT u.erp, u.name, s.program, s.intake_year
            INTO p_erp, p_name, p_program, p_intake_year
            FROM User_Table u
            JOIN Student s ON u.ERP = s.ERP
            WHERE u.email = p_identifier
              AND u.user_password = p_password
              AND u.role = 'Student';

            p_success := 1;
            p_message := 'Login successful';
```

```
        EXCEPTION
            WHEN NO_DATA_FOUND THEN
                p_message := 'Invalid email or password';
            WHEN TOO_MANY_ROWS THEN
                p_message := 'System error: Multiple accounts found';
        END;
    ELSE
        BEGIN
            SELECT u.erp, u.name, s.program, s.intake_year
            INTO p_erp, p_name, p_program, p_intake_year
            FROM User_Table u
            JOIN Student s ON u.ERP = s.ERP
            WHERE u.phone_number = p_identifier
              AND u.user_password = p_password
              AND u.role = 'Student';

            p_success := 1;
            p_message := 'Login successful';

        EXCEPTION
            WHEN NO_DATA_FOUND THEN
                p_message := 'Invalid phone number or password';
            WHEN TOO_MANY_ROWS THEN
                p_message := 'System error: Multiple accounts found';
        END;
    END IF;

END StudentLogin;
/
```

Login Page (For Admin = BuildingIncharge/ProgramOffice)



Login Page -> Login Button



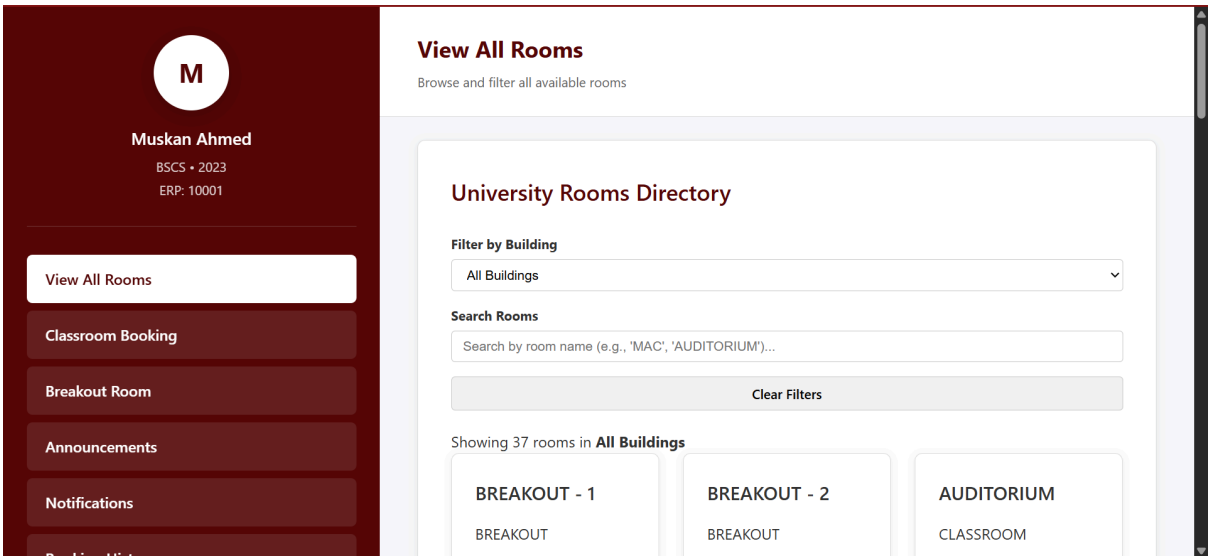

```
CREATE OR REPLACE PROCEDURE AdminLogin(
    p_identifier IN VARCHAR2,  -- Can be email OR phone
    p_password   IN User_Table.user_password%TYPE,
    p_success    OUT NUMBER,
    p_role       OUT VARCHAR2,
    p_erp        OUT NUMBER,
    p_name       OUT VARCHAR2,
    p_message    OUT VARCHAR2
)
AS
    v_is_email BOOLEAN;
BEGIN

    p_success := 0;
    p_role := NULL;
    p_erp := NULL;
    p_name := NULL;
    p_message := '';

    -- Determine identifier type
    v_is_email := INSTR(p_identifier, '@') > 0;

    IF v_is_email THEN
        -- Login with email
        BEGIN
            SELECT erp, name, role
            INTO p_erp, p_name, p_role
            FROM User_Table
            WHERE email = p_identifier
              AND user_password = p_password
              AND role IN ('ProgramOffice', 'BuildingIncharge');
```

```
            p_success := 1;
            p_message := 'Admin login successful';

        EXCEPTION
            WHEN NO_DATA_FOUND THEN
                p_message := 'Invalid admin credentials';
        END;
    ELSE
        -- Login with phone
        BEGIN
            SELECT erp, name, role
            INTO p_erp, p_name, p_role
            FROM User_Table
            WHERE phone_number = p_identifier
              AND user_password = p_password
              AND role IN ('ProgramOffice', 'BuildingIncharge');

            p_success := 1;
            p_message := 'Admin login successful';

        EXCEPTION
            WHEN NO_DATA_FOUND THEN
                p_message := 'Invalid admin credentials';
        END;
    END IF;

END AdminLogin;
/
```

Purpose:
The purpose of StudentLogin & AdminLogin is to check and ensure that the correct dashboard and right access is given to the user. From Backend, it decides which procedure to call based on the role. Then, these procedures are called which return output, based on which the respective dashboard it shown.

Student's Perspective:

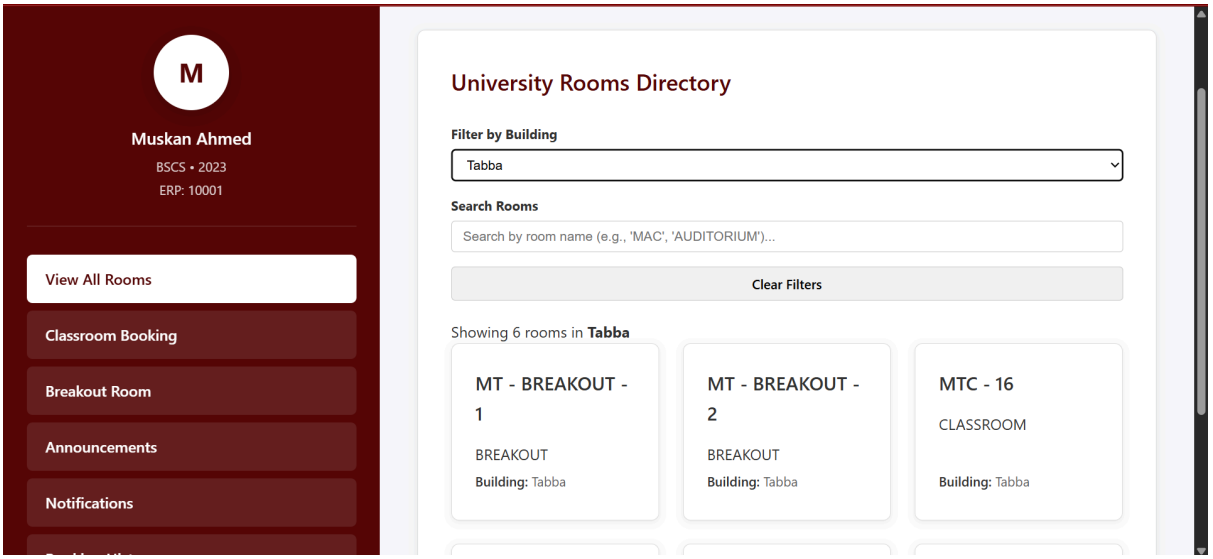
StudentDashboard -> shows view all rooms by default




```
CREATE OR REPLACE PROCEDURE ViewAllRooms(
    p_result OUT SYS_REFCURSOR
)
AS
BEGIN
    OPEN p_result FOR
        SELECT
            r.room_id,
            r.room_name,
            r.room_type,
            b.building_id,
            b.building_name
        FROM Room r
        JOIN Building b ON r.building_id = b.building_id
        ORDER BY b.building_name, r.room_type, r.room_name;

END ViewAllRooms;
/
```

The purpose of this procedure is to show all the rooms that exist in the university. This feature was implemented so that the students can see the rooms in the university as it gets hard to find rooms sometimes. This feature also gives some filters.

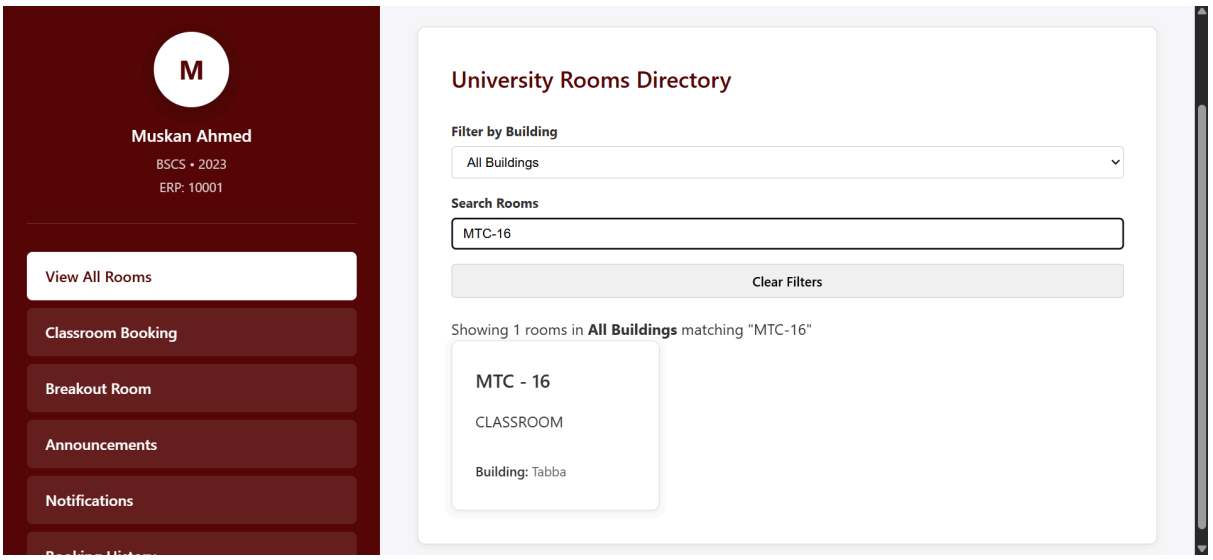
Filter -> Dropdown (Select Building (e.g. Tabba))




```sql
CREATE OR REPLACE PROCEDURE GetRoomsByBuilding(
    p_building_name IN Building.building_name%TYPE,
    p_result        OUT SYS_REFCURSOR
)
AS
    v_building_exists NUMBER;
BEGIN

    -- Check if building exists
    SELECT COUNT(*) INTO v_building_exists
    FROM Building
    WHERE UPPER(building_name) = UPPER(p_building_name);

    IF v_building_exists = 0 THEN
        -- Return empty cursor if building doesn't exist
        OPEN p_result FOR
            SELECT
                r.room_id,
                r.room_name,
                r.room_type,
                b.building_id,
                b.building_name
            FROM Room r
            JOIN Building b ON r.building_id = b.building_id
            WHERE 1 = 0; -- Always false
        RETURN;
    END IF;

    -- Get all rooms in the specified building
    OPEN p_result FOR
        SELECT
            r.room_id,
            r.room_name,
            r.room_type,
            b.building_id,
            b.building_name
        FROM Room r
        JOIN Building b ON r.building_id = b.building_id
        WHERE UPPER(b.building_name) = UPPER(p_building_name)
        ORDER BY r.room_type, r.room_name;

END GetRoomsByBuilding;
/
```

Purpose; This procedure lets user filter the rooms by the building.

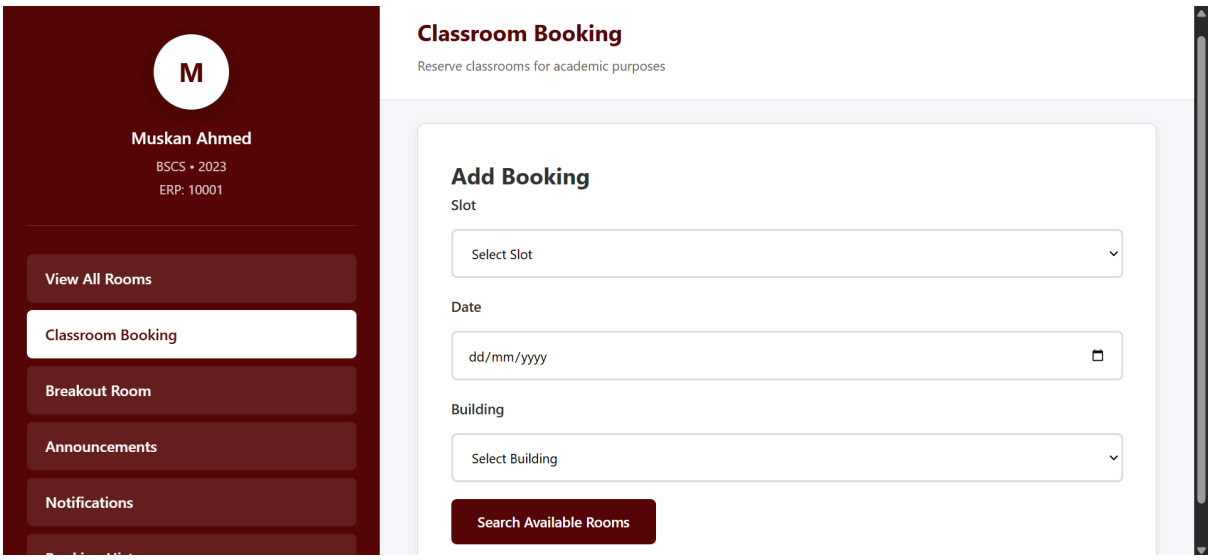
Filter -> Search any classroom



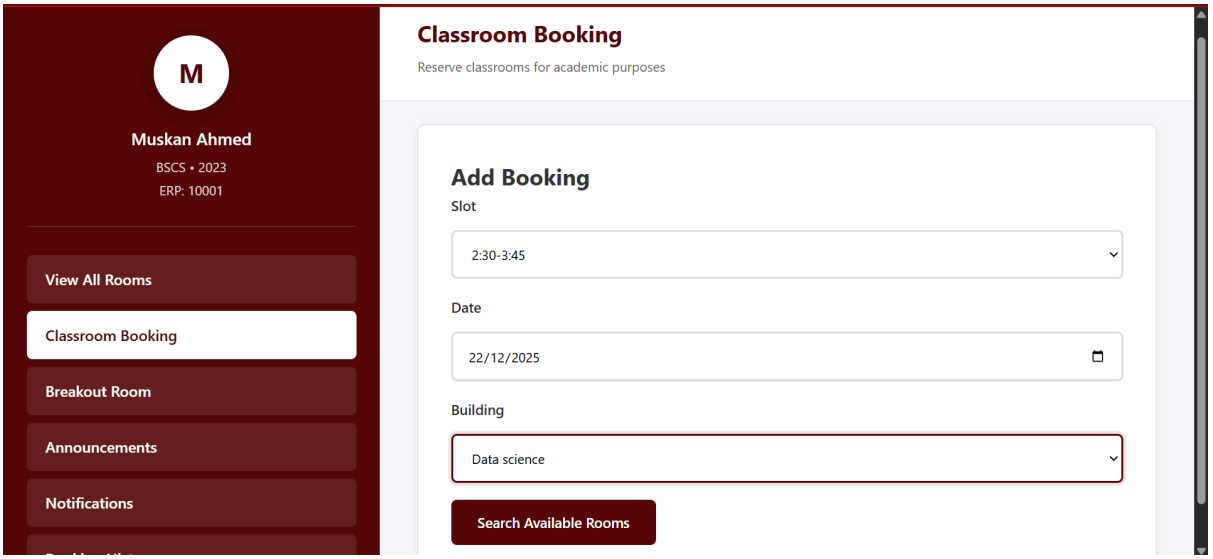

```sql
CREATE OR REPLACE PROCEDURE SearchRoomsByName(
    p_search_term IN VARCHAR2,
    p_result      OUT SYS_REFCURSOR
)
AS
BEGIN
    IF p_search_term IS NULL OR LENGTH(TRIM(p_search_term)) = 0 THEN
        -- If search term is empty, return all rooms
        OPEN p_result FOR
            SELECT
                r.room_id,
                r.room_name,
                r.room_type,
                b.building_id,
                b.building_name
            FROM Room r
            JOIN Building b ON r.building_id = b.building_id
            ORDER BY b.building_name, r.room_name;
    ELSE
        -- Search for rooms with partial match in room_name
        OPEN p_result FOR
            SELECT
                r.room_id,
                r.room_name,
                r.room_type,
                b.building_id,
                b.building_name
            FROM Room r
            JOIN Building b ON r.building_id = b.building_id
            WHERE UPPER(r.room_name) LIKE '%' || UPPER(TRIM(p_search_term)) || '%'
            ORDER BY b.building_name, r.room_name;
    END IF;

END SearchRoomsByName;
/
```

Purpose: This helps the user search a classroom. This is part of the view all rooms feature. It was implemented because sometimes it's hard to find the classroom so using this you can just search the name of the classroom ang get the details like what building it is in.

Classroom Booking -> Page to enter details to view available rooms



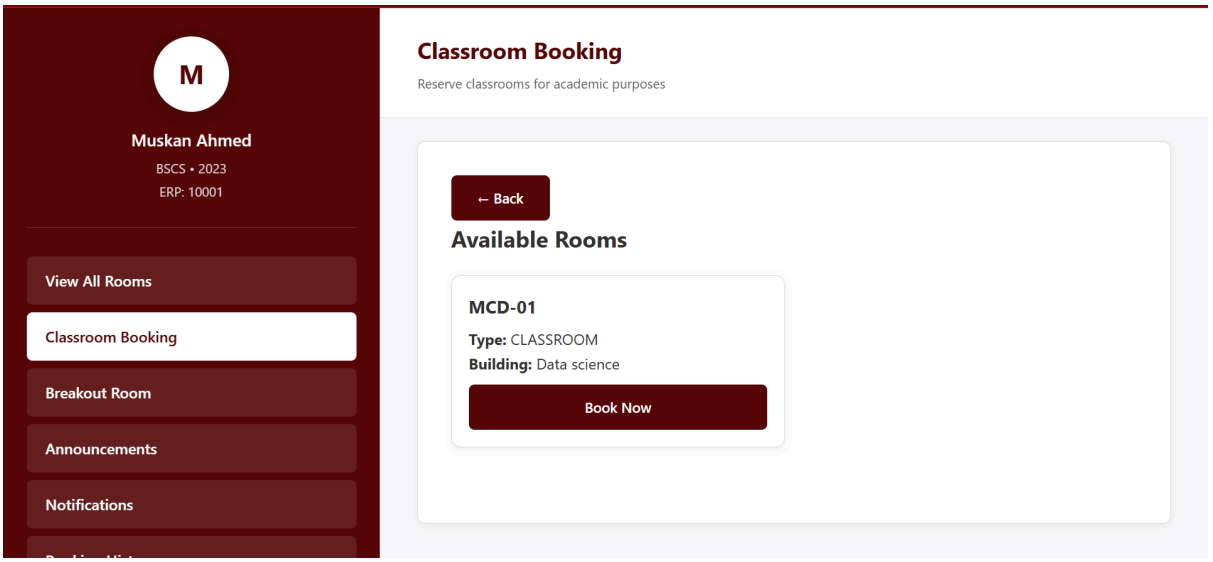
Enter details

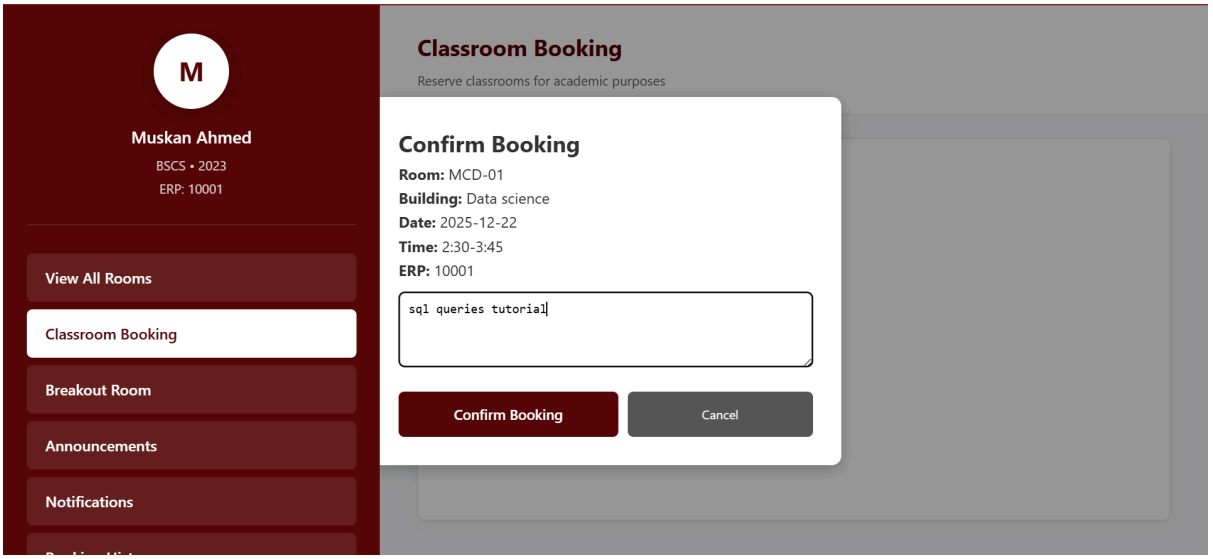


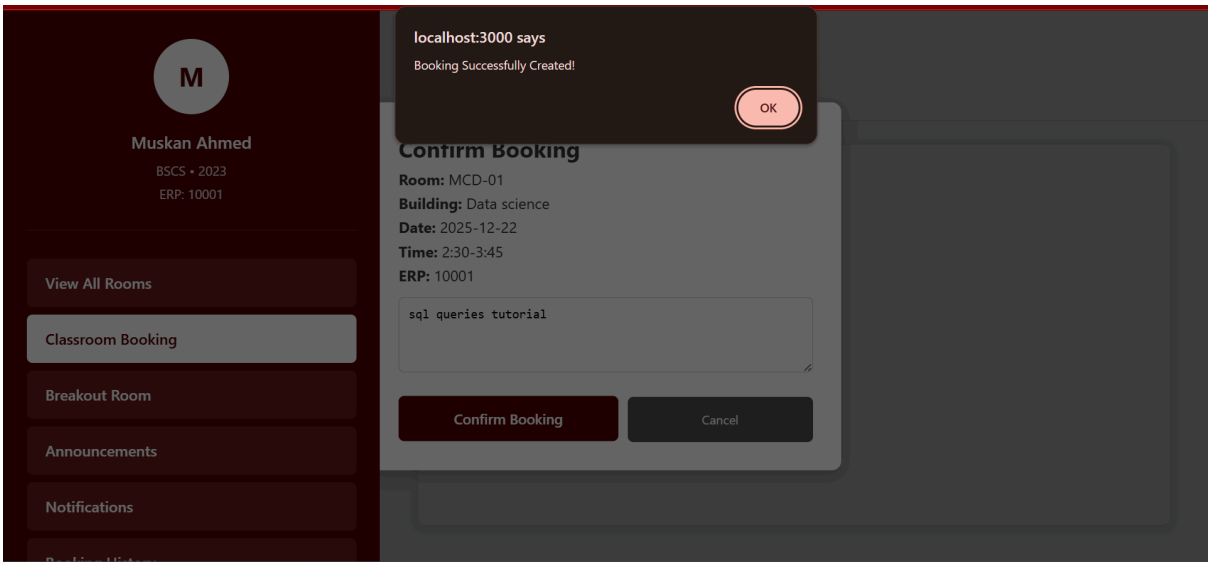
As soon as you hit search available rooms, the request is sent to backend. The backend then calls the following procedure from the database:

```sql
CREATE OR REPLACE PROCEDURE get_available_rooms(
    p_date IN DATE,
    p_start_time IN VARCHAR2,
    p_end_time IN VARCHAR2,
    p_building_id IN NUMBER,
    p_room_type IN VARCHAR2,
    p_cursor OUT SYS_REFCURSOR
)
AS
    v_start_datetime TIMESTAMP;  -- Changed from DATE to TIMESTAMP to match your schema
    v_end_datetime TIMESTAMP;    -- Changed from DATE to TIMESTAMP to match your schema
    v_day_of_week VARCHAR2(10);  -- Changed to match your Schedule.day_of_week
BEGIN
    -- Convert time strings to datetime
    v_start_datetime := TO_TIMESTAMP(TO_CHAR(p_date, 'YYYY-MM-DD') || ' ' || p_start_time, 'YYYY-MM-DD HH24:MI');
    v_end_datetime := TO_TIMESTAMP(TO_CHAR(p_date, 'YYYY-MM-DD') || ' ' || p_end_time, 'YYYY-MM-DD HH24:MI');
    v_day_of_week := UPPER(TO_CHAR(p_date, 'DAY'));  -- Get day of week

    OPEN p_cursor FOR
    SELECT
        r.room_id,
        r.room_name,
        r.room_type,
        b.building_name,
        b.building_id
    FROM Room r
    JOIN Building b ON r.building_id = b.building_id
    WHERE b.building_id = p_building_id
    AND r.room_type = p_room_type
    AND r.room_id NOT IN (
        -- Rooms already booked for this time slot (only approved bookings block availability)
        SELECT room_id FROM Booking
        WHERE booking_date = p_date  -- Changed from date_of_booking to booking_date
        AND status = 'Approved'  -- Changed to match your schema (capitalized)
        AND NOT (
            end_time <= v_start_datetime OR
            start_time >= v_end_datetime
        )
    )
    AND r.room_id NOT IN (
        -- Rooms scheduled for classes
        SELECT room_id FROM Schedule
        WHERE day_of_week = v_day_of_week  -- Changed from day to day_of_week
        AND NOT (
            end_time <= v_start_datetime OR
            start_time >= v_end_datetime
        )
    )
    ORDER BY r.room_name;
END get_available_rooms;
/
```

The purpose of this procedure is to return available rooms based on slot, date, and building.







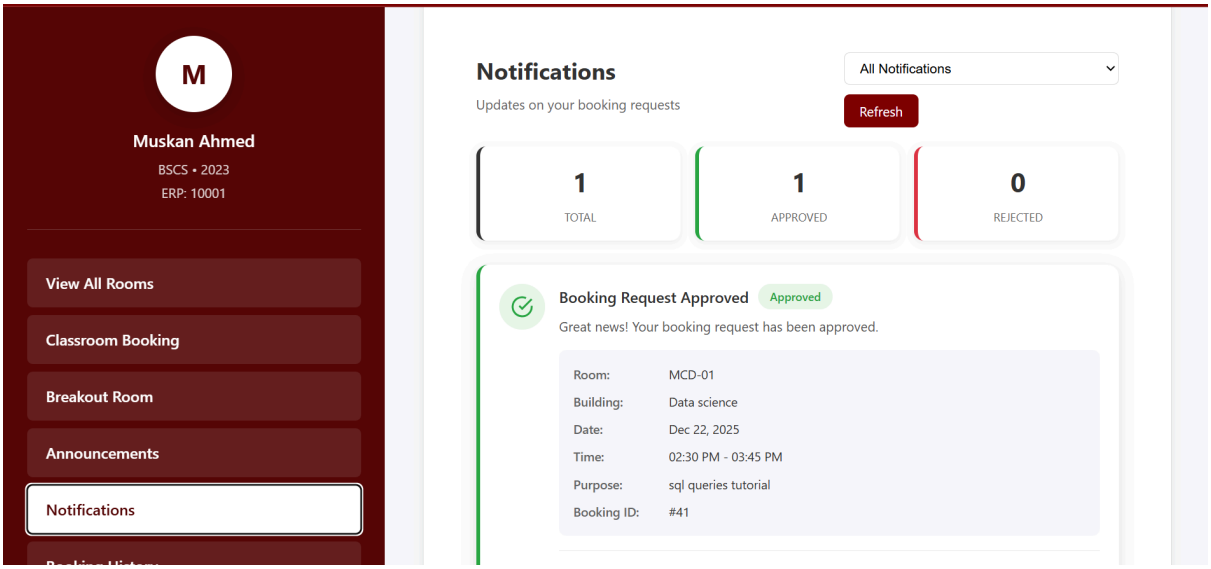
As soon as you enter the purpose and hit confirm booking, the backend sends the information as input to the following procedure:

```
CREATE OR REPLACE PROCEDURE create_booking(
    p_erp IN NUMBER,
    p_room_id IN NUMBER,
    p_date_of_booking IN DATE,
    p_start_time IN VARCHAR2,
    p_end_time IN VARCHAR2,
    p_purpose IN VARCHAR2,
    p_booking_id OUT NUMBER,
    p_success OUT NUMBER,
    p_message OUT VARCHAR2
)
AS
    v_start_datetime TIMESTAMP;  -- Changed to TIMESTAMP
    v_end_datetime TIMESTAMP;    -- Changed to TIMESTAMP
    v_conflict_count NUMBER;
    v_schedule_count NUMBER;
    v_current_date DATE := SYSDATE;
    v_day_of_week VARCHAR2(10);  -- Changed to match your schema
BEGIN
    p_success := 0;
    p_message := 'Booking failed';

    IF TRUNC(p_date_of_booking) < TRUNC(v_current_date) THEN
        p_message := 'Booking date cannot be in the past. Please select today or a future date.';
        RETURN;
    END IF;

    v_start_datetime := TO_TIMESTAMP(TO_CHAR(p_date_of_booking, 'YYYY-MM-DD') || ' ' || p_start_time, 'YYYY-MM-DD HH24:MI');
    v_end_datetime := TO_TIMESTAMP(TO_CHAR(p_date_of_booking, 'YYYY-MM-DD') || ' ' || p_end_time, 'YYYY-MM-DD HH24:MI');
    v_day_of_week := UPPER(TO_CHAR(p_date_of_booking, 'DAY'));  -- Get day of week

    -- If booking is for today, validate that start time is not in the past
    IF TRUNC(p_date_of_booking) = TRUNC(v_current_date) THEN
        IF v_start_datetime < SYSTIMESTAMP THEN  -- Changed to SYSTIMESTAMP for timestamp comparison
            p_message := 'Start time cannot be in the past for today''s booking.';
            RETURN;
        END IF;
    END IF;
```

```
    -- Validate end time is after start time
    IF v_end_datetime <= v_start_datetime THEN
        p_message := 'End time must be after start time.';
        RETURN;
    END IF;

    -- Check for booking conflicts (only check approved bookings)
    SELECT COUNT(*) INTO v_conflict_count
    FROM Booking
    WHERE room_id = p_room_id
    AND booking_date = p_date_of_booking  -- Changed from date_of_booking
    AND status = 'Approved'  -- Capitalized to match your schema
    AND NOT (
        end_time <= v_start_datetime OR
        start_time >= v_end_datetime
    );

    -- Check for schedule conflicts
    SELECT COUNT(*) INTO v_schedule_count
    FROM Schedule
    WHERE room_id = p_room_id
    AND day_of_week = v_day_of_week  -- Changed from day
    AND NOT (
        end_time <= v_start_datetime OR
        start_time >= v_end_datetime
    );

    IF v_conflict_count > 0 THEN
        p_message := 'Room already booked for this time slot';
        RETURN;
    END IF;

    IF v_schedule_count > 0 THEN
        p_message := 'Room is scheduled for classes during this time';
        RETURN;
    END IF;

    -- Insert the booking with 'Approved' status (capitalized to match your schema)
    INSERT INTO Booking (
        ERP, room_id, booking_date,  -- Changed from date_of_booking
        start_time, end_time, purpose, status
    ) VALUES (
        p_erp, p_room_id, p_date_of_booking,
        v_start_datetime, v_end_datetime, p_purpose, 'Approved'
    )
    RETURNING booking_id INTO p_booking_id;

    COMMIT;
    p_success := 1;
    p_message := 'Booking approved successfully!';

EXCEPTION
    WHEN OTHERS THEN
        ROLLBACK;
        p_message := 'Error creating booking: ' || SQLERRM;
        p_success := 0;
END create_booking;
/
```

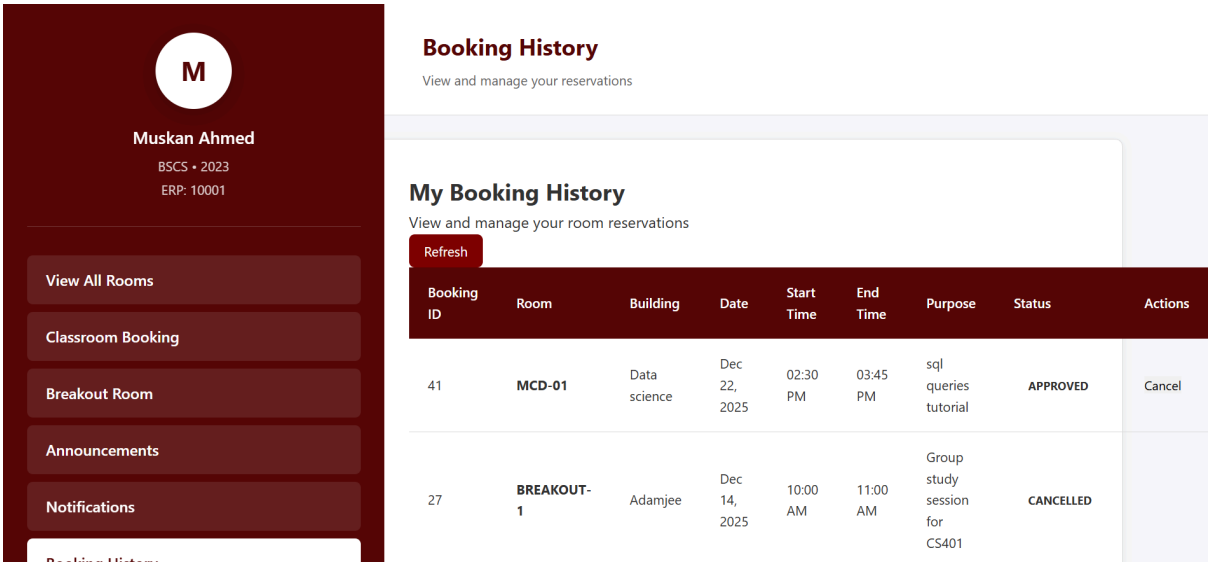
This query takes then outputs if the booking was successful or not. If it was, it says Booking Confirmed (it can be seen in the above screenshot)

As soon as the booking is confirmed, it is shown in the reservation history of the student and is also reflected in the notification tab.



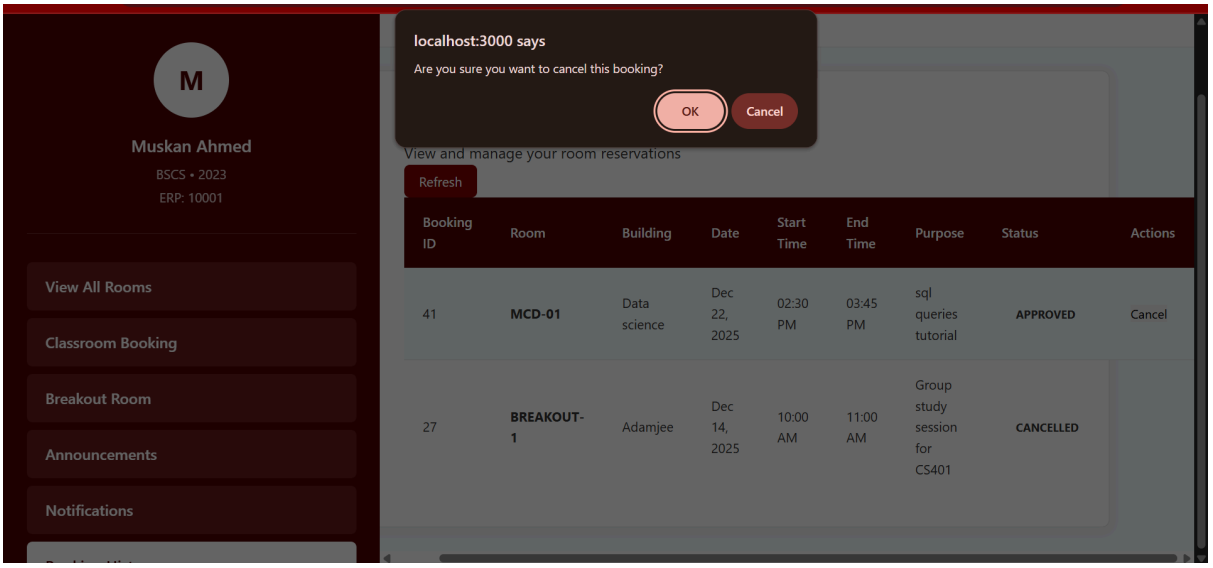

```
CREATE OR REPLACE PROCEDURE ShowReservationHistoryForStudent(
    p_erp    IN Booking.ERP%TYPE,
    p_result OUT SYS_REFCURSOR
)
AS
BEGIN
    OPEN p_result FOR
        SELECT
            b.booking_id,
            b.room_id,
            r.room_name,
            bld.building_name,
            b.booking_date,
            b.start_time,
            b.end_time,
            b.purpose,
            b.status,
            b.created_date
        FROM Booking b
        JOIN Room r ON b.room_id = r.room_id
        JOIN Building bld ON r.building_id = bld.building_id
        WHERE b.ERP = p_erp
        ORDER BY b.booking_date DESC, b.start_time DESC;
END;
/
```

And in the booking history



This Procedure displays the reservation history of the student.

The user also has the option to cancel his/her booking

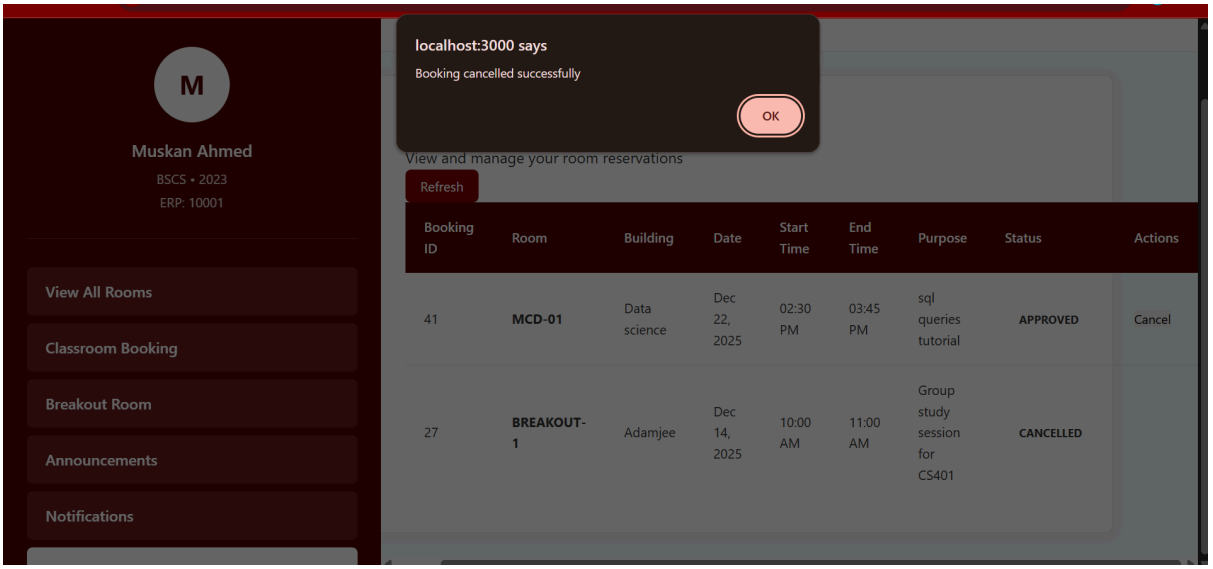


as soon as they hit okay, the backend calls the following procedure

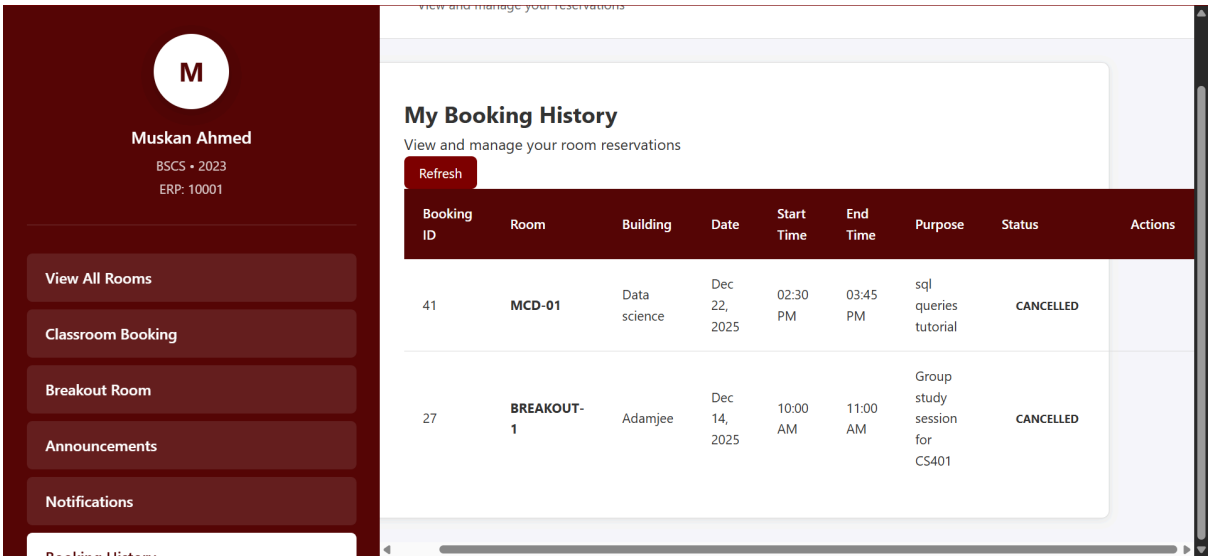
```
CREATE OR REPLACE PROCEDURE CancelBookingByStudent(
    p_booking_id IN Booking.booking_id%TYPE,
    p_erp        IN Booking.ERP%TYPE,
    p_success    OUT NUMBER,
    p_message    OUT VARCHAR2
)
AS
    v_booking_exists NUMBER;
    v_current_status Booking.status%TYPE;
    v_start_datetime TIMESTAMP;
BEGIN
    p_success := 0;
    p_message := '';
    -- First check if booking exists and belongs to student
    SELECT COUNT(*)
    INTO v_booking_exists
    FROM Booking
    WHERE booking_id = p_booking_id
      AND ERP = p_erp;

    IF v_booking_exists = 0 THEN
        p_message := 'Booking not found or does not belong to you';
        RETURN;
    END IF;

    -- Now get booking details separately
    BEGIN
        SELECT status, booking_date + (start_time - TRUNC(start_time))
        INTO v_current_status, v_start_datetime
        FROM Booking
        WHERE booking_id = p_booking_id
          AND ERP = p_erp;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            p_message := 'Booking details not found';
```

This purpose of this procedure is to check if the booking was made by the user who wants to cancel it. Once all the checks are passed, the booking is cancelled and the status in the booking table is updated from “APPROVED” to “CANCELLED”.

If it is successful, then



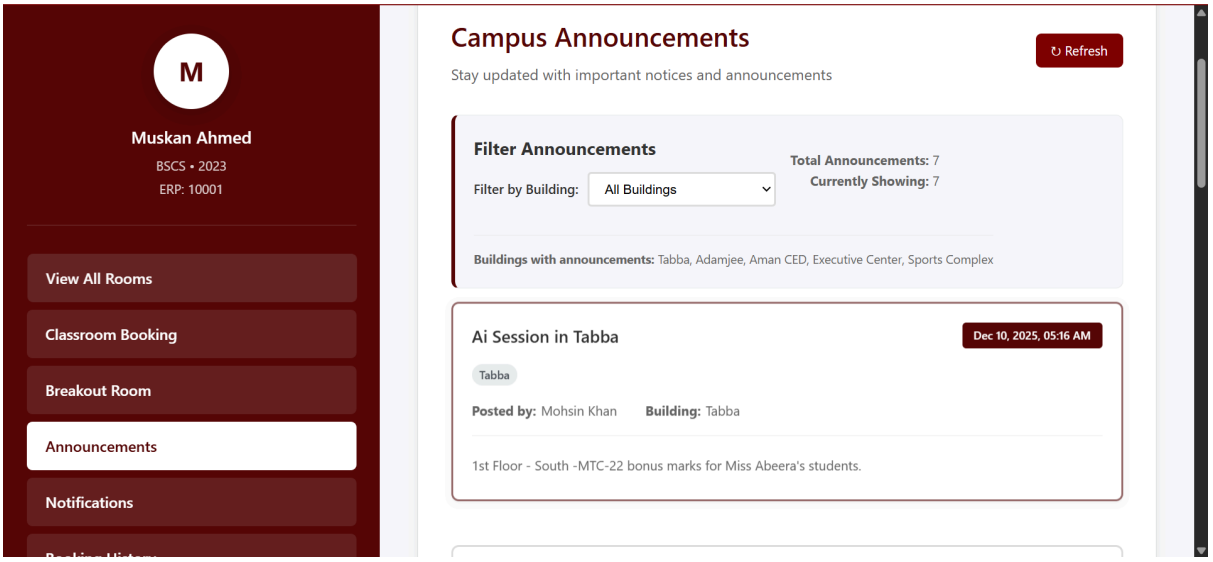
The booking gets cancelled and the reservation history is updated accordingly.



It says cancelled infront of that booking

Breakout Rooms Booking -> Same as Classrooms but the rom_type sent to the get available rooms procedure is ‘Breakout” in this case.

Announcements -> shows announcements related to the buildings.

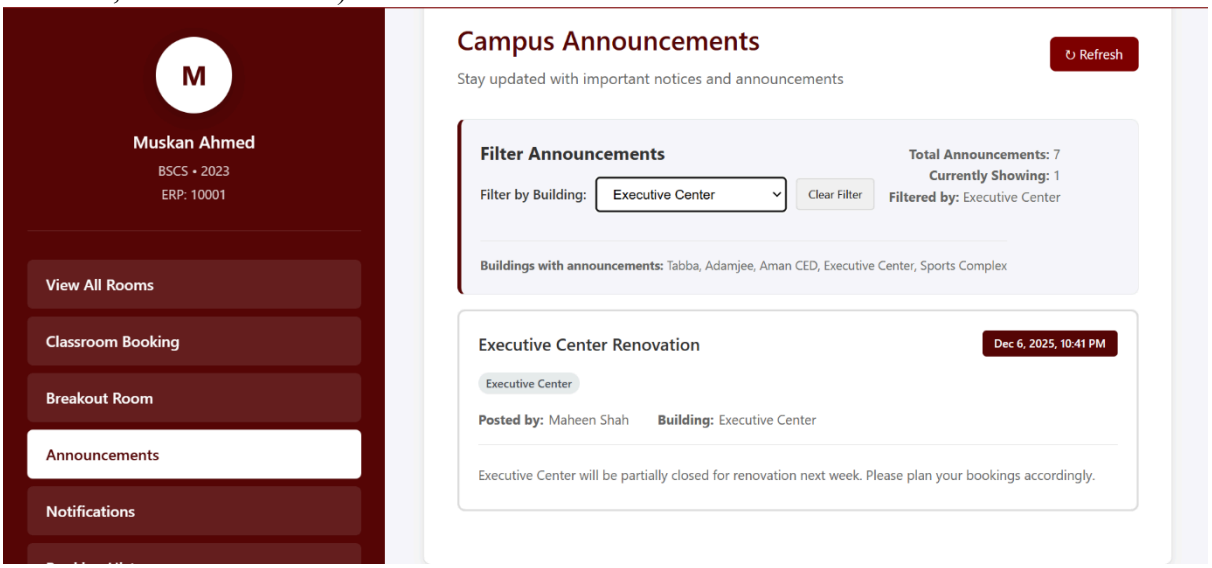


By default-> all announcements. From Backend, a relevant procedure is called.

```
CREATE OR REPLACE PROCEDURE ShowAllAnnouncements(
    p_result OUT SYS_REFCURSOR
)
AS
BEGIN
    OPEN p_result FOR
        SELECT
            a.announcement_id,
            a.title,
            a.description,
            a.date_posted,
            u.name AS posted_by,
            b.building_name
        FROM Announcement a
        JOIN User_Table u ON a.ERP = u.ERP
        LEFT JOIN Incharge i ON a.ERP = i.incharge_id
        LEFT JOIN Building b ON i.building_id = b.building_id
        ORDER BY a.date_posted DESC;
END;
/

CREATE OR REPLACE PROCEDURE FilterAnnouncementsByBuilding(
    p_building_name IN Building.building_name%TYPE,
    p_result      OUT SYS_REFCURSOR
)
AS
    v_building_exists NUMBER;
BEGIN
    -- Check building exists
    SELECT COUNT(*) INTO v_building_exists
    FROM Building
    WHERE UPPER(building_name) = UPPER(p_building_name);

    IF v_building_exists = 0 THEN
        p_result := NULL;
        RETURN;
    END IF;

    OPEN p_result FOR
        SELECT
            a.announcement_id,
            a.title,
            a.description,
            a.date_posted,
            u.name AS posted_by,
            b.building_name
        FROM Announcement a
        JOIN User_Table u ON a.ERP = u.ERP
        JOIN Incharge i ON a.ERP = i.incharge_id
        JOIN Building b ON i.building_id = b.building_id
        WHERE UPPER(b.building_name) = UPPER(p_building_name)
        ORDER BY a.date_posted DESC;

END;
/
```

The purpose of this procedure is that it shows all the announcements for all the buildings.

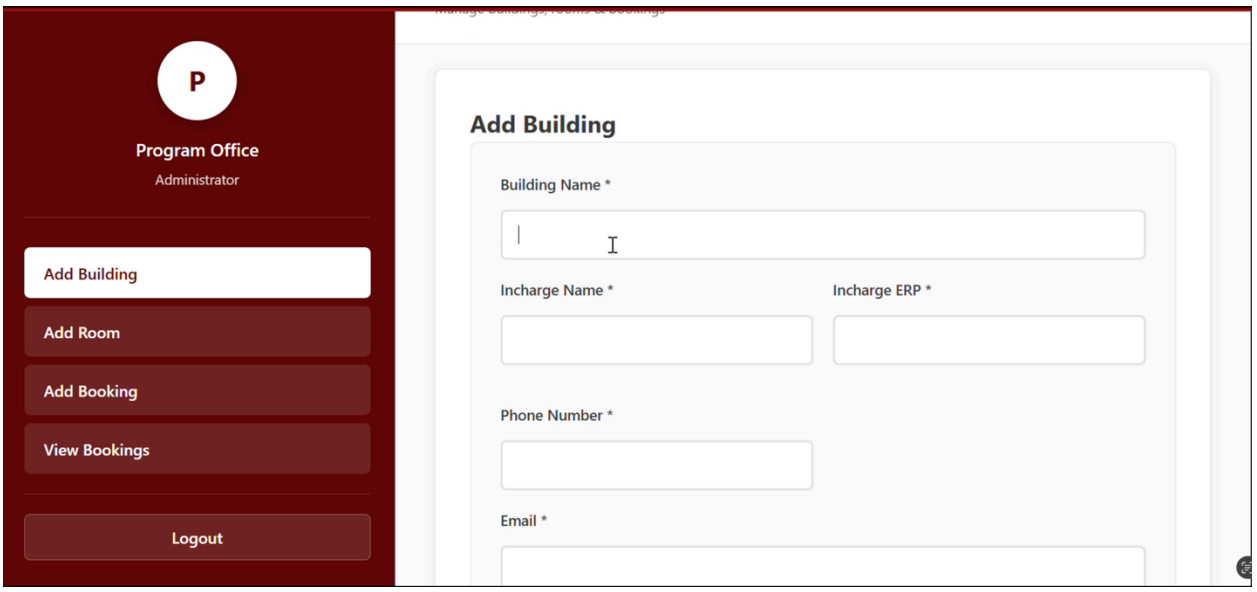
Announcements -> filter by building (in this case, executive center)



```sql
CREATE OR REPLACE PROCEDURE FilterAnnouncementsByBuilding(
    p_building_name IN Building.building_name%TYPE,
    p_result      OUT SYS_REFCURSOR
)
AS
    v_building_exists NUMBER;
BEGIN
    -- Check building exists
    SELECT COUNT(*) INTO v_building_exists
    FROM Building
    WHERE UPPER(building_name) = UPPER(p_building_name);

    IF v_building_exists = 0 THEN
        p_result := NULL;
        RETURN;
    END IF;

    OPEN p_result FOR
        SELECT
            a.announcement_id,
            a.title,
            a.description,
            a.date_posted,
            u.name AS posted_by,
            b.building_name
        FROM Announcement a
        JOIN User_Table u ON a.ERP = u.ERP
        JOIN Incharge i ON a.ERP = i.incharge_id
        JOIN Building b ON i.building_id = b.building_id
        WHERE UPPER(b.building_name) = UPPER(p_building_name)
        ORDER BY a.date_posted DESC;

END;
/
```

This procedure is called which takes the building name chosen by the user as input and outputs the announcements of that particular building.

PO POV:

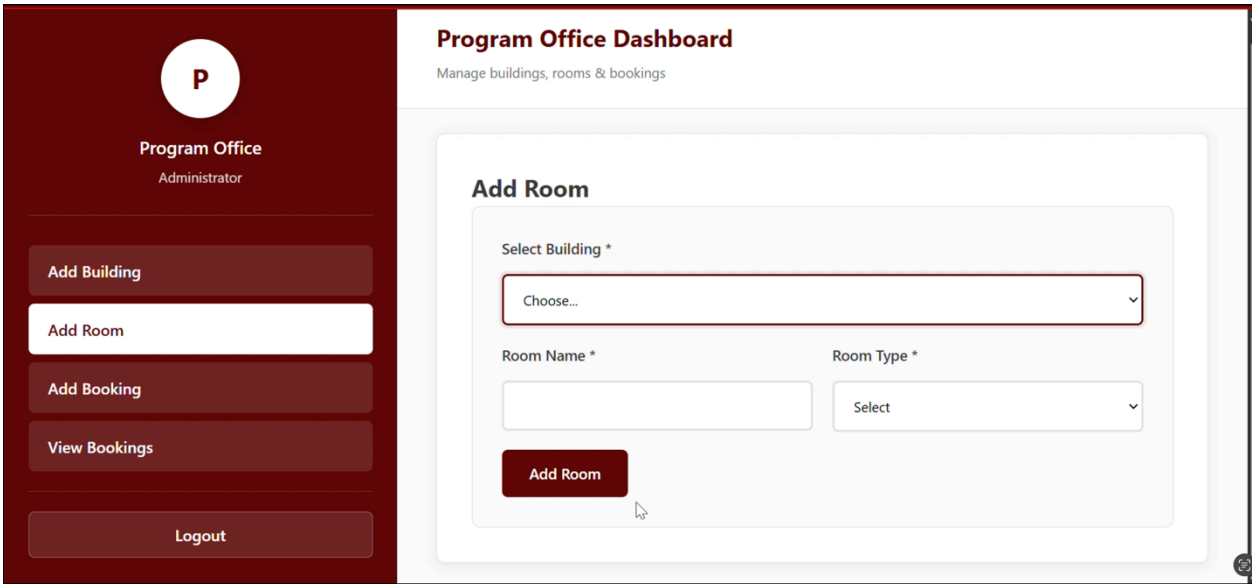

P
Program Office
Administrator
Add Building
Add Room
Add Booking
View Bookings
Logout
Add Building
Building Name *
Incharge Name *
Incharge ERP *
Phone Number *
Email *

a.sql

```sql

CREATE OR REPLACE PROCEDURE add_building(
    p_building_name IN Building.building_name%TYPE,
    p_incharge_erp IN User_Table.ERP%TYPE,
    p_incharge_name IN User_Table.name%TYPE,
    p_incharge_email IN User_Table.email%TYPE,
    p_phonenumber IN User_Table.phone_number%TYPE,
    p_result OUT VARCHAR2
)
AS
    v_building_count NUMBER;
    v_building_id Building.building_id%TYPE;
    v_user_count NUMBER;
    v_incharge_count NUMBER;
BEGIN
    -- Step 1: Check if building already exists by name
    SELECT COUNT(*) INTO v_building_count
    FROM Building
    WHERE UPPER(building_name) = UPPER(p_building_name);

    -- If building exists, return error message
    IF v_building_count > 0 THEN
        p_result := 'Building "' || p_building_name || '" already exists';
        RETURN;
    END IF;

    -- only IBA email allowed
    IF NOT (p_incharge_email LIKE '%@iba.edu.pk') THEN
        p_result := 'Only IBA incharge emails (@iba.edu.pk) allowed';
        RETURN;
    END IF;

    -- Step 2: Check if incharge ERP exists in User_Table
    SELECT COUNT(*) INTO v_user_count
    FROM User_Table
    WHERE ERP = p_incharge_erp;

    -- Step 3: Check if incharge is already assigned to another building
    SELECT COUNT(*) INTO v_incharge_count
    FROM Incharge
    WHERE incharge_id = p_incharge_erp;

    -- If incharge is already assigned to another building, return error
    IF v_incharge_count > 0 THEN
        p_result := 'Incharge with ERP ' || p_incharge_erp || ' is already assigned to another building';
        RETURN;
    END IF;

    -- Step 4: If incharge doesn't exist in User_Table, add them
    IF v_user_count = 0 THEN
        -- Use consistent role name from your existing code ('BuildingIncharge')
        INSERT INTO User_Table (ERP, name, email, user_password, role, phone_number)
        VALUES (p_incharge_erp, p_incharge_name, p_incharge_email, 'default_password', 'BuildingIncharge', p_phonenumber);
    END IF;

    -- Step 5: Insert the new building (ID will auto-generate)
    INSERT INTO Building (building_name)
    VALUES (p_building_name)
    RETURNING building_id INTO v_building_id;

    -- Step 6: Assign incharge to the building
    INSERT INTO Incharge (incharge_id, building_id)
    VALUES (p_incharge_erp, v_building_id);

    -- Step 7: Set success message with building ID
    p_result := 'Building "' || p_building_name || '" added successfully with ID: ' || v_building_id || '. Incharge assigned.';

    -- Commit the transaction
    COMMIT;

EXCEPTION
    WHEN OTHERS THEN
        -- Handle any exceptions and return error message
        p_result := 'Error adding building: ' || SQLERRM;
        ROLLBACK;
END add_building;
/
```

Purpose: The procedure **`add_building`** is designed to **add a new building to the system** and simultaneously **assign a building incharge**. It performs multiple checks to ensure data integrity, avoids duplicates, and maintains proper relationships between users and buildings.

It also provides **feedback through the `p_result` output parameter** about success or any error encountered.

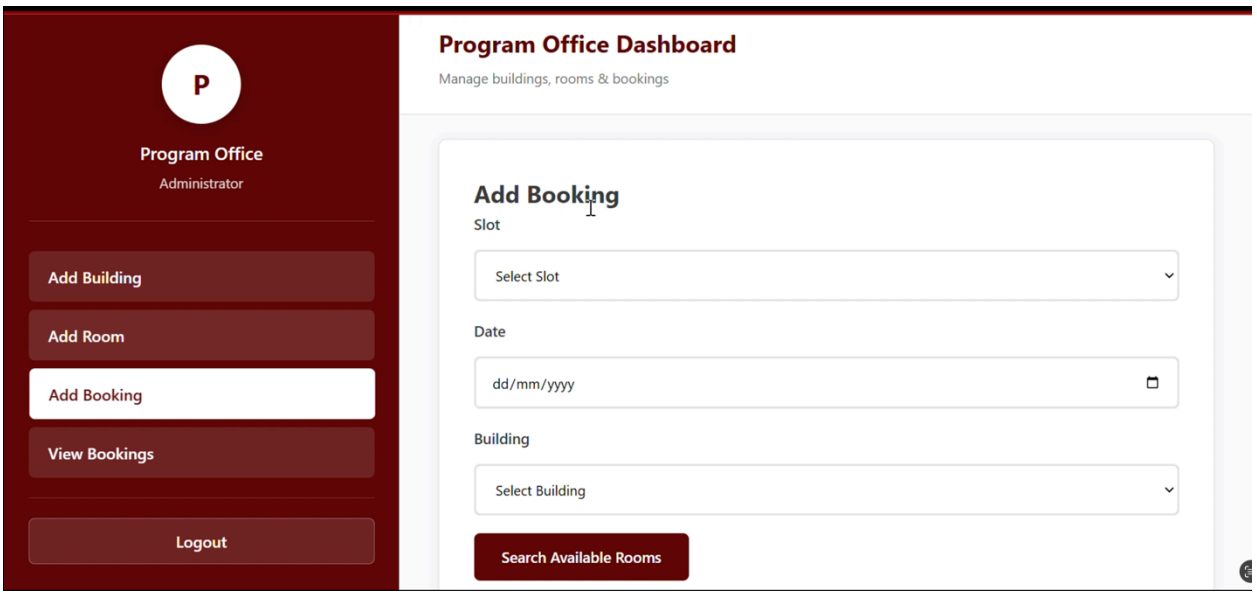


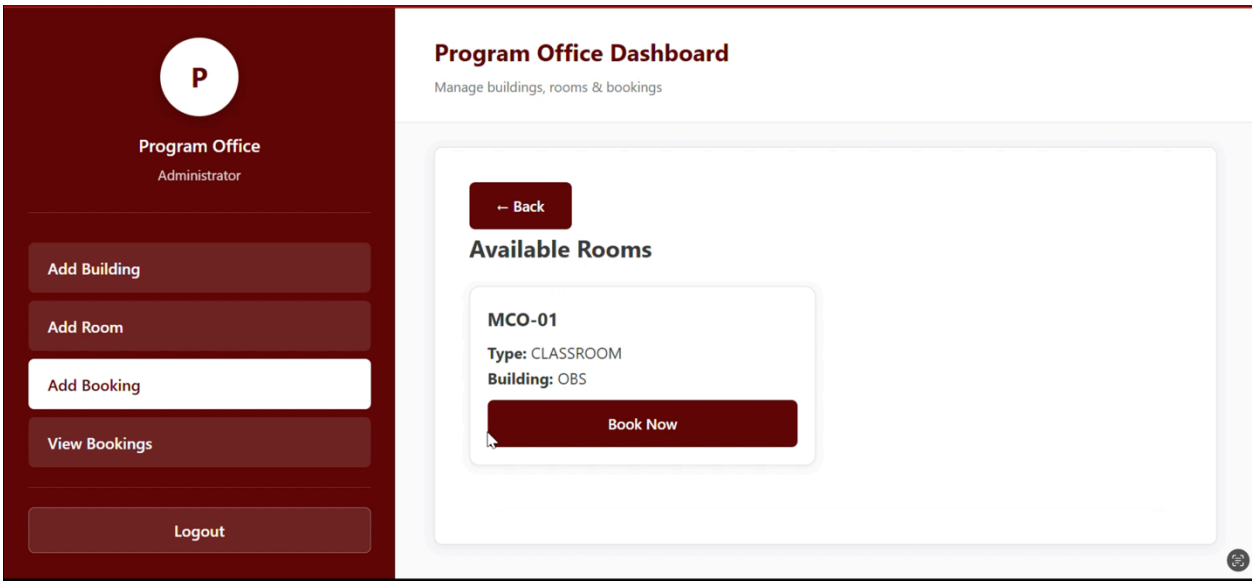
```sql
CREATE OR REPLACE PROCEDURE add_room(
    p_building_name IN Building.building_name%TYPE,
    p_room_name IN Room.room_name%TYPE,
    p_room_type IN Room.room_type%TYPE,
    p_result OUT VARCHAR2
)
AS
    v_building_count NUMBER;
    v_room_count NUMBER;
    v_building_id Building.building_id%TYPE;
    v_room_id Room.room_id%TYPE;
BEGIN
    -- Step 1: Check if building exists
    BEGIN
        SELECT building_id INTO v_building_id
        FROM Building
        WHERE UPPER(building_name) = UPPER(p_building_name);

        v_building_count := 1;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            v_building_count := 0;
    END;

    -- If building doesn't exist, return error message
    IF v_building_count = 0 THEN
        p_result := 'Building "' || p_building_name || '" does not exist. Please add building first.';
        RETURN;
    END IF;

    -- Step 2: Check if room already exists in this building
    SELECT COUNT(*) INTO v_room_count
    FROM Room
    WHERE UPPER(room_name) = UPPER(p_room_name)
    AND building_id = v_building_id;

    -- If room exists, return error message
    IF v_room_count > 0 THEN
        p_result := 'Room "' || p_room_name || '" already exists in building "' || p_building_name || '"';
        RETURN;
    END IF;

    -- Step 3: Insert the new room (ID will auto-generate)
    INSERT INTO Room (building_id, room_name, room_type)
    VALUES (v_building_id, p_room_name, p_room_type)
    RETURNING room_id INTO v_room_id;

    -- Step 4: Set success message
    p_result := 'Room "' || p_room_name || '" added successfully to building "' || p_building_name || '" with ID: ' || v_room_id;

    -- Commit the transaction
    COMMIT;

EXCEPTION
    WHEN OTHERS THEN
        -- Handle any exceptions and return error message
        p_result := 'Error adding room: ' || SQLERRM;
        ROLLBACK;
END add_room;
/
```

The **`add_room`** procedure is designed to **add a new room to a specific building** in your system. It ensures that:

1. The building exists before adding a room.
2. The room name is **unique within the building**.
3. A clear message is returned indicating success or any error.

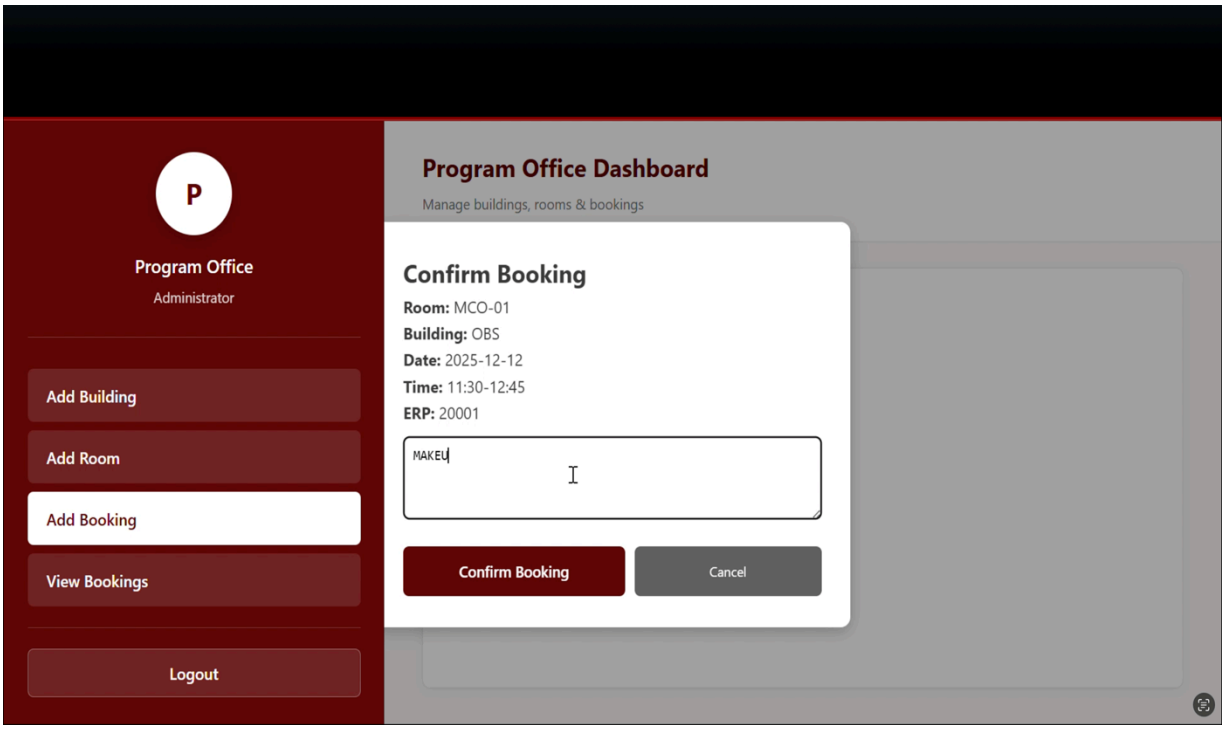
Essentially, it **automates room creation while performing all necessary validation checks**.

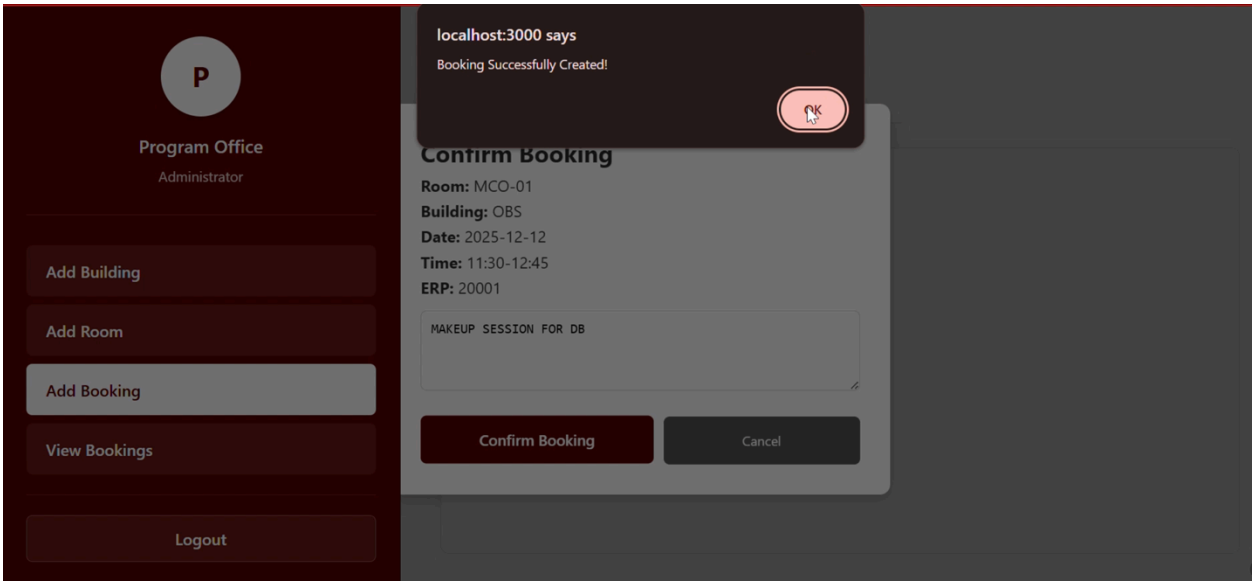




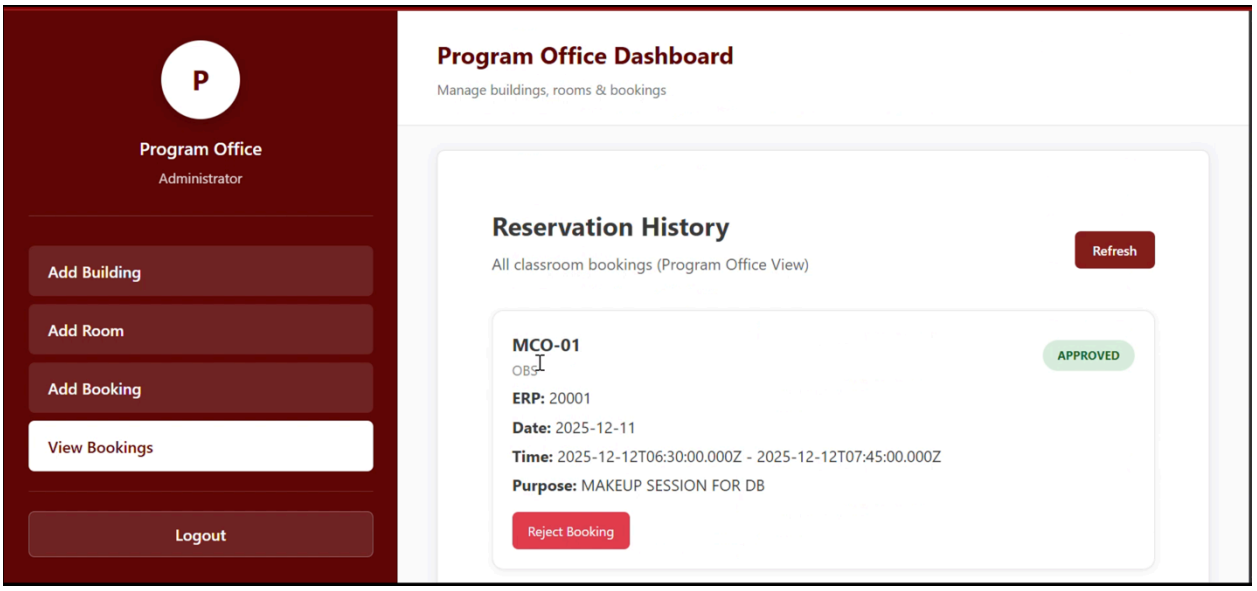
The sql query remains the same as in Student Dashboard (call procedure get_available_rooms) as soon as Search Available room is clicked. As a result, the purpose also remains the same.

The sql query remains the same as in Student Dashboard(call procedure create_booking) as soon as booking is confirmed. As a result, the purpose also remains the same.





The sql query remains the same as in Student Dashboard(call procedure create_booking) as soon as booking is confirmed. As a result, the purpose also remains the same.

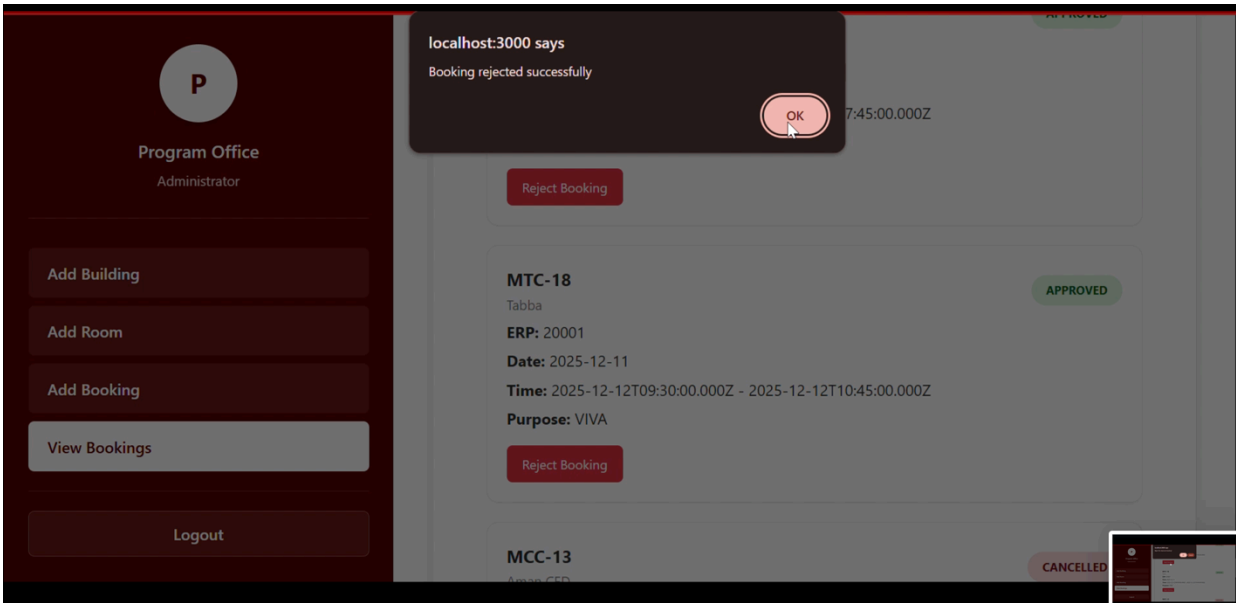


```sql
CREATE OR REPLACE PROCEDURE ShowReservationHistoryForPO(
    p_result OUT SYS_REFCURSOR
)
AS
BEGIN
    /*
    PURPOSE: Show all classroom bookings for Program Office
    CALLED BY: Program Office admin
    */

    OPEN p_result FOR
        SELECT
            b.booking_id,
            b.ERP,
            u.name AS student_name,
            r.room_name,
            bld.building_name,
            b.booking_date,
            b.start_time,
            b.end_time,
            b.purpose,
            b.status,
            b.created_date
        FROM Booking b
        JOIN Room r ON b.room_id = r.room_id
        JOIN Building bld ON r.building_id = bld.building_id
        JOIN User_Table u ON b.ERP = u.ERP
        WHERE r.room_type = 'CLASSROOM'
        ORDER BY b.booking_date DESC, b.start_time DESC;
END;
/
```

As soon as PO opens this page, backend fetches all approved reservations.

Purpose: The `ShowReservationHistoryForPO` procedure is designed to **allow the Program Office (PO) to view the complete history of classroom bookings**.

- It **retrieves all bookings for rooms of type `CLASSROOM`**.
- It provides detailed information about each booking, including **student details, room and building names, booking times, purpose, status, and creation date**.
- This procedure is typically called by **Program Office admins** for monitoring or auditing purposes.


localhost:3000 says
Booking rejected successfully
OK
P
Program Office
Administrator
Add Building
Add Room
Add Booking
View Bookings
Logout
7:45:00.000Z
Reject Booking
MTC-18
Tabba
APPROVED
ERP: 20001
Date: 2025-12-11
Time: 2025-12-12T09:30:00.000Z - 2025-12-12T10:45:00.000Z
Purpose: VIVA
Reject Booking
MCC-13
CANCELLED

```sql
CREATE OR REPLACE PROCEDURE RejectBookingByPO(
    p_booking_id IN Booking.booking_id%TYPE,
    p_success    OUT NUMBER,
    p_message    OUT VARCHAR2
)
AS
    v_room_type Room.room_type%TYPE;
    v_status    Booking.status%TYPE;
BEGIN
    /*
    PURPOSE: PO can reject classroom bookings
    RULES: Only 'Approved' classrooms can be rejected
    */

    p_success := 0;
    p_message := '';

    -- Get booking details
    SELECT r.room_type, b.status
    INTO v_room_type, v_status
    FROM Booking b
    JOIN Room r ON b.room_id = r.room_id
    WHERE b.booking_id = p_booking_id;

    -- Check it's a classroom
    IF v_room_type != 'CLASSROOM' THEN
        p_message := 'Only classroom bookings can be rejected by PO';
        RETURN;
    END IF;

    -- Check status is 'Approved'
    IF v_status != 'Approved' THEN
        p_message := 'Only Approved bookings can be rejected';
        RETURN;
    END IF;

    -- Reject the booking
    UPDATE Booking
    SET status = 'Rejected'
    WHERE booking_id = p_booking_id;

    COMMIT;

    p_success := 1;
    p_message := 'Booking rejected successfully';

EXCEPTION
    WHEN NO_DATA_FOUND THEN
        p_message := 'Booking not found';
    WHEN OTHERS THEN
        ROLLBACK;
        p_message := 'Error rejecting booking: ' || SQLERRM;
END;
```

```
CREATE OR REPLACE TRIGGER trg_booking_status_change
BEFORE UPDATE OF status ON Booking
FOR EACH ROW
BEGIN
    -- Log status changes (optional)
    IF :OLD.status != :NEW.status THEN
        DBMS_OUTPUT.PUT_LINE('Booking ' || :OLD.booking_id ||
            ' changed from ' || :OLD.status || ' to ' || :NEW.status);
    END IF;
END;
/
```

As soon as PO presses Reject Booking, after confirmation, backend fetches RejectBookingByPO.

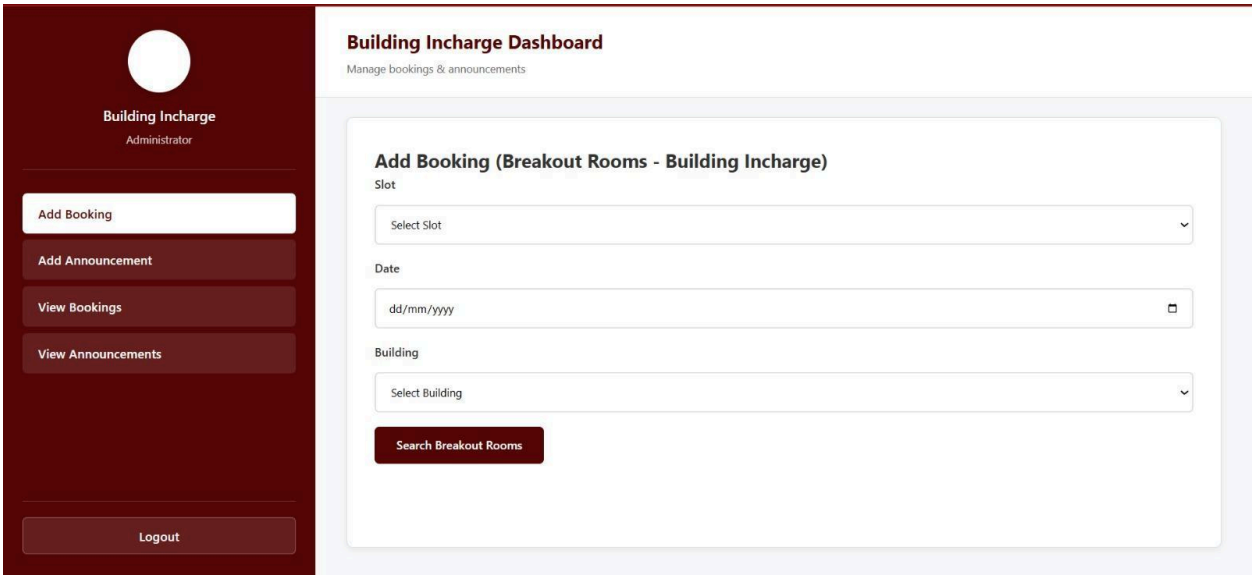
Purpose of the procedure: The **`RejectBookingByPO`** procedure allows the **Program Office (PO) to reject classroom bookings** under specific rules:

1. Only bookings for **classrooms** can be rejected by the PO.
2. Only bookings with **status 'Approved'** can be rejected.
3. Updates the booking status to **'Rejected'** if the rules are satisfied.
4. Returns **feedback** through two output parameters:
    - `p_success`: Indicates whether the rejection was successful (`1`) or failed (`0`).
    - `p_message`: Provides a descriptive message explaining the outcome.

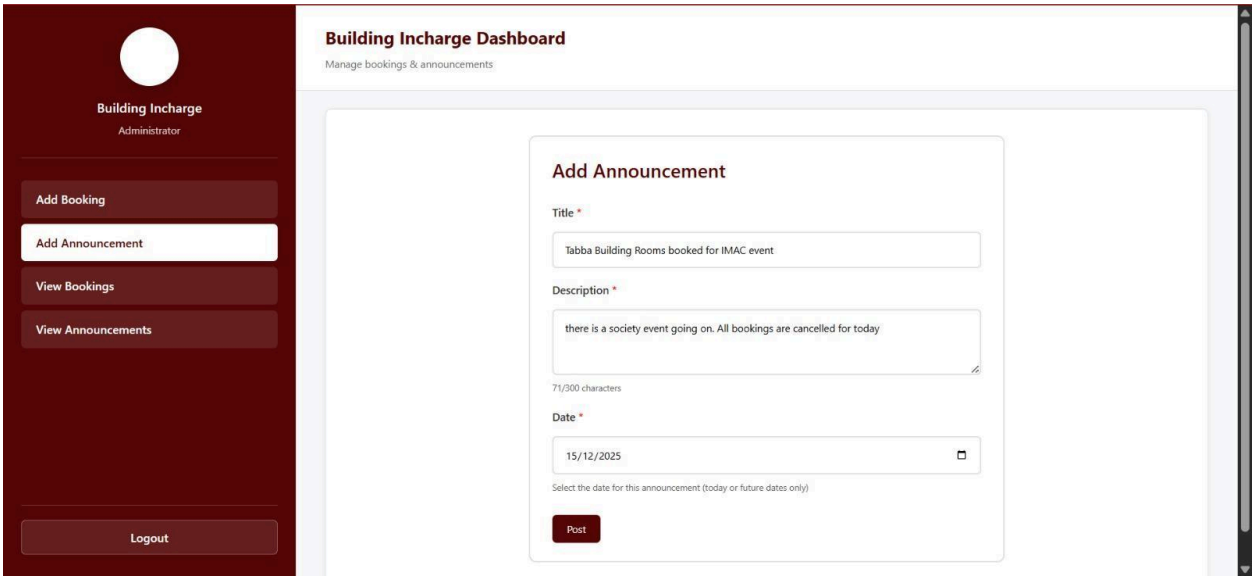
Purpose of the trigger: The **`trg_booking_status_change`** trigger is designed to **monitor and log changes to the `status` column** in the `Booking`table.

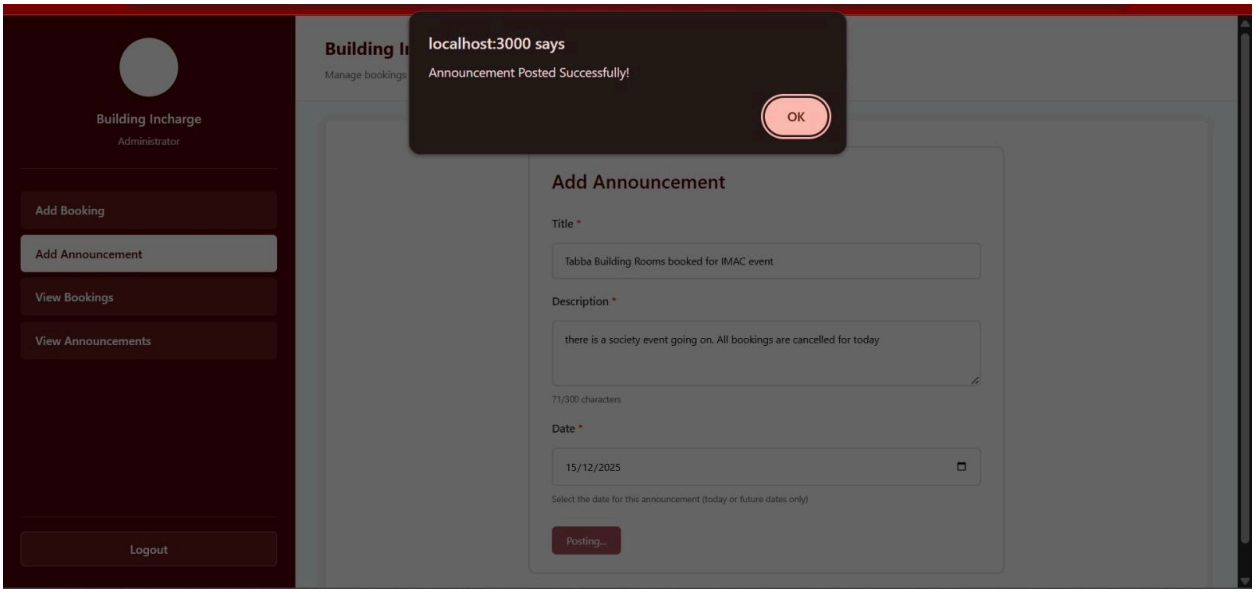
- It fires **before any update** to the `status` field of a booking.
- Its main purpose is **tracking changes in booking status**, such as `'Pending'` → `'Approved'` or `'Approved'` → `'Rejected'`.

BI POV:



Search Breakout Rooms calls the same procedure as in Student Dashboard (get_available_rooms, room_type='Breakout'). Thus, the purpose also remains the same.

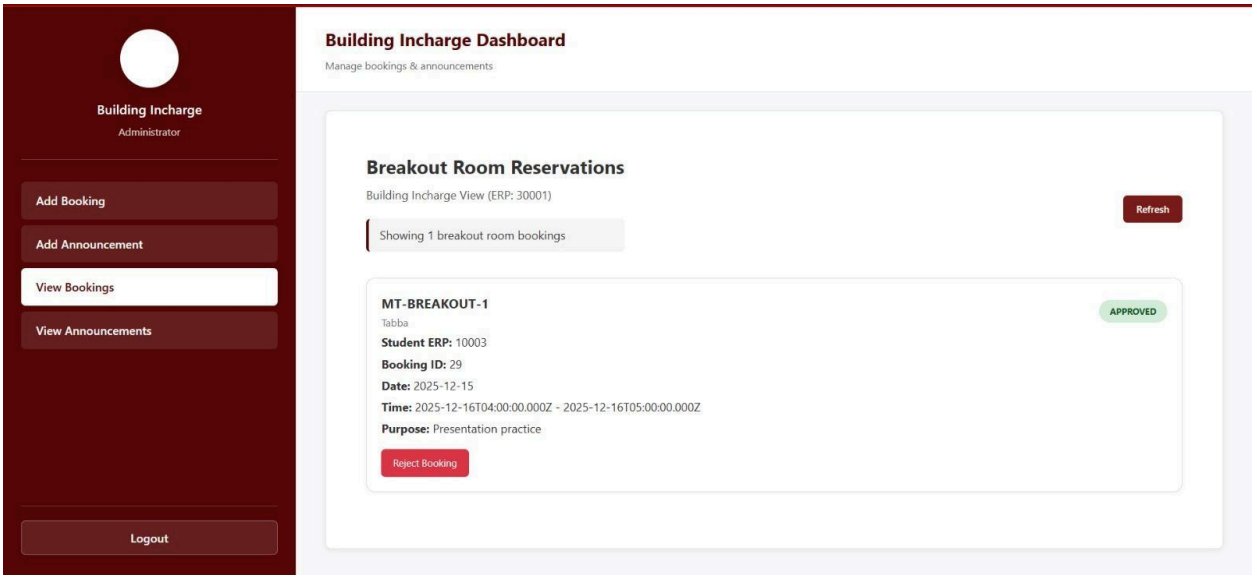


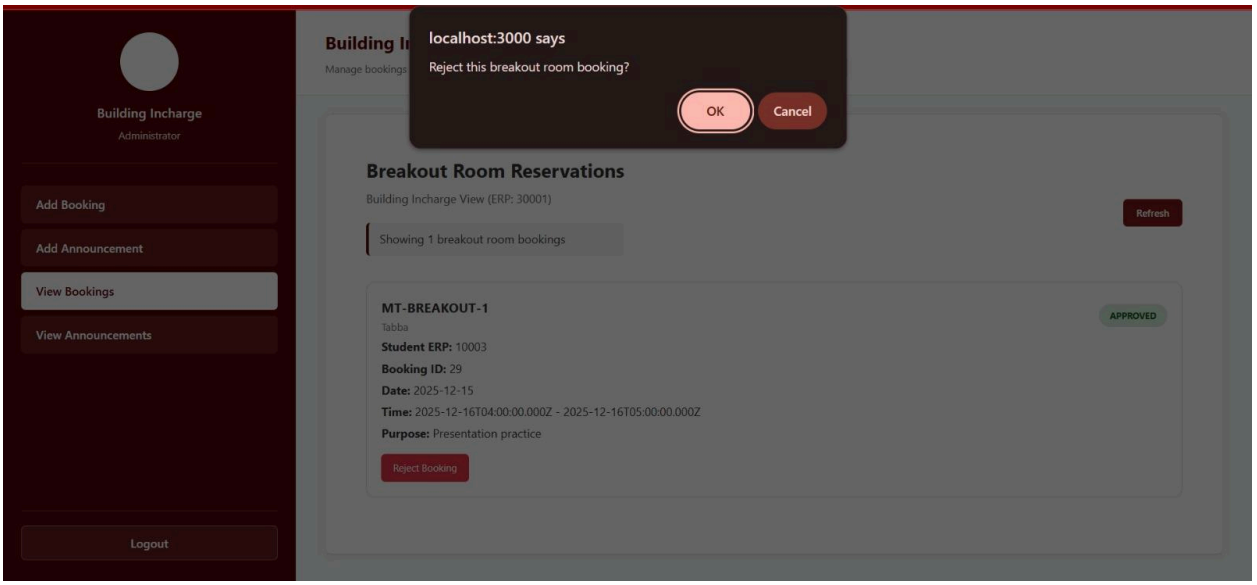

Building Incharge
Administrator
Add Booking
Add Announcement
View Bookings
View Announcements
Logout
Building I
Manage bookings
localhost:3000 says
Announcement Posted Successfully!
OK
Add Announcement
Title *
Tabba Building Rooms booked for IMAC event
Description *
there is a society event going on. All bookings are cancelled for today
71/300 characters
Date *
15/12/2025
Select the date for this announcement (today or future dates only)
Posting...

```sql
CREATE OR REPLACE PROCEDURE PostAnnouncement(
    p_erp          IN Announcement.ERP%TYPE,
    p_title        IN Announcement.title%TYPE,
    p_description  IN Announcement.description%TYPE,
    p_success      OUT NUMBER,
    p_message      OUT VARCHAR2
)
AS
    v_role User_Table.role%TYPE;
    v_incharge_exists NUMBER;
BEGIN
    /*
    PURPOSE: Building Incharge posts announcement for their building
    RULES: Only BuildingIncharge role can post announcements
    */

    p_success := 0;
    p_message := '';

    -- Check user is BuildingIncharge
    SELECT role INTO v_role
    FROM User_Table
    WHERE ERP = p_erp;

    IF v_role != 'BuildingIncharge' THEN
        p_message := 'Only Building Incharges can post announcements';
        RETURN;
    END IF;

    -- Check incharge is assigned to a building
    SELECT COUNT(*) INTO v_incharge_exists
    FROM Incharge
    WHERE incharge_id = p_erp;

    IF v_incharge_exists = 0 THEN
        p_message := 'Building Incharge not assigned to any building';
        RETURN;
    END IF;

    -- Insert announcement
    INSERT INTO Announcement (
        ERP, title, description, date_posted
    ) VALUES (
        p_erp, p_title, p_description, SYSTIMESTAMP
    );

    COMMIT;

    p_success := 1;
    p_message := 'Announcement posted successfully';

EXCEPTION
    WHEN OTHERS THEN
        ROLLBACK;
        p_success := 0;
        p_message := 'Error posting announcement: ' || SQLERRM;
END;
/
```

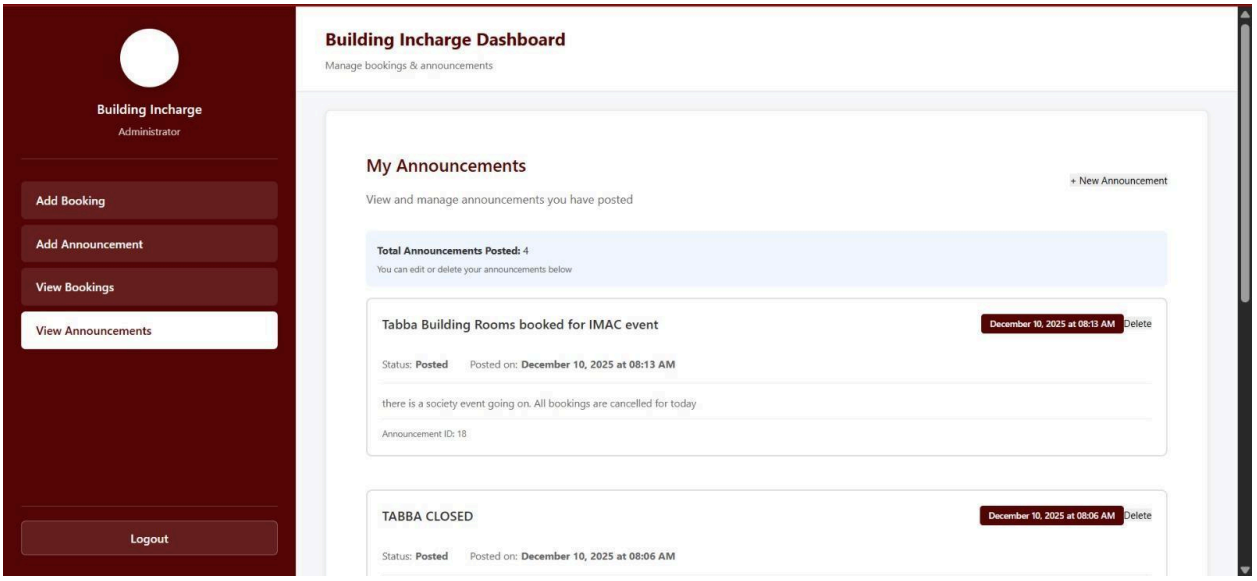
Purpose: The `PostAnnouncement` procedure allows a **Building Incharge to post announcements** for their assigned building.

- Ensures that **only users with the `BuildingIncharge` role** can post announcements.
- Validates that the incharge is **actually assigned to a building**.
- Inserts the announcement into the `Announcement` table with a timestamp.





The SQL query would be the same as PO's. for reservations. However, room_type will be 'Breakout' and incharge_id will be 'p_incharge_id'. As a result, the purpose also stays the same.


Building Incharge
Administrator
Add Booking
Add Announcement
View Bookings
View Announcements
Logout
Building Incharge Dashboard
Manage bookings & announcements
My Announcements
View and manage announcements you have posted
+ New Announcement
Total Announcements Posted: 4
You can edit or delete your announcements below
Tabba Building Rooms booked for IMAC event
December 10, 2025 at 08:13 AM
Delete
Status: Posted
Posted on: December 10, 2025 at 08:13 AM
there is a society event going on. All bookings are cancelled for today
Announcement ID: 18
TABBA CLOSED
December 10, 2025 at 08:06 AM
Delete
Status: Posted
Posted on: December 10, 2025 at 08:06 AM

```
CREATE OR REPLACE PROCEDURE ShowAnnouncementsByUser(
    p_erp     IN Announcement.ERP%TYPE,
    p_result  OUT SYS_REFCURSOR
)
AS
BEGIN
    /*
    PURPOSE: Show announcements posted by specific user
    CALLED BY: User to see their own announcements
    */

    OPEN p_result FOR
        SELECT
            announcement_id,
            title,
            description,
            date_posted,
            created_date
        FROM Announcement
        WHERE ERP = p_erp
        ORDER BY date_posted DESC;
END;
/
```

Purpose:

The **`ShowAnnouncementsByUser`** procedure is designed to **retrieve all announcements posted by a specific Building Incharge**.

- Typically called by the Building Incharge themselves to **view their own announcements**.
- Orders the announcements by **`date_posted` in descending order**, so the most recent announcements appear first.

Work Contribution

Module 1 — Student Portal & Room Booking Engine

Assigned to: Muskan

Scope

This module covers all features related to the student user experience and the core room booking functionality.

Frontend Components

studentdashboard.js

viewallrooms.js

classroombooking.js

breakoutbooking.js

bookinghistory.js

announcement.js (student view)

notifications.js

Backend Components

bookings.js

rooms.js

reservations.js

announcement.js (student-side retrieval)

Database Procedures

Booking creation and cancellation

Room availability checking

Student notification retrieval

Room listing procedures

Student announcements retrieval

Additional Contributions

Defined business rules for booking limits, time-slot logic, conflict resolution, and availability management.

Module 2 — Program Office (PO) Portal & Scheduling Management

Assigned to: Abdullah

Scope

This module manages academic scheduling processes and Program Office administrative functionalities.

Frontend Components

ProgramOffice.js

Backend Components

Schedule management endpoints

Administrative features in bookings.js

Announcement creation and management (PO privileges)

Database Procedures

Schedule creation and editing

Booking approval and oversight

Announcement posting

Reservation overrides for academic events

Timetable and booking statistics retrieval

Additional Contributions

Conducted stakeholder interviews with the Program Office.

Drafted business rules for scheduling, room prioritization, PO approval flow, and override mechanisms.

Module 3 — Building Incharge System & Authentication Management

Assigned to: Kashish

Scope

This module is responsible for building management operations and all authentication-related functionalities, including user access and security.

Frontend Components

BuildingINcharge.js

LoginForm.js

ForgotPassword.js

UserTypeSelector.js

Backend Components

building.js

auth.js

User and role administration endpoints

Database Procedures

Building management operations

Room maintenance status updates

User account management (CRUD)

Role-based access configurations

Password reset logs and associated security controls

Additional Contributions

Designed ERD sections related to user, role, and building entities.

Developed authentication flow including JWT integration, authorization rules, and reset mechanisms.