DBMS Mini-Project Report: Apartment Management System

1. Title of the Problem Statement with Team Details

- **Title:** Apartment Management System
- **Team Details:** TARUN S - PES1UG23AM919
  ADITYAA KUMAR H - PES1UG23AM025

---

2. Description about the Statement (Short Abstract)

The Apartment Management System is a comprehensive, multi-user web application designed to digitize and streamline the management of a residential apartment complex. It provides a centralized platform for administrators, apartment owners, tenants, and employees to interact and manage all aspects of apartment living. The system handles resident and staff data, financial records (maintenance and rent), complaint logging and resolution, visitor management with an approval workflow, parking allocation, and community engagement features such as event listings and amenities information. The application ensures seamless data access and manipulation through a relational database backend, with distinct dashboards and functionalities tailored to each user role.

---

3. User Requirement Specification (URS)

As per the project guidelines, the URS is as follows:

- Purpose of the Project
  The purpose of this project is to design and build a deployable, standalone web application that works seamlessly with a relational database. The system aims to replace manual record-keeping and disparate systems with a single, efficient, and centralized database application for managing an apartment complex. It facilitates core database operations (Create, Read, Update, Delete) for all apartment-related data.
- Scope of the Project
  The application serves four primary user roles: Administrator, Owner, Tenant, and Employee. The scope includes:
  - **Data Management:** Managing database records for apartment blocks, individual rooms, owners, tenants, and employees.
  - **Authentication:** Secure user login and role-based access control.
  - **Complaint System:** A full workflow for residents (Owners/Tenants) to raise complaints and for administrators to view and manage them.
  - **Financials:** Tracking maintenance payments and managing lease agreements, including monthly rent.
  - **Visitor Management:** A secure visitor tracking system that includes a

    request-and-approval workflow.
  - **Resource Management:** Allocation of resources like parking slots.
  - **Community Hub:** Providing information on community events, amenities, and local service providers.
- Detailed Description (Case Study)
  The project models a residential apartment complex. The complex is divided into one or more blocks, each with a name and number. Each block contains multiple rooms (apartments), which are identified by a room number, type (e.g., 2BHK, 3BHK), and floor. Each room has a designated parking_slot.
  The apartments are occupied by owners or rented to tenants. The system must store personal details for both, such as name, age, and proof of identity (via an identity table). For tenants, the system must also track their rental details and manage lease_agreements.
  The complex is managed by block_admins and serviced by employees (like security, janitors), whose details and salary are stored.
  A core part of the system is managing daily operations. This includes:
  1. **Complaints:** Residents must be able to file complaints associated with their room, which admins can then view and track.
  2. **Maintenance:** The system must track monthly maintenance fees for each apartment, including amount, due date, and payment status.
  3. **Visitors:** A robust visitors system is required. A resident (owner or tenant) must request entry for a visitor. This request goes into a "Pending" state and must be "Approved" by an admin before the visitor is allowed "Inside".
  4. **Community:** The system also stores information on community_events, available amenities, and trusted service_providers.

All users (admins, owners, tenants) must log in via a unified auth table that stores user credentials and links to their respective ID in the owner, tenant, or block_admin tables.

---

4. Functional Requirements

Based on the application's client-side routing and API routes, the following system functionalities are provided:

- **System Functionality 1: User Authentication**
  - **Description:** All users must log in through a secure authentication portal. The system validates their user_id and password via an API call to the /login endpoint.
  - **Deliverable:** client/src/components/Auth.jsx (Frontend), / route (Frontend), /login route (Backend).
- **System Functionality 2: Role-Based Dashboards**
  - **Description:** After logging in, each user type is redirected to the /dashboard route. The Dashboard component fetches role-specific data (e.g., total owners for admins, salary for employees) by calling API endpoints like /dashb/admin, /dashb/owner, /dashb/employee, and /dashb/tenant.

    - **Deliverable:** client/src/components/Dashboard.jsx.
- **System Functionality 3: Complaint Management**
    - **Description:** Owners and tenants can submit complaints via a dedicated form (route /raisingcomplaints). Admins have a separate view (/complaintsviewer) to see all complaints. Owners have a restricted view (/complaintsviewerowner) to see only their own submitted complaints.
    - **Deliverable:**
        - **Raising:** client/src/components/RaisingComplaints.jsx (Frontend), /dashb/raisingcomplaint (Backend).
        - **Admin View:** client/src/components/ComplaintsViewer.jsx (Frontend), /dashb/viewcomplaints (Backend).
        - **Owner View:** client/src/components/ComplaintsViewerOwner.jsx (Frontend), /dashb/ownercomplaints (Backend).
- **System Functionality 4: User Management (Admin)**
    - **Description:** The administrator has forms to create new owners, tenants, and employees. These forms make API calls to the /create route to insert new records into the database.
    - **Deliverable:** client/src/components/CreatingOwner.jsx (Frontend), client/src/components/CreatingTenant.jsx (Frontend), client/src/components/CreatingEmployee.jsx (Frontend), /create/owner, /create/tenant, /create/employee (Backend).
- **System Functionality 5: Resource & Details Management**
    - **Description:** The system allows for viewing and updating various details. Admins can view all OwnerDetails, TenantDetails, and RoomDetails. Owners can view their specific RoomDetailsOwner and update their ParkingSlot.
    - **Deliverable:**
        - **Parking Allocation:** client/src/components/ParkingSlot.jsx (Frontend), /dashb/updateParkingSlot (Backend).
        - **Viewers:** client/src/components/OwnerDetails.jsx, client/src/components/TenantDetails.jsx, client/src/components/RoomDetails.jsx.
- **System Functionality 6: Visitor Management (with Approval Workflow)**
    - **Description:** Residents can request visitor entry via the /visitors route. The system logs this request with a "Pending" status in the visitors table. An admin must approve the request (setting status to "Approved"). The system is designed to track if a visitor is "Inside" or has "Exited".
    - **Deliverable:** client/src/components/Visitors.jsx, visitors (table), GetPendingVisitorRequests (procedure).
- **System Functionality 7: Lease and Maintenance Management**
    - **Description:** The system provides modules for managing finances. Admins can manage LeaseAgreements. Residents can view their Maintenance status and PayMaintenance.
    - **Deliverable:** client/src/components/LeaseAgreements.jsx, client/src/components/PayMaintenance.jsx, client/src/components/Maintenance.jsx.

- **System Functionality 8: Community Hub**
  - **Description:** Users can browse lists of available amenities (e.g., gym, pool) via the /amenities route, community events via /communityevents, and recommended service providers via /serviceproviders. Users can also submit feedback via the /feedback route.
  - **Deliverable:** client/src/components/Amenities.jsx, client/src/components/CommunityEvents.jsx, client/src/components/ServiceProviders.jsx, client/src/components/Feedback.jsx.

---

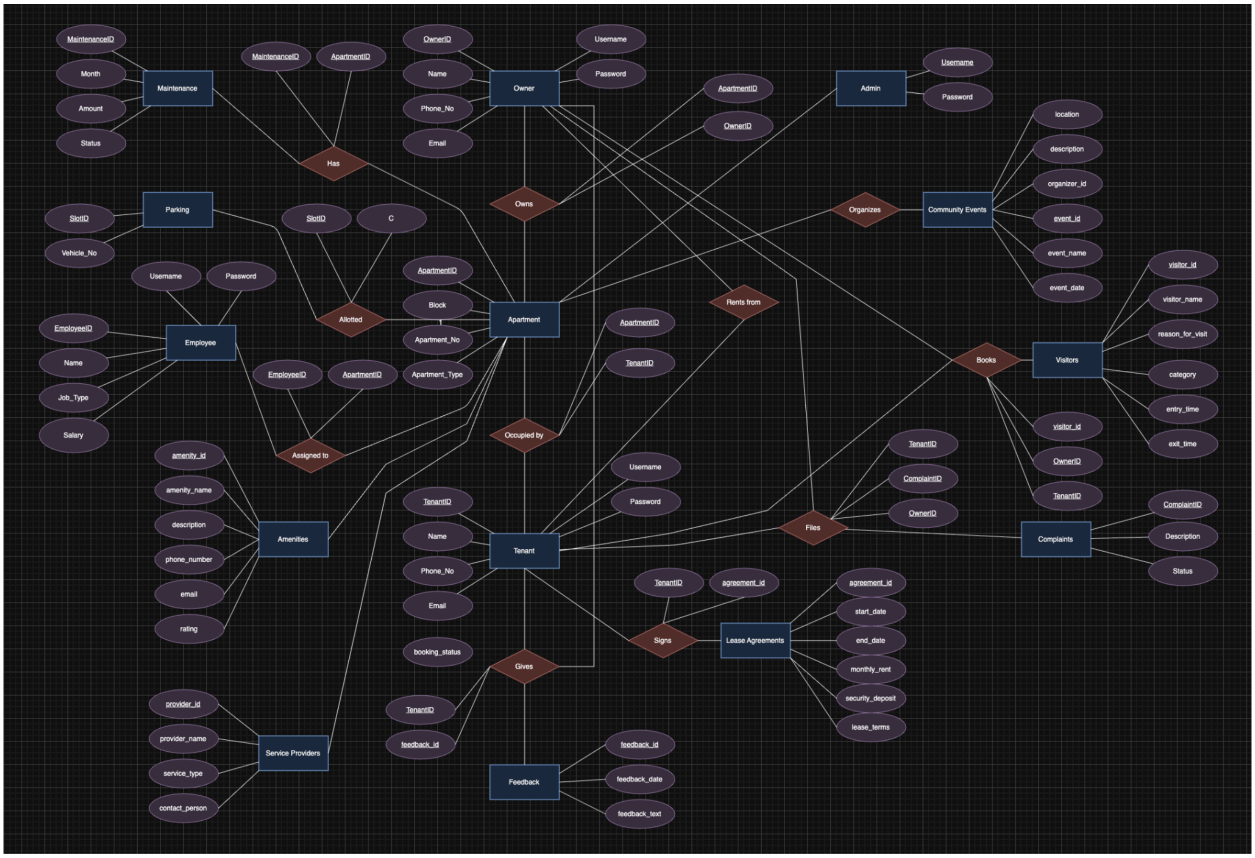
5. List of Softwares/Tools/Programming Languages Used

- **Backend Runtime:** Node.js.
- **Backend Framework:** Express.js.
- **Backend API Structure:** A RESTful API server running on http://localhost:3001. It uses express.json() for parsing JSON bodies and cors to allow cross-origin requests from the client. The API routes are modularized into /login, /create, and /dashb.
- **Database:** MySQL.
- **Database Driver:** mysql2 NPM package.
- **Database Configuration:** The server loads database credentials (host, user, password, database name) from a separate config_sql.js file.
- **Frontend Library:** React.js v18.
- **Frontend Routing:** react-router-dom v6 is used for client-side routing, defining all application paths in App.jsx.
- **Frontend HTTP Client:** axios is used to make all API calls from the client to the backend server.
- **Frontend Styling:** TailwindCSS is used for utility-first styling.
- **Frontend State Management:** React Context API is used for simple global state (managing the sidebar's open/closed state).
- **Development Tools:** nodemon for automatic server reloading and react-scripts (Create React App) for frontend development.

---

6. ER Diagram

The Entity-Relationship (ER) diagram for the database is provided in the project assets file: assets/er-diagram.png.



7. Relational Schema

The relational schema diagram is provided in assets/schema.png.

The database is named apartment_management. The final schema consists of the following 17 tables.

7.1. Entity (Table) Descriptions

Here is a description of each entity (table) and its purpose in the system:

1. **block**: This is a core entity representing a physical block or wing of the apartment complex (e.g., "A-Block", "B-Block").

2. **room**: This entity represents an individual apartment unit. It is linked to a block and stores details like room number, type (2BHK, 3BHK), floor, and its assigned parking_slot.
3. **block_admin**: This table stores information about administrators who manage the apartment complex. It is linked to a block to show which block an admin is responsible for.
4. **auth**: This is the central authentication table. It stores the user_id (username) and password for all users (admins, owners, tenants) and links to their corresponding ID.
5. **owner**: This entity stores all information related to an apartment owner, such as their name, age, and a link to the room they own.
6. **tenant**: This entity stores information for tenants (renters). It includes their personal details and a link to the room they are renting.
7. **employee**: This table stores records of apartment staff (e.g., security, janitors, maintenance crew), including their name, salary, and assigned block.
8. **identity**: This table acts as a weak entity, storing identity proof details (like an Aadhar or PAN number) and linking them to either an owner or a tenant.
9. **rental**: This table tracks the rental details for tenants, including their date of joining (doj) and monthly_rent. It is linked to both the tenant and the room.
10. **complaints**: This operational table stores all complaints raised by residents. It logs the complaint text, which room/block it came from, who reported it, and its current status (e.g., 'Pending').
11. **maintenance**: This financial table tracks monthly maintenance payments for each apartment (apartment_id, which links to room_no). It stores the amount, month, due date, and payment status ('Unpaid', 'Paid').
12. **feedback**: This table allows users (owners or tenants) to submit feedback or ratings about the apartment services or application.
13. **community_events**: This table stores information about upcoming events in the community, such as location, description, and organizer.
14. **amenities**: This entity lists the shared amenities available in the complex (e.g., "Swimming Pool", "Gym"), along with their descriptions and contact details.
15. **service_providers**: This table provides a directory of trusted external service providers (e.g., "Plumber", "Electrician") for residents, including contact info and ratings.
16. **lease_agreements**: This table manages the legal agreements for rentals. It links a tenant, an owner, and an apartment_no, and stores details like start/end dates, rent, and security deposit.
17. **visitors**: This operational table manages the visitor workflow. It stores visitor details, their purpose of visit, and, most importantly, their approval_status ('Pending', 'Approved') and visitor_status ('Requested', 'Inside', 'Exited').

---

8. DDL Commands

The following DDL commands from the database/full_updated.sql file are used to create the entire database schema.

SQL

```sql
-- Drop and create database
DROP DATABASE IF EXISTS apartment_management;
CREATE DATABASE apartment_management;
USE apartment_management;

-- ==================================================
-- CORE TABLES
-- ==================================================

-- TABLE 1: BLOCK
CREATE TABLE block (
  block_no INT NOT NULL,
  block_name VARCHAR(10) DEFAULT NULL,
  room_no INT DEFAULT NULL,
  PRIMARY KEY (block_no)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

-- TABLE 2: ROOM
CREATE TABLE room (
  room_no INT NOT NULL,
  type VARCHAR(10) DEFAULT NULL,
  floor INT DEFAULT NULL,
  parking_slot VARCHAR(10) DEFAULT NULL,
  reg_no INT DEFAULT NULL,
  block_no INT DEFAULT NULL,
  PRIMARY KEY (room_no),
  UNIQUE KEY parking_slot (parking_slot),
  UNIQUE KEY reg_no (reg_no),
  KEY fk_room_block (block_no),
  CONSTRAINT fk_room_block FOREIGN KEY (block_no) REFERENCES block (block_no) ON DELETE
CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

-- TABLE 3: BLOCK_ADMIN
CREATE TABLE block_admin (
  admin_id INT NOT NULL,
  admin_name VARCHAR(20) DEFAULT NULL,
  block_no INT DEFAULT NULL,
  PRIMARY KEY (admin_id),
  KEY fk_admin_block (block_no),
  CONSTRAINT fk_admin_block FOREIGN KEY (block_no) REFERENCES block (block_no) ON
DELETE SET NULL
```

```
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

-- TABLE 4: AUTH (Authentication)
CREATE TABLE auth (
  user_id VARCHAR(10) NOT NULL,
  password VARCHAR(20) NOT NULL DEFAULT '12345678',
  id INT NOT NULL,
  PRIMARY KEY (user_id),
  UNIQUE KEY id (id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

-- TABLE 5: OWNER
CREATE TABLE owner (
  owner_id INT NOT NULL,
  name VARCHAR(20) DEFAULT NULL,
  age INT DEFAULT NULL,
  aggrement_status VARCHAR(20) NOT NULL,
  room_no INT DEFAULT NULL,
  dob VARCHAR(15) DEFAULT NULL,
  PRIMARY KEY (owner_id),
  KEY fk_owner_room (room_no),
  CONSTRAINT fk_owner_room FOREIGN KEY (room_no) REFERENCES room (room_no) ON DELETE SET NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

-- TABLE 6: TENANT
CREATE TABLE tenant (
  tenant_id INT NOT NULL,
  name VARCHAR(30) DEFAULT NULL,
  dob VARCHAR(10) DEFAULT NULL,
  stat VARCHAR(10) DEFAULT NULL,
  room_no INT DEFAULT NULL,
  age INT DEFAULT NULL,
  PRIMARY KEY (tenant_id),
  KEY fk_tenant_room (room_no),
  CONSTRAINT fk_tenant_room FOREIGN KEY (room_no) REFERENCES room (room_no) ON DELETE SET NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

-- TABLE 7: EMPLOYEE
CREATE TABLE employee (
  emp_id INT NOT NULL,
  emp_name VARCHAR(30) DEFAULT NULL,
```

```sql
 salary INT DEFAULT NULL,
 emp_type VARCHAR(20) DEFAULT NULL,
 age INT DEFAULT NULL,
 block_no INT DEFAULT NULL,
 PRIMARY KEY (emp_id),
 KEY fk_employee_block (block_no),
 CONSTRAINT fk_employee_block FOREIGN KEY (block_no) REFERENCES block (block_no) ON
DELETE SET NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

-- TABLE 8: IDENTITY (Proof documents)
CREATE TABLE identity (
 proof VARCHAR(15) DEFAULT NULL,
 owner_id INT DEFAULT NULL,
 tenant_id INT DEFAULT NULL,
 UNIQUE KEY proof (proof),
 KEY fk_identity_owner (owner_id),
 KEY fk_identity_tenant (tenant_id),
 CONSTRAINT fk_identity_owner FOREIGN KEY (owner_id) REFERENCES owner (owner_id) ON
DELETE CASCADE,
 CONSTRAINT fk_identity_tenant FOREIGN KEY (tenant_id) REFERENCES tenant (tenant_id) ON
DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

-- TABLE 9: RENTAL
CREATE TABLE rental (
 rental_id INT NOT NULL AUTO_INCREMENT,
 doj VARCHAR(20) DEFAULT NULL,
 monthly_rent INT DEFAULT NULL,
 room_no INT DEFAULT NULL,
 tenant_id INT DEFAULT NULL,
 PRIMARY KEY (rental_id),
 KEY fk_rental_room (room_no),
 KEY fk_rental_tenant (tenant_id),
 CONSTRAINT fk_rental_room FOREIGN KEY (room_no) REFERENCES room (room_no) ON DELETE
CASCADE,
 CONSTRAINT fk_rental_tenant FOREIGN KEY (tenant_id) REFERENCES tenant (tenant_id) ON
DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

-- TABLE 10: COMPLAINTS
CREATE TABLE complaints (
 complaint_id INT NOT NULL AUTO_INCREMENT,
 block_no INT NOT NULL,
 room_no INT NOT NULL,
 complaint_text VARCHAR(500) NOT NULL,
 reported_by VARCHAR(100) NOT NULL,
 status VARCHAR(20) DEFAULT 'Pending',
 reported_date DATETIME DEFAULT CURRENT_TIMESTAMP,
 PRIMARY KEY (complaint_id),
 KEY idx_complaints_block_room (block_no, room_no),
 KEY idx_complaints_date (reported_date),
 KEY idx_complaints_status (status),
 CONSTRAINT fk_complaint_block FOREIGN KEY (block_no) REFERENCES block (block_no) ON DELETE CASCADE,
 CONSTRAINT fk_complaint_room FOREIGN KEY (room_no) REFERENCES room (room_no) ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

-- TABLE 11: MAINTENANCE
CREATE TABLE maintenance (
 maintenance_id INT NOT NULL AUTO_INCREMENT,
 month VARCHAR(20) NOT NULL,
 amount DECIMAL(10,2) NOT NULL,
 status VARCHAR(20) DEFAULT 'Unpaid',
 apartment_id INT NOT NULL,
 due_date DATE,
 created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
 updated_at DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
 PRIMARY KEY (maintenance_id),
 KEY idx_maintenance_apartment (apartment_id),
 KEY idx_maintenance_status (status),
 KEY idx_maintenance_month (month),
 CONSTRAINT fk_maintenance_room FOREIGN KEY (apartment_id) REFERENCES room (room_no) ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

-- TABLE 12: FEEDBACK
CREATE TABLE feedback (
 feedback_id INT NOT NULL AUTO_INCREMENT,
 user_id INT NOT NULL,
 user_type ENUM('owner', 'tenant') NOT NULL,
 feedback_text TEXT NOT NULL,
 feedback_date DATETIME DEFAULT CURRENT_TIMESTAMP,
 status VARCHAR(20) DEFAULT 'New',
 rating INT DEFAULT NULL,
 PRIMARY KEY (feedback_id),
 KEY idx_feedback_user (user_id, user_type),
 KEY idx_feedback_date (feedback_date),
 KEY idx_feedback_status (status)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

-- TABLE 13: COMMUNITY_EVENTS
CREATE TABLE community_events (
 event_id INT NOT NULL,
 apartment_id INT DEFAULT NULL,
 location VARCHAR(100) DEFAULT NULL,
 description TEXT,
 organizer_id INT DEFAULT NULL,
 event_name VARCHAR(100) DEFAULT NULL,
 event_date DATETIME DEFAULT NULL,
 created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
 PRIMARY KEY (event_id),
 KEY idx_event_date (event_date),
 KEY fk_event_apartment (apartment_id),
 CONSTRAINT fk_event_apartment FOREIGN KEY (apartment_id) REFERENCES room (room_no) ON DELETE SET NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

-- TABLE 14: AMENITIES
CREATE TABLE amenities (
 amenity_id INT NOT NULL,
 amenity_name VARCHAR(100) NOT NULL,
 description TEXT,
 phone_number VARCHAR(15) DEFAULT NULL,
 email VARCHAR(100) DEFAULT NULL,
 rating DECIMAL(3,2) DEFAULT NULL,
 created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
 PRIMARY KEY (amenity_id),
 KEY idx_amenity_name (amenity_name)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

-- TABLE 15: SERVICE_PROVIDERS
CREATE TABLE service_providers (
 provider_id INT NOT NULL,
 provider_name VARCHAR(100) NOT NULL,
 service_type VARCHAR(50) DEFAULT NULL,
 contact_number VARCHAR(15) DEFAULT NULL,
 email VARCHAR(100) DEFAULT NULL,
 rating DECIMAL(3,2) DEFAULT NULL,
 created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
 PRIMARY KEY (provider_id),
 KEY idx_service_type (service_type),
 KEY idx_provider_name (provider_name)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

-- TABLE 16: LEASE_AGREEMENTS
CREATE TABLE lease_agreements (
 agreement_id INT NOT NULL AUTO_INCREMENT,
 tenant_id INT NOT NULL,
 owner_id INT NOT NULL,
 apartment_no INT NOT NULL,
 start_date DATE NOT NULL,
 end_date DATE NOT NULL,
 monthly_rent DECIMAL(10,2) NOT NULL,
 security_deposit DECIMAL(10,2) NOT NULL,
 lease_terms TEXT,
 status VARCHAR(20) DEFAULT 'Active',
 created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
 updated_at DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
 PRIMARY KEY (agreement_id),
 KEY idx_lease_tenant (tenant_id),
 KEY idx_lease_owner (owner_id),
 KEY idx_lease_apartment (apartment_no),
 KEY idx_lease_status (status),
 KEY idx_lease_dates (start_date, end_date),
 CONSTRAINT fk_lease_tenant FOREIGN KEY (tenant_id) REFERENCES tenant (tenant_id) ON DELETE CASCADE,
 CONSTRAINT fk_lease_owner FOREIGN KEY (owner_id) REFERENCES owner (owner_id) ON DELETE CASCADE,
 CONSTRAINT fk_lease_apartment FOREIGN KEY (apartment_no) REFERENCES room (room_no) ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

-- TABLE 17: VISITORS (WITH APPROVAL WORKFLOW)
CREATE TABLE visitors (
 visitor_id INT NOT NULL AUTO_INCREMENT,
 visitor_name VARCHAR(100) NOT NULL,
 apartment_no INT NOT NULL,
 owner_id INT DEFAULT NULL,
 tenant_id INT DEFAULT NULL,
 requested_by VARCHAR(20) NOT NULL,
 requester_id INT NOT NULL,
 entry_time DATETIME NOT NULL,
 exit_time DATETIME DEFAULT NULL,
 purpose VARCHAR(200) DEFAULT NULL,
 contact_number VARCHAR(15) DEFAULT NULL,
 id_proof_type VARCHAR(50) DEFAULT NULL,
 id_proof_number VARCHAR(50) DEFAULT NULL,
 approval_status VARCHAR(20) DEFAULT 'Pending',
 approved_by INT DEFAULT NULL,
 approved_at DATETIME DEFAULT NULL,
 rejection_reason VARCHAR(500) DEFAULT NULL,
 visitor_status VARCHAR(20) DEFAULT 'Requested',
 created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
 PRIMARY KEY (visitor_id),
 KEY idx_visitor_apartment (apartment_no),
 KEY idx_visitor_owner (owner_id),
 KEY idx_visitor_tenant (tenant_id),
 KEY idx_visitor_entry (entry_time),
 KEY idx_visitor_approval_status (approval_status),
 KEY idx_visitor_status (visitor_status),
 CONSTRAINT fk_visitor_apartment FOREIGN KEY (apartment_no) REFERENCES room (room_no) ON DELETE CASCADE,
 CONSTRAINT fk_visitor_owner FOREIGN KEY (owner_id) REFERENCES owner (owner_id) ON DELETE SET NULL,
 CONSTRAINT fk_visitor_tenant FOREIGN KEY (tenant_id) REFERENCES tenant (tenant_id) ON DELETE SET NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
```

9. CRUD Operations (Only few are listed below)

The project implements full CRUD (Create, Read, Update, Delete) functionality. The backend's mysql_connect.js file defines the raw SQL queries, which are exposed as functions. These are called by the API route handlers in server/routes/dashb.js and triggered by axios calls from the frontend React components.

- **CREATE:**
  - **Operation:** A resident (owner) raises a new complaint.
  - **Frontend (React):** The RaisingComplaints.jsx component makes a POST request to the API.
  - **API Endpoint:** POST http://localhost:3001/dashb/raisingcomplaint.
  - **SQL Query:** INSERT INTO complaints (complaint_text, reported_by, block_no, room_no) VALUES (?, ?, ?, ?).

- **CREATE (Inferred):**
  - **Operation:** An admin creates a new owner.
  - **Frontend (React):** The CreatingOwner.jsx component makes a POST request.
  - **API Endpoint:** POST http://localhost:3001/create/owner.
  - **SQL Query:** (The SQL query is in the un-provided server/routes/create.js file, but it is an INSERT into the owner table).
- **READ (Collection):**
  - **Operation:** An admin views all complaints.
  - **Frontend (React):** The ComplaintsViewer.jsx component fetches data on load.
  - **API Endpoint:** GET http://localhost:3001/dashb/viewcomplaints.
  - **SQL Query:** SELECT * FROM complaints.
- **READ (Filtered):**
  - **Operation:** An owner views only their own complaints.
  - **API Endpoint:** POST http://localhost:3001/dashb/ownercomplaints.
  - **SQL Query:** SELECT * FROM complaints WHERE reported_by=?.
- **READ (Aggregate):**
  - **Operation:** The admin dashboard gets the total count of owners.
  - **API Endpoint:** POST http://localhost:3001/dashb/admin.
  - **SQL Query:** SELECT COUNT(*) FROM owner.
- **UPDATE:**
  - **Operation:** An owner updates their assigned parking slot.
  - **Frontend (React):** The ParkingSlot.jsx component makes a POST request on submit.
  - **API Endpoint:** POST http://localhost:3001/dashb/updateParkingSlot.
  - **SQL Query:** UPDATE room SET parking_slot = ? WHERE room_no = ?.
- **DELETE:**
  - **Operation:** Deleting a room record.
  - **Implementation:** While no direct delete endpoints are exposed in the provided route files, the database schema is designed with **cascading deletes**. If a record from the room table is deleted, all corresponding records in complaints, maintenance, lease_agreements, and visitors will be automatically deleted by the database.
  - **SQL (DDL):** CONSTRAINT fk_complaint_room FOREIGN KEY (room_no) REFERENCES room (room_no) ON DELETE CASCADE.

---

10. Triggers, Procedures/Functions, Nested query, Join, Aggregate queries

The following advanced SQL features are implemented in the file database/views.sql.

Triggers

Trigger 1: after_lease_insert

- **Code:**
  SQL
  ```
  CREATE TRIGGER after_lease_insert
  ```

```
AFTER INSERT ON lease_agreements
FOR EACH ROW
BEGIN
  INSERT INTO rental (doj, monthly_rent, room_no, tenant_id)
  VALUES (NEW.start_date, NEW.monthly_rent, NEW.apartment_no, NEW.tenant_id)
  ON DUPLICATE KEY UPDATE
    monthly_rent = NEW.monthly_rent,
    doj = NEW.start_date;
END //
```

- **Reason Used:** To maintain data consistency and reduce redundancy. When a new lease_agreements record is created, this trigger automatically inserts or updates the corresponding record in the simpler rental table. This ensures the tenant's primary rent information is synchronized without requiring a separate application-level query.

Trigger 2: before_lease_check_expiry

- **Code:**
  SQL
  ```
  CREATE TRIGGER before_lease_check_expiry
  BEFORE UPDATE ON lease_agreements
  FOR EACH ROW
  BEGIN
    IF NEW.end_date < CURDATE() AND OLD.status = 'Active' THEN
      SET NEW.status = 'Expired';
    END IF;
  END //
  ```
- **Reason Used:** To automate business logic at the database level. This trigger automatically checks if a lease's end date has passed every time the record is updated. If it has, it automatically changes the lease status from 'Active' to 'Expired', ensuring data integrity and preventing manual checks by an admin.

Trigger 3: after_visitor_approval

- **Code:**
  SQL
  ```
  CREATE TRIGGER after_visitor_approval
  AFTER UPDATE ON visitors
  FOR EACH ROW
  BEGIN
    IF NEW.approval_status = 'Approved' AND OLD.approval_status != 'Approved' THEN
      UPDATE visitors
      SET visitor_status = 'Approved'
      WHERE visitor_id = NEW.visitor_id;
  ```

```
    END IF;

    IF NEW.approval_status = 'Rejected' AND OLD.approval_status != 'Rejected' THEN
        UPDATE visitors
        SET visitor_status = 'Rejected'
        WHERE visitor_id = NEW.visitor_id;
    END IF;
END //
```

- **Reason Used:** To manage the multi-step visitor workflow. The approval_status is changed by an admin, but the visitor_status is changed by security (e.g., to 'Inside'). This trigger automatically moves the visitor to the next stage ('Approved' or 'Rejected') as soon as the admin makes a decision, decoupling the admin's action from the security's action.

Stored Procedures

Procedure 1: GetExpiringLeases(IN days_param INT)

- **Code:**

SQL

```
CREATE PROCEDURE GetExpiringLeases(IN days_param INT)
BEGIN
    SELECT
        la.*, t.name as tenant_name, t.age as tenant_age,
        o.name as owner_name, r.type as room_type, r.floor, b.block_name,
        DATEDIFF(la.end_date, CURDATE()) as days_remaining
    FROM lease_agreements la
    INNER JOIN tenant t ON la.tenant_id = t.tenant_id
    INNER JOIN owner o ON la.owner_id = o.owner_id
    INNER JOIN room r ON la.apartment_no = r.room_no
    INNER JOIN block b ON r.block_no = b.block_no
    WHERE la.status = 'Active'
    AND la.end_date BETWEEN CURDATE() AND DATE_ADD(CURDATE(), INTERVAL
days_param DAY)
    ORDER BY la.end_date ASC;
END //
```

- **Reason Used:** For complex, reusable reporting. This procedure encapsulates a complex 5-table **JOIN** query. It allows the application to easily fetch a report of all active leases that will expire within a user-specified number of days (days_param). This is far more efficient and secure than writing the join in the application code.

Procedure 2: GetLeaseDetails(IN lease_id_param INT)

- **Code:**
  SQL
  ```
  CREATE PROCEDURE GetLeaseDetails(IN lease_id_param INT)
  BEGIN
    SELECT
      la.*, t.name as tenant_name, t.age as tenant_age, t.dob as tenant_dob,
      o.name as owner_name, o.age as owner_age,
      r.type as room_type, r.floor, r.parking_slot,
      b.block_no, b.block_name,
      DATEDIFF(la.end_date, CURDATE()) as days_remaining,
      DATEDIFF(CURDATE(), la.start_date) as days_elapsed,
      DATEDIFF(la.end_date, la.start_date) as total_lease_days
    FROM lease_agreements la
    INNER JOIN tenant t ON la.tenant_id = t.tenant_id
    INNER JOIN owner o ON la.owner_id = o.owner_id
    INNER JOIN room r ON la.apartment_no = r.room_no
    INNER JOIN block b ON r.block_no = b.block_no
    WHERE la.agreement_id = lease_id_param;
  END //
  ```

- **Reason Used:** To encapsulate a complex data retrieval query. This procedure gets all possible details for a single lease agreement by its ID. It **joins** 5 tables and also calculates derived data (like days_remaining, days_elapsed) on the fly, simplifying the application logic.

Procedure 3: GetCurrentVisitorsInside()

- **Code:**
  SQL
  ```
  CREATE PROCEDURE GetCurrentVisitorsInside()
  BEGIN
    SELECT
      v.*, r.type as room_type, r.floor, b.block_name,
      CASE
        WHEN v.requested_by = 'owner' THEN o.name
        WHEN v.requested_by = 'tenant' THEN t.name
        ELSE 'Unknown'
      END as requester_name,
      TIMESTAMPDIFF(MINUTE, v.entry_time, NOW()) as minutes_inside
    FROM visitors v
    INNER JOIN room r ON v.apartment_no = r.room_no
    INNER JOIN block b ON r.block_no = b.block_no
  ```

```
  LEFT JOIN owner o ON v.requester_id = o.owner_id AND v.requested_by = 'owner'
  LEFT JOIN tenant t ON v.requester_id = t.tenant_id AND v.requested_by = 'tenant'
  WHERE v.visitor_status = 'Inside'
  ORDER BY v.entry_time DESC;
END //
```

- **Reason Used:** For live monitoring and security dashboards. This procedure provides a real-time list of all visitors who are currently 'Inside' the complex. It **joins** multiple tables to show *who* they are (visitor name) and *where* they are (room, block, host name), and calculates how long they have been inside (minutes_inside).

Procedure 4: GetPendingVisitorRequests()

- **Code:**

  SQL

```
CREATE PROCEDURE GetPendingVisitorRequests()
BEGIN
  SELECT
    v.*, r.type as room_type, r.floor, b.block_name,
    CASE
      WHEN v.requested_by = 'owner' THEN o.name
      WHEN v.requested_by = 'tenant' THEN t.name
      ELSE 'Unknown'
    END as requester_name,
    TIMESTAMPDIFF(HOUR, v.created_at, NOW()) as hours_pending
  FROM visitors v
  INNER JOIN room r ON v.apartment_no = r.room_no
  INNER JOIN block b ON r.block_no = b.block_no
  LEFT JOIN owner o ON v.requester_id = o.owner_id AND v.requested_by = 'owner'
  LEFT JOIN tenant t ON v.requester_id = t.tenant_id AND v.requested_by = 'tenant'
  WHERE v.approval_status = 'Pending'
  ORDER BY v.created_at ASC;
END //
```

- **Reason Used:** To power the administrator's core workflow. This procedure selects all visitors whose approval_status is 'Pending', effectively creating the "approval queue" for the admin dashboard. It calculates hours_pending to help admins prioritize requests.

Procedure 5: GetApartmentVisitorHistory(IN apartment_param INT, IN days_param INT)

- **Code:**

  SQL

```
CREATE PROCEDURE GetApartmentVisitorHistory(IN apartment_param INT, IN days_param
INT)
BEGIN
```

```sql
  SELECT
    v.*,
    CASE
      WHEN v.requested_by = 'owner' THEN o.name
      WHEN v.requested_by = 'tenant' THEN t.name
      ELSE 'Unknown'
    END as requester_name,
    TIMESTAMPDIFF(MINUTE, v.entry_time, v.exit_time) as visit_duration_minutes
  FROM visitors v
  LEFT JOIN owner o ON v.requester_id = o.owner_id AND v.requested_by = 'owner'
  LEFT JOIN tenant t ON v.requester_id = t.tenant_id AND v.requested_by = 'tenant'
  WHERE v.apartment_no = apartment_param
  AND v.entry_time >= DATE_SUB(CURDATE(), INTERVAL days_param DAY)
  ORDER BY v.entry_time DESC;
END //
```

- **Reason Used:** For auditing and user-facing reporting. This allows a resident or admin to see a complete history of all visitors to a specific apartment (apartment_param) over a given number of days (days_param).

Procedure 6: GetMyVisitorRequests(IN user_type_param VARCHAR(20), IN user_id_param INT)

- **Code:**

  SQL

```sql
CREATE PROCEDURE GetMyVisitorRequests(IN user_type_param VARCHAR(20), IN user_id_param INT)
BEGIN
  SELECT
    v.*, r.type as room_type, r.floor, b.block_name,
    CASE
      WHEN v.approved_by IS NOT NULL THEN
        (SELECT admin_name FROM block_admin WHERE admin_id = v.approved_by)
      ELSE NULL
    END as approved_by_name
  FROM visitors v
  INNER JOIN room r ON v.apartment_no = r.room_no
  INNER JOIN block b ON r.block_no = b.block_no
  WHERE v.requested_by = user_type_param
  AND v.requester_id = user_id_param
  ORDER BY v.created_at DESC;
END //
```

- **Reason Used:** To provide user-specific, secure data. This procedure allows a *specific* owner or tenant to see the status of all visitor requests *they* have submitted. It uses a **Nested Query** (subquery) in the SELECT statement to get the name of the admin who approved the request, which is more efficient than another JOIN.

User-Defined Functions

Function 1: GetTotalLeaseValue(lease_id_param INT) RETURNS DECIMAL(12,2)

- **Code:**
  SQL
  ```sql
  CREATE FUNCTION GetTotalLeaseValue(lease_id_param INT)
  RETURNS DECIMAL(12,2)
  DETERMINISTIC
  BEGIN
    DECLARE total_value DECIMAL(12,2);
    DECLARE months_count INT;
    DECLARE monthly_amount DECIMAL(10,2);

    SELECT
      TIMESTAMPDIFF(MONTH, start_date, end_date),
      monthly_rent
    INTO months_count, monthly_amount
    FROM lease_agreements
    WHERE agreement_id = lease_id_param;

    SET total_value = months_count * monthly_amount;
    RETURN total_value;
  END //
  ```
- **Reason Used:** For complex, reusable calculations. This function calculates the total monetary value of a single lease (monthly rent multiplied by the number of months in the lease term). This logic can now be easily reused in any query (e.g., SELECT GetTotalLeaseValue(101)).

Function 2: GetApartmentVisitorCount(apartment_param INT, days_param INT) RETURNS INT

- **Code:**
  SQL
  ```sql
  CREATE FUNCTION GetApartmentVisitorCount(apartment_param INT, days_param INT)
  RETURNS INT
  DETERMINISTIC
  BEGIN
    DECLARE visitor_count INT;
  ```

```
    SELECT COUNT(*) INTO visitor_count
    FROM visitors
    WHERE apartment_no = apartment_param
    AND entry_time >= DATE_SUB(CURDATE(), INTERVAL days_param DAY);
    RETURN visitor_count;
END //
```

- **Reason Used:** For quick statistics and embedding in other queries. This function returns a single integer: the total number of visitors a specific apartment has had in the last days_param days. It's perfect for use in a dashboard or a larger report.

Nested, Join, and Aggregate Queries

These queries are used extensively within the views, procedures, and application code.

- **Join Query:** The active_leases_view is a prime example of a complex join, combining 5 tables to create a comprehensive virtual table.
  - **Code:**

SQL

```
CREATE OR REPLACE VIEW active_leases_view AS
SELECT
    la.agreement_id, la.apartment_no, la.start_date, la.end_date,
    la.monthly_rent, la.security_deposit, la.status,
    t.tenant_id, t.name as tenant_name,
    o.owner_id, o.name as owner_name,
    r.type as room_type, r.floor, b.block_name,
    DATEDIFF(la.end_date, CURDATE()) as days_remaining
FROM lease_agreements la
INNER JOIN tenant t ON la.tenant_id = t.tenant_id
INNER JOIN owner o ON la.owner_id = o.owner_id
INNER JOIN room r ON la.apartment_no = r.room_no
INNER JOIN block b ON r.block_no = b.block_no
WHERE la.status = 'Active';
```

- **Aggregate Query:** The visitor_approval_stats view uses COUNT, SUM, and AVG to create a statistical summary of the visitor system.
  - **Code:**

SQL

```
CREATE OR REPLACE VIEW visitor_approval_stats AS
SELECT
    COUNT(*) as total_requests,
    SUM(CASE WHEN approval_status = 'Pending' THEN 1 ELSE 0 END) as pending_count,
    SUM(CASE WHEN approval_status = 'Approved' THEN 1 ELSE 0 END) as approved_count,
    SUM(CASE WHEN approval_status = 'Rejected' THEN 1 ELSE 0 END) as rejected_count,
    SUM(CASE WHEN visitor_status = 'Inside' THEN 1 ELSE 0 END) as currently_inside,
    AVG(TIMESTAMPDIFF(HOUR, created_at, approved_at)) as avg_approval_time_hours
FROM visitors;
```

- **Aggregate Queries (from Application):** The admin dashboard relies on simple aggregate queries.
  - **Code:**

```
SQL
module.exports.totalowner = (callback) => {
  con.query("SELECT COUNT(*) FROM owner", callback);
};

module.exports.totaltenant = (callback) => {
  con.query("SELECT COUNT(*) FROM tenant", callback);
};

module.exports.totalemployee = (callback) => {
  con.query("SELECT COUNT(*) FROM employee", callback);
};
```

- **Nested Query (Subquery):** The GetMyVisitorRequests procedure uses a subquery in its SELECT list.
  - **Code:**
- SQL

```
CASE
    WHEN v.approved_by IS NOT NULL THEN
        (SELECT admin_name FROM block_admin WHERE admin_id = v.approved_by)
    ELSE NULL
END as approved_by_name
```

---

11. Code snippets for invoking the Procedures/Functions/Trigger

Triggers are invoked automatically by the database on DML operations (INSERT, UPDATE). Procedures and queries are invoked by the application. The following shows the full-stack invocation for raising a complaint:

1. Frontend (React Component): The user fills a form and clicks "Submit," triggering this axios call.

- **File:** client/src/components/RaisingComplaints.jsx
  JavaScript

```
Axios.post("http://localhost:3001/dashb/raisingcomplaint", {
    complaint: complaint,
    userId: localStorage.getItem("userId"),
    block: localStorage.getItem("block"),
    room: localStorage.getItem("room"),
}).then((res) => {
    alert(res.data.message);
    navigate('/dashboard');
});
```

2. Backend (Express Server Entry): The server receives the request and routes it.

- **File:** server/index.js
  JavaScript
  ```
  const dashb = require("./routes/dashb");
  app.use("/dashb", dashb);
  ```

3. Backend (Express Route): The /dashb router handles the specific endpoint.

- **File:** server/routes/dashb.js
  JavaScript
  ```
  router.post("/raisingcomplaint", (req, res) => {
    console.log("Received complaint:", req.body);
    const values = [
      req.body.complaint,
      req.body.userId,
      req.body.block,
      req.body.room
    ];

    db.registercomplaint(values, (err, result) => {
      // ... (sends response)
    });
  });
  ```

4. Backend (Database Abstraction Layer): The route calls the specific function from the mysql_connect.js file, which contains the raw SQL.

- **File:** server/mysql_connect.js
  JavaScript
  ```
  module.exports.registercomplaint = (values, callback) => {
   con.query("INSERT INTO complaints (complaint_text, reported_by, block_no, room_no) VALUES (?, ?, ?, ?)", values, callback);
  };
  ```

This flow demonstrates how the frontend client, backend server, and database are connected.

---

12. SQL queries (.sql file)

All SQL queries used to build and define the project's database, triggers, procedures, and functions are contained within the following project files:

- **Main Schema DDL:** database/full_updated.sql
- **Views, Triggers, Procedures, Functions:** database/views.sql
- **Iterative/Feature Schema Files:** database/add_features.sql, database/maintenance_feedback.sql, database/visitor_leese aggrement.sql, database/update_complaints.sql, database/add_rating_to_service_providers.sql
- **Raw Queries (Application Layer):** server/mysql_connect.js

---

13. Github repo link

https://github.com/tarun0648/Apartment-Management-System-DBMS