Visitor Management System Report

1. Project Overview

1.1 Overall Project Description

The Chinese University of Hong Kong, Shenzhen (CUHK(SZ)) Visitor Management System is an application designed to modernize and streamline the visitor registration, appointment scheduling, and approval processes within the university premises.

The system serves two distinct user groups: external **Visitors**, who require a convenient interface to submit, track, and modify their visit requests; and **Administrators**, who need robust tools for real-time review, monitoring, and operational data reporting. This document provides a comprehensive analysis of the system requirements, design architecture, implementation details, and overall project outcomes.

2. Requirements Analysis

The system's requirements were derived from the need to facilitate smooth visitor access while maintaining strict security control. The functionality is divided into two primary user roles, ensuring role-based access control is implemented throughout the application.

2.1 Functional Requirements

2.1.1 Visitor Functions

Visitors primarily interact with the system to manage their personal profile and schedule appointments. Key functionalities include:

1. **Authentication and Profile:** Visitors must `register` using a unique username and password, `log in` through an authenticated session, and `submit/manage personal identifying information` (name, phone, affiliation) within a dedicated profile section.

1. **Reservation Management:** Visitors can `create a new reservation` by specifying the date, time, location, and purpose of their visit.
2. **Status Tracking and Modification:** Users can `view the real-time status of their submissions` (pending, approved, or denied). Critically, `modifying an existing appointment` requires the reservation status to be automatically reset to *pending*, necessitating a fresh administrative review. Visitors can also `cancel existing appointments`, subject to status constraints.
3. **History:** The system maintains and displays a complete history of all past and future reservations.

2.1.2 Administrator Functions

Administrators manage all visitor traffic and operational data. Core functions are focused on review and reporting:

1. **Authentication and Profile:** Administrators `log in` via dedicated credentials and possess the ability to `update their personal contact information` (e.g., email and phone number).
2. **Review and Approval:** The primary duty involves `reviewing all reservation records`, which feature powerful `filtering` capabilities by date, location, or approval status. They can instantly approve or

deny any visitor request.

3. **Operational Monitoring:** The system provides two essential dashboards: a real-time `view of all approved visitors scheduled for the current day`, and a statistical report displaying aggregated `daily visitor data`, including the total count and most popular visit locations.

3. Database and SQL Design

The system employs a relational database structure, utilizing **SQLite** for simplicity and reliability. The design comprises five primary tables to manage user accounts, personal details, reservations, administration details, and user notifications.

3.1 Database Architecture

The system follows a three-tier architecture: the Client (Frontend) communicates via the **Network** with the **Server (Backend)**, which manages persistent data storage in the **SQLite Database**.

3.2 SQL Schema Design

The following five tables enforce data integrity and relationship constraints using Primary Keys, Foreign Keys, and CHECK constraints:

1. **Users Table:** Stores authentication credentials for all visitors.
   - Fields: `user_id` (PK, Auto), `username` (Unique), `password_hash`, `created_at`.
2. **Visitor_Info Table:** Stores the visitor's personal, non-authentication data.
   - Fields: `visitor_id` (PK, Auto), `user_id` (FK to Users), `name`, `phone`, `affiliation`, `updated_at`.
3. **Reservations Table:** The core table managing all visit requests and their status.
   - Fields: `reservation_id` (PK, Auto), `user_id` (FK to Users), `visit_date`, `visit_time`, `location`, `purpose`, `status` (CHECK constraint: 'pending', 'approved', 'denied'), `admin_id` (FK to Admins), `review_comment`, `created_at`, `updated_at`.
4. **Admins Table:** Stores authentication and profile data for system administrators.
   - Fields: `admin_id` (PK, Auto), `username` (Unique), `password_hash`, `phone`, `created_at`.
5. **Admin_Info Table:** Stores the Admin's personal data.
   - Fields: `admin_id` (PK, Auto), `username` (Unique, Not Null), `email` (Unique, Not Null), `phone` (Unique, Not Null).

The design strategically utilizes foreign keys to ensure that reservation and profile data are tightly linked to their respective user accounts. Meanwhile, the status and read_status fields employ check constraints to maintain data consistency.

3.3 Data Consistency

SQLite ensures **data consistency** through its built-in database locking mechanism. Whenever a write operation occurs, SQLite automatically applies a **WRITE LOCK**, preventing any other write attempts until the transaction is fully committed. This guarantees that concurrent users cannot modify the same data simultaneously, avoiding potential inconsistencies.

All backend write operations are executed within transactions (`execute` + `commit`), ensuring that changes are applied atomically, so partial updates never occur. Although SQLite does not support advanced row-level locking, its default locking model is fully sufficient for the scale of this system.

In addition to locking, overall data integrity is maintained through foreign key constraints, NOT NULL fields, controlled reservation status values, automatic timestamps, input validation, and parameterized SQL queries. Together, these mechanisms ensure that the system's data remains consistent, reliable, and structurally correct.

4. Front and Back-End Design and Implementation

4.1 Backend Design and Implementation

The backend is built upon **Python** using the **Flask** framework. It's built with a structure that uses Flask Blueprints, which makes the code well-organized and easy to extend. The entire system operates on a **RESTful API** model, with all data exchange handled via **JSON payloads over HTTP**.

4.1.1 Backend Architecture Overview

The codebase is structured to enforce a strict separation of concerns:

1. `app.py`: Serves as the application entry point, handling configuration, initialization, and registering Blueprints.
2. `db.py`: Manages the SQLite database connection, providing a consistent `get_db()` function and ensuring connections are reliably closed at the end of each request by `close_db()`.
3. `models.py`: The Data Abstraction Layer, containing all raw SQL queries wrapped into high-level Python functions (e.g., `update_reservation`, `admin_review_reservation`). This centralizes database interaction and enhances testability.
4. `utils.py`: Handles system utilities, primarily password hashing and verification using secure techniques (e.g., PBKDF2 via a security library).
5. **Route Modules (Blueprints):** Functionality is categorized into distinct Blueprints:
    - `auth_routes.py`: Handles user registration and login, issuing tokens upon successful authentication.
    - `visitor_routes.py`: Manages all visitor-specific operations, including profile updates, reservation creation, viewing, modification, and deletion.
    - `admin_routes.py`: Implements all administrator functions, including fetching filtered reservation lists, executing approval/denial actions, and providing operational metrics (e.g., today's visitors, daily reports).

4.1.2 Security and Logic

All API modules enforce authorization checks, ensuring that visitors can only access or modify data associated with their `user_id`, and administrative APIs are restricted to users with the `admin` role. The system ensures transaction integrity through database commits and implements automatic status resetting (e.g., when a visitor modifies an approved reservation, the status automatically reverts to *pending*).

4.2 Frontend Design and Implementation

The frontend is a lightweight Single-Page Application (SPA) style experience, relying on **HTML**, **Vanilla JavaScript**, and **Tailwind CSS** for styling. The front-end's primary role is to communicate with the backend via asynchronous `fetch` calls, render JSON responses, and manage the user interface state.

4.2.1 Frontend Structure

The design utilizes distinct HTML pages tailored for each major user action, providing a clear user flow:

- `login.html`: Serves as the unified entry point for both visitors and administrators.
- `visitor_home.html` & `visitor_profile.html`: The primary interface for visitors to view their dashboards, reservation lists, and manage their personal contact information.
- `reservation_create.html` & `reservation_list.html`: Dedicated pages for submitting new visit requests and tracking the status of all past and future reservations.
- `admin_dashboard.html`: The central hub for administrators, featuring a metrics dashboard (pending counts, daily visitors), a filterable reservation review table, and quick action buttons for approval/denial.

4.2.2 Technology and User Experience

The use of **Tailwind CSS** allows for rapid styling and a modern, responsive interface with minimal custom CSS. **Vanilla JavaScript** handles all client-side logic, including:

1. **Session Management:** Storing authentication tokens and user IDs in `localStorage` for persistent session state.
2. **Data Fetching and Rendering:** Implementing asynchronous functions (e.g., `fetchReservations`, `fetchMetrics`) to retrieve data from the backend APIs and dynamically render HTML table rows and dashboard metrics.
3. **Event Handling:** Managing user interactions, such as form submissions, filter changes, and dynamic button clicks (e.g., changing a reservation status from 'pending' to 'approved').
4. **Role Separation:** The frontend uses the stored role information to route users to the correct dashboard (`visitor_home.html` or `admin_dashboard.html`) after they log in.

5. Summary and Harvest

5.1 Project Summary

The CUHK(SZ) Visitor Management System successfully meets all defined functional requirements for both visitor and administrator roles. By integrating a `Flask-based RESTful API` with a structured `SQLite database` and a responsive, `JavaScript-driven frontend`, the project delivers a comprehensive digital solution for campus access management.

Key achievements include:

1. Establishing a robust, role-based authentication and authorization mechanism.
2. Implementing secure data operations, notably through password hashing and controlled database access via the model layer.
3. Developing dynamic dashboards for administrators, providing actionable insights (e.g., real-time visitor counts, popular location statistics).
4. Ensuring application resilience, successfully addressing critical issues like database locking and ensuring robust error handling across both client and server (e.g., handling missing request parameters).

5.2 Lessons Learned and Future Harvest

The development process yielded valuable insights into full-stack application development, particularly concerning transaction management and data consistency.

1. **Database Concurrency:** The initial encounter with the `sqlite3.OperationalError: database is locked` emphasized the necessity of meticulous resource management in SQLite. The solution, which involved implementing application context teardown handlers (`app.teardown_appcontext`) and ensuring timely connection closure (`conn.close()`), demonstrated the critical link between backend framework practices and database performance.
2. **Schema and Frontend Synchronization:** Recurring errors related to `no such column` or `undefined` data highlighted the importance of strictly synchronizing the database schema (e.g., `email`, `denied` status) with frontend JavaScript variable names (`item.email`, `item.denied`). This iterative debugging process reinforced the value of using standardized data structures and strict API contracts.

Moving forward, the system is highly extensible. Future enhancements could include adding features such as **Email/SMS notifications** upon status change, integrating an **Admin role for location managers** (restricting view to specific locations), and implementing a **data visualization library** to enhance the daily statistical report beyond simple tables.