PROJECT REPORT

Hospital Management System (HMS)

Course: Modern Application Development 1
Student Name: ADITYA SINGH
Roll Number: 24F2008224
Date: 28-11-2025

---

1. Declaration of AI Usage

I hereby declare that this project, **Hospital Management System**, was developed by me. I have used Artificial Intelligence (AI) tools to assist in the development process.

- **AI Tool Used:** Google Gemini / ChatGPT
- **Approximate AI Contribution:** 40%
- **Nature of Usage:**
    - **Code Assistance:** Used AI to generate boilerplate code for Flask models and initial HTML templates.
    - **Debugging:** Used AI to analyze stack traces (specifically SQLite integrity errors, Flask-Login session issues, and Chart.js rendering bugs).
    - **Logic Optimization:** Used AI suggestions to implement the logic for the "Appointment Conflict Prevention" algorithm (1-hour buffer window).
    - **Content Generation:** Used AI to help structure this report and generate dummy data for testing.

The core business logic, database schema modifications, testing, manual refactoring, and final integration were performed by me.

ADITYA SINGH

---

2. Abstract

The Hospital Management System (HMS) is a full-stack web application designed to digitize and streamline the administrative and clinical operations of a hospital. The system addresses the challenges of manual record-keeping, appointment conflicts, and inefficient data retrieval.

Built using the **Flask** framework (Python), the application features a robust **Role-Based Access Control (RBAC)** system with three distinct portals: **Admin**, **Doctor**, and **Patient**. Key functionalities include secure authentication, dynamic appointment booking with conflict prevention, digital prescription management, and interactive data visualization using Chart.js. The system ensures data integrity through backend validation and provides a responsive user interface using Bootstrap 5.

---

3. Introduction

3.1 Problem Statement

Traditional hospital management often relies on paper-based records or disjointed legacy systems. This leads to critical issues such as:

- **Double-booking:** Doctors are often overbooked due to lack of real-time scheduling checks.
- **Data Loss:** Patient medical history is difficult to track across multiple visits.
- **Lack of Visibility:** Administrators lack immediate insights into hospital statistics (e.g., total active patients).
- **Inconvenience:** Patients must physically visit or call to book appointments.

3.2 Project Objectives

The primary objective of this project is to develop a centralized platform that:

1. **Simplifies Administration:** Allowing admins to manage doctors and view real-time hospital analytics.
2. **Empowers Doctors:** Providing a digital interface to manage schedules and issue digital prescriptions.

3. **Assists Patients:** Enabling easy online booking and access to personal medical history.
4. **Ensures Security:** Implementing secure login, password hashing, and strict session management.

---

4. System Architecture

The application follows the **Model-View-Controller (MVC)** architectural pattern, implemented using Flask Blueprints for modular code organization.

4.1 Technology Stack

- **Backend:** Python 3.x, Flask (Micro-framework).
- **Database:** SQLite (Relational Database), SQLAlchemy (ORM), Flask-Migrate.
- **Frontend:** HTML5, CSS3, Bootstrap 5 (Responsive Framework), Jinja2 Templating.
- **JavaScript Libraries:**
  - **Chart.js:** For data visualization (Admin/Doctor analytics).
  - **Flatpickr:** For modern, user-friendly date/time selection.
- **Version Control:** Git & GitHub.

4.2 Key Libraries Used

- **Flask-Login:** For session management and route protection.
- **Flask-WTF:** For secure form handling and CSRF protection.
- **Flask-Migrate:** For handling database schema changes.
- **Werkzeug:** For password hashing security .

---

5. Features and Modules

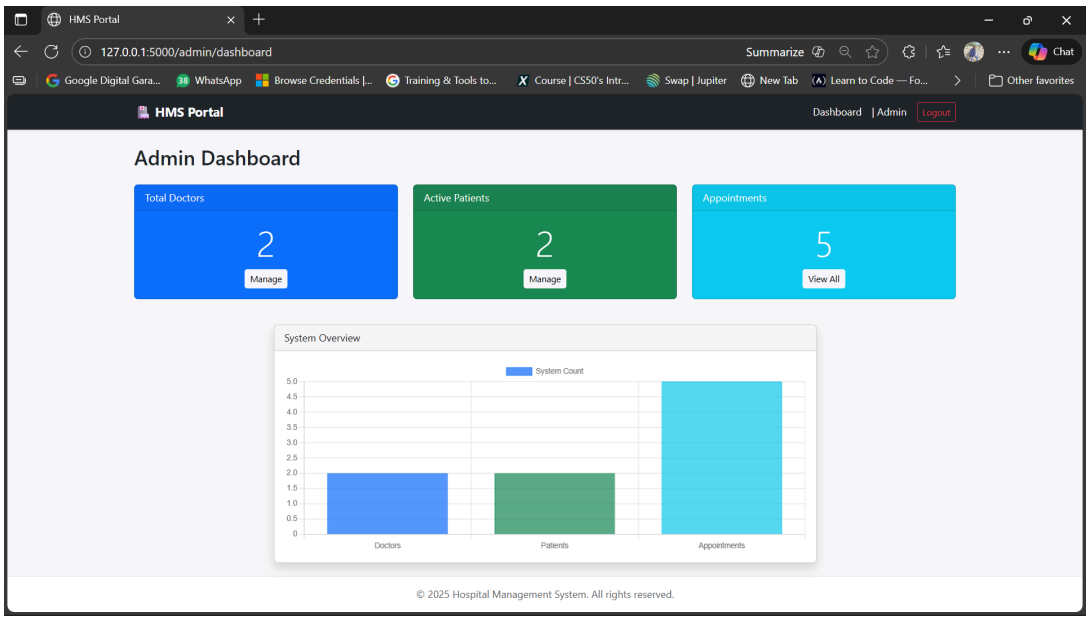
The project is divided into distinct modules based on user roles.

5.1 Authentication Module

- **Secure Login/Logout:** Users are authenticated against hashed passwords.
- **Role-Based Redirection:** Upon login, the system detects the user role (Admin, Doctor, or Patient) and redirects them to their specific dashboard.
- **Registration:** Public registration is available for Patients. Doctors are added securely by Admins.

5.2 Admin Module

- **Dashboard:** Displays cards with total counts and a bar graph comparing Doctors vs. Patients vs. Appointments.
- **Doctor Management:** Admins can Add, Update, and Delete doctor profiles.
- **Patient Management:** Admins can view list of patients and remove access if necessary.
- **Medical Records Access:** Admins have read-only access to view patient treatment histories for oversight.

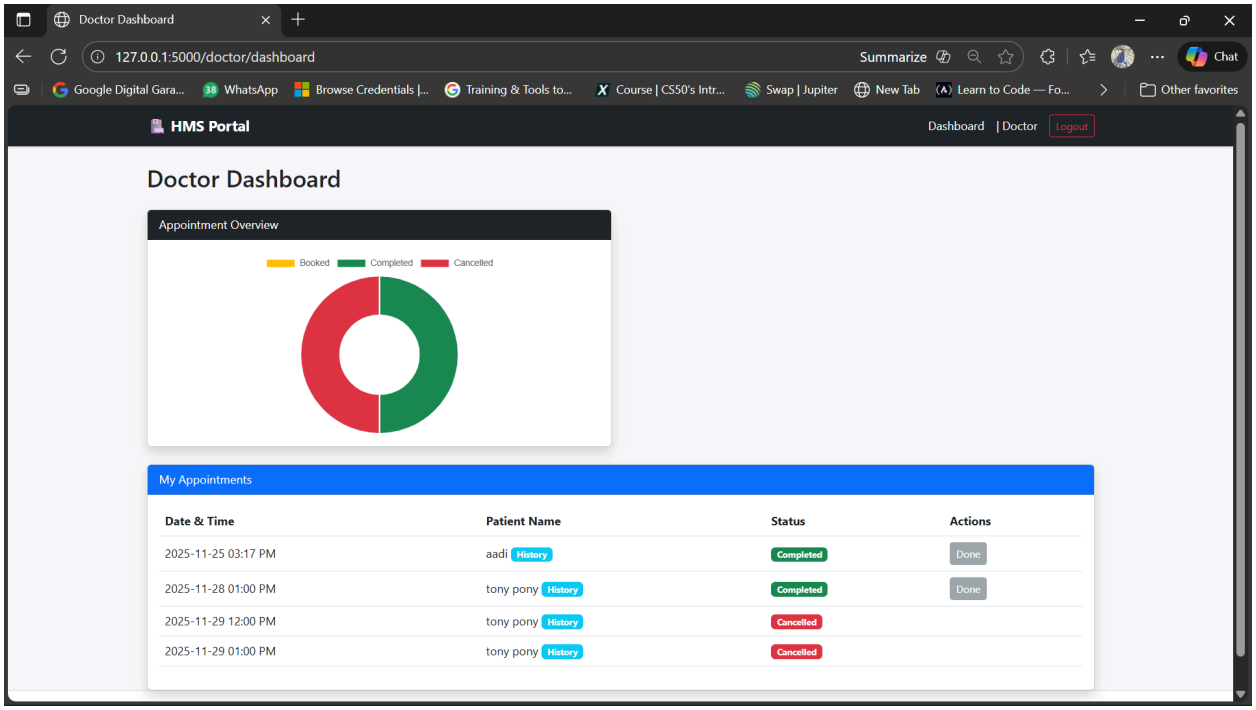


5.3 Doctor Module

- **Appointment Management:** Doctors can view their specific schedule and Approve/Reject pending requests.
- **Treatment Logic:** Doctors can mark appointments as "Completed" by entering a

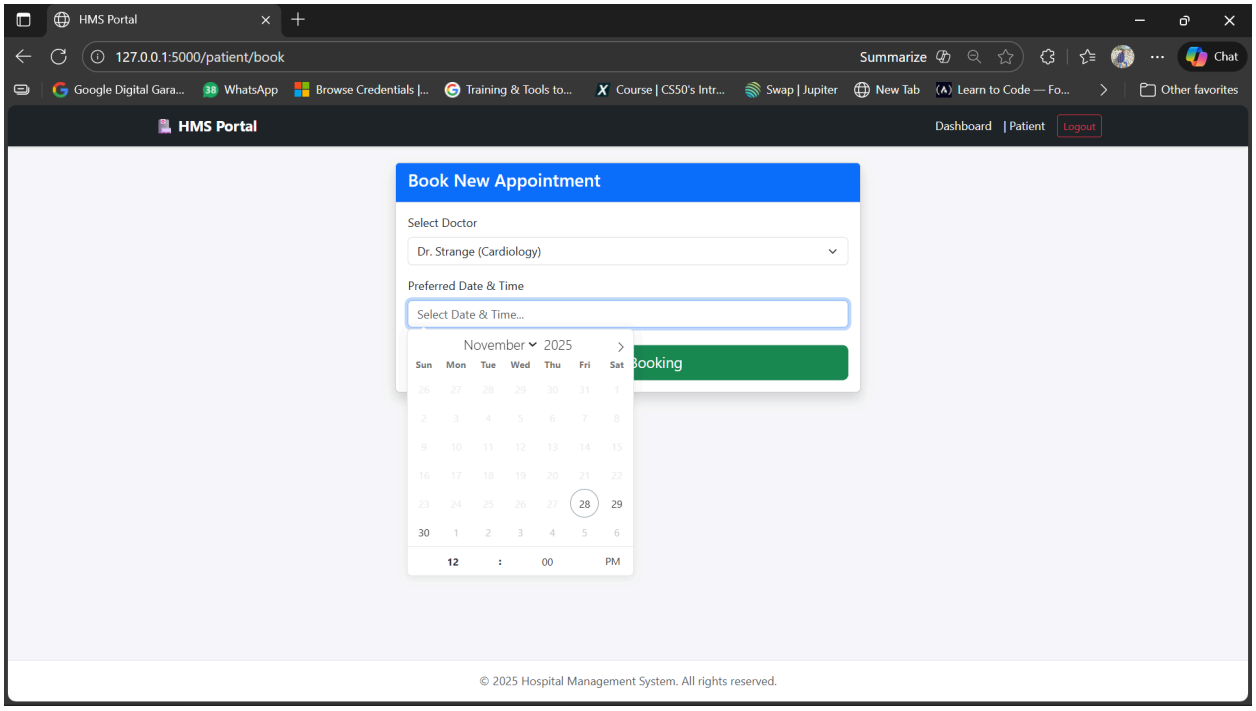
diagnosis and prescription.

- **Analytics:** A doughnut chart displays the doctor's workload (Booked vs. Completed vs. Cancelled).



5.4 Patient Module

- **Booking System:** Patients can select a department, a doctor, and a time slot using a modern date picker.
- **Conflict Prevention:** The system checks the database before confirming to ensure the doctor isn't double-booked.
- **Medical History:** Patients can view a chronological list of past diagnoses and prescriptions.



---

6. Database Schema (ER Design)

The database is programmatically created using SQLAlchemy models. The key entities are:

1. **Admin:** Stores predefined admin credentials.
2. **Department:** Stores medical specializations (e.g., Cardiology).
3. **Doctor:** Linked to Department (Many-to-One). Includes phone_contact and joining_date.
4. **Patient:** Stores personal details and credentials.
5. **Appointment:** connects Patient and Doctor. Stores time and status (Pending_Approval, Booked, Completed).
6. **Treatment:** Linked to Appointment (One-to-One). Stores diagnosis and prescription.

---

7. API Integration

To support future scalability (e.g., mobile apps), the system exposes RESTful API endpoints:

- GET /api/doctors: Public endpoint to fetch the list of doctors.
- GET /api/appointments: Protected endpoint returning appointments for the logged-in user.

- POST /api/appointments: Allows programmatic booking.
- PUT /api/doctors/<id>: Admin endpoint for updates.
- DELETE /api/appointments/<id>: Endpoint for cancellations.

---

8. Challenges Faced & Testing

During the development lifecycle, several critical technical issues were encountered. These were resolved through rigorous testing and debugging.

8.1 The "ID Collision" Issue

- **Problem:** Initially, logging in as a Doctor with ID 1 would conflict with Admin ID 1. The Flask-Login user_loader would confuse them, leading to "Permission Denied" errors.
- **Solution:** I implemented a **Session-Based Role Check**. Upon login, the user's role is stored in session['role']. The load_user function checks this session variable first to query the correct database table.

8.2 Database Schema Migration

- **Problem:** Mid-development, I needed to add a phone_contact field to the Doctor table. Running the code caused OperationalError: no such table and IntegrityError because the old database schema did not match the new Python models.
- **Solution:** I implemented a "Clean Slate Protocol." I deleted the old hms.db and migrations/ folder, then re-initialized the database using flask db init, migrate, and upgrade. I wrote a custom Python script to repopulate the database with dummy data to restore the testing environment.

8.3 Visual Chart Rendering

- **Problem:** The Chart.js graphs appeared blank/invisible on the dashboard, despite the data being correct.
- **Solution:** Inspecting the HTML revealed the canvas height was collapsing to 0. I created a custom CSS class .chart-container with explicit height properties and wrapped the

canvas elements, ensuring they render correctly on all devices.

8.4 Date Format Inconsistencies

- **Problem:** The default HTML5 date picker looked different on Mobile vs. Desktop and caused parsing errors in Python (ValueError).
- **Solution:** I replaced the native input with the **Flatpickr** library. This provided a unified UI and allowed me to enforce a strict Y-m-d H:i K (12-hour) format, which matches the backend parsing logic perfectly.

---

9. Conclusion

The Hospital Management System project was successfully completed, meeting all core requirements and optional enhancements. The application demonstrates a robust understanding of backend logic, database management, and frontend responsiveness. The testing phase was particularly valuable, as it highlighted the importance of database migrations and session management in multi-user environments.

---

10. References

1. Flask Documentation: https://flask.palletsprojects.com/
2. Bootstrap 5 Docs: https://getbootstrap.com/
3. Chart.js Docs: https://www.chartjs.org/
4. Flatpickr Docs: https://flatpickr.js.org/