INDIAN INSTITUTE OF TECHNOLOGY, MADRAS
CHENNAI, TAMIL NADU, INDIA– 600036

Modern Application Development- 1 Project Report

Hospital Management System

Name: BIPLOV ADHIKARI
Roll Number: 23f1001103
Student email ID: 23f1001103@ds.study.iitm.ac.in

Description:

The **Hospital Management System (HMS)** is a multi-user web application built to streamline hospital operations by connecting three distinct user roles—Admins, Doctors, and Patients—onto a single, centralized platform.

The system is designed to manage common hospital workflows, such as appointment scheduling, patient record management, and staff coordination. Admins have a secure panel to manage hospital staff and view system-wide statistics. Doctors are given a personal dashboard to manage their work schedules, handle patient appointments, and update medical records. Patients use a self-service portal to search for doctors and book appointments based on real-time availability.

Technologies Used:

- **Backend:** I used **Flask (Python)** as the core of my project. It handles all the URL routing, backend logic, and user authentication.
- **Frontend:** I used **Jinja2 templates** to dynamically render the HTML pages. The styling was done with standard HTML5 and simple, custom **CSS** which I wrote directly into the base template.
- **Database:** I used **SQLite** with the **Flask-SQLAlchemy** extension. This allowed me to define my data models in Python (like User and Appointment) and let SQLAlchemy handle the database interactions.

Architecture and Features:

The application is built on a standard Model-View-Controller (MVC) pattern:

- **Model:** This is my SQLite database, defined in models.py. It includes all the necessary tables, such as User (which handles all three roles), Department (for doctor specializations), Appointment, Treatment, and DoctorAvailability.
- **View:** These are the Jinja2 templates in my templates folder (like admin_dashboard.html, doctor_dashboard.html, etc.) that render the HTML page the user sees.
- **Controller:** This is my main app.py file. It contains all the Flask routes (like /login or /book_appointment) that handle user requests, process data, and decide which

template to show.

Key Features:

- **Role-Based Authentication:** I built a single, secure login system that checks a user's role (admin, doctor, or patient) after they log in and redirects them to the correct dashboard. All dashboards are protected with @login_required to prevent anyone from accessing a page they shouldn't.
- **Full-Featured Admin Dashboard:** This is the "control center" for the hospital. The admin can:
  - View live statistics (total patients, doctors, etc.).
  - Manage doctors: Add new doctors, edit their profiles, and deactivate/blacklist them.
  - Manage patients: Deactivate/blacklist a patient's account.
  - Search for any user in the system by name, email, or specialization.
  - View a complete log of all appointments and check their treatment details.
- **Doctor's Personal Workflow:** I built a complete interface for doctors to manage their day:
  - A 7-day calendar to set their own work hours and availability.
  - An organized list of their "Upcoming" and "Past" appointments.
  - The ability to "Complete" an appointment, which opens a form to enter a Treatment record (diagnosis and prescription).
  - The ability to view the *entire* medical history for any of their patients.
- **Patient Booking System (with Conflict Prevention):** This was a key challenge. The patient-facing system allows a user to:
  - Register, log in, and update their own personal info (name, age, contact, etc.).
  - Search the database for all active doctors.
  - Book an appointment by selecting a slot from a list of the doctor's *real-time* availability. The system automatically filters out times the doctor has marked as "unavailable" or that are already booked, preventing any double-bookings.
  - Cancel their own upcoming appointments.
  - View their complete appointment history and read the doctor's notes from completed visits.

AI / LLM Usage Declaration

I used an LLM (Gemini) as a paired programmer and debugging helper. I wrote the main backend logic (Flask routes, database models, authentication) myself. The AI helped mainly with:

1. Helping with the "Boilerplate": I used the AI to help generate the initial "boilerplate" code for the HTML templates and the basic CSS styling. This was a time-saver that let me focus more on the Python code.
2. Debugging and Refactoring (My main use): This is where the AI was most valuable. After I wrote a feature, I would often get stuck on complex errors. I would provide my code and the error log to the AI to help me find the problem. For example:
   - It helped me fix several TemplateSyntaxError issues by finding mismatched {% endfor %} or {% endif %} tags.
   - It helped me solve a complex AttributeError when my is_active database column was conflicting with a property from the Flask-Login library. The AI helped me understand the problem, which led to the solution of renaming my column.

- It also helped me fix a TemplateAssertionError (the strptime filter) by explaining that I should move that Python logic from the template into my Flask route and more.

Project Presentation Video Link : MAD_1_Hospital_Management_System