App Dev Project Report

STUDENT DETAILS

NAME : RAUSHAN KUMAR

ROLL NUMBER : 2023104341

EMAIL : 24f2000297@ds.study.iitm.ac.in

ABOUT ME : I am an undergraduate student pursuing Btech from NIT Nagaland as well as BS in Data Science and Applications by IIT Madras . I am very passionate about web development and building meaningful applications using modern tools and technologies.

PROJECT DETAILS

Project Title : Hospital Management App

Problem Statement : To build a Hospital Management System(HMS) web application that allows Admin, Doctors and Patients to interact with the system based on their roles.

APPROACH

The app is built as a server-side rendered web application using Flask. The design emphasizes clear separation of concerns:

- `models.py` defines the database schema using Flask-SQLAlchemy (models and relationships).
- `forms.py` defines the form classes and validation rules using Flask-WTF / WTForms.
- `app.py` contains controllers (Flask routes), request handling, session management, business logic and database operations.
- Templates (Jinja2) render views and UI; `static/` provides styling and client assets.

The workflow: users authenticate (role-based), interact via HTML forms and pages. Server-side validation and ORM-backed persistence ensure data correctness. Time-slot availability is computed at booking time to avoid double-booking.

Technologies used

Technology	Usage(%)	Purpose
Python (3.8+)	17%	Core backend language; fast development, mature ecosystem, wide library support.
Flask (2.0.1)	6.66%	Lightweight web framework for routing, templating, and request handling.
Flask-SQLAlchemy (2.5.1)	3.47%	ORM for defining models, relationships, and performing CRUD operations with a Pythonic API.
SQLite (via SQLAlchemy)	0.02%	Lightweight file-based database; ideal for development and portable deployments.
Flask-WTF (0.15.1) & WTForms (2.3.3	0.39%	Provides secure form handling with CSRF protection and field validation.
Werkzeug (built into Flask)	0.15%	Password hashing (generate_password_hash, check_password_hash) and HTTP utilities.
Jinja2 (bundled with Flask)	58.71%	Server-side HTML templating for injecting dynamic data into pages.
HTML / CSS	66.92%	Frontend interface design and styling of web pages.
Git & GitHub	0.75%	Version control, collaboration, and remote repository hosting.

DB Schema Design

Tables and columns (constraints and notes):

1) `user` (authentication + role)

- `id` (Integer, PK)
- `username` (String(50), unique, not null)
- `email` (String(100), unique, not null)

- `password` (String(200), not null) — store hashed password
- `role` (String(20), default='patient') — values: `patient`, `doctor`, `admin`
- Relationships: one-to-one with `patient` or `doctor` via `user_id` foreign key.

Reasons / constraints:

- Unique username/email enforced to identify accounts.
- Passwords stored hashed (Werkzeug) for security.
- Role field simplifies authorization checks at route-level.

2) `patient` (patient profile)

- `id` (Integer, PK)
- `user_id` (Integer, FK -> user.id, not null)
- `name` (String(100), not null)
- `dob` (Date, not null)
- `gender` (String(10), not null)
- `phone` (String(15), not null)
- `address` (String(200), nullable)
- Relationships: one-to-many `appointments`, one-to-many `medical_records`.

Reasons:

- Separate patient table separates authentication from profile data and enables richer patient-specific fields.
- Foreign key enforces referential integrity with `user`.

3) `doctor` (doctor profile)

- `id` (Integer, PK)
- `user_id` (Integer, FK -> user.id, not null)
- `name` (String(100), not null)
- `specialization` (String(100), not null)
- `department_id` (Integer, FK -> department.id, not null)
- `gender` (String(10), nullable)

- `phone` (String(15), nullable)
- `fees` (Float, default=500.0)
- Relationships: one-to-many `appointments`.

Reasons:

- Doctor-specific metadata lives here; `fees` and `specialization` enable admin management and patient display.
- `department_id` groups doctors by department for browsing and assignment.

4) `department`

- `id` (Integer, PK)
- `name` (String(100), not null)
- Relationships: one-to-many `doctors`.

Reason:

- Logical grouping for doctors and administration.

5) `appointment`

- `id` (Integer, PK)
- `patient_id` (Integer, FK -> patient.id, not null)
- `doctor_id` (Integer, FK -> doctor.id, not null)
- `appointment_date` (Date, not null)
- `time_slot` (String(20), not null) — format `HH:MM`
- `period` (String(20), not null) — Morning/Afternoon/Evening
- `status` (String(20), default='Scheduled') — Scheduled/Completed/Cancelled
- Relationship: one-to-one to `medical_record`.

Reasons / constraints:

- Time-slot stored as string to allow a small fixed set of slots; queries filter on date + time_slot to detect conflicts.
- Status allows lifecycle tracking.

6) `medical_record`

- `id` (Integer, PK)
- `patient_id` (Integer, FK -> patient.id, not null)
- `appointment_id` (Integer, FK -> appointment.id, not null)
- `diagnosis` (Text, not null)
- `prescription` (Text, not null)
- `notes` (Text, nullable)

Reasons:

- Records are linked to appointments so that reports are auditable to a visit.

Design choices and reasons summary:

- Normalized schema with separate `user` table for authentication and `patient`/`doctor` for profile data keeps concerns separate and simplifies role checks.
- Using ORM (SQLAlchemy) makes queries, relationships and migrations easier to manage.
- Storing appointments with `date + time_slot` simplifies availability checks and prevents double-booking by querying for existing appointments with same doctor/date/slot.

API Design

Brief overview of API-like endpoints (server-side routes that return HTML or JSON):

- Authentication & Sessions

- `GET/POST /register` — registration flow (form validation via Flask-WTF)
- `GET/POST /login` — login and role selection
- `GET /logout` — clears session

- Patient-facing

- `GET /patient_dashboard` — view dashboard
- `GET/POST /book_appointment` — book appointments (form + server-side availability check)
- `GET /get_available_slots/<doctor_id>/<date>` — returns available slots (JSON)
- `GET /view_appointments` — list patient appointments

- `GET /view_medical_records` — list medical records

- Doctor-facing

- `GET /doctor_dashboard` — today's and upcoming appointments
- `GET /doctor_appointments` — full list
- `POST /complete_appointment/<id>` — mark appointment completed
- `GET/POST /add_medical_record/<appointment_id>` — create medical record
- `GET /view_medical_record/<appointment_id>` — view specific record

- Admin-facing

- `GET /admin_dashboard` — system overview
- `GET/POST /manage_doctors` — add/list doctors
- `POST /edit_doctor/<id>`, `POST /delete_doctor/<id>` — edit/delete doctor
- `GET /manage_patients`, `POST /delete_patient/<id>` — patient management
- `GET/POST /manage_departments`, `POST /edit_department/<id>`, `POST /delete_department/<id>` — department management
- `GET /manage_appointments`, `POST /delete_appointment/<id>` — appointment oversight

Implementation notes:

- Most endpoints render templates and use form submission; the `get_available_slots` endpoint returns JSON for client-side dynamic slot loading.
- Authentication uses session variables (`session['user_id']`, `session['role']`) and route-level role checks.
- Input validation is performed server-side using WTForms validators; route handlers perform additional checks (e.g., checking appointment conflicts).

Architecture and Features

- `app.py` — Controllers (Flask routes), application configuration (`SECRET_KEY`, `SQLALCHEMY_DATABASE_URI`), session handling, and business logic. This is the main entry point.
- `models.py` — ORM models using Flask-SQLAlchemy. All table definitions, relationships and DB `db` instance live here.

- `forms.py` — WTForms classes used across views for registration, login, and appointment creation.
- `templates/` — Jinja2 templates for all views (landing, login/register, dashboards, appointment flows, management pages).
- `static/` — static assets (CSS, JS, images). Styling is in `static/css/styles.css`.
- `hospital_management.db` — SQLite database file generated at runtime.

Features implemented (how, and categorization):

Default / Core Features

- Authentication & role-based access: implemented using Flask sessions; routes check `session['role']` for access control.
- Patient registration & profile: WTForms + SQLAlchemy to persist user and patient records.
- Appointment booking: form to choose doctor/date/time; server verifies availability by querying `appointment` table for same doctor/date/slot; saves appointment on success.
- Doctor workflows: doctor dashboard shows today's and upcoming appointments via filtered queries; doctors can mark appointments completed and add medical records.
- Medical records: doctors create records linked to an appointment; records stored in `medical_record` table.
- Admin management: create departments, add/edit/delete doctors and patients with dependency checks (e.g., prevent deleting a doctor with appointments).

Additional / Convenience Features

- Default admin creation on first run (auto-insert admin user if not present).
- Simple time-slot categorization (Morning/Afternoon/Evening) for UI grouping.
- Basic conflict handling and user-facing flash messages on errors/success.

Security & Data Integrity

- Passwords hashed with Werkzeug functions; never stored in plain text.
- CSRF protection via Flask-WTF on all forms.
- DB constraints and foreign keys maintain referential integrity.

Video

Drive Link: https://drive.google.com/file/d/1CmnIbSCSdEIkPur-uSukqkdCKS663u54/view