# CarePlus - A Microservices-Based Hospital Management System

CarePlus is a modern, web-based hospital management platform designed to streamline patient and doctor interactions. It features separate, dedicated portals for patients, doctors, and administrators. The system is built on a robust microservices architecture, ensuring scalability, maintainability, and independent deployment of its core functionalities, including appointment scheduling, prescription management, and lab test bookings etc.

# Architecture Used - Microservices

The application is built using a **Microservices Architecture**. This approach breaks down the complex hospital management system into smaller, independent services that communicate with each other.

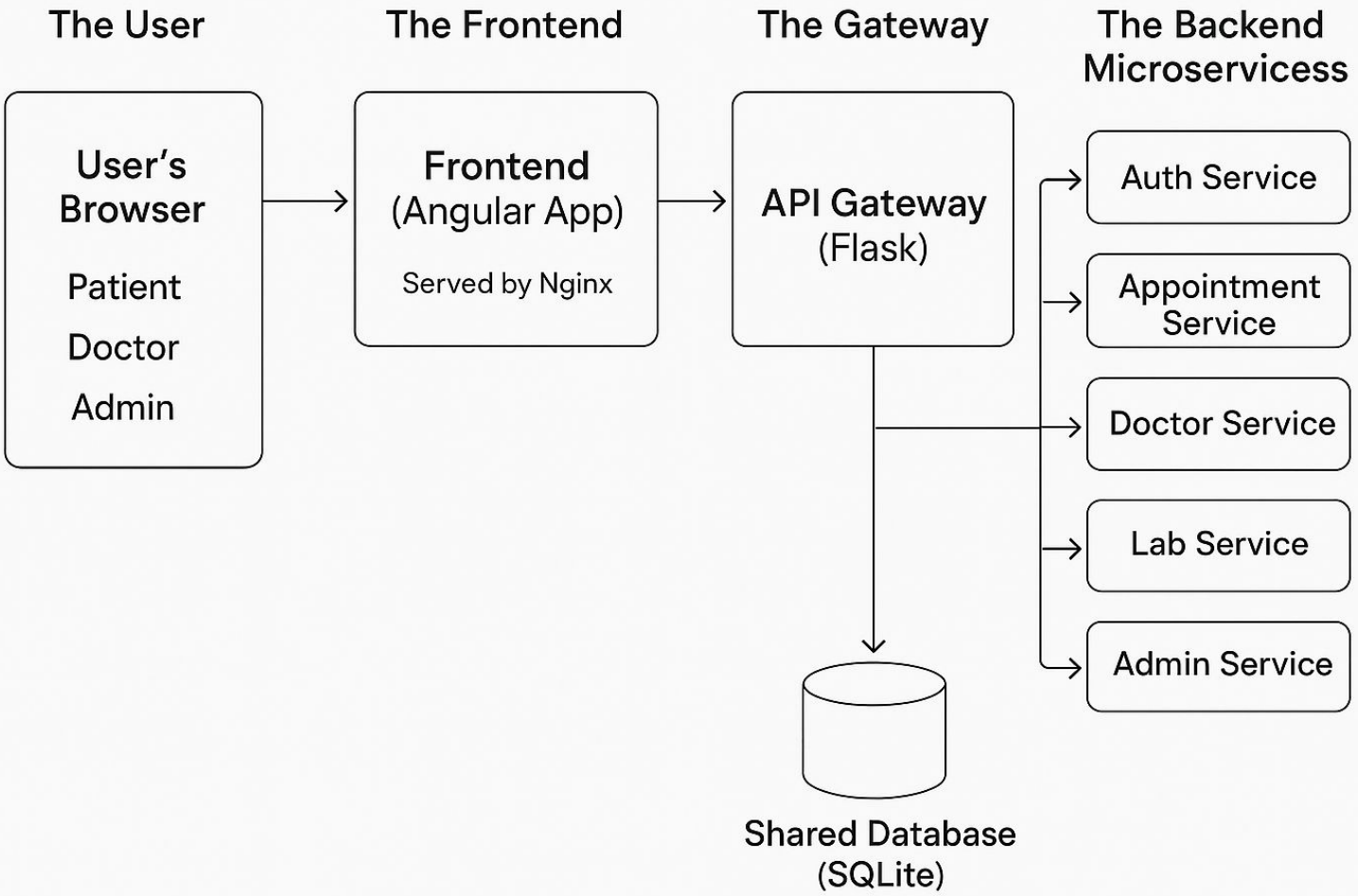
**Key Advantages:**

- **Scalability:** Each service (e.g., appointments, labs) can be scaled independently based on demand.
- **Maintainability:** It's easier to update, fix, and manage smaller, focused services.
- **Technology Flexibility:** Different services could potentially be written in different technologies if needed.

# Architecture Diagram

Our architecture consists of four main layers:

1. **Frontend (Client):** A single-page application built with Angular that provides the user interface for patients, doctors, and admins.
2. **API Gateway:** A single entry point for all backend requests. It receives requests from the frontend and routes them to the appropriate microservice. This simplifies the frontend code and adds a layer of security.
3. **Backend Microservices:** Five independent Flask services, each handling a specific business domain.
4. **Database:** A single, shared SQLite database that all services connect to, ensuring data consistency.


The User
The Frontend
The Gateway
The Backend Microservicess
User's Browser
Patient
Doctor
Admin
Frontend
(Angular App)
Served by Nginx
API Gateway
(Flask)
Auth Service
Appointment Service
Doctor Service
Lab Service
Admin Service
Shared Database
(SQLite)

# Overview of Each Service & APIs Exposed

### a. Auth Service

This service handles all user authentication and registration.

- **POST** `/auth/register` - Creates a new user account (patient, doctor, or admin).
- **POST** `/auth/login` - Authenticates a user and returns a JWT token.

### b. Appointment Service

This service manages all appointment-related operations for both patients and doctors.

- **GET** `/api/appointments` - Lists appointments for the logged-in patient.
- **POST** `/api/appointments` - Books a new appointment.
- **PUT** `/api/appointments/:id` - Reschedules an existing appointment.
- **PUT** `/api/appointments/:id/cancel` - Marks an appointment as cancelled.
- **GET** `/api/doctor/appointments` - Lists appointments for the logged-in doctor.
- **PUT** `/api/doctor/appointments/:id` - Updates an appointment's status (e.g., Confirmed).

▪

### c. Doctor Service

This service manages doctor profiles, prescriptions, and referrals.

- **GET/PUT** `/api/doctor/profile` - Manages a doctor's professional profile.
- **POST** `/api/doctor/prescriptions` - Creates a new prescription for a patient.
- **POST** `/api/doctor/referrals` - Creates a new patient referral.
- **GET** `/api/doctor/referrals` - Lists patients referred by the logged-in doctor.

### d. Lab Service

This service handles lab test listings and bookings.

- **GET** `/api/labs/tests` - Lists all available lab tests.
- **POST** `/api/labs/bookings` - Books a new lab test for a patient.
- **GET** `/api/labs/bookings` - Lists lab bookings for the logged-in patient.

### e. Admin Service (Part of Doctor Service)

This service provides administrative functionalities.

- **GET** `/api/admin/stats` - Retrieves dashboard statistics (total doctors/patients).
- **GET/POST** `/api/admin/doctors` - Manages the list of all doctors in the system.

.

# Containerization - Docker

Each component of the application is containerized using Docker for consistency and portability.

- **Frontend:** A multi-stage `Dockerfile` is used to build the Angular application and serve it via a lightweight Nginx web server.
- **Backend:** A single, reusable `Dockerfile` is used for all five Python microservices. It intelligently installs the specific dependencies for each service at build time.

This ensures that the application runs the same way on any machine, from a developer's laptop to a production server.

# Deployment - Docker Compose

For local development and deployment, we use a `docker-compose.yml` file to define and run the entire multi-container application with a single command.

- It defines all **six services** (frontend + five backend).
- It maps the container **ports** to the host machine (e.g., `4200` for the frontend, `5000` for the gateway).
- It uses a **volume** to persist the SQLite database file, ensuring data is not lost when containers are restarted.
- It shares the `.env` file with all backend services to provide the `SECRET_KEY`.

**Command to run:** `docker compose up --build`

# Thankyou Everyone

Before I conclude, I would like to express my sincere gratitude. First and foremost, to my mentor, **Praveen Sir**. Sir, your expert guidance and invaluable training have been instrumental throughout this entire process, and I am incredibly thankful for your contribution to my learning and the success of this project.

I also want to extend a warm thank you to everyone in the audience. Thank you for your time and attention today. I hope you found this demonstration insightful.