Medifusion Project Analysis

1. Components & Technologies

Frontend (User Interface)

Role: The visual interface users interact with. It manages state, navigation, and API calls.

* **React**: A JavaScript library for building user interfaces based on components.
* **React Router (`react-router-dom`)**: Handles client-side routing, enabling navigation between pages (e.g., Dashboard, Login) without reloading the browser.
* **TailwindCSS**: A utility-first CSS framework for rapid UI styling directly in class names.
* **Framer Motion**: A library for React that powers the animations and transitions (e.g., the animated title).
* **Axios**: A promise-based HTTP client used to make requests to the Backend API (GET, POST).
* **Leaflet / React-Leaflet**: Used for interactive maps, likely for locating hospitals or services.
* **Lucide React**: Provides the icon set used throughout the application.
* **jsPDF & jspdf-autotable**: Libraries used to generate PDF reports directly in the browser (e.g., prescriptions or history).

Backend (Application Logic)

Role: Processes requests, enforces business rules, manages data, and integrates with AI.

* **FastAPI**: A modern, high-performance web framework for building APIs with Python. It provides automatic interactive documentation (Swagger UI).
* **Uvicorn**: An ASGI web server implementation for Python, used to run the FastAPI application.
* **SQLAlchemy**: The Object Relational Mapper (ORM) that translates Python classes (Models) into SQL queries for the database.
* **Pydantic**: Data validation and settings management using Python type hints. Ensures API request/response data is valid.
* **Celery**: An asynchronous task queue used for background processing (e.g., sending emails, long-running AI tasks) so the UI doesn't freeze.
* **Google Gemini AI (`google-generativeai`)**: The AI model integrated for "Symptom Analysis" and "Disease Detection".
* **FastAPI-Mail**: A library specifically for sending emails from FastAPI applications (used for Welcome emails).
* **Passlib & Bcrypt**: Used for secure password hashing.

Database & Storage

Role: Persistent storage for all application data.

* **PostgreSQL 15**: A powerful, open-source object-relational database system. Stored in the `db` container.
* **Psycopg2-binary**: The PostgreSQL adapter for Python, allowing SQLAlchemy to communicate with the database.

Infrastructure & DevOps

Role: Orchestration, networking, and deployment.

* **Docker & Docker Compose**: Containerization platform. `docker-compose.yml` defines all services (`backend`, `frontend`, `db`, `redis`, etc.) and how they run together.

* **Nginx**: A high-performance web server acting as a **Reverse Proxy**. It sits in front of the application, routing traffic:

* `/api` requests -> Backend container

* All other requests -> Frontend container

* **Redis**: In-memory data store used as a "Message Broker" and "Result Backend" for Celery. It coordinates tasks between the main backend and the worker.

* **RabbitMQ**: A dedicated message broker that receives task messages from the backend and distributes them to Celery workers.

2. Data Storage & Access

Where is data stored?

- **User Data**: Stored in the **PostgreSQL** database container (`medifusion-postgres`).
- **Volume**: Data is persisted in the Docker volume `postgres_data` mapping to `/var/lib/postgresql/data` inside the container.

How to access it?

1. **Via Application**: The Frontend displays data fetched from the Backend API.
2. **Via Database Client**: You can connect to the database running on `localhost:5432`.
   - **Host**: `localhost`
   - **Port**: `5432`
   - **User**: `medifusion_user`
   - **Password**: `securepassword123`
   - **Database**: `medifusion_db`
3. **Via CLI**:

```
    docker exec -it medifusion-postgres psql -U medifusion_user -d medifusion_db
```

3. Program Flow

General Flow

1. **Entry Point**: User accesses `http://localhost`. Nginx serves the Frontend.
2. **Navigation**: React Router (`App.js`) handles client-side routing.
   - **Public Pages**: `/heart-health` (and Login/Signup).
   - **Protected Pages**: `/` (Dashboard), `/profile`, `/history`, `/medicines`, etc. require a valid JWT token.

Authentication Flow

1. **Signup/Login**: User submits form -> Frontend sends POST to `/api/signup` or `/api/login` (Note: Issue identified in `api.js` using incorrect path).
2. **Token Issuance**: Backend verifies credentials via `backend/routers/auth.py`. Returns `access_token`

(JWT).

3. **Session state**: Frontend stores token (likely in Context/LocalStorage) and includes it in the `Authorization` header for subsequent requests (`Bearer <token>`).

Feature Flows

- **Symptom Analysis**: User uploads file/text -> API `/api/analyze` (or similar) -> Backend processes with Gemini AI -> Result returned.
- **Medicine Tracking**: User adds medicine -> API POST -> Stored in `medicines` table -> Displayed on Dashboard/Medicine Tracker.

4. Common Errors & Challenges

Based on the project structure and common issues encountered in such setups:

1. **Network Error (Signup/Login)**:
   - **Cause**: The Frontend `api.js` is configured to hit `http://localhost:8000/login`, but the Backend router is mounted at `/api/login`.
   - **Fix**: Update `API_BASE_URL` or endpoints in `api.js` to include the `/api` prefix.
2. **Localhost Access / Deployment**:
   - **Cause**: Docker container communication issues or Nginx misconfiguration. Nginx must correctly proxy `http://frontend:3000` and `http://backend:8000`.
   - **Observation**: Recent context suggests debugging why `localhost` wasn't accessible, often due to port mapping or container health checks (e.g., Database taking too long to start).
3. **CORS & Routing**:
   - **Cause**: `localhost` vs container names (`backend`, `frontend`). Browser runs on host, so it hits `localhost`. Internal containers talk via service names. Nginx bridges this.