Software Requirements and Design Document

- Increment 2 -
Group #2

Authors :
Daivion Brooks
Brooks Berry
Jason Lee
Nadav Peper
Nandagopal Nair

1. Overview (5 points)

Database

- *Runs through Docker*
- *Yml file specifies the services such as volumes, environment, and ports*
- *SQL Schema - includes user profile table, authentication table, and medical data tables such as labs, symptoms, diet, and treatments. All tables include fields that we deemed necessary for the initial foundation of the app.*
- *Models implemented in backend to match the schema*
- *Enabled alembic through flask so that schema changes can be handled through migrations instead of raw SQL*

Backend

- *Runs through Flask*
- *Written in Python*
- *Flask app initialization*
- *Models - align with database schema*
- *Routes - Modular CRUD implementation for database fields - routes directory*
- *Secure authentication with JWT tokens and hashed password using Werkzeug library*

Frontend

- *Runs through React (requires Node.js)*
- *Written in TypeScript*
- *React components configured to Vite with tailwind CSS*
- *Sign up, Home, and Profile pages created with full integration with the backend API/database*

- *Treatments, symptoms, food logs frontend skeleton pages created and designed*

2. Functional Requirements (10 points)

Functional requirements by priority

- *(high) CRUD implementation for treatments, symptoms, medications, doctors, etc. Displayed via frontend on different pages in the app*
- *(low)Data should be able to be read on an embedded calendar. For example, they should see data such as upcoming treatments/appointments, or their medication schedule*
- *Embedded Calendar will have a single page in the app, where the user is able to read different fields of data that are paired with a specific date*
- *(low)Should be able to filter which fields you can see. Freedom to display all together or select any number of types*
- *(medium)Data should also be read on other pages via list view*
- *(low)Filter feature to display only specified data (upcoming/completed treatments, daily symptoms, specific lab entries, appointments, etc.)*
- *(low)Graphs using Numpy library to show trends for measurable data*
- *(medium)Web scrape to find doctors in the area*
- *(medium)Web scrape for research related to user condition/illness*
- *(high)Scheduled Push notifications for treatment reminders*

- *(low)Feature to export specified user data to a pdf file*
- *(medium) implement RBAC*
- *user types would include patients and doctors*
- *(low)Potentially implement messaging between doctors and patients with push notifications*
- *(high) Pages/Views for all backend features*
- *(high) Symptoms data CRUD: users will log the date, user notes, and user rating (how do you feel out of 10) (for numerical tracking)*
- *(medium) Treatments: user will log scheduled treatments as well as completed treatments. They will store the date, notes, and completion flag. Push notifications for approaching date - (high) Lab data (for example, rbc count, blood pressure, etc.):*
- *(high)User data: user can update profile information such as email, age, weight, medications, insurance, picture, etc.*

3. Non-functional Requirements (10 points)

Security*: All database queries must be secure and protected against SQL injection and unauthorized access. User credentials and personal health data should be stored safely using encryption and secure authentication methods.*

Maintainability*: The codebase should remain clean, modular, and well-documented so that all team members can easily update, debug, and extend system functionality in future increments.*

Performance*: API endpoints should respond within 1–2 seconds under normal usage conditions to ensure a smooth and efficient user experience.*

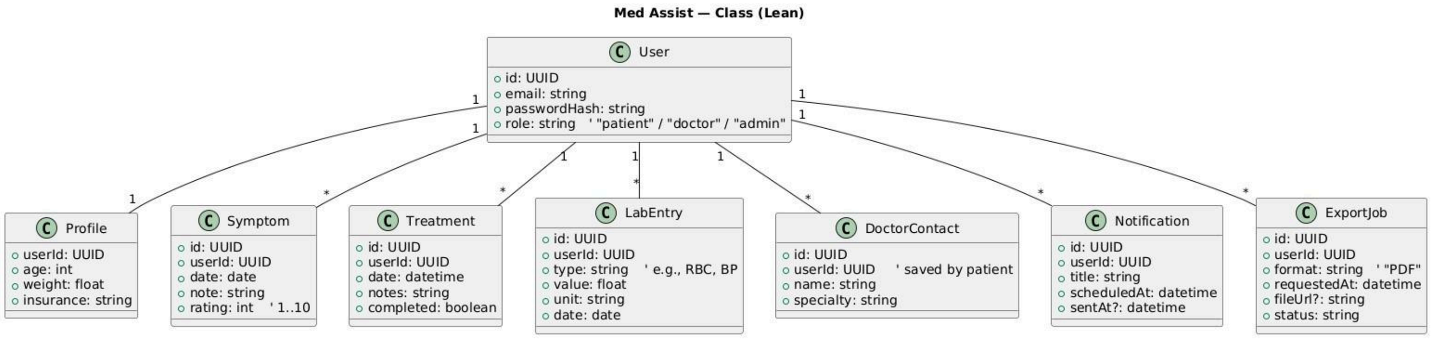
Usability*: The interface should be intuitive and accessible, allowing users to easily navigate and log their symptoms, treatments, and food data without confusion.*

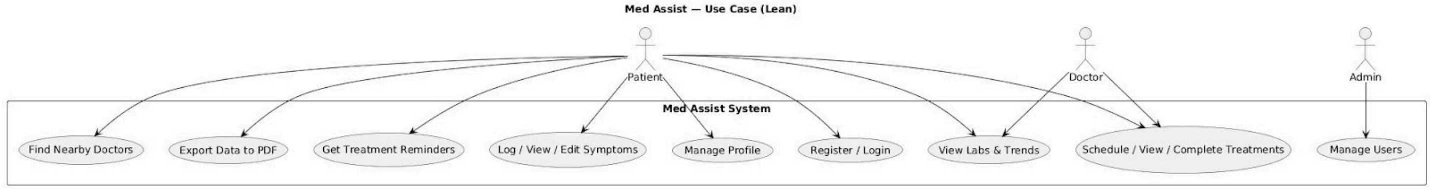
Reliability*: The system should consistently store, update, and retrieve data accurately across all sessions. Data integrity must be maintained after database migrations or updates to prevent information loss.*

Scalability: *The system should be designed to handle an increasing number of users and larger volumes of data without a significant drop in performance.*

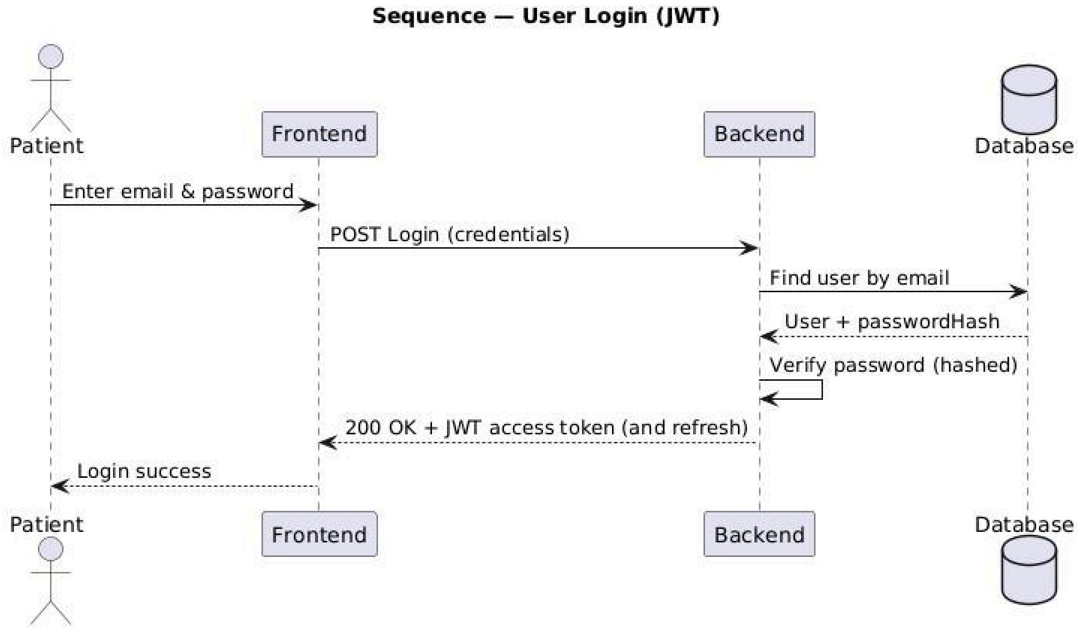
Portability*: The application should be compatible across different operating systems and web browsers, ensuring users can access the medical diary from various devices.*

4. Use Case Diagram (10 points)

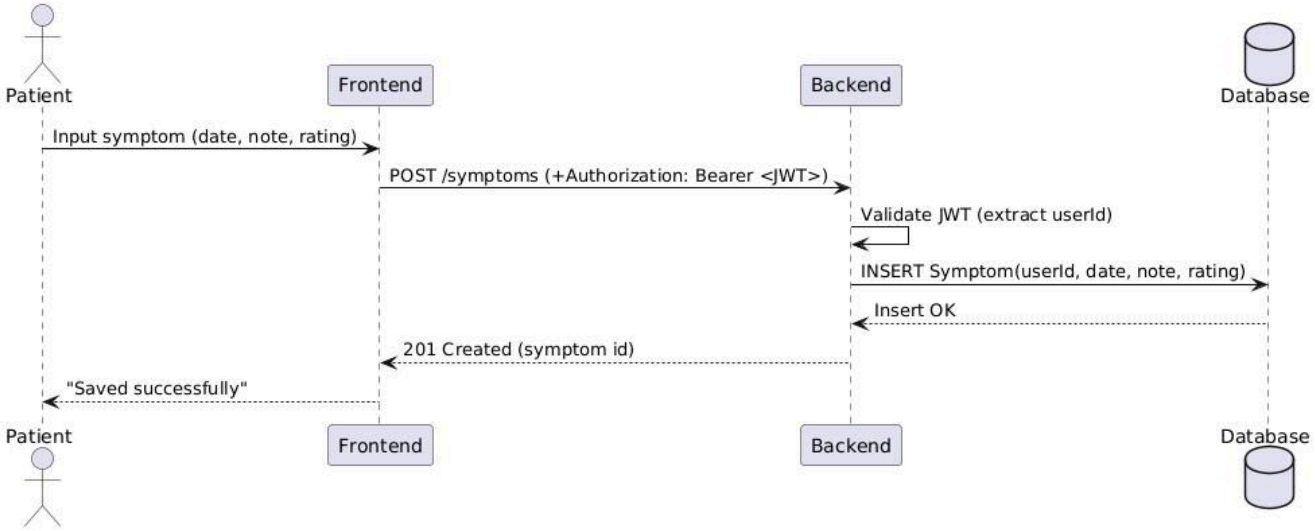




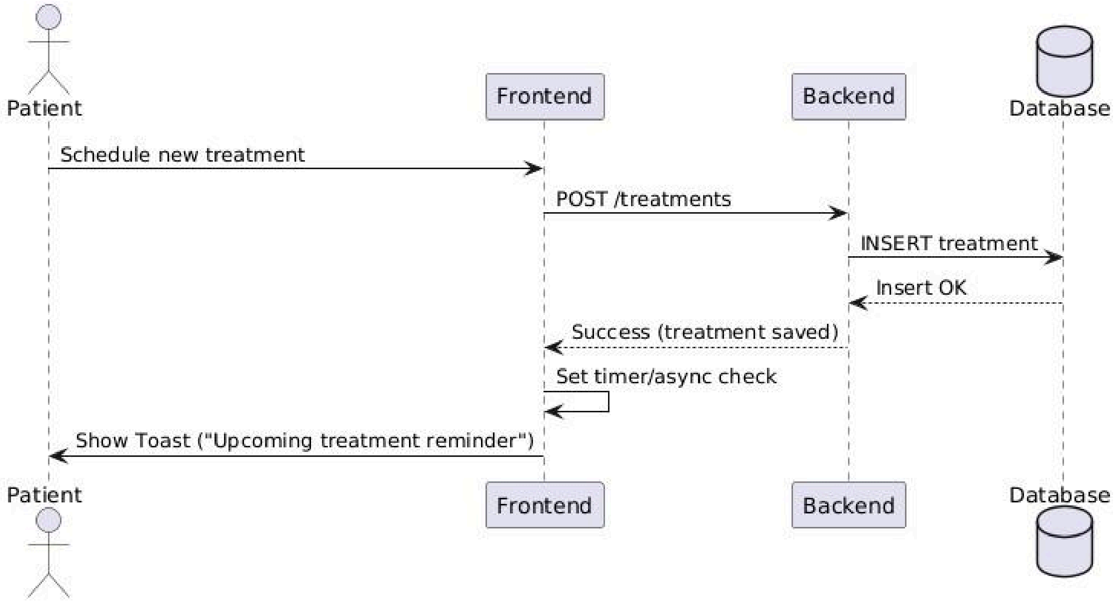
5. Class Diagram and/or Sequence Diagrams (15 points)



Sequence — Log Symptom (Authenticated)


Patient
Frontend
Backend
Database
Input symptom (date, note, rating)
POST /symptoms (+Authorization: Bearer <JWT>)
Validate JWT (extract userId)
INSERT Symptom(userId, date, note, rating)
Insert OK
201 Created (symptom id)
"Saved successfully"
Patient
Frontend
Backend
Database

Sequence — Treatment Reminder (UI Toast)



6. Operating Environment (5 points)

Hardware Platform: *The frontend can run on standard personal computers (Windows, macOS, Linux) currently. The backend will operate on a server environment capable of running Docker containers.*

Operating System: *The frontend is platform independent and only requires a supported browser. The backend can be deployed on any OS that supports Docker. Development was performed primarily on Windows and macOS machines.*

Web Browser Support: *The application is compatible with major browsers including Google Chrome, Microsoft Edge, and Safari.*

7. Assumptions and Dependencies (5 points)

Assumptions

Team members will continue using Docker to keep development environments consistent across machines.

The database schema will not undergo major changes beyond small adjustments (such as adding columns) during this increment.

The backend and frontend will communicate through JSON, and all future features will follow that.

Manual testing with Postman and browser-based checks is needed.

Users are expected to input their own data (symptoms, treatments) directly, since integration with external medical systems is not planned for this increment.
Each user's data is stored securely and privately, accessible only to their account.

Internet connectivity is required for accessing and saving data through the web application.

The application assumes users will have basic technical knowledge to navigate forms and input fields correctly.

Any data entered by users (like symptoms or food logs) is assumed to be accurate and self-reported.

Dependencies

The system depends on external libraries/frameworks including Flask, Marshmallow, JWT, React/TypeScript, and PostgreSQL.

The project relies on Docker for containerization and environment consistency.

GitHub is required for version control and collaboration.

The future "Find a Doctor" feature will rely on a third-party API, which may cause some usage limits or require a lot more configuration.

The system relies on SQLAlchemy as an ORM to interact with PostgreSQL database and Alembic to handle database schema migrations.