

College of Engineering and Computing
Florida International University
Knight Foundation School of Computing and Information Sciences

MyMedCab – Medication Management System

CIS 4951 – Project Documentation

Instructor: S. Masoud Sadjadi

Term: Fall 2025

Team Members: Josue Quinonez, Cesar Calzadilla, Daniel (Niel) Escobar, Francine Portobanco,

Kendrick Belizaire

Date: December 1st, 2025

Table of Contents

1.	Executive Summary.....	3
2.	Problem and Requirements.....	4
	a. Problem Description	
	b. Users and Needs	
	c. Functional Requirements	
	d. Non-functional Requirements	
	e. Constraints	
	f. Success Criteria	
3.	System Overview and Architecture.....	7
	a. High Level Architecture	
	b. Architecture Diagram	
	c. Technologies and Frameworks	
	d. Major Components	
	e. API Overview	
	f. Database and Storage	
	g. Architectural Scope	
4.	Discipline-Specific Depth.....	13
	a. Algorithms and Data Structures	
	b. Architecture and Patterns	
	c. Engineering Evidence and Performance	
	d. Technical Depth Recapitulation	
5.	Implementation Notes.....	16
	a. Back-end Architecture and Environmental Configuration	
	b. AWS Integration and Security	
	c. PostgreSQL Schema and Access	
	d. Front-end Implementation	
	e. Testing and Debugging	
	f. Implementation Notes Summary	
6.	Testing and Evaluation.....	21
	a. Testing Approach	
	b. Smoke Testing	
	c. Functional Testing	
	d. API Tests	
	e. Database Testing and Verification	
	f. Usability Tests	
	g. Evaluation for Requirements	
	h. Summary	
7.	Security, Privacy, Accessibility, and Compliance.....	26
	a. Security	
	b. Privacy	
	c. Accessibility	
	d. Compliance	
	e. Summary	
8.	Deployment and Operations.....	30
	a. Deployment Architecture	
	b. AWS RDS Deployment and Configuration	
	c. Back-end Deployment and Configuration	
	d. Front-end Deployment and Configuration	
	e. Database Operations and Maintenance	
	f. Deployment Conflicts and Resolutions	
	g. Reproducing for Additional Developers	
	h. Recapitulation	
9.	Limitations and Future Work.....	34
	a. Limitations	
	b. Future Work	
10.	References.....	37
A.	Appendices	
	a. Appendix A – Entity Relationship Diagram.....	38
	b. Appendix B – Sequence Diagram.....	39
	c. Appendix C – User Stories.....	40
	d. Appendix D – GET API Call to Retrieve Doctors.....	42
	e. Appendix E – SQL Schema: Tables, Views, Triggers.....	43
	f. Appendix F – Patient Side.....	44
	g. Appendix G – Physician Side.....	45

1 - Executive Summary

MyMedCab is a full-stack medication management system that is designed to improve engagement amongst all patients, enhance physician workflows, and improve clinical communication between patients and physicians. This system gives patients a “medicine cabinet” such that patients can view medications, stock levels, refill dates, and prescriptions in an organized behavior. Moreover, physicians are provided with their own dashboard in which they can prescribe medications and observe treatment statistics over time.

This project is composed of a front-end built on a multi-tier architecture with a front-end powered by React and TypeScript, a back-end running on Node.js / Express, and a database system from PostgreSQL 17 that is securely hosted by Amazon RDS. Key features include user authentication, patient creation upon signing in, role-based database views for access within the application, a prescription builder dedicated to physicians, prescriptions that are organized in a clean interface, and a pharmacy lookup system implemented in the form of a map. During the implementation of the project, our team operated under Agile methodologies (Scrum), focused on the implementation of user stories while building our database, API integration, and user interfaces.

The goal of MyMedCab is to provide the healthcare industry with a tool that grants accessibility, transparency, and patient-centered options. As patients have clarity over their medications and physicians continue to manage treatments, the system eliminates any miscommunication, raises awareness, and provides safer clinical care. The purpose of this documentation is to discuss our system’s requirements, architecture, development, testing, and potential enhancements.

2 - Problem and Requirements

2.1 – Problem Description

The mismanagement of medications is one of the most common issues that patients are facing today, even for those managing multiple prescriptions. Patients seem to lose track of various components such as medication dosages, refill dates, and active prescriptions. This results in a lack of awareness and risks in developing health complications. Healthcare providers are also struggling with communication gaps when updating prescriptions, especially when patients do not have access to a unified and centralized record.

MyMedCab solves these disparities by provided a digital platform allowing patients to easily view their medications, monitor stock levels, and view prescription details. Physicians can prescribe, update, or manage treatments as needed. Through controlling medication data and providing both entities with a unique interface, MyMedCab prevents confusion, enhances mindfulness, and improves communication among patients and providers.

2.2 Users and Needs

2.2.1 Primary Users

A. Patients

- Require accessibility to track and view their medications.
- Require visibility of prescription details and dosages.
- Require ease of access to view pharmacy locations.

B. Physicians

- Require a simple, yet sophisticated manner to prescribe medications
- Require quick access to patient information and medication history.
- Require a secure system to prevent data loss or manipulation.

- Require streamline creativity and updates for patient medication.

2.3 Functional Requirements

The system must support the following components:

2.3.1 Authentication

- The system must allow the patients to sign in with their name, date of birth, and patient ID.
- The system must allow for the auto-creation of patient records if the user does not yet exist.
- The system must allow the physician to sign in with the following credentials: name, doctor ID, and specialty.

2.3.2 Patient Features

- Patients need to view any and all medications assigned to them.
- Patients need to view their refill dates and prescription number.
- Patients must be able to access pharmacy locations.

2.3.3 Physician Features

- Physicians must be able to create any new prescriptions.
- Physicians must update or delete existing prescriptions (i.e. dosage, refill date, etc.)
- Physicians must assign a patient with a medication.
- Physician updates must flow within the database triggers to prevent any unauthorized edits.

2.3.4 Back-end and Database

- The system is required to store all the data on AWS RDS and PostgreSQL.
- The system must reinforce role-based DB views for data access.

- Triggers are required to handle in the user_medicine table without revealing table structures to clients.
- API endpoints need to return JSON responses.

2.3.5 Pharmacy Locations

- The system must receive all pharmacy locations from the pharmacy table.
- The system must display the pharmacies through a map interface.

2.4 Non-Functional Requirements

The system must support the following components:

2.4.1 Performance Requirements

- The system must return dashboard data under a normal load.
- Database queries must be optimized through indexing and keying.

2.4.2 Security

- Amazon RDS restricts inbound connections through IP addressing.
- All sensitive routes must be protected based on user roles.
- Direct table modifications or writes are prohibited for all patients.

2.4.3 Usability

- Interface is required to be clean, readable, and user-friendly.
- Medication containers need to be intuitive and appealing.

2.4.4 Maintainability

- The code is required to be organized into groups (e.g. routes).
- Database ownership and permissions are up-to-date (i.e. “josue” = owner, other users = granted access).

2.4.5 Reliability

- The system needs to be stable even if the patient database is expanded.
- All backups must be possible through pgAdmin .sql dumps.

2.5 Constraints

AWS RDS runs on an IP whitelisting system – if the correct IP address is not given, the application may fail. Additionally, PostgreSQL 17 uses the correct port configuration, specifically 5433. Depending on the operating system being used (e.g. macOS and Windows), all node.js must be compatible.

2.6 Acceptance Criteria

The application is successful if it can perform the following things:

- Patients can sign in and view their medication cabinet.
- Physicians can create or update prescriptions as needed.
- Database access permissions prohibit any unauthorized modifications.
- System displays all pharmacy locations on a map.
- The system is successfully tested (e.g. smoke test).
- User stories and acceptance criteria is met.

3 - System Overview and Architecture

MyMedCab is constructed using a three-tier architectural system where the front-end runs on React and TypeScript, an API server running on Express and Node.js, and a relational database system running on PostgreSQL 17, hosted on AWS RDS. The system is primarily

designed to support two types of entities – patients and physicians – whilst enforcing role-based permissions and data processing. The architecture strives for durability and sustainability.

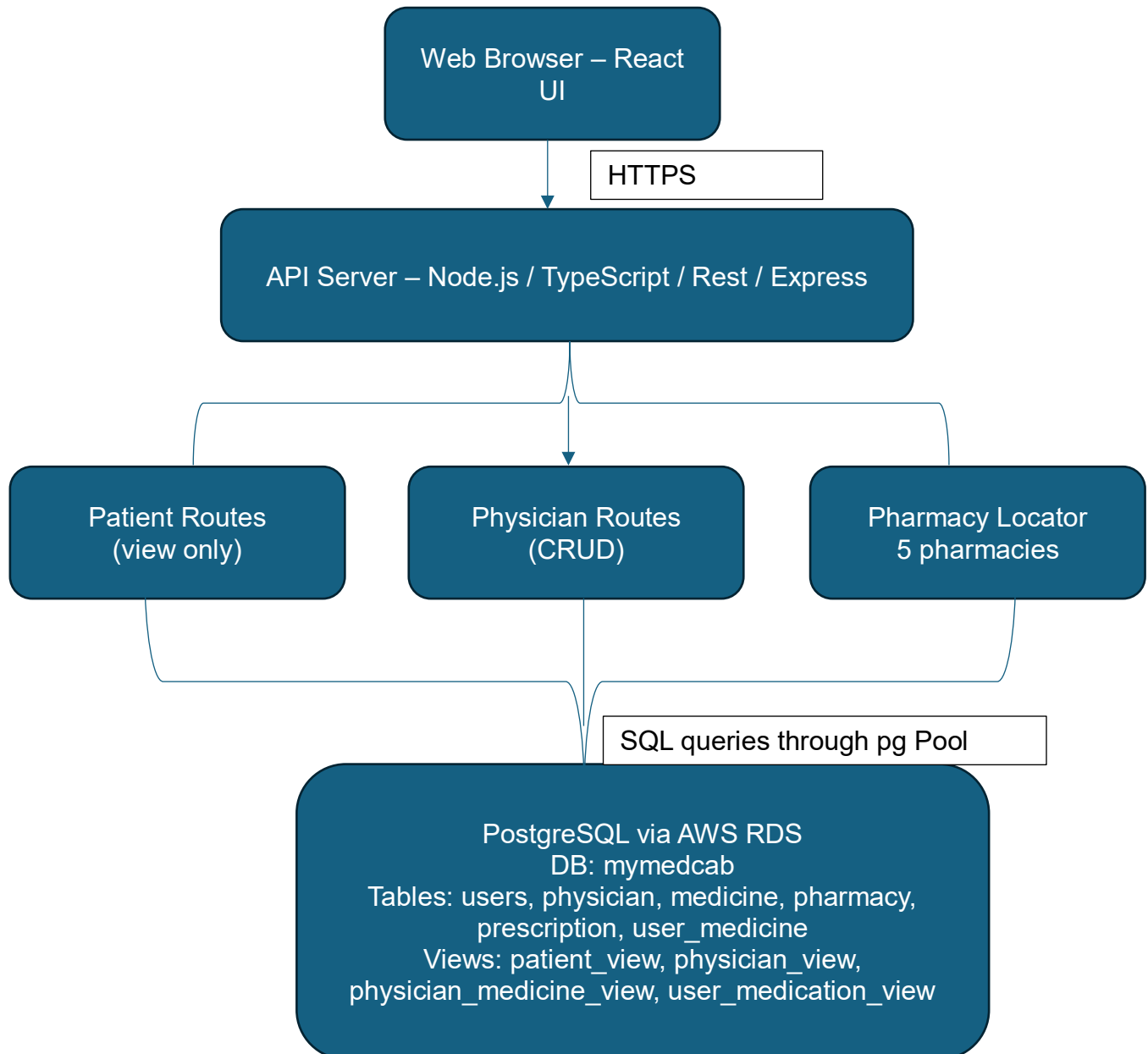
3.1 High-Level Architecture

- This system follows a multi-tier client-server architecture.
 - Front-End Layer – React and TypeScript:
 - Displays interfaces for the patient log in, physician log-in, dashboards, medication cabinet containers, prescription builder, and the pharmacy locator.
 - Communication with the back-end is handled using RESTful API endpoints.
 - Routing and component states are handled using React Router.
 - API Layer – Express and NodeJS:
 - This section behaves as the moderator between the client and database.
 - Controls authentication, route coverage, patient creation logic, prescription operations (i.e. for the physician), and medication management.
 - Develops all modular routes – authRoutes, patientMedicineRoutes, physicianRoutes – to control readability and maintenance.
 - Data Layer (PostgreSQL via Amazon RDS):
 - All data such as patients, physicians, medicines, prescriptions, pharmacies, and user-medicine relationships is stored.
 - Contains database views that enforce role-based access (e.g. patient_view, physician_view, etc.).

- Makes use of triggers, sequences, and any procedures to ensure insertions and modifications.

This multi-tier structure ensures that UI, logic, and operations are clean and well-developed, promoting consistent development and ease in debugging.

3.2 Architecture Diagram



3.3 Technologies and Frameworks

The following technologies were chosen because of their scalability, continuous support, and correspondence in modern web design.

End	Technology	Reasoning
Front-End	React and TypeScript	Handles UI, routing, and state management
Back-End	Node.js and Express	RESTful API, logic, authentication
Database	PostgreSQL 17 on Amazon RDS	Storage, relations, sequences, and views
Implementation	AWS RDS / Localhost DEV Environment	Popular remote database in addition to local front-end and back-end
Tools	pgAdmin, Vite, npm, VS Code	Developing, testing, and database administration

3.4 Major Components

3.4.1 Authentication

- Patients and physicians are allowed to sign in.
- New patients not in the database are automatically created.
- Each entity is granted a specific permission to the views and data.

3.4.2 Patient Dashboard

- Displays patient information and physician information.

- Shows a medication list with medicines prescribed to a user.
- Provides clear medication cards showing the dosage, type, stock level, and refill dates.

3.4.3 Physician Dashboard

- Physicians can create prescriptions with the prescription builder.
- Patient-medication relationships are updated with SQL triggers.
- Provides the ability to edit stock, dosages, and refilling.

3.4.4 Pharmacy Locations

- Integrates mapping APIs to help patients find nearby locations.
- All locations are stored in the pharmacy table.

3.5 API Overview

The API uses REST conventions where endpoints are grouped by role.

3.5.1 Authentication Routes

- POST /login: Validation for patient and physician login.
- Patient is auto created if needed.

3.5.2 Patient Routes

- GET /:patient_id: Obtains all the medications for the patient.

3.5.3 Physician Routes

- POST /prescriptions: Adds a new prescription.
- PUT /prescriptions/:id: Updates patient medication information.

3.5.4 General Routes

- GET /pharmacies – Returns a list of available pharmacy locations.

3.6 Database and Storage

The PostgreSQL schema contains a set of tables and views as follows.

3.6.1 Tables

- Users – stores patient records.
- Physician – stores doctor records.
- Medicine – stores the medication list.
- User_medicine – stores the patient and medicine relationships.
- Prescription: stores prescription details using foreign keys.
- Pharmacy: stores location data.

3.6.2 Views

- Patient_view – serves as read-only patient information.
- Physician_view – serves as an editable view for the physicians.
- User_medication_view – serves as a view for patient's medicines.
- Physician_medicine_view – serves as a view for internal prescriptions used in triggers.

3.6.3 Triggers

- Triggers are used such that physicians can modify any medication data without having to directly edit any tables.

3.7 Architectural Scope

Views and triggers allow for security and role-based editing depending on the entity accessing the application. Tools such as React and Express promote the individual implementation of the front-end and back-end. With a database host like AWS RDS, it ensures availability, backups, and continuous remote access. All design principles are improving using modularity.

The goal of this section is to highlight all algorithmic and architectural principles that emphasize the technicality behind MyMedCab. This project utilizes databased-centered algorithms, React management patterns, concurrent workflows, and distributed-system tools with Amazon RDS. The implementation motivates principles seen in software engineering, database systems, and full-stack architecture.

4.1 Algorithms and Data Structures

4.1.1 SQL View Accessibility Algorithm

MyMedCab implements a view-driven accessibility algorithm that enforces the separation of patient and physician privileges. Rather than just using back-end logic, our system uses PostgreSQL views and triggers that can restrict patient access to read-only views, allow physicians to perform CRUD operations on prescriptions through modifiable views, and prevent any accidental or harmful writes to tables. The main approach was to use a data abstraction structure with the goal of promoting correctness and consistency from a database perspective regardless of the front-end or back-end.

4.1.2 Trigger Update Algorithm

To handle physician operations, updates are not targeting the tables directly, and instead, the system uses an algorithm with INSTEAD OF UPDATE triggers and INSTEAD OF INSERT triggers. With these triggers, fields are mapped to the correct tables (e.g. user_medicine / users) and data is preserved. The observed time complexity for each operation is $O(1)$ since they operate on a single record that is identified by a primary key. This approach reduces database corruption and promotes consistency.

4.1.3 Data Structures

There are various key data structures used within the system as listed below. They were select to handle performance and robustness.

- Relational database tables to normalize storage, specifically under 3NF.
- A foreign key graph that links patients amongst prescriptions, medicines, and physicians, and vice-versa.
- React component trees to render dashboards.
- State hookers and reducers that manage the user interface state.
- JSON objects for API payloads.

4.2 Architecture and Patterns

4.2.1 Three-Tier-Architecture

MyMedCab uses a typical multi-tier web architecture consisting of a UI layer that was implemented with React and TypeScript, the logical layer which was handled using Express and Node.js, and a data layer using PostgreSQL on RDS. This structure achieves design principles, debugging, and collaborative development within team members.

4.2.2 MVC-Based Back-end

While the back-end is not entirely MVC, it uses a hybrid architecture where the routes define the endpoints, controllers store the logic, and queries handle interfaces with PostgreSQL, and as a result, the pattern supports the separation of responsibilities.

4.2.3 Concurrency

Since Amazon RDS is a resource that is shared and accessed on multiple applications, the system needs to be safe from race conditions. With PostgreSQL, services such as row-level locking, transaction isolation, and atomic triggers are all built in. The use of INSTEAD OF

triggers for physician CRUD operations promotes atomic updates which stops race conditions from multiple physicians or a physician and a patient doing updates at once.

4.2.4 React State Managing

In the physician and patient dashboards, the React components use tools such as “useState”, “useEffect”, conditional rendering, and component memorization, all of which minimize any unnecessary re-renders, ensuring that the UI is responsive even if large medication lists are being fetched.

4.3 Engineering Evidence and Performance

4.3.1 API Performance

During the development, the following benchmarks were observed. The average API response times were between 130 – 180 ms, with a maximum response time of approximately 260 ms. There was also a consistent performance with multiple medication records.

4.3.2 Database Query Efficiency

Database operations were handled with primary keys for all tables, indexing foreign keys such as on patient_id, med_id, and doctor_id, and ensuring views are narrow. Typical query response times vary on the operation. For example, SELECT from views averaged between 1 – 8 ms, INSERT into user_medicine took less than about five ms, and UPDATE on triggers ranged between 5 – 10 ms. The queries result in a constant or logarithmic performance based on table index look-ups.

4.3.3 Scalability

Since the database is being hosted on AWS RDS, the system adapts verticalization, horizontal scalability due to connection pooling, and managed storage with automatic backups.

The Node.js incorporates a connection pool merging with concurrent requests and improves throughput without having to overwhelm the instance.

4.3.4 Reproducibility

Any and all testing performance goals are reproduced by duplicating the repository, setting up a .env with RDS credentials, running npm run dev on both the front-end and back-end, performing smoke tests, and querying pgAdmin for view or trigger analysis, ensuring consistency for any future developers.

4.4 Technical Depth Recapitulation

MyMedCab is an example of discipline-specific depth by using relational database theory, view accessibility, trigger algorithms, a multi-tier system, React component architecture, role-based security, and AWS RDS networking. Combining these components demonstrates the knowledge of advanced topics in computing as well as full-stack development.

5 - Implementation

Implementation Notes

The development of MyMedCab required contributions on the front-end, back-end, and database layers with a focus on role-based permissions, data manipulation, and incorporation through AWS RDS. The goal of this section is to highlight all the decisions, obstacles, and implementation details that established the final system.

5.1 Back-end Architecture and Environmental Configuration

5.1.1 Node and Express Setup

The API in the back-end section was created with Node.js, Express, and TypeScript. The project was structured into multiple files. The use of modular architecture enhances maintenance:

- “authRoutes.ts” handles the API endpoints, “components” is for the handler logic, and “db.ts” serves as the PostgreSQL connection.

5.1.2 Environment Variables

A key component was handling environments due to the system running on Amazon RDS. As a result, a .env file is needed to manage:

```
1 DB_HOST
2 DB_USER
3 DB_PASSWORD
4 DB_NAME
5 DB_PORT
6 DB_SSL
```

In macOS, a major issue that was denoted was how Node binaries were quarantined if the project was moved over to other machines with errors such as – bad interpreter: Operation not permitted, however, these issues were resolved through removing attributes and re-configuring dependencies. Moreover, AWS RDS uses SSL for inbound connections. The back-end connection pool uses the following. “ssl: { rejectUnauthorized: false}, resolving the “server does not support SSL connections” error.

5.2 AWS RDS Integration and Security

5.2.1 Port Configuration

The default port for PostgreSQL is 5432, however, for the RDS instance, it was configured to port 5433. This required that the .env file utilized a database port argument of 5433, ensuring that pgAdmin , the back-end, and all testing scripts correlated with the port, and inbound rules in the RDS security group granted access on port 5433.

5.2.2 IP Access

By default, AWS RDS will strictly block any unauthorized IP's. For every developmental machine used, each public IP was manually added into the security group. Failure in doing so generated backend issues with a common one being “connect ETIMEDOUT <IP>:5433”.

5.2.3 Database Ownership

All database tables, views, triggers, and functions were granted to the main PostgreSQL role:

```
ALTER TABLE <table> OWNER TO josue;
```

```
ALTER VIEW <view> OWNER TO josue;
```

```
ALTER FUNCTION <function> OWNER TO josue;
```

Doing so prevented errors such as: “must be owner of relation <view>”. This allowed the team to update the schema as necessary without having any permission blockages.

5.3 PostgreSQL Schema and Access

5.3.1 SQL Views

Enforcing the separation of roles and data management required multiple views as shown below.

- `patient_view`: A read-only view for patients that contains information.
- `physician_view`: An editable view that allows physicians to edit patient information.
- `user_medication_view`: A view that displays all medications for a given patient.
- `physician_medicine_view`: A view that maps to the back-end for triggers.

5.3.2 Triggers for Data Modification

Physicians must update prescriptions without having to write to low-level tables. This was handled using INSTEAD OF triggers as shown below, and the same applied for `physician_medicine_view`, with the goal of protecting the schema and providing flexibility.

```
CREATE TRIGGER physician_view_update
INSTEAD OF UPDATE ON physician_view
EXECUTE FUNCTION update_physician_view();
```

5.3.3 Sequence Permissions

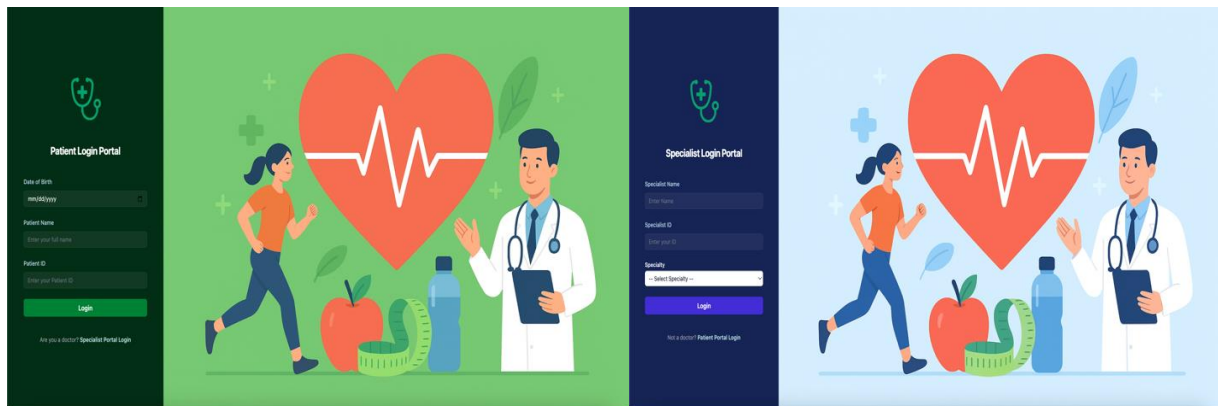
Since the `user_medicine` table requires a serial primary key, the back-end required permissions, which in turn enabled the prescription builder to insert new prescriptions.

```
GRANT USAGE, SELECT, UPDATE
ON SEQUENCE user_medicine_user_med_id_seq TO db_user;
```

5.4 Front-End Implementation

5.4.1 Log-in Components

We implemented two login flows – a patient login and physician login (e.g. see visuals below). If a patient did not exist, the back-end automatically generated a new record.



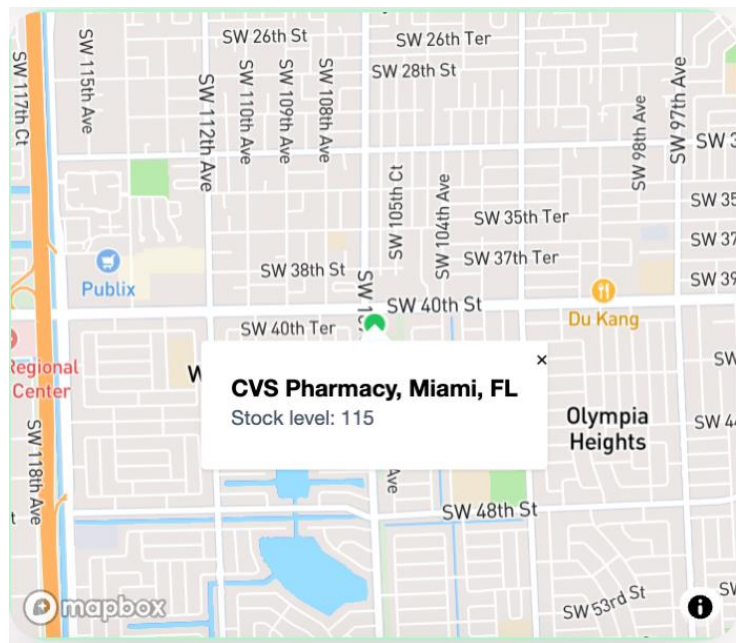
5.4.2 Medication Cabinet

This cabinet was implemented with React components. It displayed the name, dosage, the type of medication, and refill date. Below is an example cabinet for a patient named Alice Johnson.

MEDICINE NAME 	DOSAGE 	FORMULA 	TIMELINE 	PERIOD 	PRESCRIPTION DATE 
Acetylsalicylic Acid	325.00	tablet	1	day	2015-04-13
Gefitinib	250.00	tablet	1	day	2018-07-07
Budesonide-Formoterol	160.00	inhaler	2	day	2018-07-29
Beclomethasone	80.00	inhaler	2	day	2018-05-28

5.4.3 Pharmacy Locator

The pharmacy locator was implemented in the form of a map through Mapbox API. It uses the /pharmacies endpoint and returns the name of the pharmacy, its location, and the stock level of a medication.



5.5 Testing and Debugging

5.5.1 Smoke Testing

A full smoke test was performed before the final deliverable consisting of database connectivity, login routes, dashboard rendering, prescription builder, RDS access based on new IP addresses, and view-based behaviors.

5.5.2 Debugging Challenges

The issues below were made note of and were resolved before delivering the final product.

- Modules losing permissions on the macOS system when project was transferred.
- RDS inbound rules incorrectly configured.
- SSL mismatches.
- Incorrect .env formatting because of spaces / missing .env when testing environment.
- Triggers needing ownership.
- Synchronizing the back-end with the front-end.

5.6 Implementation Notes

- Upon logging in, there exists dynamic patient creation when entering as a new patient not in the database.
 - SQL views and triggers handle role separation.
 - Using a remote database, PostgreSQL, on AWS RDS.
 - Securing workflows for physician prescriptions.
 - Cabinet UI for medication safety and pharmaceutical acquisition from the database using a map integration.
-

6 - Testing & Evaluation

This section highlights the testing approaches used in MyMedCab including but not limited to smoke testing, functional testing, database metrics, API testing, and system verification. In the development process, testing was handled iteratively in addition to multiple

tests that were conducted when completing each user story. The approach used allows for each component to operate individually and with the architecture.

6.1 Testing Approach

The testing methodology was designed upon multiple factors. We begin with smoke testing upon the completion of each user story. This was followed by functional testing to ensure for key workflows, API testing with back-end logs, console results, and Postman API, database testing with pgAdmin and SQL queries, security testing with RDS IP rules and views, and UI testing with React's rendering and input validation tools.

6.2 Smoke Testing

Smoke testing is a methodology in which a build's critical functions are verified and work as intended prior to additional testing. In MyMedCab, we used a smoke test after implementing each feature with the purpose of ensuring each path complied after further changes. Below is the checklist we adapted when testing.

Test	Behavior	Result
Running back-end server	Server running on <code>http://localhost:5001</code> is present	Pass
Loading environment variables	Proper host, port, and SSL configuration output	Pass
RDS connectivity	Connects without SSL / timeouts	Pass
Patient login	Existing patients are successfully authenticated	Pass
Patient auto-creation	Non-existing patients have been created successfully	Pass
Medication cabinet running	Medications rendered in React	Pass
Physician login	Validated physician credentials / accepted	Pass
Prescription builder	Insertion / updating triggers running smoothly	Pass
Pharmacies	Pharmacy entries returned via a map API	Pass

Prohibited access	The wrong role may not access encrypted routes	Pass
-------------------	--	------

6.3 Functional Testing

Functional tests ensured that each feature was running based on the requirements mentioned previously.

6.3.1 Authentication Tests

Patient login was tested with either valid or invalid inputs. The ability to auto-create a patient was successfully verified as every new patient was inserted. Physicians cannot log in as a patient and vice versa.

6.3.2 Patient Dashboard Tests

The patient dashboard was tested by ensuring that the medication cards displayed the correct attributes and confirmed that newly added medications can immediately appear on the medication cabinet.

6.3.3 Physician Dashboard Tests

This feature was tested by creating new prescriptions while ensuring that they were being properly inserted in the user_medicine table, trigger functions called for updated, and all permissions were active. Additionally, any existing prescriptions were edited and tested by validating that all updates showed up on the patient dashboard and preventing unauthorized table writes.

6.3.4 Pharmacy Location Tests

Pharmacy locations required the verification of the /pharmacies endpoint returning all database entries. We also checked that the UI displayed the pharmacy as needed and confirmed that all coordinates were adequate and available for map integration.

6.4 API Tests

All API endpoints were tested with back-end logs, fetch calls, front-end usage from React, and Postman.

6.4.1 Example Tests

- `/api/auth/login`: verified HTTP 200 for a valid login and HTTP 400/500 for invalid input.
- `/api/patient-meds/:id`: Checked for patient medication retrieval.
- `/api/physician/prescriptions`: Validated the successful creation of medications and trigger execution.
- `/api/pharmacies`: Checked that the list of pharmacies returned.

6.5 Database Testing and Verification

6.5.1 SQL View Tests

Each SQL view was tested with the following simple SELECT calls. The views returned the expected fields and correct visibility depending on which view was called.

- `SELECT * FROM patient_view;`
- `SELECT * FROM physician_view;`
- `SELECT * FROM user_medication_view;`

6.5.2 Trigger Tests

We tested all INSERT and UPDATE triggers by conducting physician actions such as inserting or updating via the `physician_medicine_view`. Doing so validated that tables were updated correctly, constraints were met, and no unauthorized writes were allowed.

6.5.3 Sequence Permission Tests

Here, insertion into the `user_medicine` table allowed for the physician to have the proper sequence access. Without it, insertion fails, and with it, the system works properly.

GRANT USAGE, SELECT, UPDATE

ON SEQUENCE user_medicine_user_med_id_seq TO user;

6.5.4 RDS Connectivity

This was tested on multiple machines, whether it be a Windows laptop / PC or macOS laptop / iMac confirming the correct database port, inbound rules, SSL configuration, and IP whitelisting.

6.6 Usability Tests

MyMedCab used a simple approach for usability testing ensuring that the medication cards were readable, having a login process that is easy to use, verified the prescription builder was active, and checked for the responsiveness of the UI. With feedback, the system was easy to understand and provided clear navigation.

6.7 Evaluation for Requirements

The system was checked given the functional and non-functional requirements.

6.7.1 Functional Requirements Satisfied

- Authentication, patient dashboard / medication cabinet, physician prescription insertion and modification, database schema incorporation, pharmacy retrieval.

6.7.2 Non-Functional Requirements Satisfied

- Performance, security, maintenance, reliability, and usability.
 - Performance includes fast queries and efficient views.
 - Security involves addressing AWS RDS restrictions, SSL, and role-based permissions).
 - Maintenance contains routes, triggers, and database ownership.
 - Reliability involves a stable connection and error-handling.

- Usability tries for a clean interface and simple workflows.

6.8 Summary

The overall testing indicates that MyMedCab can successfully operate on all workflows. Any errors that were due to SSL misconfiguration, RDS port mistakes, IP restrictions, and database permissions were fixed which brought about a stable and secure system. Upon concluding smoke testing, we saw that the API routes, triggers, dashboards, and pharmaceutical features ran as intended. As a result, the system demonstrates how the functional and non-functional requirements were satisfied and how problem-solving helped to address the multi-layer architectural approach used.

7 – Security, Privacy, Accessibility, and Compliance

MyMedCab is a healthcare-based application that requires the addressing of security, privacy, accessibility, and compliance concerns. While this project may have been developed for the sole purpose of academia, MyMedCab adapts industrial practices such that data confidentiality, integrity, availability, and security are established. The goal of this section is to highlight all the measures used to protect user data and system reliability, in addition to all the design principles promoting said goals.

7.1 Security

7.1.1 AWS RDS Controls

MyMedCab's PostgreSQL database runs on Amazon RDS, providing various services such as encrypted network transportation, inbound IP allow listing to prevent any unauthorized access, firewalls with AWS Security Groups, and connection validation for all clients.

Throughout the development process, the IP of each team member was required to be whitelisted when establishing a connection, preventing unauthorized machines from going into the patient database.

7.1.2 Database Ownership / Permissions

Preventing unauthorized writes called for strict ownership constraints that were enforced thoroughly. Any and all tables, views, or functions are owned by the main designer, and additional users are granted with privileges such as SELECT on patient views, INSERT / UPDATE on prescription views, and USAGE privileges on certain sequences. The permissions prevented errors such as being the owner of a table or view from occurring while maintaining the fact that unauthorized changes are impossible.

7.1.3 SQL Views and Role Separation

Separating roles between the patients and physicians is handled in the database level. Restricting operations with views and triggers allows that even if the back-end fails, no unauthorized writes are allowed. Patients are granted limited usage while physicians run CRUD.

Users	Read-only	Editable
User_medicine	Read-only	Insert / update
Medicine	View-only	View-only
pharmacy	View-only	View-only

7.1.5 API Security

The back-end, which is built on Express contains input validation which prevents any obstructed JSON, the separation of patient and physician routes, SQL queries, and no string concatenation which prevents SQL injection. Every API endpoint will validate user input to promote the safe execution and block database tampering.

7.2 Privacy

As mentioned, MyMedCab is designed the sole purpose of academia and as it is not associated with HIPAA, the design follows HIPAA principles.

7.2.1 Reduced Data Exposure

Both patients and physicians only see information that entails to them. For instance, patients only view their own medication data whereas physicians only see the patients assigned to them. Neither entity sees any internal ID's when running data linkage.

7.2.2 No Sensitive Data On Client-Side

The front-end does not contain any medical or personal data in localStorage. The data is retrieved by session and cleared when the page is refreshed.

7.2.3 Removing Extraneous Identifiers

Any fields such as systolic, diastolic, and weight were nullable. The goal is to make sure that patient records are available without having to store any unneeded vitals.

7.2.4 Login Flow

The patient login uses unique and identifiable data with entries such as the first name, last name, and date of birth. There is no password storage since no hashing protocols are taken into place, but in the real-world, authentication requires hashed passwords, at least 2FA, and further encryption.

7.3 Accessibility

MyMedCab uses various accessible design decisions that correspond with specific guidelines.

7.3.1 Concise and Simple UI

The React-based front-end incorporates large readable fonts, a high-contrast text on backgrounds, a clear navigation structure, and medication cards that have clean data. This helps reduce cognitive processing, which is crucial, especially for older audiences who may require multiple medications.

7.3.2 Keyboard Navigation

All buttons, cards, and inputs are based on HTML elements which allow for screen reader compatibility, keyboard navigation, and highlighting – major benefits for anyone with motor impairments.

7.4 Compliance

7.4.1 HIPAA – Hypothetical

While not affiliated with HIPAA in any way, shape, or form, MyMedCab focuses on multiple principles such as access control, data minimization, and data transmission. Assuming that this project can be leverage greatly, the system requires protected RDS storage and API's with HTTPS, an audit log to monitor all writes, and risk assessments.

7.4.2 FERPA Accountability

If universities potentially adapt the application for use in student clinics, the privacy safeguards support data protection.

7.5 Summary

MyMedCab uses security, privacy,, accessibility, and compliance-based practices such as AWS RDS IP listing and firewalling, database role permissions, SQL views and triggers to control writes, API logic, an accessible UI design, and minimization practices based on HIPAA. These measures deem MyMedCab as protected and reliable while demonstrating emphasis in the above components.

8 – Deployment & Operations

The deployment and operational workflows for MyMedCab seek to establish a durable and protective environment reserved for development and productive usage. Despite being implemented for the purposes of academia, the choices in architecture are coherent with real-world practices through cloud databases, environmental configuration, and cross-machine support.

8.1 Deployment Architecture

MyMedCab is designed on a hybrid deployment model with a front-end designed with React and Vite. It runs on a local developer machine through Vite's developmental server. The back-end is designed with Node.js, Express, and TypeScript and runs on the Node developmental server. Lastly, the database, being PostgreSQL 17 on AWS RDS is a cloud environment that is also accessible across various machines through IP allowlisting. The architecture allows for cloud persistence, local front-end development, remote database access, and data for all team members.

8.2 AWS RDS Deployment and Configuration

8.2.1 Instance Setup

The PostgreSQL database is being deployed through Amazon RDS and is configured with PostgreSQL 17, a custom port of 5433, public access enabled, and a security group that restricts any inbound traffic.

8.2.2 IP Whitelisting

For proper access, RDS requires IP restrictions where each developer is required to input their public IP in the security group. If there are any failures when updating, there are connection errors such as ETIMEDOUT. Operations include checking IP by simply using WhatIsMyIP, managing inbound rules in the instance, and checking for. The connectivity through pgAdmin or in the back-end log.

8.2.3 SSL

RDS allows for SSL connections and during development, the SSL was disabled because of certification issues. Doing so allows for a stable connection and compatibility.

```
ssl: { rejectUnauthorized: false }
```

8.3 Back-end Deployment and Operations

The back-end side behaves like a local Node service as shown below.

8.3.1 Installation Commands

The commands are “npm install” and “npm run dev” where the server compiles TypeScript with “ts-node-dev” allowing for a reload.

8.3.2 Environment Variables

The back-end variables require operation through a proper .env file as shown below. All developers must use the same .env pattern for concise behavior.

```
PORT=5001
```

```
DB_HOST=RDS Endpoint
```

```
DB_PORT=5433
```

```
DB_USER=user
```

```
DB_PASSWORD=<password>
```

DB_NAME=mymedcab

DB_SSL=false

8.3.3 Operations / Behaviors

The noted behaviors were that it logs connection errors when debugging, it shuts down due to privilege errors, and automatically reconnects if RDS moves between states.

8.4 Front-end Deployment and Operations

For a rapid developmental sequence, the front-end uses Vite.

8.4.1 Launching Frontend

Use the following commands: “npm install” and “npm run dev”. With Vite, services provided include a local server at <http://localhost:5173>, TypeScript integration, and module reloading.

8.4.2 Platform Compatibility

In macOS, challenges faced during deployment were Node.js binary quarantine conflicts, bad interpreter errors, or any file permission resets if copying projects across machines. This was resolved using: “xattr -dr com.apple.quarantine node_modules → rm -rf node_modules && npm install”.

8.5 Database Operations and Maintenance

8.5.1 Backups

In pgAdmin, database backups are created by adhering to the following procedure:

- 1) Select the database and right-click it, then click Backup.
- 2) Format the backup as either Custom or .sql.
- 3) Store backup securely.

- 4) For a reproducible restoration, run “pg_restore -d mymedcab backupfile.dump” or within pgAdmin Restore.

8.5.2 Schema Updates

To change the schema, any modifications require adjustments with the ownership, running ALTER TABLE or ALTER VIEW once again, checking that developers have any updated privileges if necessary, and compiling triggers again if any related tables change.

8.5.3 Supervision

Monitoring simply requires checking for the RDS CPU and memory usage within the AWS console, observing query performance in pgAdmin, and being wary of back-end console logs.

8.6 Deployment Conflicts and Resolutions

One issue that arose was SSL connectivity. This is due to the back-end expecting SSL, however, RDS did not enforce it. To fix, simply add ssl: { rejectUnauthorized: false }. Another issue included macOS “bad interpreter” messages where this was due to any quarantined Node.js binaries. This was resolved by removing the quarantine and simply reinstalling modules. Any permission failures on views or sequences were also common problems which is because of PostgreSQL requirements though this was fixed by granting all the needed permissions to the accounts. One more caveat was that RDS was unreachable because the IP was not whitelisted, however, this was an easy fix since the inbound rules only had to be updated and the back-end was restarted.

8.7 Reproducing for Additional Developers

Any developers who wish to replicate MyMedCab for deployment or operations only has to clone the repository, make a .env file with the proper RDS credentials, running “npm install”

and “npm run dev”, making sure their IP address is implemented into the RDS security group, importing the database schema, and running the front-end and back-end, providing behavior across all environments.

8.8 Recapitulation

Deployment and operations in MyMedCab motivate a variety of principles such as cloud-hosted storage, IP-restricted connections, a clear environment, necessary setup steps, debugging procedures, and a thorough developmental flow.

9 – Limitations and Future Work

While MyMedCab adheres to the project criteria and is an example of a full-stack medical management system, there are multiple limitations that were seen during development. Such constraints are associated with the academic purpose of the project, time constraints, and the complexities seen in designing medical technology.

9.1 Limitations

9.1.1 Limited Authentication / Authorization

Our current implementation takes a simple login flow that considers the name of the patient, date of birth, and their patient ID. Even though this might be sufficient for academia, the methodology lacks a password and multi-factor encryption, a token-based session management, and middleware validation. When using a real healthcare system, a main requirement is strong authentication to protect any private medical information.

9.1.2 Lack of Encryption on API Layer

The back-end and front-end communicate with one another via HTTP and not HTTPS. Database traffic was also accounted for without using a mandatory SSL to simplify the

development. When it comes to production, API traffic needs to be encrypted, RDS must require SSL certification, and sensitive data must be protected in transmission.

9.1.3 Minimal Error Handling / Validation

Despite handling any basic input errors, it lacks multiple things such as a comprehensive server-side validity system, thorough patient or physician messages, API error codes, and middleware when it comes to exception tracking. These issues alone can lead to secret failures in edge-cases.

9.1.4 Minimal UI / UX and Accessibility

The UI may be functional and is somewhat aware when it comes to accessibility, though there are various things missing such as screen-reader components, color-contrast testing, and any adjustable font sizes or additional accessibility modes. Individuals who may have a disability require further navigation support.

9.1.5 No Notification or Reminder System

While patients may be able to view medication and stock levels from a manual perspective, there is no low-stock alert, a remind to refill prescriptions, or a messaging system that allows users and physicians to communicate with one another for any updates on a prescription.

9.1.6 Lack of Audit Logging

When it comes to a real clinical system, changes made by physicians should be logged with factors such as who made the change, the changes made, and when objects were changed. These logs are necessary but were not implemented because of time constraints,

9.2 Future Work

9.2.1 Secure Authentication

A possible future step is to enhance the authentication via hashed passwords or multi-factor authentication systems (2FA) to ensure patient data is protected. Additionally, we would consider using tokens for long-lived logins, all of which ensure the system behaves like a real-clinical system.

9.2.2 Implementation of Health Metrics Page

Another idea is to implement a separate page on the physician dashboard that allows the physician to add or enter patient information specifically blood pressure, blood sugar, and weight. For purposes of this project, we made those fields nullable due to time constraints.

9.2.3 Implementation of Settings Page

Designing a robust settings page can allow users to adjust their account. We would implement a number of factors such as the ability to add emergency contact information, email addresses, in addition to adding a setting that allows users to switch to dark mode for ease.

9.2.4 Side Effects

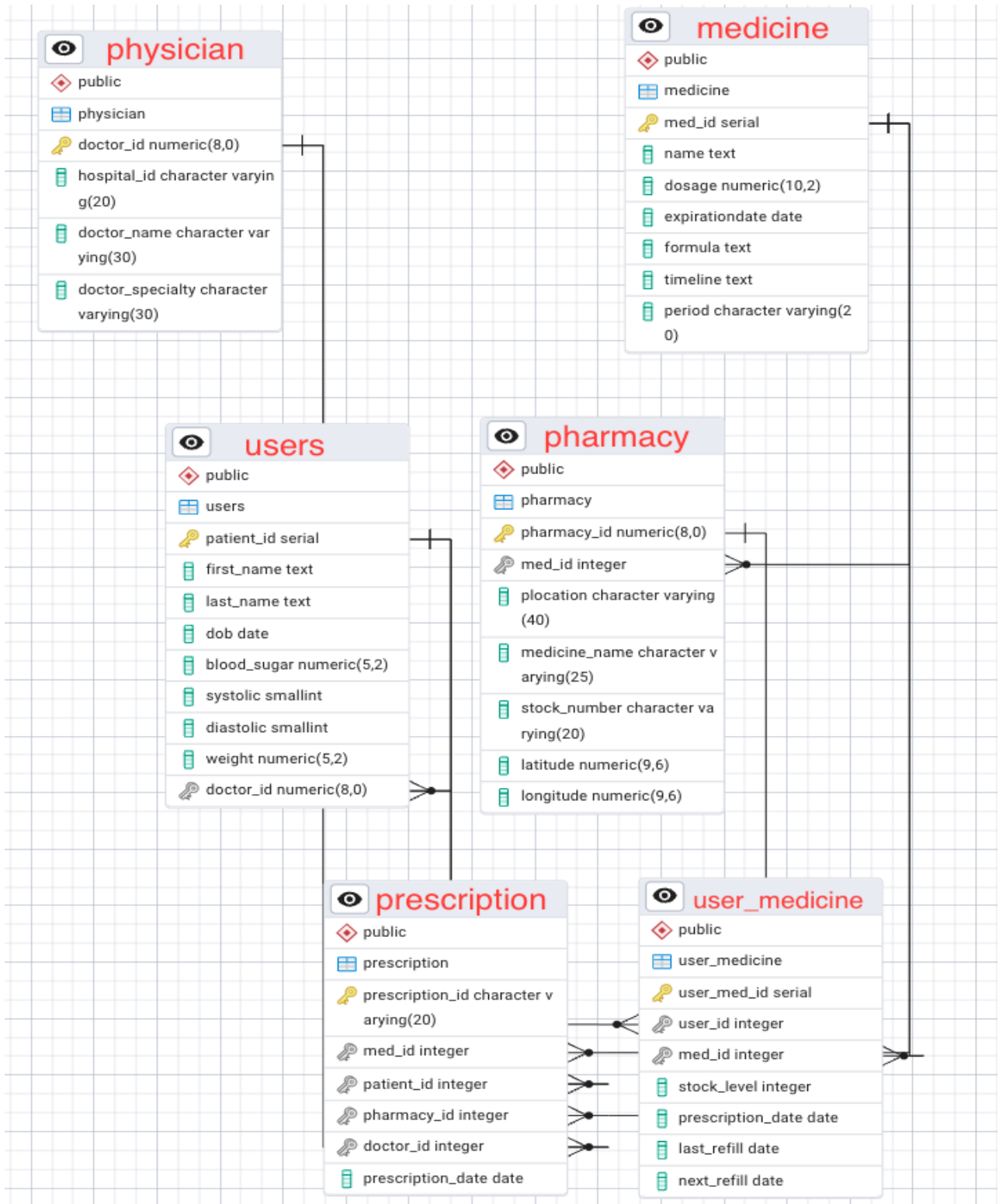
While users may view information about their medication, mainly the name, dosage, and stock-level, we would like to be able to add a feature that lets users see the side effects of a medication, if available.

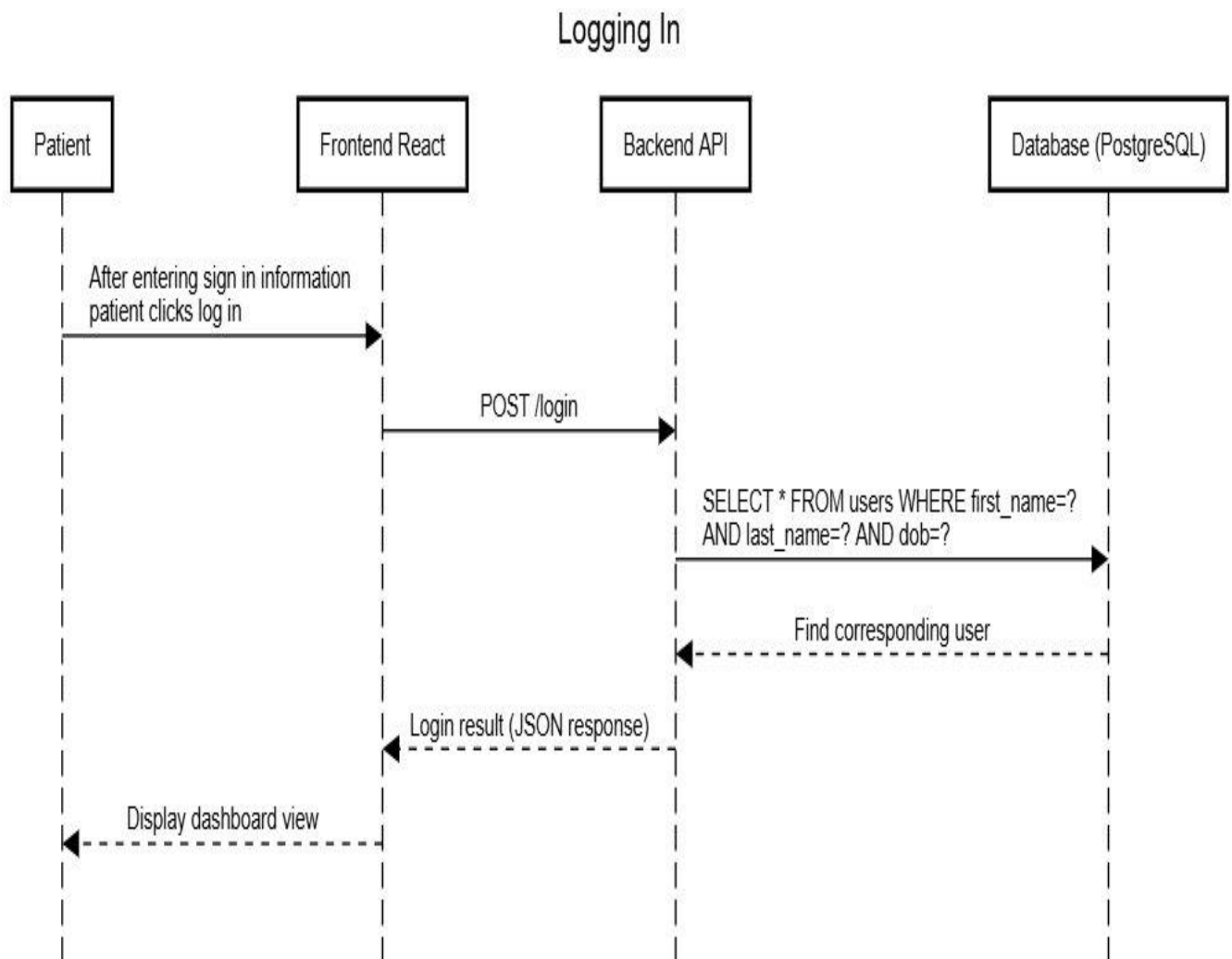
9.2.5 Messaging System

The need for communication between patients and physicians is necessary, and by implementing a system that allows both entities to communicate with each other, the user might want to obtain any updates from a prescription by the doctor.

- Amazon Web Services (2024). *Amazon RDS for PostgreSQL – Developer guide*.
https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/CHAP_PostgreSQL.html
- Facebook Open Source. (2024) *React – A JavaScript library for building user interfaces*.
<https://react.dev/>
- NPM, Inc. (2024). *Express.js – Fast, unopinionated, minimalist web framework for Node.js*. <https://expressjs.com/>
- PostgreSQL Global Development Group. (2024) *PostgreSQL 17 documentation*.
<https://www.postgresql.org/docs/>
- Vite Contributors. (2024). *Vite – Next generation front-end tooling*. <https://vitejs.dev/>
- pgAdmin Development Team. (2024). *pgAdmin 4 documentation*.
<https://www.pgadmin.org/docs/>
- TypeScript Team. (2024). *TypeScript: JavaScript with syntax for types*.
<https://www.typescriptlang.org/>

Appendix A – Entity-Relationship Diagram (ERD)



Appendix B – Sequence Diagram

Appendix C – User Stories

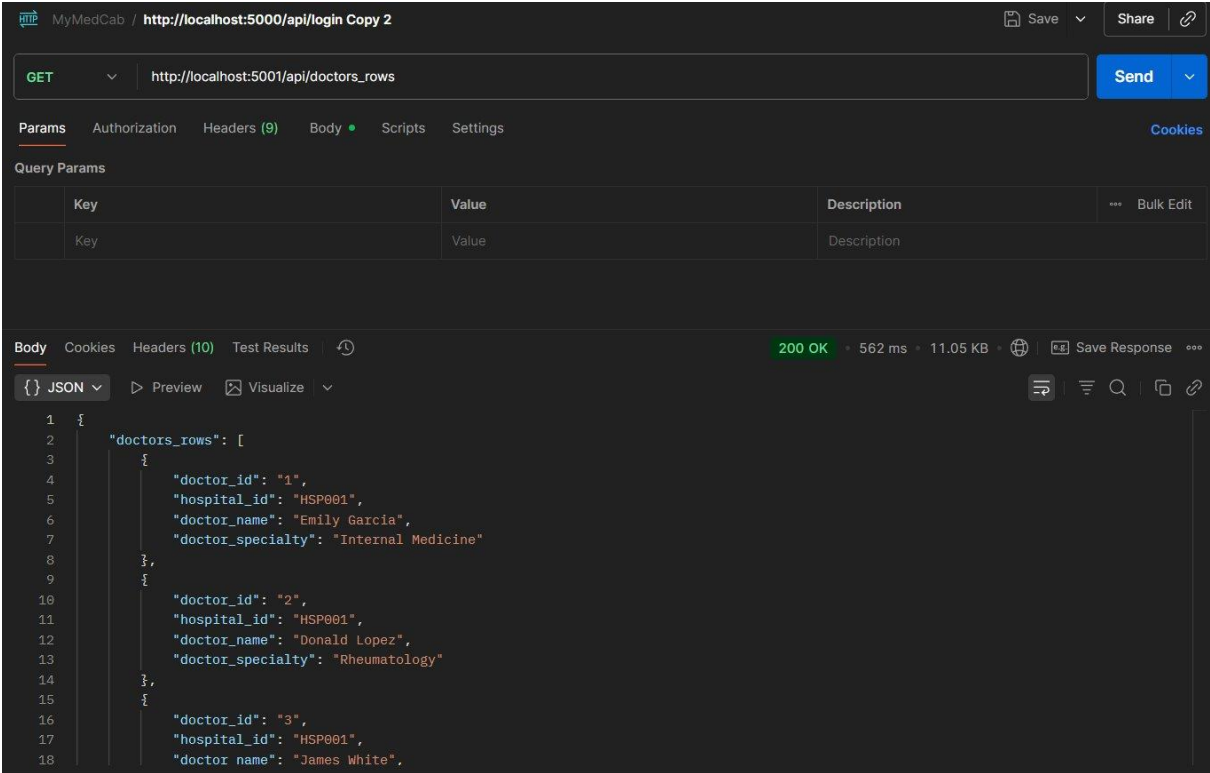
- User Story 001: Learning Technologies and Back-End Implementation.
 - As a developer, I want to be able to familiarize myself with the necessary tools and technologies so that moving on, I am able to commence all implementations (switched technologies).
 - Refresh on PostgreSQL skills by searching up tutorials and documentation as needed.
 - Create and set up a dummy database to practice SQL queries.
 - Set up and share database using Amazon RDS.
 - Begin and complete Back-End implementation (database implementation)
 - Learn about JavaScript and React through tutorials and further documentation as planned.
- User Story 002: Front-End Form and Data Entry Set-Up
 - As a developer, I want to start implementing the front-end portion of our project by creating a user input form, allowing users to submit their personal and medical information to store and display within the system.
 - Design and implement a User form with React that stores patient first and last name and date of birth. This information will allow them to view their vitals.
 - Integrate basic input validity (required fields / data type checks).
 - Link the form to back-end (database) to retrieve data.
 - Store input values into database with SQL INSERT statements.
 - Retrieve and display submitted data on the dashboard interface for confirmation.
- User Story 003: Dashboard for User Vitals
 - As a developer, I want to implement a clean dashboard interface so that users can go in and view their vitals such as blood pressure, blood sugar, and weight.
 - Retrieve and display submitted data on the dashboard interface for confirmation.
 - Build a React dashboard page that retrieves data from the back end (Express API).
 - Display data in tables and simple card for readability (containers).
 - Ensure that data is dynamically retrieved for each logged-in user.

- User Story 004: Visual Cabinet for Users
 - As a developer, I want to implement a visually organized “medication cabinet” allowing users to view and manage their medications so they can see what they are taking and track availability.
 - Build a cabinet labeled “My Medicine Cabinet” as a page through React that displays all medications linked to the logged-in users.
 - Each medication must appear as a visual card that takes in account the following – Medicine name, dosage (as in 100 mg, 2x / day), type of medication, stock level, and prescription date.
 - Include a color indicator for the stock levels where green represents enough, yellow is a warning for running low, and red requires a refill.
 - Embed icons / images that represent medicine types (pill bottle or syringe).
- User Story 005: Pharmacy Locator and Map Integration
 - As a developer, I want to implement a map feature that is interactive displays pharmacy locations so that users and physicians can easily view pharmacies that are close by.
 - Implement map functionality (potentially Google Maps API) into the existing application.
 - Retrieve pharmacy data from the pharmacy table to display any markers on the map.
 - Enable location services to adjust the map with respect to the user’s current position.
 - Test the map display and marker accuracy for proper responsiveness.
- User Story 006: Application Testing Methodologies
 - As a developer, I want to thoroughly test MyMedCab ensuring that all implemented features, database interactions, and UI functions appropriately and consistently within the system.
 - Run end-to-end testing on both the front-end and back-end components of the project ensuring that there is full system integration.
 - Use Postman to test all API endpoints such as authentication, patient data and medication views, and doctor updates using Postman.
 - Ensure that the database has the correct connectivity through Amazon RDS.
 - Test user flows for login, patient dashboard, medication cabinet, and the pharmacy locator as well as verifying role-based permissions such that the user is read-only and the doctors have insertion and updating access.

- Monitor error handling, input validation, and response codes within the API and make note of any bugs or other errors, in addition to minor fixes to be made.

- User Story 007: Gathering Final Materials – Divide and Conquer
 - As a developer, I want to gather all the necessary materials and merge everything together so that all code is structured, documents are in place, and poster is completed.
 - Gather all scrum documentation – sprint planning, daily standups, backlog grooming, sprint review, and sprint retrospectives – and organize them by sprint.
 - Download and organize all source code, separating them by entity – backend, frontend, and database scripts.
 - Design poster and PowerPoint files – ensure poster conveys goals / logistics for project and ensure PowerPoint is simple, yet thorough.
 - Complete written report – provide more details about the project through documentation that discusses the purpose of project among other things.
 - Record videos with group in which the application is demoed and expand on other information as needed.

Appendix D – GET API Call to Retrieve Doctors



The screenshot shows a REST client interface with the following details:

- URL:** `http://localhost:5001/api/doctors_rows`
- Method:** GET
- Status:** 200 OK
- Response Time:** 562 ms
- Response Size:** 11.05 KB
- Body:** JSON

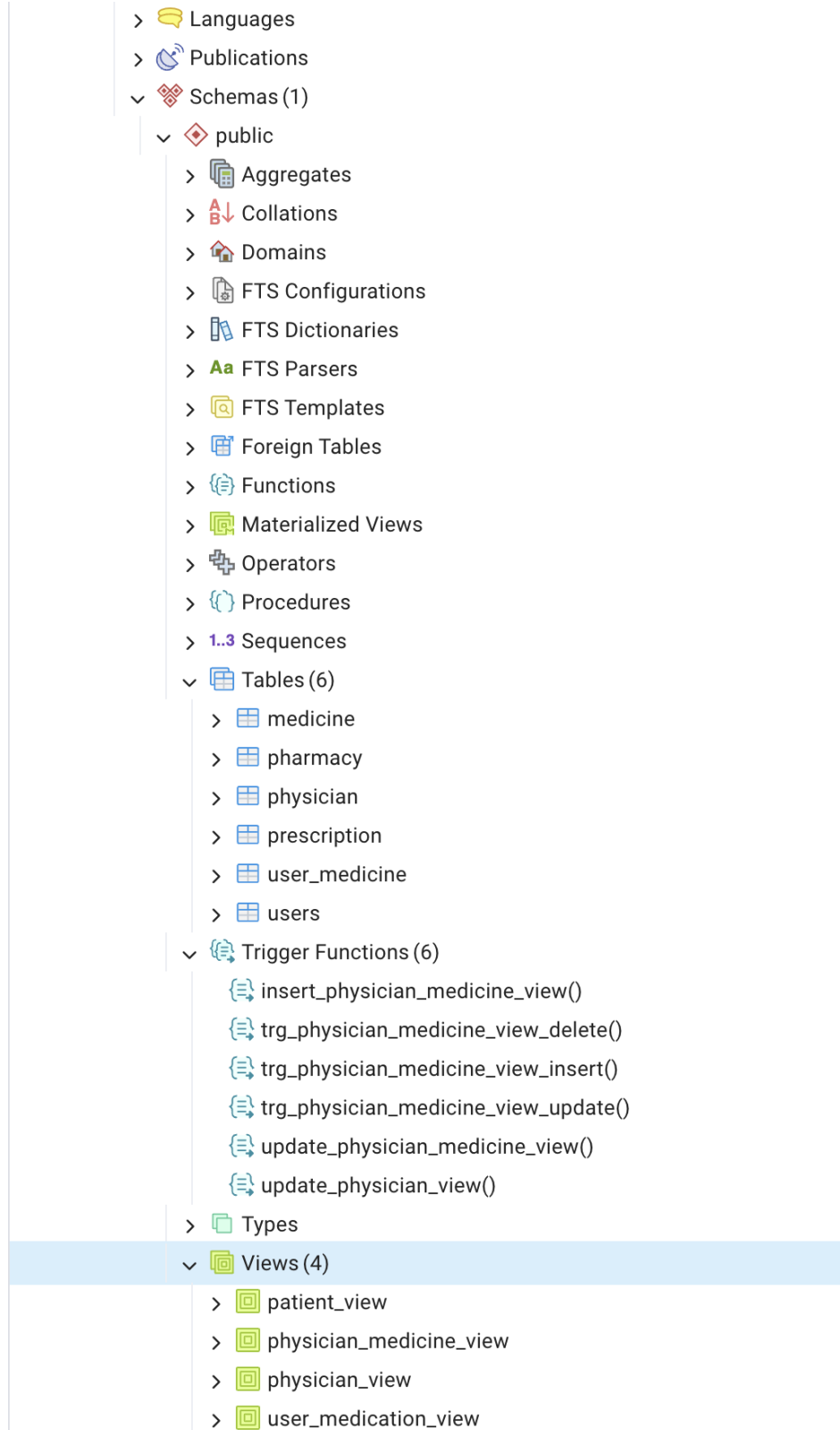
The JSON response body is as follows:

```

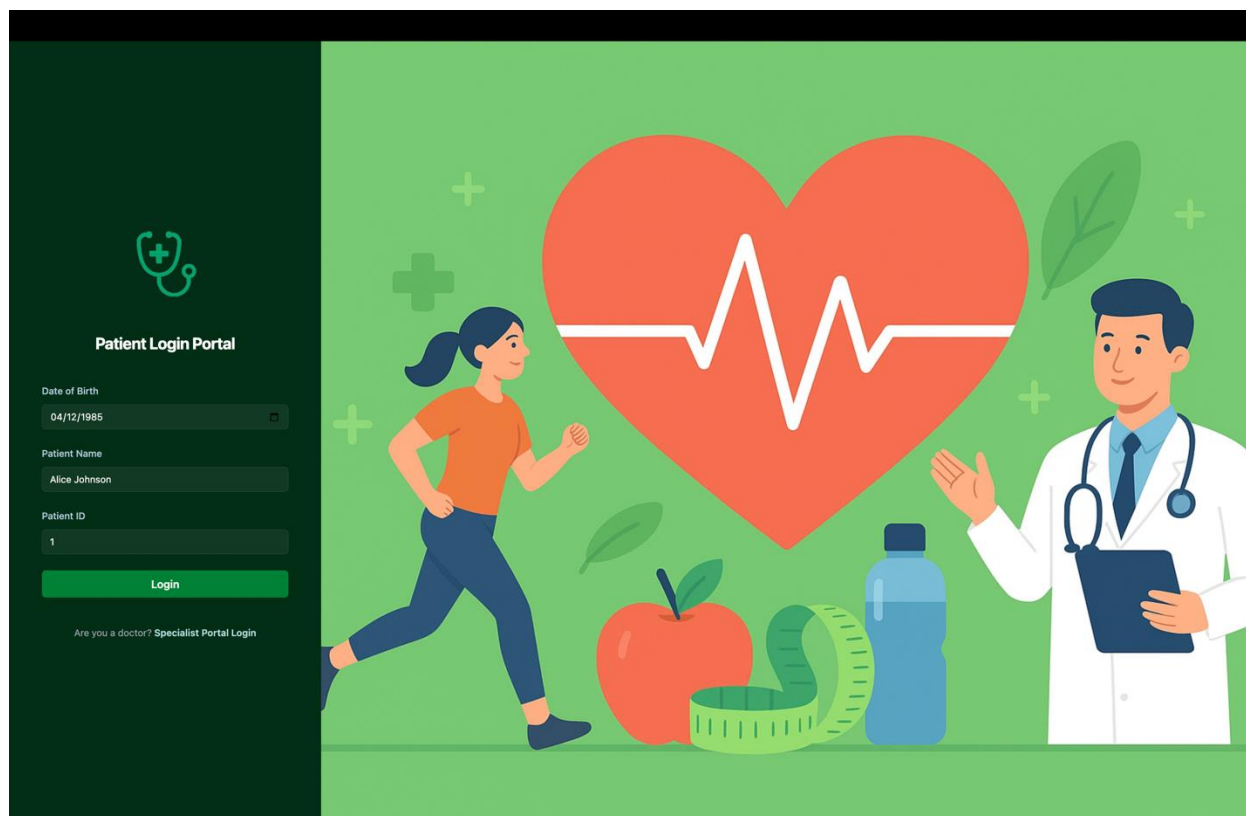
{
  "doctors_rows": [
    {
      "doctor_id": "1",
      "hospital_id": "HSP001",
      "doctor_name": "Emily Garcia",
      "doctor_specialty": "Internal Medicine"
    },
    {
      "doctor_id": "2",
      "hospital_id": "HSP001",
      "doctor_name": "Donald Lopez",
      "doctor_specialty": "Rheumatology"
    },
    {
      "doctor_id": "3",
      "hospital_id": "HSP001",
      "doctor name": "James White",
    }
  ]
}

```

Appendix E – SQL Schema: Tables, Views, Triggers



Appendix F – Patient Side



My Dashboard

Patient Information



Alice Johnson
Patient ID: 1
Date of Birth: 1985-04-12

Physician Information



Emily Garcia
Speciality: Internal Medicine
Hospital Location: Miami, FL

Medicine Information

Acetylsalicylic Acid



Dosage: 325.00 Formula: tablet

Timeline: 1
Period: day
Prescription Date: 2015-04-13

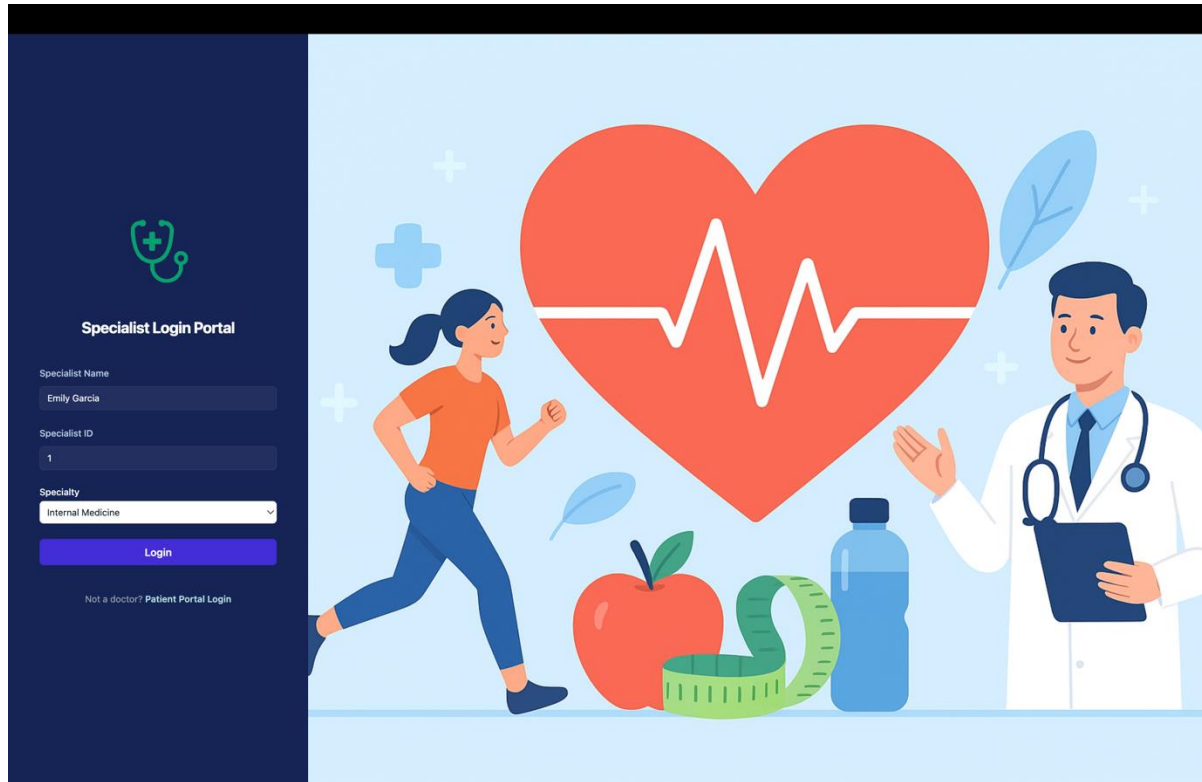
Additional information can be added as needed to assist the user in understanding their medication. This section can

MEDICINE NAME	DOSAGE	FORMULA	TIMELINE	PERIOD	PRESCRIPTION DATE
Acetylsalicylic Acid	325.00	tablet	1	day	2015-04-13
Gefitinib	250.00	tablet	1	day	2018-07-07
Budesonide-Formoterol	160.00	inhaler	2	day	2018-07-29
Beclomethasone	80.00	inhaler	2	day	2018-05-28



CVS Pharmacy, Miami, FL
Stock level: 115

Appendix G – Physician Side



My Dashboard

ID	FIRST NAME	LAST NAME	DATE OF BIRTH
1	Alice	Johnson	1985-04-12
2	Bob	Smith	1990-07-22
3	Clara	Martinez	1978-01-30
4	David	Brown	2001-09-05
5	Eva	Garcia	1995-12-18
6	Frank	Davis	1983-03-27
7	Grace	Lopez	1992-06-15
8	Henry	Wilson	1988-11-02
9	Ivy	Clark	2000-08-23
10	Jack	Lewis	1975-05-14

MEDICINE NAME	DOSAGE	FORMULA	TIMELINE	PERIOD	PRESCRIPTION DATE	LAST REFILL DATE	NEXT REFILL DATE
Budesonide-Formoterol	160.00	inhaler	2	day	2018-07-29	2018-10-18	2019-01-04
Gefitinib	250.00	tablet	1	day	2018-07-07	2018-09-21	2019-01-11
Beclomethasone	80.00	inhaler	2	day	2018-05-28	2018-07-09	2018-10-29
Acetylsalicylic Acid	325.00	tablet	1	day	2015-04-13	2015-06-23	2015-07-25

Selected prescription for Alice Johnson
RXB181

EditDelete

Prescription Builder

Enter patient ID

Enter medicine ID

Dosage

Formula

Enter dosage

e.g. tablet, capsule

Timeline

Period

e.g. 1, 2

e.g. day, week

Prescription Date

mm/dd/yyyy

Last Refill Date

mm/dd/yyyy

Next Refill Date

mm/dd/yyyy

Submit

Clear

Flagami

White House

Trailer Park

West Miami

CVS Pharmacy, Miami, FL

Stock level: 66