HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION COMMUNICATION TECHNOLOGY

CAPSTONE PROJECT REPORT:

CLINIC MANAGEMENT WEB APP

Supervised by:
Ph. D Bui Thi Mai Anh

Group number: 6

Name	ID
Bui Khac Chien	20220059
Trinh Hoang Anh	20225470
Tran Quang Minh	20225512
Vo Ta Quang Nhat	20225454
Bach Nhat Minh	20225509

TABLE OF CONTENTS

Chapter 1: Project Introduction

1.1 Problem Statement

The client, a local clinic, was operating with an outdated clinic management web application that suffered from limited functionalities, a manual appointment booking system, and an insufficient database. These shortcomings led to inefficiencies in daily operations, patient care coordination, and overall service quality. The manual processes were prone to errors and consumed significant staff time. The existing database lacked the structure and capacity to support modern healthcare data management needs, including comprehensive electronic medical records and efficient data retrieval. This project was initiated to address these challenges by developing a modern, robust, and feature-rich web-based clinic management system.

1.2 Objectives and Scope

The primary objective of this project was to design, develop, and deploy an improved web-based clinic management system. This system aims to streamline clinic operations, enhance patient experience, and provide better data management capabilities.

The scope of the completed project encompasses the following core functionalities:

- **User Authentication & Role-Based Access Control:** Secure registration and login for Patients, Doctors, Clinic Staff, and Administrators, with distinct permissions for each role.
- **Patient Management:** Comprehensive patient registration, profile management (personal information, contact details, basic medical history notes), and search capabilities.
- **Online Appointment Booking:** Patients can search for available slots (by service, date, doctor) and book appointments online.
- **Appointment Management:** Clinic Staff and Doctors can view, schedule, reschedule, confirm, and cancel appointments.
- **Electronic Medical Records (EMR):** Doctors can create, view, and update electronic medical records for patients, including consultation notes, diagnoses, and prescriptions. Patients have limited access to view their own records (e.g., appointment history, prescriptions).
- **Chatbot Support:** An AI-powered chatbot integrated into the system to answer frequently asked questions (clinic information, booking help), provide basic symptom guidance, and assist staff with EMR note-taking.
- **OTC Medication Records:** Clinic staff can create records for patients purchasing over-the-counter medication.

1.3 Solution Approach

The project followed an Agile development methodology, emphasizing iterative and incremental development. The development process was structured into key phases:

- **Requirement Analysis and Design:** Detailed analysis of existing system limitations and user needs, followed by system design.
- **Database Development:** Design and implementation of the PostgreSQL database schema.
- **Backend API Development:** Building RESTful APIs using FastAPI and Python.
- **Frontend UI Development:** Creating the user interface with React and TypeScript.
- **Integration and Testing:** Integrating frontend and backend components.
- **Deployment:** Deploying the application for use.

The technological stack selected for the solution includes:

- **Frontend:** React, TypeScript, Node.js/npm, Axios for API communication, React Router for navigation, and react-toastify for notifications.
- **Backend:** Python, FastAPI, SQLAlchemy (ORM), Pydantic (data validation), Uvicorn (ASGI server), python-jose (JWT handling), passlib/bcrypt (password hashing), and Google Generative AI SDK for the chatbot.
- **Database:** PostgreSQL.

The system architecture is a client-server model, with a decoupled frontend and backend, facilitating independent development and scalability.

1.4 Report Outline

Chapter 1: Project Introduction: Provides an overview of the project, including the problem statement, objectives, scope, solution approach, and the structure of this report.

Chapter 2: Requirements Analysis: Details the analysis of the system, identifies system actors and their functionalities, presents use case diagrams and detailed specifications, and lists the non-functional requirements that were met.

Chapter 3: Technologies Used: Describes the key technologies, frameworks, libraries, and design patterns employed in the successful development of the project.

Chapter 4: Application Development and Deployment: Covers the architectural design, detailed design of UI, classes, and database, discusses implementation aspects, and outlines the deployment process of the completed application.

Chapter 5: Conclusion and Future Development: Summarizes the project's achievements against its objectives, reflects on its success, and suggests potential directions for future enhancements.

Chapter 2: Requirements Analysis

2.1 Functional Overview

The Clinic Management System is designed to serve multiple user roles with specific functionalities tailored to their needs.

Actors:

- **Patient:** Individuals seeking medical services from the clinic.
- **Doctor:** Medical professionals responsible for diagnosis, treatment, and maintaining patient medical records.
- **Clinic Staff:** Administrative personnel (e.g., receptionists, clinic managers) handling patient registration, appointment scheduling, and other operational tasks.
- **Administrator:** System superuser responsible for managing user accounts, system configurations, and overall system oversight.

Main Functionalities per Actor:

Patient:

- Secure self-registration and login.
- View and update personal profile information.
- Search for available appointment slots by service, date, and doctor.
- Book new appointments online.
- View upcoming and past appointment history.
- View personal medical information, including prescriptions (with appropriate access controls).
- Cancel booked appointments (subject to clinic policy).
- Interact with an AI-powered chatbot for FAQs (clinic info, services, hours), appointment booking assistance, and basic symptom-based guidance.

Doctor:

- Secure login.
- View daily/weekly appointment schedule.
- Access and review patient profiles and comprehensive medical histories.
- Conduct consultations and document findings.

- Create, update, and manage electronic medical records (EMR), including consultation notes, diagnoses (selectable from a predefined list or custom entry), and treatment plans.
- Create and manage electronic prescriptions, selecting medications, dosages, and quantities.
- View a list of patients scheduled for consultation.

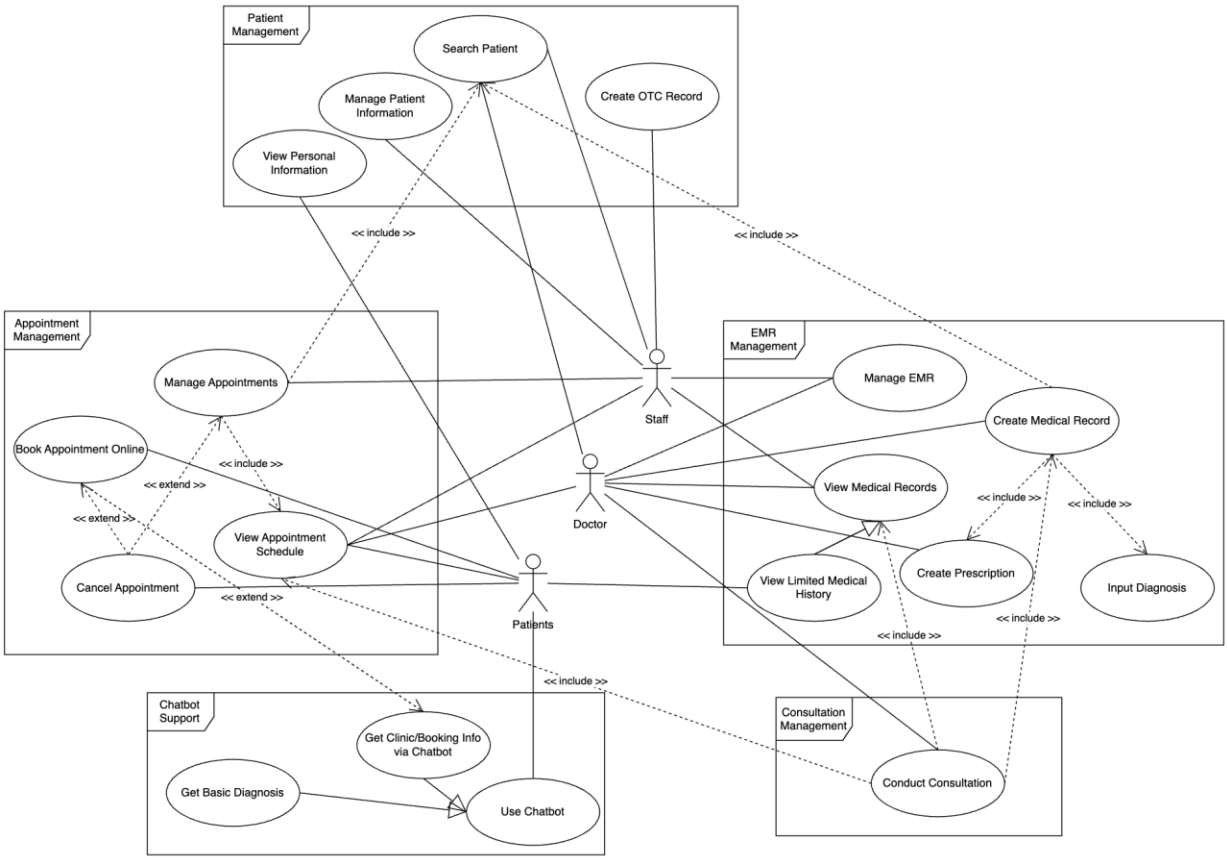
Clinic Staff:

- Secure login.
- Manage patient registration and update patient profiles.
- Search for patients by various criteria (name, phone, ID).
- Manage the overall appointment schedule: book, reschedule, confirm, and cancel appointments for patients.
- View appointment calendars (daily, weekly).
- Create records for patients purchasing over-the-counter (OTC) medication.

Administrator:

- Secure login.
- Manage user accounts across all roles (create, view, update, delete users; manage roles and permissions).
- Configure clinic services and doctor schedules (if applicable).
- Oversee general system settings and potentially view system logs and basic analytics.

2.1.1 Overall Use Case Diagram



Key high-level use cases depicted include:

- **Patient:** Book Appointment Online, View Medical History (Limited), Use Chatbot, Manage Personal Information, Cancel Appointment.
- **Doctor:** Conduct Consultation, Create Medical Record, Input Diagnosis, Create Prescription, View Medical Records, Manage EMR, Search Patient, View Appointment Schedule.
- **Staff:** Manage Appointments (Book, Reschedule, Cancel), Manage Patient Information, Search Patient, Create OTC Record, Use Chatbot, View Appointment Schedule.
- **Administrator:** Manage Users, Manage System Settings.

2.1.2 Workflows

The system supports several key business workflows:

Online Appointment Booking Workflow (Patient):

- Patient logs in or accesses public booking.
- Patient searches for services/doctors/availability.

- System displays available slots.
- Patient selects a slot and provides necessary information.
- System confirms the booking and updates the schedule.

Consultation and EMR Creation Workflow (Doctor):

- Doctor selects a patient from their schedule.
- Doctor reviews existing patient EMR.
- Doctor conducts consultation and inputs new findings, diagnosis, and treatment plan into the EMR.
- If medication is required, Doctor creates an electronic prescription.
- Doctor saves the EMR. The system updates the patient's record.

Appointment Management Workflow (Staff):

- Staff logs in.
- Staff views the clinic's appointment schedule.
- Staff receives a call/request from a patient.
- Staff searches for the patient or registers a new patient.
- Staff finds an available slot based on patient/doctor preference and service.
- Staff books/reschedules/cancels the appointment in the system.
- System updates the schedule and relevant records.

Chatbot Interaction Workflow (Patient/Staff):

- User initiates chat.
- User types a query.
- Chatbot (via Google Gemini) processes the query, potentially accessing EMR data or general knowledge base.
- Chatbot provides a response (information, guidance, or prompts for more details).
- Notes from the chat interaction related to symptoms can be logged into the patient's EMR.

2.2 Use Case Specifications

Use Case 1: Patient Book Appointment Online

Use Case ID: UC001

Brief Description: This use case describes how a Patient interacts with the Clinic Management Website to search for available appointment slots and book a new appointment online.

Actor(s): Primary: Patient.

Preconditions:

- The Patient is logged into the system or accessing a public booking interface if supported.
- Clinic services, doctor schedules, and available time slots are configured and up-to-date in the system.

Basic Flow / Main Success Scenario:

1. Patient navigates to the "Book Appointment" section.
2. System displays options to filter/search by service type, preferred date range, and optionally by specific doctor.
3. Patient enters their search/filter criteria.
4. System queries the database and displays a list or calendar view of matching available time slots.
5. Patient selects a desired time slot.
6. System displays a summary of the selected appointment details (service, doctor, date, time) for confirmation.
7. Patient confirms the booking.
8. System creates a new appointment record, updates the doctor's and clinic's schedule to mark the slot as booked, and displays a success confirmation message to the Patient. A notification (e.g., email) is sent to the Patient.

Alternative Flows / Exceptions:

4a. No available slots match the Patient's criteria: System displays a "No slots available" message and allows the Patient to modify search criteria (Return to Step 3).
7a. Patient decides not to confirm or cancels before confirmation: System discards the pending booking and the selected slot remains available (Return to Step 4 or allow new search).
8a. System error during saving (e.g., database connectivity issue, concurrent booking conflict): System displays an appropriate error message. The appointment is not booked, and the slot remains available (or is handled by a conflict resolution mechanism). (Return to Step 7 or inform user to retry).

Input Data:

Data fields	Description	Mandatory	Valid condition	Example

Service Type	Clinic service needed	Yes	From defined list	"General Checkup"
Preferred Date	Desired date/range	Yes	Valid date format	"2025-05-10"
Preferred Doctor	Optional doctor choice	No	Valid doctor	"Dr. Tran"
Selected Slot	Chosen time slot	Yes	Must be available	"10:30 AM, 2025-05-10"

Output Data:

Data fields	Description	Display format	Example
Available Slots	List of open appointment times	Calendar/List	"10:30 AM", "11:00 AM"
Confirmation Msg	Booking success/failure message	Text	"Appointment Confirmed!"
Appt. Details	Summary of booked appointment	Text Summary	Service, Time, Doctor

Postconditions:

- On Success: A new appointment record is created in the database with "Scheduled" status. The selected time slot is marked as unavailable in the relevant schedules. The patient's appointment history is updated.
- On Failure: The appointment is not created. The system state regarding schedules remains unchanged from before the attempt.

Use Case 2: Doctor Create Medical Record

Use Case ID: UC002

Brief Description: Describes how a Doctor creates a new electronic medical record (EMR), including consultation notes, diagnosis, and prescription, for a patient during or after a consultation.

Actor(s): Primary: Doctor.

Preconditions:

- Doctor is logged into the system.
- The relevant Patient's record has been selected or is active in the Doctor's context (e.g., from an appointment list or patient search).

Basic Flow / Main Success Scenario:

1. Doctor selects the option to create/update a medical record for the current patient/consultation.
2. System displays the EMR interface, pre-filling patient identifiers.
3. Doctor enters/updates consultation notes, observations, and patient-reported symptoms.
4. Doctor selects or enters a diagnosis (potentially searching a predefined list of diagnoses or adding a new one if permitted).
5. If medication is required, Doctor selects the option to add/edit a prescription.
6. System displays the prescription entry interface.
7. Doctor searches for and selects medication(s) from the drug database, entering dosage, quantity, frequency, and instructions.
8. System adds/updates the medication(s) in the EMR's prescription section.
9. Doctor reviews the complete EMR (notes, diagnosis, prescription).
10. Doctor saves the medical record.
11. System validates all entered data, saves the EMR to the database, associating it with the patient and the specific consultation/appointment, and displays a success message.

Alternative Flows / Exceptions:

4a. Diagnosis not in predefined list: System allows Doctor to enter a new diagnosis text (if system configuration permits). (Continue to Step 5)
7a. Medication not found in database: System allows adding a new medication to the formulary (if Doctor has permission and feature is supported) or flags it for review. (Continue to Step 7 or require selection of existing)
10a. Doctor decides to cancel creation/update before saving: System discards any unsaved changes (after confirmation). (Return to patient dashboard or appointment list)
11a. Validation error (e.g., missing required field, invalid dosage format): System highlights errors and prompts Doctor to correct them. (Return to relevant input field in Steps 3-9)
11b. Error saving the record (e.g., database error): System displays a system error message and allows Doctor to retry saving. (Return to Step 10)

Input Data:

Data fields	Description	Mandatory	Valid condition	Example
Consultation Notes	Doctor's observations	Yes	Text	"Patient reports cough..."
Diagnosis	Medical diagnosis	Yes	Selected/Text	"Bronchitis"
Medication Name	Name of prescribed drug	Yes (if Rx)	From DB or new	"Amoxicillin 500mg"
Dosage	How much/often to take	Yes (if Rx)	Text	"1 tablet 3 times daily"
Quantity	Amount to dispense	Yes (if Rx)	Number	"21"

Output Data:

Data fields	Description	Display format	Example
EMR Form	Interface for entering data	Web Form	Fields for notes, Dx, Rx
Diagnosis List	Search results for diagnosis	Dropdown/ List	"Bronchitis", "Pneumonia"
Medication List	Search results for drugs	List	"Amoxicillin", "Paracetamol"
Confirmation Msg	Record saved success/error message	Text	"Medical Record Saved."

Postconditions:

- On Success: A new EMR (or an update to an existing one) is created/updated in the database and linked to the patient's overall health history and the specific appointment.
- On Failure: No new EMR is created, or if updates were being made, the record reverts to its previous state (unless drafts are supported and saved).

Use Case 3: Use Chatbot

Use Case ID: UC003

Brief Description: Describes how a Patient (or Staff) interacts with the website's Chatbot to get answers to FAQs, clinic information, booking assistance, or basic symptom guidance.

Actor(s): Primary: Patient, Staff.

Preconditions:

- User is viewing a page with the Chatbot interface accessible.
- Chatbot service (Google Gemini integration) is active and configured.

Basic Flow / Main Success Scenario (Getting Clinic Info):

1. User clicks to open/activate the Chatbot window/interface.
2. System (Chatbot) displays a welcome message and may offer prompt options or an input field.
3. User types a query (e.g., "What are your clinic hours?").
4. System sends the query to the backend, which then forwards it to the AI service (Google Gemini).
5. AI service processes the query, identifies the intent (e.g., requesting clinic hours).
6. AI service retrieves relevant information from its configured knowledge base or general capabilities.
7. Backend receives the AI's response and forwards it to the frontend.
8. System (Chatbot) displays the answer to the user.
9. Chatbot may ask if further assistance is needed.

Alternative Flows / Exceptions:

5a. Chatbot doesn't understand the query or cannot find relevant information: Chatbot displays a message like "Sorry, I couldn't understand that. Could you please rephrase?" or suggests common topics/questions. (User returns to Step 3)

5b. Query is about booking an appointment: Chatbot provides instructions on how to book, a direct link to the booking page, or initiates a guided booking flow if implemented. (May end use case or lead to UC001)

5c. Query describes symptoms (Symptom Checker functionality):

- If user is Staff and has a patient context: Chatbot (AI) uses EMR data and the new symptom input to provide potential insights or differential diagnoses for the staff member's consideration. Notes from this interaction can be logged to the EMR.
- If user is Patient: Chatbot provides general information about potential conditions related to symptoms and advises to consult a doctor for an actual diagnosis, possibly offering to help book an appointment.

5d. Query requires access to personal data (and user is not authenticated or lacks context): Chatbot instructs the user to log in or navigate to their profile page for such information.

Input Data:

Data fields	Description	Mandatory	Valid condition	Example
User Query	Text typed by the patient	Yes	Text	"How to book online?"
Topic Selection	Choice from predefined menu	No	Valid option	"Services"

Output Data:

Data fields	Description	Display format	Example
Chatbot Response	Text answer from the bot	Text	"Our hours are 8 AM - 5 PM."
Guidance/Links	Instructions or hyperlinks	Text/Link	"Click here to book: [link]"

Postconditions:

- The user has received information or guidance from the Chatbot.
- If the interaction involved staff and a patient context for symptom checking, relevant notes from the chat may be logged into the patient's EMR.

Use Case 4: Doctor Conduct Consultation

Use Case ID: UC004

Brief Description: Describes the process of a Doctor conducting a consultation using the system, which includes accessing patient information and creating/updating their medical records.

Actor(s): Primary: Doctor; Secondary: Patient (whose record is being managed).

Preconditions:

- Doctor is logged into the system.
- The patient for the consultation is identified and their record is accessible (e.g., selected from the Doctor's appointment schedule or via patient search).

Basic Flow / Main Success Scenario:

1. Doctor selects the patient from their daily appointment list or searches for and selects the patient.
2. System displays the patient's dashboard/overview, including summaries of past visits, allergies, ongoing conditions, and options for the current consultation.
3. Doctor reviews the patient's relevant medical history, previous diagnoses, lab results, and prescriptions displayed by the system (<> View Medical Records).
4. Doctor initiates the process to document the current consultation (this effectively is <> Create/Update Medical Record - UC002).
5. Doctor interacts with the EMR interface to input subjective complaints, objective findings, assessment (diagnosis), and treatment plan, including creating any necessary prescriptions (following the detailed flow of UC002, steps 3-10).
6. Doctor finalizes and saves the consultation record within the EMR.
7. System confirms that the EMR has been successfully saved and updated.
8. Doctor concludes the consultation process in the system for this patient (e.g., marks appointment as completed).

Alternative Flows / Exceptions:

3a. Patient is new or has no prior medical history in the system: System indicates no previous records found. Doctor proceeds with creating a new EMR. (Proceed to Step 4)

5a. Consultation requires ordering external lab tests or imaging: Doctor uses a dedicated function/section within the system to order these tests (this may be a separate use case or an extension of EMR functionality). The EMR may be partially saved pending results. (May pause EMR entry or save as draft, then return to Step 5 or proceed to Step 8)

6a. Doctor needs to complete or update the record later (e.g., pending test results): Doctor saves the current EMR as a draft (if the system supports this). (Proceed to Step 8)

7a. Error saving the record (e.g., network issue, database error): System displays an error message and allows the Doctor to retry saving. (Return to Step 6)

Input Data:

Data fields	Description	Mandatory	Valid condition	Example
Patient Selection	Selected patient from search or list	Yes	Selected	“Mr. A”
Consultation Notes	Doctor's observations	Yes	Text	"Patient reports cough..."
Diagnosis	Medical diagnosis	Yes	Selected/ Text	"Bronchitis"
Medication Name	Name of prescribed drug	Yes (if Rx)	From DB or new	"Amoxicillin 500mg"
Dosage	How much/often to take	Yes (if Rx)	Text	"1 tablet 3 times daily"
Quantity	Amount to dispense	Yes (if Rx)	Number	"21"

Output Data:

Data fields	Description	Display format	Example
Patient Dashboard	Overview for the patient	Web UI	
Display of Past Medical History	Review of patient's previous medical records	Web UI	
EMR Interface	Form for creating or updating medical records	Web Form	Fields for notes, Dx, Rx
Confirmation Messages	Message indicating success/failure of save action	Text	"Medical Record Saved."

Postconditions:

- On Success: A new or updated Electronic Medical Record is saved in the database for the patient, linked to the specific consultation/appointment. The patient's overall medical history is updated. The appointment status may be updated to "Completed."
- On Failure/Draft: The EMR may not be saved, or may be saved in an incomplete/draft state if supported by the system.

Use Case 5: Staff Manage Appointment

Use Case ID: UC005

Brief Description: This use case describes how Clinic Staff interact with the system to view the clinic schedule, and schedule, reschedule, or cancel patient appointments.

Actor(s): Primary: Clinic Staff; Secondary: Patient (whose appointment is managed), Doctor (whose schedule is affected).

Preconditions:

- Clinic Staff is logged into the system with appropriate permissions.
- System has up-to-date information on clinic services, doctor schedules, and existing patient data.

Basic Flow / Main Success Scenario (Schedule New Appointment):

1. Clinic Staff accesses the appointment management section or calendar view.
2. System displays the clinic's schedule (e.g., daily/weekly view, filterable by doctor).
3. Clinic Staff searches for an existing patient by name, ID, or phone number, or initiates the process for booking for a new patient (<> Search Patient or an implicit new patient registration flow if integrated).
4. If an existing patient is found, the system displays their key details. If new, Staff enters essential patient information to create a basic profile or link to a full registration process.
5. Clinic Staff selects the required service type, preferred doctor (if any), and desired date/time range.
6. System queries for and displays available time slots matching the criteria.
7. Clinic Staff selects a suitable time slot from the available options.
8. System presents a summary of the appointment details (patient, service, doctor, date, time) for confirmation.
9. Clinic Staff confirms the appointment booking.
10. System saves the new appointment, updates the clinic and doctor schedules, marks the slot as unavailable, and displays a success message. A confirmation might be sent to the patient (e.g., email/SMS if implemented).

Alternative Flows / Exceptions:

3a. Patient not found (when searching for an existing patient): System indicates "Patient not found." Staff may refine the search, or proceed to register the patient as new (Return to Step 3 or proceed to new patient registration).

6a. No suitable slots available for the given criteria: System displays a message indicating no availability. Staff may adjust search criteria (e.g., different date, doctor, or time) or inform the patient. (Return to Step 5)

Reschedule Existing Appointment:

10a.1. Staff searches for and selects an existing appointment to reschedule.

10a.2. Staff chooses the "Reschedule" option.

10a.3. System proceeds similarly to steps 5-10 of the basic flow for selecting a new date/time slot for the existing appointment. Upon confirmation, the original slot is freed, and the new slot is booked.

Cancel Existing Appointment:

10b.1. Staff searches for and selects an existing appointment to cancel.

10b.2. Staff chooses the "Cancel" option.

10b.3. System prompts for confirmation and optionally a reason for cancellation.

10b.4. Staff confirms the cancellation.

10b.5. System updates the appointment status to "Cancelled," makes the time slot available again, and logs the cancellation. (End Use Case)

10c. Error saving/updating appointment (e.g., database error, concurrent modification): System displays an error message. (Return to Step 9 or provide options to retry).

Input Data:

Data fields	Description	Mandatory	Valid condition	Example
Patient Search Criteria	Name, ID, phone, etc.	If existing	Text/Number	"John Doe" / "1001"
New Patient Info	Name, contact, DOB, etc.	If new	Valid formats	"Jane Smith, 0123456789"
Service Type	Clinic service needed	Yes	From defined list	"Dental Cleaning"
Date/Time Criteria	Preferred appointment time	Yes	Valid date/time	"2025-06-16, Afternoon"
Selected Doctor	Doctor for the appointment	If applicable	Valid doctor	"Dr. Emily White"
Selected Time Slot	Chosen appointment slot	Yes	Must be available	"2:30 PM, 2025-06-16"
Action	Schedule, Reschedule, Cancel	Yes	Enum value	"Schedule"

Output Data:

Data fields	Description	Display format	Example

Appointment Schedule	Clinic/Doctor calendar view	UI Calendar/List	
Patient Search Results	List of matching patients	UI List	
Available Time Slots	Open slots for booking	UI List/Calendar	
Appointment Details	Summary for confirmation	UI Modal/Display	
Success/Error Messages	Feedback on action performed	UI Text/Modal	"Appointment Scheduled Successfully."

Postconditions:

- On Successful Scheduling/Rescheduling: An appointment record is created or updated in the database. The relevant schedules (clinic, doctor) are updated to reflect changes in slot availability.
- On Successful Cancellation: The appointment record's status is updated to "Cancelled." The previously booked time slot becomes available.
- On Failure: The system state regarding the specific appointment remains unchanged or reflects the error condition.

2.3 Non-Functional Requirements

The completed Clinic Management System successfully addressed the following non-functional requirements:

Error Handling and Messaging: The system provides clear, user-friendly error messages and feedback. Frontend components utilize toast notifications for immediate user feedback on actions (login, registration, data submission). Backend API responses include detailed error messages and appropriate HTTP status codes to aid in frontend error handling and debugging.

Security (Authentication, Authorization):

- **Authentication:** A robust JWT-based authentication system is implemented. User credentials are not stored in plain text; passwords are securely hashed using bcrypt. Tokens have defined expiration times.
- **Authorization:** Role-Based Access Control (RBAC) is enforced across the application. API endpoints are protected, and access to specific functionalities and data is restricted based on user roles (Patient, Doctor, Clinic Staff, Admin). This ensures that users can only perform actions and access data relevant to their roles.

Usability (Display Formats, Fonts, Colors): The frontend, built with React and TypeScript, offers a modern, intuitive, and responsive user interface. A consistent design language is applied across components, with standardized fonts, a professional color palette, and clear display formats for data. Navigation is role-based and straightforward.

Performance (Response Time): The system is designed for good performance. FastAPI (backend) and React (frontend) are known for their efficiency. Database queries are optimized through SQLAlchemy. Asynchronous operations are utilized in the backend to handle concurrent requests effectively, ensuring acceptable response times under typical load conditions.

Scalability (Concurrent Users): The chosen technology stack (FastAPI, PostgreSQL, React) and the decoupled architecture allow for both vertical and horizontal scaling of the backend and frontend independently to accommodate a growing number of users and data.

Reliability/Availability: The system is built with reliable technologies. Standard deployment practices, including database backups and potentially redundant server setups (in a production environment), ensure high availability and quick recovery from failures.

Data Security (Encryption):

- Passwords are cryptographically hashed.
- JWTs are signed to prevent tampering.
- All sensitive data transmission between the client and server is secured using HTTPS in the production environment.
- Database access is controlled through secure credentials.

Maintainability (Coding Standards, Complexity Metrics):

- The codebase is well-structured with a clear separation of concerns (frontend/backend, and within each, by feature/module like routers, models, components).

- Consistent coding standards, type hinting (Python/TypeScript), and commenting practices were followed throughout development, facilitating easier understanding, debugging, and future enhancements.
- The use of an ORM (SQLAlchemy) and Pydantic models simplifies data management and validation, contributing to maintainability.

Chapter 3: Technologies Used

This chapter details the key technologies, frameworks, libraries, and design patterns that were instrumental in the successful development and completion of the Clinic Management System.

3.1 FastAPI

Description: FastAPI is a modern, high-performance web framework for building APIs with Python 3.7+, leveraging standard Python type hints.

Reasons for Choosing & Advantages: Selected for its exceptional speed, ease of development, automatic data validation and serialization (via Pydantic integration), automatic interactive API documentation (Swagger UI and ReDoc), and robust dependency injection system. Its asynchronous capabilities (async/await) were crucial for building a responsive and scalable backend.

Community Support: Benefits from a large, active community and comprehensive documentation, ensuring ample resources and support.

3.2 React

Description: React is a declarative, efficient, and flexible JavaScript library for building user interfaces, particularly single-page applications.

Reasons for Choosing & Advantages: Chosen for its component-based architecture, which promotes code reusability, modularity, and easier maintenance of the frontend. The virtual DOM ensures efficient UI updates, leading to a smooth and responsive user experience. The vast ecosystem of libraries (e.g., React Router, Axios) and strong community support were also key factors.

Relevant History: Developed and actively maintained by Meta (formerly Facebook) and a large community of individual developers and companies.

3.3 PostgreSQL

Description: PostgreSQL is a powerful, open-source object-relational database system known for its reliability, feature robustness, and performance.

Reasons for Choosing & Advantages: Selected for its ACID compliance, strong support for complex queries and transactions, extensibility, and a wide array of data types

suitable for healthcare data. Its proven stability and scalability make it an excellent choice for managing sensitive patient information.

Drawbacks: While powerful, initial setup and advanced administration can be more complex compared to simpler database systems, though this was managed effectively for the project's needs.

3.4 SQLAlchemy

Description: SQLAlchemy is a comprehensive Python SQL toolkit and Object Relational Mapper (ORM).

Reasons for Choosing & Advantages: Utilized to provide an abstraction layer over SQL, allowing developers to interact with the PostgreSQL database using Python objects and methods. This simplified data access logic (CRUD operations), improved code readability, and helped prevent SQL injection vulnerabilities. Its flexibility supports both ORM patterns and raw SQL execution when needed.

Community Support: A mature and widely-used library with extensive documentation and a strong community.

3.5 Pydantic

Description: Pydantic is a data validation and settings management library that uses Python type hints to validate data.

Reasons for Choosing & Advantages: Integral to FastAPI, Pydantic was used for defining clear data schemas for API request and response bodies. This enabled automatic data validation, serialization, and documentation, significantly reducing boilerplate code and improving data integrity at the API boundaries.

Relevant History: Has become a standard for data validation in modern Python web frameworks.

3.6 Uvicorn

Description: Uvicorn is an ASGI (Asynchronous Server Gateway Interface) server, built using uvloop and httptools for high performance.

Reasons for Choosing & Advantages: Chosen as the ASGI server to run the FastAPI application, enabling its asynchronous capabilities and ensuring efficient handling of concurrent requests.

3.7 Node.js/npm

Description: Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine. npm (Node Package Manager) is the default package manager for Node.js.

Reasons for Choosing & Advantages: Essential for the frontend development workflow. Used to manage React project dependencies, run the development server (react-scripts), build the production version of the frontend application, and utilize a vast ecosystem of JavaScript libraries.

3.8 TypeScript

Description: TypeScript is an open-source language which builds on JavaScript, one of the world's most used tools, by adding static type definitions.

Reasons for Choosing & Advantages: Adopted for the frontend React application to enhance code quality, maintainability, and developer productivity. Static typing helps catch errors early in the development process, improves code readability, and provides better autocompletion and refactoring capabilities in IDEs.

3.9 Vercel

Description: Vercel is a cloud platform for static sites and Serverless Functions that enables developers to host web projects with high performance and easy deployment.

Reasons for Choosing & Advantages: Chosen for deploying the frontend React application due to its seamless integration with GitHub for continuous deployment (CI/CD). Vercel automatically detects Create React App projects, simplifying configuration. It provides features like SSL, a global CDN, and easy environment variable management, making it ideal for hosting modern frontend applications like the one developed for this project.

3.10 Render

Description: Render is a unified cloud platform to build and run applications and websites. It supports deploying web services, static sites, databases, and more.

Reasons for Choosing & Advantages: Selected for deploying the backend FastAPI application. Render offers straightforward GitHub integration for CI/CD, simplifying the deployment process for Python applications. It supports specifying root directories, runtimes, build commands (e.g., pip install -r requirements.txt), and start commands (e.g., for Uvicorn). Render also provides easy management of environment variables, crucial for database connections and API keys.

3.11 Design Patterns Used

Design patterns are general, reusable solutions to commonly occurring problems within a given context in software design. The following patterns were employed in this project:

3.11.1 Repository Pattern

Explanation: This pattern mediates between the domain logic and data mapping layers (database). It provides a collection-like interface for accessing domain objects, abstracting the underlying data storage and retrieval mechanisms.

How and why it's used in the project: Implemented in the backend/app/crud.py module. Functions like create_user, get_patient_by_user_id, update_appointment, etc., encapsulate the SQLAlchemy database interactions for their respective entities. This decouples the API endpoint logic (in routers/) from the specifics of database operations, making the business logic cleaner, more testable, and easier to modify if the data source changes. It centralizes data access logic, improving maintainability.

3.11.2 Dependency Injection

Explanation: A design pattern where an object or function receives other objects or functions (dependencies) that it needs, rather than creating them internally.

How and why it's used in the project: FastAPI heavily utilizes dependency injection. This is evident in API route handlers, such as db: Session = Depends(get_db) and current_user: models.User = Depends(get_current_active_user). FastAPI manages the creation and provision of these dependencies (like database sessions and authenticated user objects), making the route functions more focused on their specific tasks, easier to test (by mocking dependencies), and promoting cleaner code.

3.12.3 Model-View-Controller (MVC) / Model-View-ViewModel (MVVM) like (Frontend)

Explanation: Architectural patterns that separate application concerns. The View is responsible for the UI, the Model manages the data and business logic, and the Controller/ViewModel acts as an intermediary.

How and why it's used in the project: The React frontend application (frontend/src/) adheres to principles similar to MVC/MVVM:

Model: Represents the application data, often fetched from the backend API and managed within React component state (e.g., using useState, useEffect hooks to manage patient lists, appointment details, chat messages in components like DashboardPage.tsx).

View: Comprises the React components that render the user interface based on the current state and props (e.g., LoginPage.tsx, Sidebar.tsx, PatientDashboard.tsx).

Controller/ViewModel Logic: Resides within the React components themselves (or custom hooks/services). This logic handles user interactions (e.g., button clicks, form submissions), makes API calls (using Axios) to fetch or update data, and updates the component's state, triggering UI re-renders. This separation improves code organization, testability, and reusability of UI components.

Chapter 4: Application Development and Deployment

This chapter outlines the architectural design, detailed design specifications, implementation process, and deployment strategy for the completed Clinic Management System.

4.1 Architectural Design

4.1.1 Software Architecture Selection

The project successfully implemented a Client-Server Architecture. This architecture distinctly separates the user interface (client-side) from the business logic and data management (server-side).

- **Client (Frontend):** A Single Page Application (SPA) developed using React and TypeScript. It is responsible for rendering the user interface, handling user interactions, and communicating with the backend API.
- **Server (Backend):** A RESTful API developed using Python with the FastAPI framework. It handles business logic, data processing, database interactions, and AI service integration.

Within the backend, a **Layered Architecture** was adopted to further promote modularity and separation of concerns:

- **Presentation Layer (API Routers):** Located in backend/app/routers/. This layer is responsible for handling incoming HTTP requests, validating request data using Pydantic schemas, invoking appropriate business logic, and formatting HTTP responses.
- **Service/Business Logic Layer:** Primarily encapsulated within backend/app/crud.py and specific logic within router files or dedicated service modules. This layer contains

the core application logic, orchestrates data operations, and implements business rules.
- **Data Access Layer (DAL):** Implemented using SQLAlchemy ORM (backend/app/models.py, backend/app/database.py). This layer abstracts the database interactions, allowing the application to work with Python objects instead of raw SQL queries, and manages database connections and sessions.
- **Database Layer:** PostgreSQL serves as the persistent data store for all application data.

Advantages of this architecture:

- **Separation of Concerns:** Frontend and backend concerns are clearly separated, allowing for independent development, testing, and deployment.
- **Scalability:** Both frontend and backend can be scaled independently based on demand.
- **Technology Flexibility:** Different technologies can be used for frontend and backend.
- **Maintainability:** The layered approach within the backend makes the codebase easier to understand, modify, and maintain.

4.1.2 Overview Design

The system is composed of two primary deployable units: the frontend React application and the backend FastAPI application.

Backend Component: The backend consists of several key modules:

- **API Routers (auth, users, patients, chat):** Define API endpoints.
- **CRUD Module:** Centralizes data access operations.
- **Models Module:** Defines SQLAlchemy ORM classes.
- **Schemas Module:** Defines Pydantic data validation models.
- **Database Module:** Manages database connections and sessions.
- **Config Module:** Handles application settings.
- **Dependencies Module:** Provides reusable authentication/authorization logic.
- **AI Integration (within chat router):** Connects to Google Gemini for chatbot functionality.

Frontend Package: The frontend is structured with:

- **Components (components/):** Reusable UI elements (e.g., forms, layout elements, specific feature components).
- **Pages (pages/):** Top-level components representing different views/screens of the application.
- **Services/API Layer (implicit):** Axios instances and functions for making API calls to the backend.
- **Routing:** Managed by react-router-dom for navigation within the SPA.

- **State Management:** Primarily using React's built-in state and context, or potentially a dedicated state management library if complexity grew.
- **Types (types/):** TypeScript definitions for data structures.

4.1.3 Detailed Design of Functions / Package-Level Design

User Authentication & Authorization (backend/app/routers/auth.py, dependencies.py):

- Handles user registration, password hashing, JWT token generation upon successful login, and token validation for protected routes.
- Dependencies define access control based on user roles (ADMIN, DOCTOR, PATIENT, CLINIC_STAFF).

Patient Management (backend/app/routers/patients.py, crud.py, models.py, schemas.py):

- Defines API endpoints for creating, reading, updating, and deleting patient records.
- Includes logic for assigning doctors to patients and updating EMR summaries.
- SQLAlchemy models define the Patient table and its relationships. Pydantic schemas validate patient data.

Appointment Management (Conceptual, based on proposal):

- API endpoints (likely in a dedicated appointments.py router) allow for creating, viewing (by patient, doctor, or all for staff/admin), updating, and canceling appointments.
- The Appointment model in models.py stores appointment details and links to Patient and Doctor models.

EMR Management (Integrated into patients.py, chat.py, crud.py):

- Doctors can create/update EMRs via specific API endpoints.
- The MedicalReport model likely stores detailed consultation notes, diagnoses, and links to prescriptions.
- The chat.py router also interacts with EMR by logging chat notes.

Chatbot Functionality (backend/app/routers/chat.py):

- Receives messages from the frontend.
- Constructs prompts for the Google Gemini API, incorporating patient EMR data (if available and authorized) and the user's query.
- Sends requests to Gemini and returns the AI's response to the frontend.
- Logs relevant chat interactions or notes into the patient's EMR.

Frontend UI Rendering and Interaction (frontend/src/):

- React components are responsible for rendering different views based on application state and user role.
- User interactions trigger state updates and API calls to the backend.
- DashboardPage.tsx is a key component for staff, integrating patient listing, EMR viewing, and the AI chat interface.

4.2 Detailed Design

4.2.1 UI Design

The user interface was designed to be clean, intuitive, and role-specific, ensuring ease of use for all user types.

Screen Dimensions & Responsiveness: The application is designed as a responsive web application, primarily targeting desktop and tablet use, ensuring usability across various screen sizes.

Color Palette and Typography: A professional and calming color palette was used, primarily featuring blues, grays, and whites, with accent colors (green for success, red for errors) for important actions and feedback. Typography prioritizes readability with clear sans-serif fonts (system defaults like Segoe UI, Roboto, Helvetica Neue).

Common UI Element Styling:

- **Forms:** Consistently styled input fields, labels, and buttons with clear visual hierarchy.
- **Navigation:** A persistent sidebar for authenticated users (Sidebar.tsx), providing role-based navigation links. A top header (BaseDashboard.tsx) includes branding and user account options.
- **Data Display:** Tables and lists are used for displaying patient lists, appointment schedules, and medical records, with clear formatting.
- **Modals/Dialogs:** Used for confirmations, detailed views, or form inputs where appropriate.

Feedback Mechanisms: Visual feedback is provided through toast notifications for actions like successful login/registration or data saving. Loading indicators (spinners or text) are used during data fetching or processing. Form validation messages are displayed inline.

Key UI Screens (Screenshots would typically be included here):

Login Page: A clean interface with fields for email and password, and a link to the registration page.

Registration Page: Form for new patients to create an account.

- **Patient Dashboard:** (For Patients) View upcoming appointments, access limited medical records (prescriptions, appointment history), manage profile, and interact with the chatbot.
- **Doctor Dashboard:** (For Doctors) View appointment schedule, access patient EMRs, tools for creating/updating consultation notes, diagnoses, and prescriptions.
- **Staff Dashboard:** (For Clinic Staff) View and manage patient lists, patient details including EMR summaries, manage clinic appointment schedule, and use the AI symptom checker chat.
- **Admin Dashboard:** (For Administrators) User management interface, system settings, and potentially reporting tools.
- **Chat Interface:** A dedicated modal for interacting with the AI chatbot, displaying conversation history.

4.2.2 Class Design

The primary classes in the system are represented by the SQLAlchemy ORM models (backend/app/models.py) and Pydantic schemas (backend/app/schemas.py).

Backend Data Models (SQLAlchemy - models.py):

- User: Represents all users with attributes like user_id, username, email, hashed_password, role, full_name.
- Patient: Linked to User, stores patient-specific details like date_of_birth, gender, address, phone_number, assigned_doctor_id, and EMR summary (though detailed EMRs might be in MedicalReport).
- Doctor: Linked to User, stores doctor-specific details like major, hospital_id.
- Hospital: Stores hospital information.
- Appointment: Stores appointment details, linking Patient and Doctor, including appointment_day, appointment_time, reason.
- MedicalReport: Stores detailed consultation records, diagnoses, treatment plans, linked to Patient and Doctor.
- ChatMessage: Stores messages for the chat functionality, linked to User.

Backend Data Schemas (Pydantic - schemas.py): These define the structure for API request and response data, ensuring validation. Examples:

- UserCreate, UserSchema, UserUpdate
- PatientCreate, PatientSchema, PatientUpdate, PatientEMRUpdate
- AppointmentCreate, AppointmentSchema, AppointmentUpdate

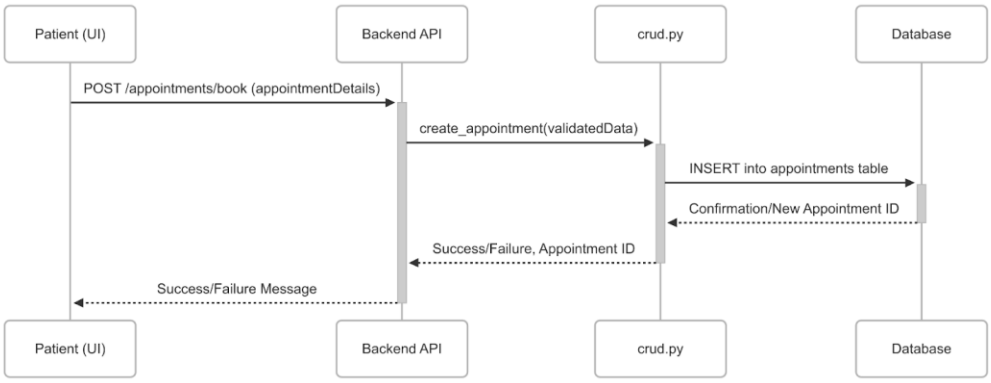
- Token, TokenData
- ChatMessageCreate, ChatResponse
- MedicalReportCreate, MedicalReportSchema, MedicalReportUpdate

Frontend Component Structure (React - frontend/src/):

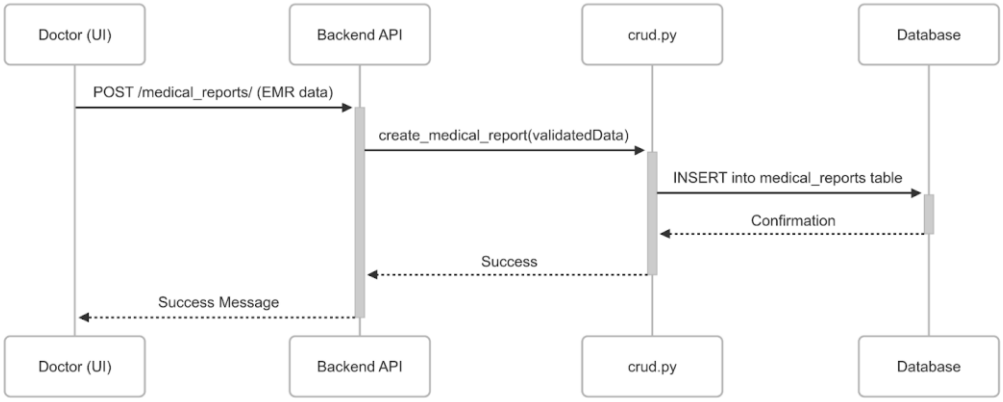
- Container components (in pages/) manage state and logic for specific views.
- Presentational components (in components/) are responsible for rendering UI elements and receiving data via props.
- Service/utility functions handle API calls and data transformations.

Sequence Diagrams:

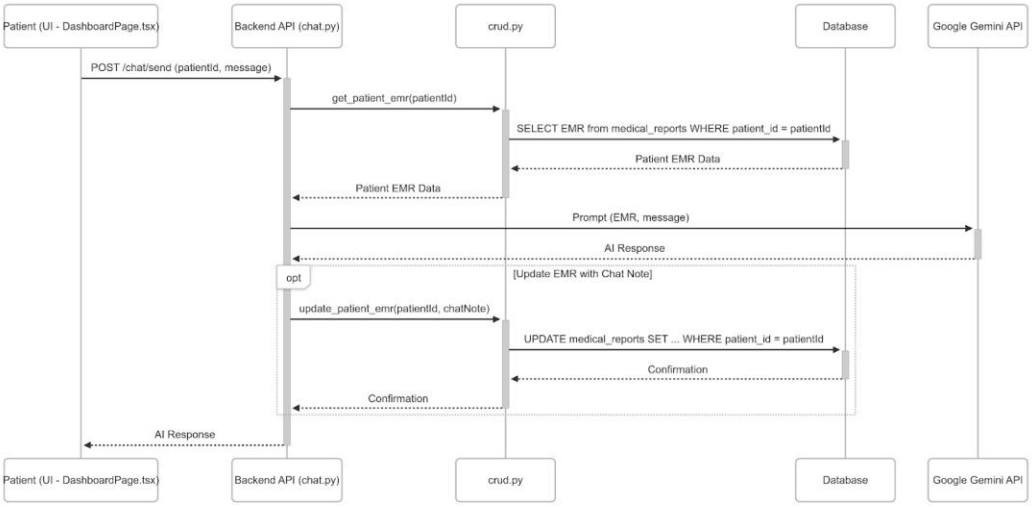
Patient Books Appointment:



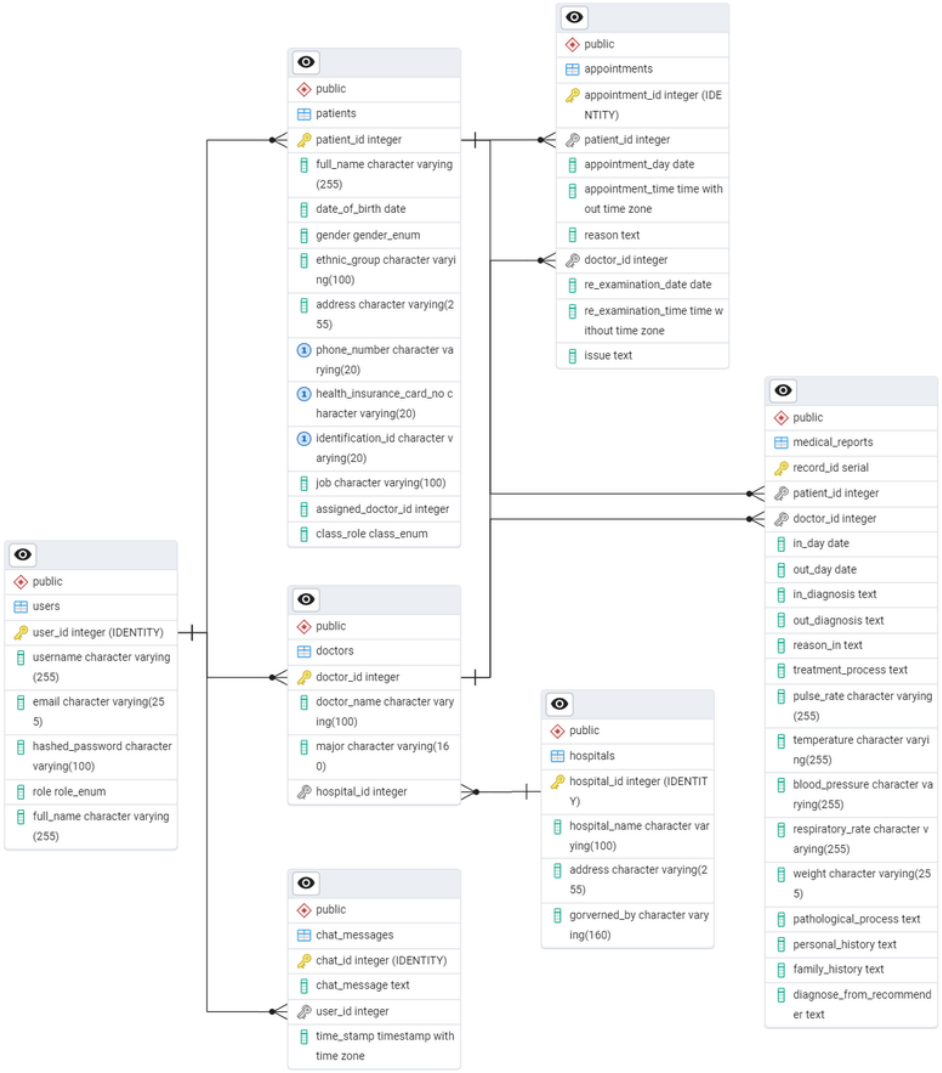
Doctor Creates Medical Record:



Patient Uses Chatbot for Symptom Check:



4.2.3 Database Design



4.3 Implementation

4.3.1 Libraries and Tools Used

Backend (requirements.txt):

Purpose	Tool Name/Version
Web Framework	FastAPI (version 0.115.12)
ORM	SQLAlchemy (version 2.0.41)
Database Driver (PostgreSQL)	psycopg2-binary (version 2.9.10)
Data Validation	Pydantic (version 2.11.5)
ASGI Server	Uvicorn (version 0.34.2)
Password Hashing	passlib, bcrypt
JWT Handling	python-jose
Environment Variables	python-dotenv
AI SDK	google-generativeai(version 0.8.5)

Frontend (frontend/package.json):

Purpose	Tool Name/Version
UI Library	React (version 19.1.0)
HTTP Client	Axios (version 1.9.0)
Routing	react-router-dom (version 7.6.0)

Language	TypeScript (version 4.9.5)
Build/Dev Tool	react-scripts (version 5.0.1)
Notification Library	react-toastify (version 11.0.5)
Markdown Rendering	react-markdown (version 10.1.0)

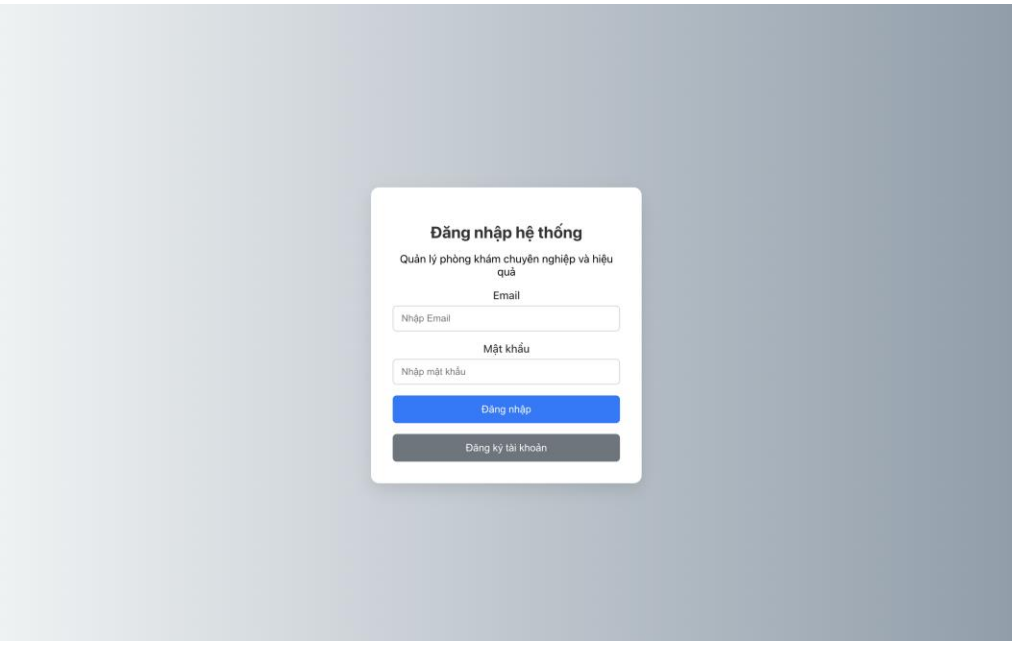
Development Tools:

- **IDE:** Visual Studio Code (assumed common choice).
- **Version Control:** Git & GitHub (repository bnmbanhmi/clinic-management).

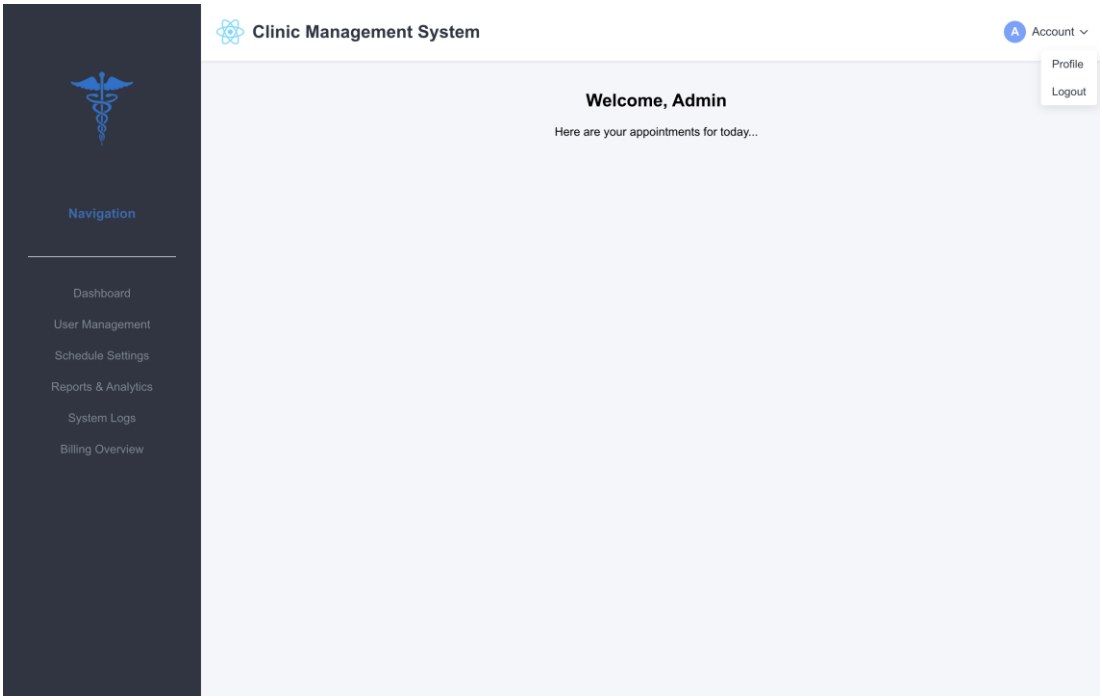
Database Management: PostgreSQL server, pgAdmin or DataGrip.

4.3.2 Demonstration of Main Functions

User Registration & Login:



Role-Based Dashboards:



4.4 Deployment

The Clinic Management System has been successfully deployed, with the frontend hosted on Vercel and the backend on Render, utilizing a Google Cloud SQL for PostgreSQL instance for data persistence. Both services are configured for auto-deployment from the GitHub repository.

Application Structure:

- **Frontend:** React application located in the frontend directory.
  Live URL: clinic-management-nine-lime.vercel.app
- **Backend:** FastAPI (Python) application located in the backend directory.
  Live URL: clinic-management-be3h.onrender.com
- **Database:** Google Cloud SQL for PostgreSQL.
- **Source Code:** github.com/bnmbanhmi/clinic-management

4.4.1 Frontend Deployment (React Application on Vercel)

The React-based frontend application is deployed and hosted on Vercel, leveraging its seamless integration with GitHub for continuous deployment.

1. **Vercel Project Setup:**

A Vercel project was created and connected to the GitHub repository: https://github.com/bnmbanhmi/clinic-management. This connection enables automatic builds and deployments upon code changes to the specified branch.

2. **Project Configuration (via Vercel UI):**
   - **Framework Preset:** Vercel auto-detected the project as a "Create React App."
   - **Root Directory:** Configured to frontend, ensuring Vercel uses the correct directory for build and deployment processes.
   - **Build Command & Output Directory:** Standard Create React App settings were used (npm run build command, and frontend/build as the output directory).
3. **Environment Variables (Frontend Project on Vercel UI):**
   REACT_APP_BACKEND_URL: Set to the live backend URL (clinic-management-be3h.onrender.com) to enable communication between the frontend and backend services.
4. **Deployment & URL:**
   Vercel automatically deploys changes pushed to the main branch (or the designated production branch) of the connected GitHub repository.
5. **Live Frontend URL:** The deployed frontend is accessible at clinic-management-nine-lime.vercel.app.

4.4.2 Backend Deployment (FastAPI Application on Render)

The FastAPI backend application is deployed as a web service on Render, which also supports continuous deployment from GitHub.

1. **Render Project Setup:**
   A "Web Service" was created on Render and linked to the same GitHub repository: github.com/bnmbanhmi/clinic-management.
2. **Service Configuration (via Render UI):**
   - **Root Directory:** Set to backend to specify the location of the backend application code.
   - **Runtime:** Python, automatically detected by Render due to the presence of backend/requirements.txt.
   - **Build Command:** pip install -r requirements.txt (executed relative to the backend directory) to install all necessary Python dependencies.
   - **Start Command:** uvicorn app.main:app --host 0.0.0.0 --port $PORT --workers 1 to run the FastAPI application using Uvicorn, making it accessible on the port assigned by Render.
3. **Backend Dependencies (backend/requirements.txt):**
   A comprehensive requirements.txt file located at backend/requirements.txt lists all Python dependencies for the backend, critically including psycopg2-binary for PostgreSQL database connectivity.
4. **Environment Variables (Backend Service on Render UI):**

- DATABASE_URL: The complete PostgreSQL connection string for the Google Cloud SQL database. Format: postgresql://db_user:db_password@34.67.156.97:5432/db_name.
- SECRET_KEY, ALGORITHM, ACCESS_TOKEN_EXPIRE_MINUTES: Variables critical for JWT authentication and security.
- FRONTEND_URL: Set to the live frontend URL (clinic-management-nine-lime.vercel.app) for CORS configuration and potentially other backend-to-frontend communications.
- MAIL_USERNAME, MAIL_PASSWORD, MAIL_FROM, etc.: Configuration for email services (if implemented).
- GEMINI_API_KEY: API key for accessing the Google Gemini service for the chatbot functionality.
- PYTHON_VERSION: Specified if a particular Python version is necessary for the application.

5. **Deployment & URL:**
   Render automatically deploys new versions of the backend when changes are pushed to the main branch of the GitHub repository.
6. **Live Backend URL:** The deployed backend API is accessible at clinic-management-be3h.onrender.com.

4.4.3 Database Deployment (Google Cloud SQL for PostgreSQL)

The application relies on a Google Cloud SQL for PostgreSQL instance for persistent data storage.

Instance Configuration:

- **Public IP Address:** 34.67.156.97 (used in the DATABASE_URL for the backend).
- **Authorized Networks:** Currently configured to allow access from any IP address (0.0.0.0/0) for ease of development and connection from Render. *For enhanced security in a production environment, this should be restricted to Render's outbound IP addresses or specific necessary ranges.*
- **Hardware:** 1 vCPU, 614.4 MB Memory, 10 GB HDD Storage (Auto storage increase enabled).
- **Edition & Version:** Enterprise edition, PostgreSQL 17.5.
- **Backup & Recovery:**
  Automated backups are enabled.
  Point-in-time recovery is disabled.
  Instance deletion prevention is enabled.
  Backup retention after deletion is disabled.
- **Location & Availability:** The instance is located in us-central1-f and is configured as a zonal (not highly available) instance.

Network Configuration: The Render backend service connects to this Cloud SQL instance using its public IP address.

SSL/TLS: Google Cloud SQL provides SSL certificates. The backend application (via psycopg2) should be configured to use secure connections, potentially by appending ?sslmode=require or similar parameters to the DATABASE_URL if not handled by default by the driver or Cloud SQL proxy.

4.4.4 Post-Deployment Configuration & Connection

1. **Frontend to Backend Link:** The REACT_APP_BACKEND_URL environment variable on Vercel correctly points the frontend to the live backend URL (clinic-management-be3h.onrender.com).
2. **Backend to Frontend Link:** The FRONTEND_URL environment variable on Render ensures the backend is aware of the frontend's live URL (clinic-management-nine-lime.vercel.app), primarily for Cross-Origin Resource Sharing (CORS) configuration.
3. **CORS Configuration:** The FastAPI application's CORS middleware is configured to allow requests from the frontend. For development, the origins list was set to allow all origins.
   In a production setting, this list should be restricted to the specific frontend URL (clinic-management-nine-lime.vercel.app) for security.

4.4.5 Auto-Updates from GitHub (CI/CD)

Both Vercel (monitoring the frontend directory) and Render (monitoring the backend directory) are configured for Continuous Integration/Continuous Deployment (CI/CD).

Any push to the main branch of the https://github.com/bnmbanhmi/clinic-management repository that includes changes within these respective directories will automatically trigger new builds and deployments on Vercel and Render. This ensures that the live applications are always up-to-date with the latest stable code.

Chapter 5: Conclusion and Future Development

5.1 Conclusion

The Clinic Management System project has culminated in the successful delivery of a comprehensive and modern web application, addressing all core requirements initially proposed. This system effectively modernizes the client's previous operations by introducing robust functionalities, including patient management, online appointment booking, electronic medical record (EMR) management, and an innovative AI-powered chatbot for user support.

The chosen client-server architecture, featuring a React frontend and a FastAPI backend, has proven highly effective. This structure ensures a clear separation of concerns, allows for independent scalability of components, and promotes a maintainable codebase. The system's robustness and reliability are further enhanced by the utilization of technologies such as SQLAlchemy for object-relational mapping, Pydantic for data validation, and JSON Web Tokens (JWT) for secure authentication. A significant value addition is the integration of Google Gemini for the chatbot, which provides immediate assistance to users and aids clinic staff with preliminary symptom assessment.

The application successfully implements role-based access control, guaranteeing that Patients, Doctors, Clinic Staff, and Administrators have appropriate and secure access to system features and data. The user interface has been designed with a focus on intuitiveness and user-friendliness, contributing to an enhanced overall user experience. Ultimately, the project has achieved its primary objectives of streamlining clinic operations, improving data management capabilities, and enhancing the quality of service delivery.

The system has been successfully deployed and is fully operational. The frontend React application is hosted on Vercel and is accessible at clinic-management-nine-lime.vercel.app. The backend FastAPI application is deployed on Render, available at clinic-management-be3h.onrender.com. Data persistence is managed by a Google Cloud SQL for PostgreSQL instance. Continuous integration and continuous deployment (CI/CD) pipelines are established through GitHub, ensuring that both Vercel and Render automatically deploy updates from the bnmbanhmi/clinic-management repository, keeping the live applications current with the latest stable code. Environment variables on Vercel link the frontend to the backend, and CORS policies on the backend are configured to permit requests from the live frontend URL, ensuring seamless communication between the services. This successful deployment marks the project's readiness for operational use.

5.2 Future Development Directions

While the current system fulfills its core objectives, the platform is well-positioned for future enhancements and expansions. Potential future development directions include:

Advanced EMR Features:

- Integration with standardized medical coding systems (e.g., ICD-10, SNOMED CT) for diagnoses.
- Support for image attachments (X-rays, lab reports) within the EMR.
- Customizable EMR templates for different specialties or consultation types.
- Drug interaction checking and allergy alerts within the prescription module.

Enhanced Patient Portal:

- Secure messaging between patients and doctors/clinic.
- Online payment integration for appointments or services.
- Patient education resources linked to their conditions or treatments.
- Ability for patients to fill out pre-appointment questionnaires.

Telemedicine Capabilities:

- Integration of video conferencing tools for remote consultations.

Inventory Management:

- Module for managing clinic supplies and pharmacy stock.

Mobile Application:

- Develop native mobile applications (iOS and Android) for patients and doctors to provide on-the-go access.

Enhanced AI Chatbot:

- Expand the chatbot's knowledge base and conversational capabilities.
- Enable more complex interactions, such as triaging patients based on symptoms or providing personalized health tips (with appropriate disclaimers).

The successful completion of this project provides a strong foundation for these future enhancements, ensuring the Clinic Management System can continue to evolve and meet the growing needs of the clinic and its patients.