Ain Shams University
Faculty of Engineering

Graduation Project

Online Clinic(C1)

Under the Supervision of:

Prof. Bassem A. Abdullah

Team members:

Name	ID
Ahmed Yasser Hassan	1809986
Mohamed Elsayed Hassan	1804989
Ahmed Essam Eldin Essa	1808787
Ahmed Magdy Hassan Ali	1807100
Waleed saber Mohamed Hassan	1701664
Salwa Ashraf Mohamed Abd-Allah	1806836
Sally Hesham Mohamed	1804676
Mina Medhat Mounir Luka	1801782
Neven Nabil Shokri	1601626
Antonius Adel Ragheb Naann	1805824

Abstract

The healthcare industry is experiencing constant evolution, particularly in the digital era where all fields are being digitized. To meet the growing demand for improved services and accessibility, this project focuses on developing a comprehensive platform for patients, revolutionizing the way they schedule doctor appointments and communicate with experienced physicians from the comfort of their homes. The platform incorporates various functionalities to enhance accessibility and facilitate medical assistance for patients.

The primary objective of this project is to create an interactive environment that empowers patients to effortlessly find the nearest clinic and explore available appointment slots. Moreover, it aims to provide patients with detailed profiles of physicians, showcasing their extensive experience and the range of services they offer. The platform also facilitates transparency by displaying the fees associated with these services, granting users a wide array of options to choose from.

By leveraging this platform, patients can easily navigate through a user-friendly interface, simplifying the process of finding suitable healthcare providers and scheduling appointments. Additionally, physicians benefit from increased visibility and accessibility to a broader patient base, establishing a seamless connection between medical professionals and individuals seeking medical assistance.

Ultimately, this project strives to enhance the healthcare experience by leveraging digital solutions and bridging the gap between patients and healthcare providers. Through improved convenience, transparency, and accessibility, this comprehensive platform aims to facilitate better and easier access to medical services, ultimately improving the overall healthcare landscape.

1.Table of Contents

2) Introduction

2.1) Purpose

The purpose of this Document to understand platform That include Registration of Doctors and Patient's registration that storing their details including health status into the system and shows how establishing connection between doctors and patients also explain how patients can search and book appointment to get correct diagnoses, all of these features can be shown in front in diagrams that proved in these documents into more details.

2.2) Overview

Platform that allows patients to access healthcare services and resources from anywhere, at any time, through the internet. Online clinics typically provide a range of services, including virtual consultations with healthcare providers, online appointment scheduling and telemedicine

The software for online clinics is designed to provide a secure, user-friendly, and efficient platform for healthcare services. Key features of the software may include:

Patient management: The software should allow patients to create and manage their profiles, including their medical history, test results, and treatment plans

Virtual consultations: The software should provide a platform for virtual consultations with healthcare providers, including video conferencing, chat as communication tools.

Appointment scheduling: The software should allow patients to schedule appointments with healthcare providers cross time and place, either in person or virtually.

Medical records: The software should provide a secure and accessible platform for storing and managing medical records, including test results, treatments, and other relevant information.

Payment processing: The software should provide a platform for secure and efficient payment processing, allowing patients to pay for services and products.

Analytics and reporting: The software should provide data analytics and reporting tools, allowing healthcare providers and administrators to monitor and analyze patient data.

3) General Description

The purpose of this section is to provide essential description of the product's perspective, General Capabilities, General Constraints, User Characteristics and Environment Description, Assumptions and Dependencies

3.1) Product Perspective

Product is one of those medical technologies that serve the live of patients, doctors, and administration easier because patients can easily book their appointment also product help in establishing communication between all users (patients, Doctors, administration) so they can achieve their goals and requirements easily in the context of the medical facility where operations are manually and digitally executed.

the solution is a completely independent framework that includes four separate and interrelated modules that allow the medical facility to perform the following functions:

- Database management
- Medical Personal management
- consultation management
- Payment management

3.2) General Capabilities

Online Clinic provide booking Appointment for patients and change or cancel your Appointment also provides visit user's profile additional to patients will be able to communicate with their doctors and see their profiles.

3.3 General Constraints

- ✓ The website must be speed and organized
- ✓ User Experience and Usability: Prioritize a user-friendly interface and intuitive navigation to facilitate ease of use for patients
- ✓ the system is available all the time

3.4 User Characteristics

Users at all ages can join. But that platform is developed to patients that in specific region.
Busy individuals who have limited time to visit a doctor's office.
People with mobility issues who have difficulty traveling to a doctor's office.
People living in remote or rural areas with limited access to healthcare services.
Patients who prefer the privacy and comfort of receiving medical care from home.
People who are looking for a more cost-effective alternative to traditional healthcare services.
Individuals who are seeking quick and easy access to medical advice or treatment options.
Patients who are seeking a second opinion from a medical professional.

3.5 Environment Description

It must have healthy environment, understandable and simple, we must block the toxic and malicious people.

3.6 Assumptions and Dependencies

The medical facility must have a domain, that can be used for the web interface of the patient module. This web interface must have the rights to access the database of the medical facility.

For the web interface used by patient has a workstation with internet connection.

3.7 Tools

Web Development:

- Front-end: HTML, CSS, JavaScript, and a web development framework such React
- Back-end: Laravel.
- Database management: A relational database management system such as MySQL

Project Management: Knowledge of project management methodologies, such as Agile, Scrum, or Waterfall, and tools such as JIRA or Trello.

Quality Assurance (QA): Knowledge of QA methodologies and tools such as automated testing frameworks and manual testing practices.

Hint: we didn't use automated testing we did manual testing for now

Security: Familiarity with security best practices and tools such as encryption, SSL certificates, and firewalls.

Collaboration and Communication: Familiarity with collaboration and communication tools, such as Slack, Asana, or Google Workspace. Web Development: HTML, CSS, JavaScript, and a web development framework such as Laravel and React

4) Survey for Online Clinics

This section comprises a comprehensive collection of compiled data sourced from multiple reputable websites, offering valuable insights into the dynamic realm of the online clinical system.

The system serves as a remarkable platform, facilitating the exchange of perspectives from both esteemed healthcare professionals and discerning patients, shedding light on the ever-evolving phenomenon of digitalization within the healthcare landscape.

The data encompasses a wide array of viewpoints, spanning a spectrum of opinions, experiences, and observations, all of which contribute to a holistic understanding of the progressive surge in the adoption of digital clinics.

Moreover, the data delves deep into the significant strides made in recent years, highlighting the steady growth and proliferation of digitalization within the healthcare sector, underscoring its transformative impact on the industry.

4.1) Analysis for the general use of online resources for health

This survey presents a concise yet insightful overview of a few noteworthy studies that have examined the increasing usage of online clinical services in recent years. These studies have focused on investigating the resultant behavioral changes within the patient-physician relationship. The findings highlight the growing acceptance and adoption of digital platforms in healthcare and shed light on the evolving dynamics between patients and healthcare professionals.

Background

During the epidemic, isolation became necessary, leading to an increased demand for medical consultancy. Authorities have responded by developing online hospitals, leveraging technology to bridge the gap between patients and healthcare providers.

These online hospitals offer telemedicine, online consultations, and remote monitoring, ensuring accessible and comprehensive care. The development of

online hospitals acknowledge the lasting impact of the epidemic on healthcare and addresses the growing preference for convenient and technology-driven healthcare solutions.

This commitment to patient well-being and safety, along with the use of digital platforms, enhances healthcare outcomes and empowers patients.

Purpose:

The purpose of this essay is to comprehensively assess the diverse viewpoints of several outpatients about their experiences and perceptions of online clinics. By delving into the multifaceted dimensions of this emerging healthcare paradigm, the essay aims to conduct a meticulous analysis of the factors that contribute to the overall agreeability and satisfaction associated with reusing these online systems.

Through a comprehensive examination of patient feedback, the essay seeks to gain valuable insights into the efficacy, convenience, accessibility, and overall effectiveness of these digital platforms in meeting patients' healthcare needs and expectations.

Methods:

The statistics presented in this study were obtained through a comprehensive questionnaire that was administered to a total of 191 patients who had utilized the online clinic services provided by a renowned tertiary hospital located in the Sichuan province.

This survey spanned an extensive period from January to July 2019, ensuring a broad representation of patients' experiences and opinions. Following the collection of this valuable data, a meticulous analysis was conducted, considering various factors that contribute to patient satisfaction.

Results

The study findings revealed that most of the participants, specifically 92.7%, belonged to the young or middle-aged demographic, highlighting the appeal of the online clinic service among these age groups. Additionally, it was noteworthy that 42.9% of the participants possessed a college degree or higher, suggesting a higher level of education among the user base.

When it came to overall satisfaction with the online clinical system, a significant proportion of the respondents, approximately 72.2%, expressed their contentment. Impressively, 31.4% of the participants reported very positive results, indicating a substantial level of satisfaction with the service provided.

Moreover, an overwhelming majority, 91.1% of the participants acknowledged the role of online clinics in enhancing awareness and self-management of their health. This finding highlights the potential impact of online clinic systems in empowering individuals to take a more proactive approach to their healthcare.

Additionally, an encouraging statistic emerged from the study, with 92.1% of the participants expressing their intention to use online clinic systems again in the future. This strong inclination towards continued utilization suggests a high level of confidence and trust in the online clinic service, emphasizing its effectiveness and convenience as an alternative healthcare option.

These findings collectively underscore the positive reception and potential benefits associated with online clinic services, reinforcing the importance of further research and development in this rapidly growing field.

Conclusion:

Comparatively, the findings of this study revealed a lower level of satisfaction among participants with their online outpatient clinic experience, in contrast to similar surveys conducted on traditional outpatient services. This disparity suggests that there are specific areas where the online clinic service may fall short in meeting patient expectations and needs.

However, it is important to note that despite the lower satisfaction levels, many participants reported a positive impact on their health-related self-management awareness after using the online outpatient clinic. This implies that despite some shortcomings, patients still perceive value and benefit in utilizing the online clinic service, leading to an improved sense of empowerment and control over their healthcare.

To address the factors that affected participants' willingness to reuse the online outpatient clinic, these clinics must focus on addressing the imperfections and limitations that hinder patient satisfaction. Key areas of improvement could include streamlining the registration process to make it more user-friendly and efficient, enhancing the functionality of the online platform, and expanding service options such as online examination appointments and follow-up visits.

By simplifying and optimizing these aspects, online outpatient clinics can work towards meeting patient expectations, improving overall satisfaction, and fostering a stronger inclination for patients to continue utilizing these services. This highlights the importance of ongoing development and refinement of online clinic systems to ensure they effectively meet patient needs and deliver a seamless and satisfactory healthcare experience.

4.2) Impact of Internet Use on Health-Related Behaviors

Abstract:

Patients' use of online resources to seek health information is on the rise, but there is a lack of large-scale studies examining the patient-physician relationship in primary care or osteopathic medical settings within this context.

This knowledge gap limits our understanding of how online health information-seeking impacts patient-provider dynamics. Comprehensive research in this area could shed light on communication, shared decision-making, and patient satisfaction. It could also explore the challenges and benefits of online health information seeking, such as misinformation and patient empowerment.

By conducting large-scale studies, we can gain insights that inform evidence-based guidelines and strategies to optimize the use of online resources and improve patient experiences in primary care and osteopathic medical settings.

Objective:

The primary objective of this study is to examine the behaviors related to seeking health information online and assess the outcomes of this information on both the patient-physician relationship and self-care. The research methodology employed in this study involved the distribution of a survey comprising eight questions. The survey focused on healthcare and internet usage and was administered to patients attending three osteopathic primary care clinics.

By gathering data from patients within these clinic settings, the study aimed to gain insights into the patterns and preferences associated with seeking health information online. The survey questions likely covered various aspects such as the frequency of internet use for healthcare purposes, specific information sought, preferred online sources, and the impact of this information on patient interactions with their physicians.

By analyzing the survey responses, the study aimed to evaluate the effects of online health information seeking on the patient-physician relationship. This assessment likely included examining the dynamics of communication, shared decision-making, and patient satisfaction within the context of online health

information use. Additionally, the study likely sought to understand the influence of this information on patients' self-care behaviors and the extent to which it empowered patients to take a more active role in managing their health.

Results:

Out of the 154 survey respondents, a significant proportion of 89 individuals (58%) reported utilizing the Internet as a resource to find health information. Among

these individuals, slightly over half (55%) reported experiencing a change in their perspective regarding their health because of the information they found online.

Furthermore, most of these individuals (46%) reported making subsequent changes in their health-related behaviors. Within this subgroup, the largest segment consisted of individuals aged 31 to 45 years (57%).

These individuals indicated various health-related behavioral changes, including asking more questions during office visits (66%), adhering more closely to physician advice (54%), and making self-directed dietary changes (54%).

Notably, a significant proportion of these patients (73%) informed their physicians about the changes they made based on the health information they obtained online. This suggests a willingness to engage in open dialogue with their healthcare providers, with the belief that physicians are receptive to discussing information obtained online (84%).

Regarding age distribution, approximately 60% of the participants fell within the 31 to 60 years age range, while the remaining respondents were evenly distributed across other age groups. The use of the Internet to seek health information demonstrated an inverse relationship with age, with younger individuals more likely to engage in this behavior.

When it comes to reasons for office visits, most participants (75%) reported seeking medical care for their health concerns.

In summary, of the 154 survey respondents, a significant portion (58%) reported utilizing the Internet to find health information. This subgroup experienced changes in their thinking about health and made subsequent behavioral changes, with a majority being aged 31 to 45 years. They displayed an inclination to discuss these changes with their physicians, reflecting a belief that healthcare providers are

open to conversations about online health information. The age distribution of participants demonstrated that younger individuals were more likely to use the Internet for health information-seeking purposes.

Conclusion

The significant shift in patient-physician dynamics brought about by online information gathering raises various concerns. However, it is important to recognize the higher potential that online information gathering holds in terms of engaging patients in their health maintenance.

By harnessing the power of the web, patients can access a vast amount of health-related information conveniently and efficiently. This empowers individuals to take a more proactive role in managing their own health. With online information, patients can gather insights, educate themselves about various conditions, treatments, and preventive measures, and make informed decisions about their well-being.

Online information gathering also has the potential to enhance health maintenance by promoting preventive care and healthy lifestyles. Patients can access resources that provide guidance on preventive measures, healthy habits, and self-care practices. With readily available information, individuals can make lifestyle changes, monitor their health indicators, and take proactive steps to prevent the onset of certain health conditions.

However, it is crucial to acknowledge the need for critical evaluation and discernment when accessing online health information. The reliability, accuracy, and trustworthiness of online sources vary, and patients should exercise caution and consult reputable sources or healthcare professionals for guidance.

In conclusion, despite concerns surrounding the changing dynamics between patients and physicians due to online information gathering, it is important to recognize the higher potential it holds in terms of patient engagement and health maintenance. With responsible information seeking, patients can empower themselves, actively participate in their healthcare decisions, and adopt preventive measures for improved overall well-being.

5.) System requirements

The digital healthcare booking appointment platform that will be developed aims to fulfill various requirements to ensure a seamless and efficient user experience. Some of the key features and technologies that will be incorporated in the platform include:

5.1) Functional Requirements

The digital healthcare booking appointment platform will include a comprehensive set of features to cater to the needs of doctors, clinics, and patients. The platform will incorporate the following functionalities:

1. Doctor/Clinic Registration: Doctors and clinics can sign up and create their profiles on the platform, providing information about their services, fees, and availability for appointments.

2. Patient Registration: Patients can sign up and add their personal information, including their medical history and relevant details.

3.Doctor/Patient/Admin login: they can normally login with their registered accounts

4. Search and Filter Functionality: Patients can search for doctors or clinics based on their area, specialization, or doctor's name. The platform will provide advanced filtering options to help patients find the right healthcare professional cross time and place.

5. Global chatting: Users can use fast-chatting property to share a massage with all available doctors/users in the website and save the answers of doctors.

6. Appointment Booking: Patients will be able to choose a doctor, view their available time slots, and manage their appointments accordingly. The platform will offer a seamless appointment booking process for patients.

7. Appointment Cancellation: Patients should have the ability to cancel existing appointments if needed, providing flexibility and convenience.

8. Rating and Feedback: Patients can rate and comment on the services they receive once to each doctor, allowing others to make informed decisions and providing valuable feedback to doctors and clinics.

9. Patient Transfer: Admin can transfer patients to another doctor if necessary and notify the patients about the change.

10. Access to Medical History: Doctors will have access to patients' medical history, enabling them to provide personalized care and make informed decisions.

11. Appointment Management for Doctors: Doctors can manage their appointments, view their schedule, and make necessary adjustments.

12. Dashboard Management: The platform will include features for managing doctors' and patients' Dashboard, allowing for efficient administration of healthcare professionals.

13. Statistical Information Collection: Administrators will be able to collect statistical information from the system, enabling analysis and insights for decision-making purposes.

14. Online Payment: Patients can pay for their medical services securely through the platform, providing a convenient payment option.

15. Test Results Accessibility: The system will display test results to patients and store them in their records, accessible to doctors or authorized individuals responsible for managing patient records.

These features will collectively create a robust and user-friendly digital healthcare booking appointment platform that enhances the overall experience for doctors, clinics, and patients while ensuring efficient management of medical services.

5.2) Non-functional Requirements

System Availability:

1. The system should maintain high availability, ensuring it is always accessible.

2. Clinics should be available during normal working hours.

3. The online clinic system should be accessible to doctors during their designated working hours or available time.

Performance of the System:

4. The system should provide a fast and responsive user experience, allowing every user to sign in within 5 seconds.

Security of the System:

5. The online clinic system must prioritize the privacy and confidentiality of test results, ensuring they are securely transmitted to patients or informal caregivers.

6. Patients should be able to make secure payments for their appointments within the system.

Consistency of the System:

7. If a patient misses their appointment, they should not be able to register for a new appointment without fulfilling the payment requirement.

8. Doctors profiles and accounts should be verified by admins

These non-functional requirements set expectations for the system's availability, performance, security, consistency, and the technologies to be used in its development. Adhering to these requirements will contribute to a reliable, efficient, secure, and consistent online clinic system.

6. Requirements Validation

6.1 Source Traceability Matrix

REQ	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	1	2	3	4	5	6	7
1																						
2																						
3	D	D			R	D	R	R				D		D							R	R
4																					R	R
5	R	R	D																		R	R
6			D				R							R							R	R
7			D			R															R	R
8			D																		R	R
9	D	D				R				R		R									R	R
10	D	D	R			R					D											
11	R		D			R	R															
12	R	R	D										R									
13			D			R		R							R							
14	R	R	D			D	R				R											
15	R	R	D			R															R	R
1				R									R				R	R				
2				R												R		R				
3				R												R	R					
4			D				D									R	R	R				
5	R	R				R									R							
6						R								D								
7	R	R	D		R									D								
8	D																					

D: Requirement on the row depends on the requirement on the corresponding column.
R: Requirement on the row has other weaker relationship with the requirement on the corresponding column.

6.2 Requirements Traceability Matrix

Requirement	User	Front-End	Back-End	Database
FR 1	✓	✓	✓	✓
FR 2	✓	✓	✓	✓
FR 3	✓	✓	✓	
FR 4	✓		✓	
FR 5	✓		✓	
FR 6	✓		✓	
FR 7	✓		✓	
FR 8	✓		✓	
FR 9	✓	✓		
FR 10	✓	✓	✓	
FR 11	✓			✓
FR 12	✓	✓	✓	✓
FR 13	✓	✓	✓	✓
FR 14	✓			✓
FR 15	✓		✓	✓
FR 16	✓	✓	✓	✓
NFR 1				✓
NFR 2				✓
NFR 3			✓	✓
NFR 4			✓	✓
NFR 5		✓		✓
NFR 6			✓	✓

7. Database Management

MySQL

7.1 Introduction

MySQL is a feature-rich and widely acclaimed open-source relational database management system (RDBMS) that has been driving countless applications and websites across the globe for several decades. Since its inception, MySQL has established itself as a robust and flexible solution for storing, managing, and retrieving vast amounts of data efficiently.

At its core, MySQL follows the principles of a relational database, organizing data into structured tables. These tables consist of rows and columns, with each column representing a specific attribute and each row containing a set of related data.

One of MySQL's standout features is its support for the structured query language (SQL), a universal standard for interacting with relational databases. SQL provides a comprehensive set of commands and statements that enable developers to create, modify, and query databases. MySQL's adherence to SQL ensures compatibility with existing applications and simplifies the development process for those already familiar with the language.

MySQL's client-server architecture facilitates concurrent access to the database, allowing multiple users to work with the data simultaneously. The server component, often referred to as the MySQL server or MySQL daemon, handles the storage, retrieval, and manipulation of data. Clients, which can be applications or other servers, communicate with the MySQL server to perform operations on the database.

The versatility of MySQL is evident in its cross-platform compatibility, supporting a wide range of operating systems and development environments. Whether you are working on a Windows, macOS, Linux, or Unix system, MySQL can seamlessly integrate into your preferred setup.

MySQL's integration capabilities extend beyond the realm of operating systems. It seamlessly interfaces with numerous programming languages, including but not limited to PHP, Python, Java, and C++.

Security is a paramount concern when it comes to managing data, and MySQL addresses this with a robust set of security features. It provides mechanisms for user authentication, access control, and data encryption, ensuring that sensitive information remains protected from unauthorized access.

In addition to security, MySQL also offers backup and recovery options to ensure data durability and reliability. Regular backups allow you to protect your data against accidental loss, hardware failures, or other unforeseen events. With proper backup strategies in place, you can quickly restore your database to a previous state, minimizing downtime and maintaining data integrity.

7.2 Comparison (SQL vs no-SQL)

	SQL	NoSQL
Data Model	Relational model, with tables and predefined schema.	Diverse data models, including document-based (key-value pairs, JSON-like documents), columnar, graph, and more.
Structure	Structured and fixed schema.	Dynamic and flexible schema, allowing for schema-less data.
Querying	Structured Query Language (SQL)	Various query languages, including proprietary query languages specific to the database. Some NoSQL databases also support SQL-like querying.
Scalability	Vertical scaling (hardware upgrades) and Horizontal scaling (shading).	Horizontal scaling (shading) for high scalability.
Transactions	ACID properties supported for maintaining data integrity in complex transactions.	Limited ACID properties. NoSQL databases typically sacrifice some ACID properties for higher scalability and performance.
Flexibility	Rigid structure, where changes in schema require modifying the entire database.	Flexible structure, allowing for easy addition, modification, and deletion of fields without impacting other documents.
Data Integrity	Strong data consistency and referential integrity constraints.	Eventual consistency model.
Joins	Supports complex joins between tables.	NoSQL databases generally do not support complex joins across collections.
Scalability	Vertical scaling (hardware upgrades) and Horizontal scaling (shading).	Horizontal scaling (shading) for high scalability.
Use Cases	Traditional applications with complex relationships and transactions.	Large-scale, high-traffic web applications, real-time analytics, content management systems.
Examples	MySQL, PostgreSQL, Oracle, Microsoft SQL Server.	MongoDB, Cassandra, Redis, Couchbase, Amazon DynamoDB.

7.3) Design of database

7.3.1) Design Requirements

We designed a database for an online clinic describes:

- The system manages users' information and doctors' data also their clinics and available appointments
- The system provides publishing posts, comments and likes.
- The appointments can be either chatting or threw video call.
- Each user has the ability of chatting with doctor.
- Each user can rate and give feedback to doctors.

The system support user reports with report details.

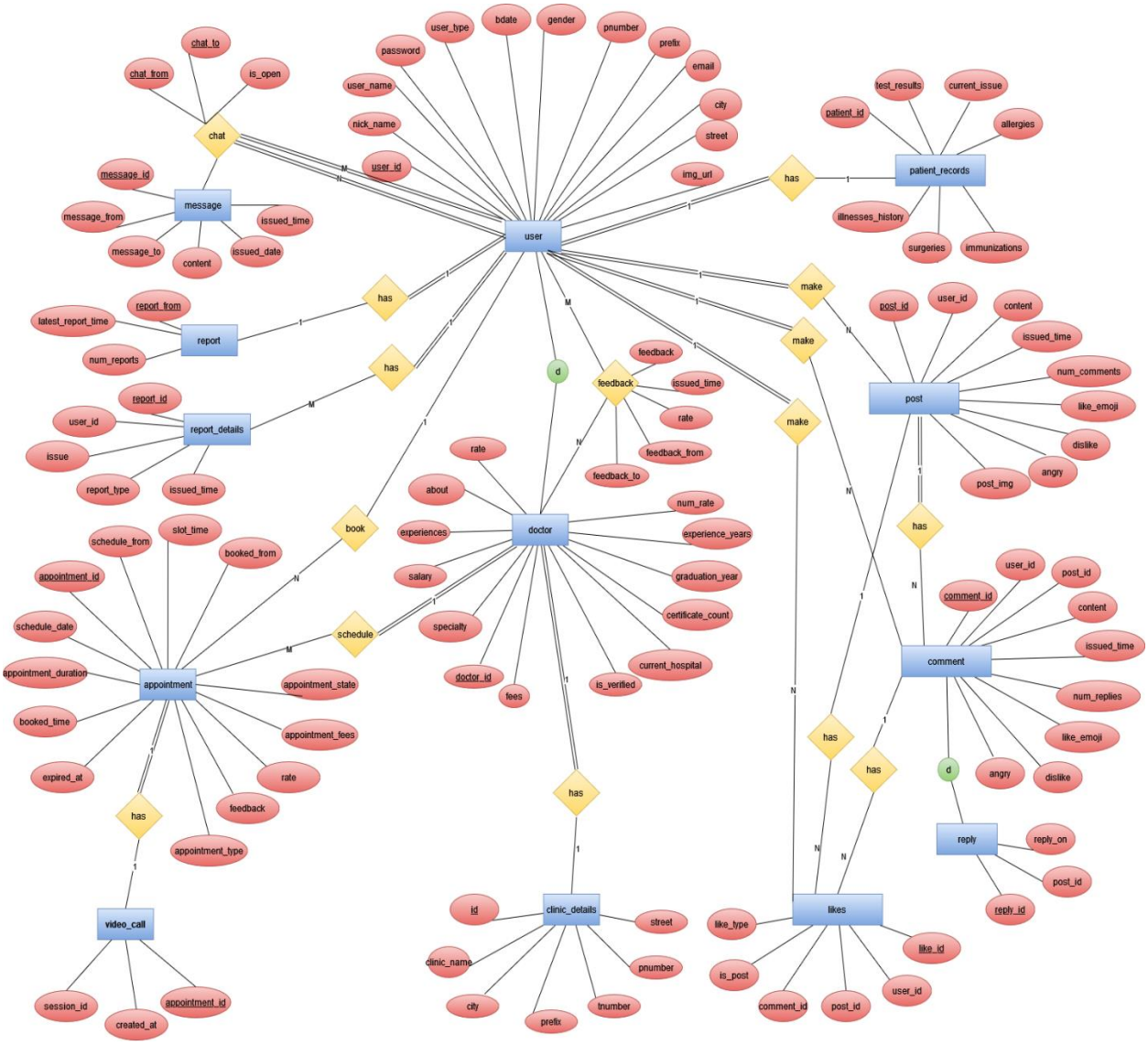
7.3.2) Constraints

- Users may be classified as doctors.
- Each doctor may have one clinic.
- Each clinic must belong to one doctor only.
- Each doctor may schedule one or more appointments.
- Only one doctor must schedule each appointment.
- Each user may book one or more appointments.
- Only one user may book each appointment.
- Each appointment may have a video call.
- Each video call must belong to one appointment.
- Each user may have one report.
- Each report must belong to one user.
- Each user may have one or more report details.
- Each report details must belong to one user.
- Each user may have one patient record.
- Each patient record must belong to only one user.
- Each user can make one or more posts.

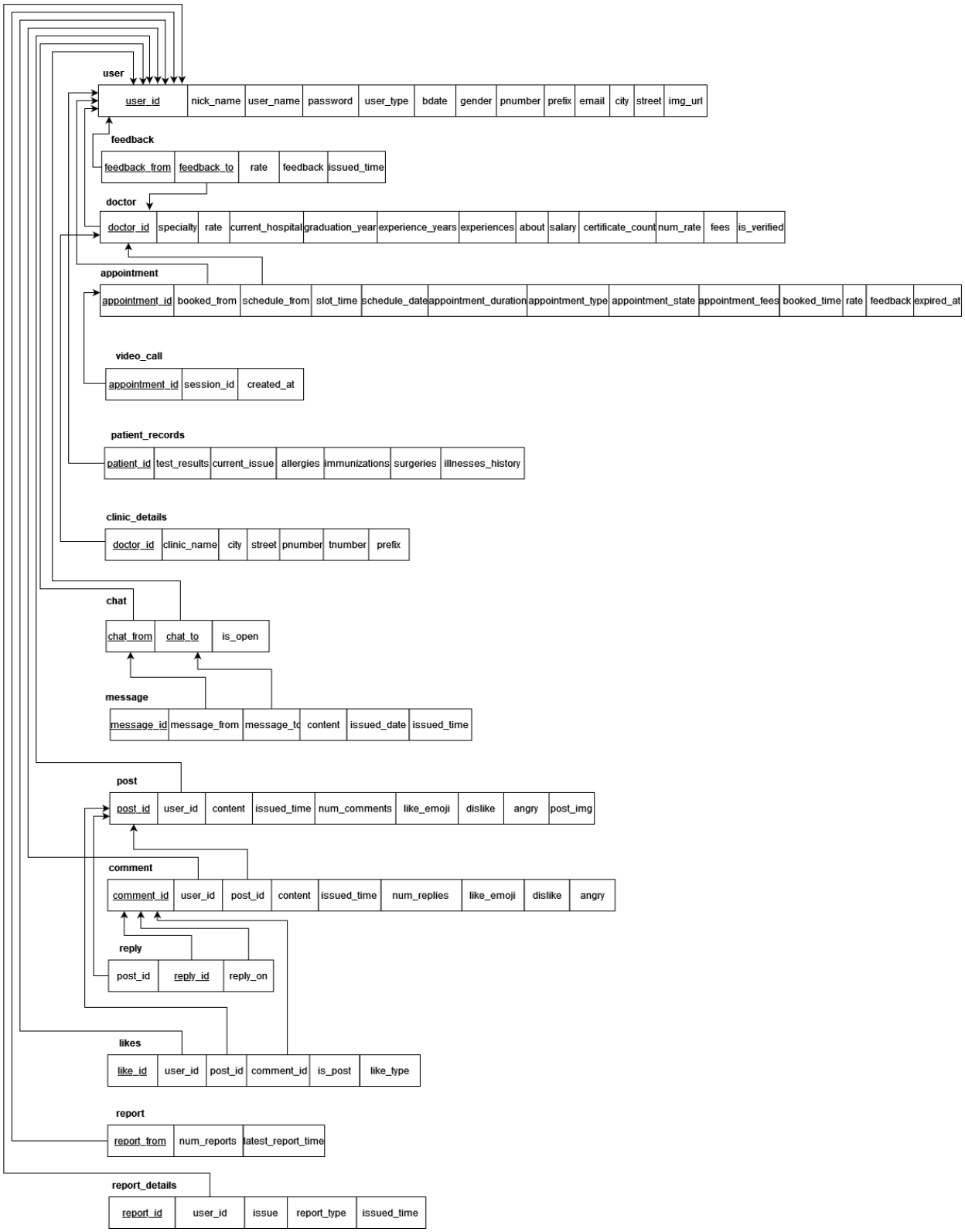
- Each post must belong only to one user.
- Each post may have one or more comments.
- Each comment must belong to only one post.
- Each user can make one or more comments.
- Each comment must belong to only one user.
- Comment may be classified as reply.
- Each comment may have one or more likes.
- Each post may have one or more likes.
- Each like must belong to only one comment or one post.
- Each user may have one or more likes.
- Each like must belong to only one user.
- Each user may give doctors one or more feedback.
- Each doctor may take from users one or more feedback.
- Each user must have one or more chat with another user.
- Each chat may contain one or more messages.

7.4) Database Diagrams

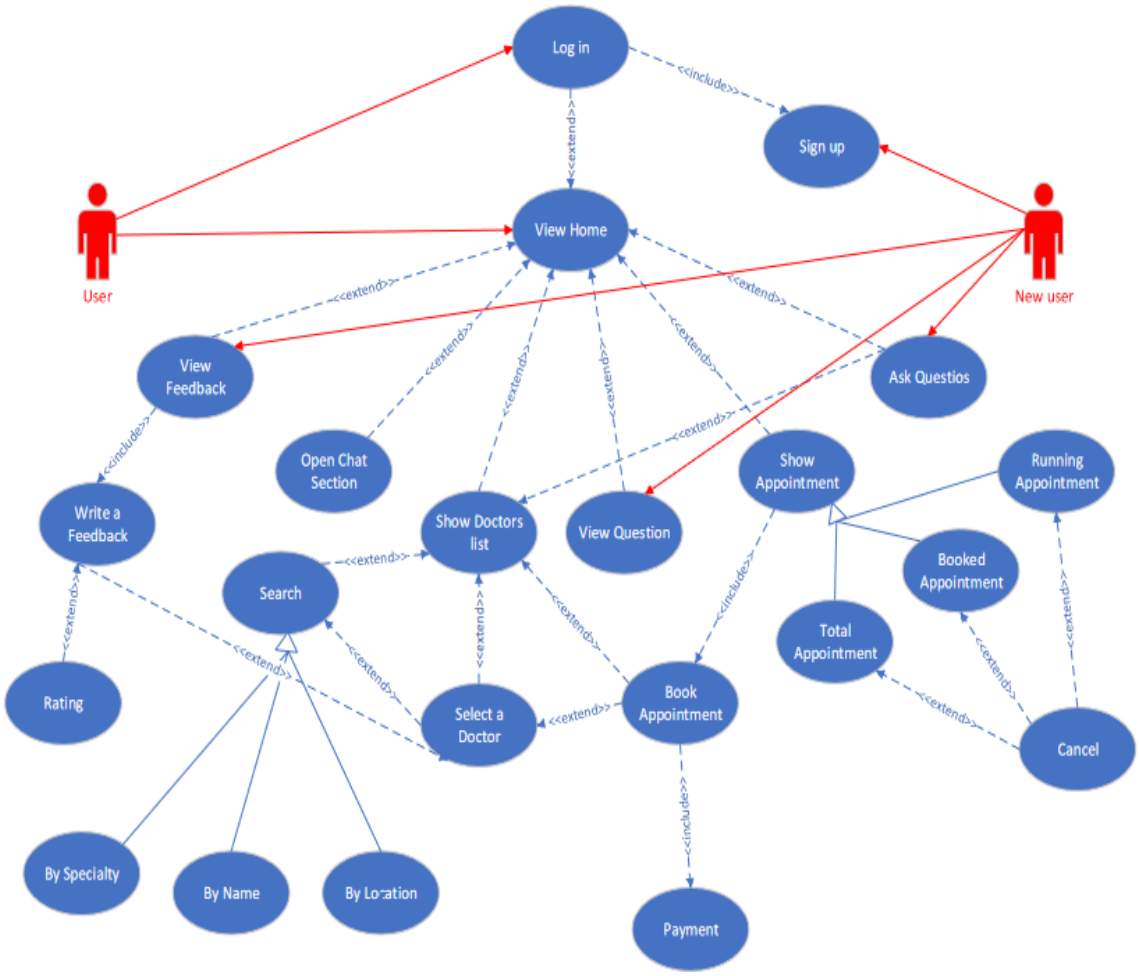
7.4.1) EER diagram

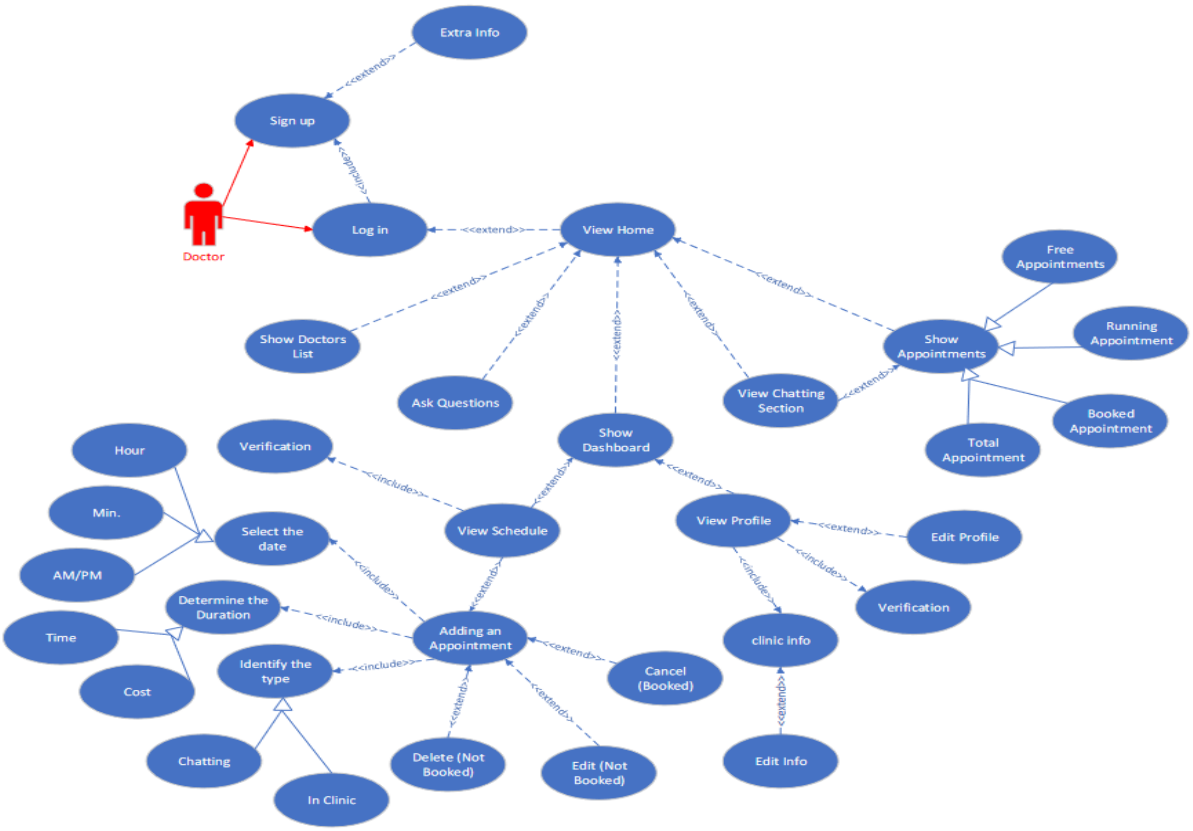


7.4.2) Referential schema diagram



7.4.3) Use Case Diagrams




Extra Info
Sign up
Doctor
Log in
View Home
Free Appointments
Show Doctors List
Show Appointments
Running Appointment
Ask Questions
View Chatting Section
Booked Appointment
Total Appointment
Hour
Verification
Show Dashboard
Min.
Select the date
View Schedule
View Profile
Edit Profile
AM/PM
Determine the Duration
Verification
Time
Adding an Appointment
clinic info
Identify the type
Cancel (Booked)
Cost
Chatting
Delete (Not Booked)
Edit (Not Booked)
Edit Info
In Clinic
<<extend>>
<<include>>


Admin
Log in
View Home
View Catting Section
View Feedback
View Doctors List
View Ratings
View Dashboard
View Questions
Search by name
Doctor Management
View Profile
Verify or Reject
User Management
View Details
Restrict
Search by name
Delete
<<extend>>
<<include>>

7.4.4) Tabular Description of Use Cases

Use Case Name	**Sign up**
Related Requirements	-
Goal in Context	A new user wants to sign up to the site.
Preconditions	-
Failed End Condition	The user signed up and Login screen is viewed.
Primary Actors	New user/ New Doctor
Main Flow	1. Open sign-up Screen. 2. Enter nickname, username, password, confirm password, gender, birthday, address (city and street), phone, email and photo. 3. Select user or doctor. 4. Press submit. 5. Login screen is viewed.

Use Case Name	**Login**
Related Requirements	Sign up
Goal in Context	An existing user wants to login to the site.
Preconditions	The user has already signed up on the website.
Failed End Condition	The user cannot login and stay in login screen.
Primary Actors	User/ Doctor
Main Flow	1. Open login Screen. 2. Enter username and password. 3. Press submit. 4. Include::Sign up: Check if the user has signed up by checking the username and password in the database. 5. Home screen is viewed.

Use Case Name	**Ask Questions**
Related Requirements	-
Goal in Context	A new, an existing user, or a doctor wants to ask
Preconditions	The user has already signed up on the website.
Successful End Condition	The question is posted successfully on the site
Failed End Condition	The question is not posted successfully on the site
Primary Actors	A new, an existing user, or a doctor
Main Flow	1. Open the website. 2. Enter the question in the questions section. 3. Press post.

Use Case Name	**View Questions**
Related Requirements	Ask question.
Goal in Context	A new, an existing user, or a doctor wants to view
Preconditions	A new, an existing user, or a doctor wants to view.
Successful End	The user viewed questions successfully.
Failed End Condition	The user sees a blank page.
Primary Actors	A new, an existing user, or a doctor.
Main Flow	1. Open the website. 2. Open questions section.

Use Case Name	**Show User Appointments**
Related Requirements	Booking Appointment.
Goal in Context	A user wants to show his appointments.
Preconditions	The user has already booked an appointment.
Successful End	The user sees the details of his appointments.
Failed End Condition	The user sees a blank page on his appointment page.
Primary Actors	User
Main Flow	1. Open my appointment page. 2. Include::Book Appointment: Check if the user has booked an appointment by checking the user database. **3.** See the appointment details in the total appointment tab (status, day, date, appointment time, remaining time, and doctor details).

Use Case Name	**Book an Appointment**
Related Requirements	Login
Goal in Context	A user wants to book an appointment
Preconditions	Choose a specific slot from the available
Successful End	The user booked the appointment successfully.
Failed End Condition	The booking process failed.
Primary Actors	User
Main Flow	1. Open the selected doctor profile. 2. Include: The system Checks available slots. 3. The available slots appear on the screen. 4. Select the appropriate slot. 5. Click on book your appointment. 6. Include: Payment 7. The user paid. 8. A successful booking message viewed on the screen with the appointment details.

Use Case Name	**Payment**
Related Requirements	Login and book an appointment.
Goal in Context	A user wants to pay for their booking.
Preconditions	The user has a valid credit card.
Successful End	The user paid successfully.
Failed End Condition	The system rejects the payment.
Primary Actors	User
Main Flow	1. Click Book Now. 2. The payment screen appears. 3. The user enters his email, card number, expiration date, CVC, and the country. 4. The user clicks on pay Now. 5. Include:: The system checks the user information. 6. The appointment has been booked. 7. Include:: The system adds this booking in your appointment page.

Use Case Name	**Show Doctors' List**
Related Requirements	-
Goal in Context	A user wants to see doctors' list.
Preconditions	There are doctors available on the site.
Successful End	The user saw doctors' list successfully.
Failed End Condition	The user sees a blank page.
Primary Actors	User/ Doctor
Main Flow	1. Open the site. 2. Select doctors' list. 3. See all the doctors' details.

Use Case Name	**View Feedback**
Related Requirements	Write a Feedback.
Goal in Context	A Feedback has already been written.
Preconditions	The user viewed Feedbacks successfully.
Successful End	The user viewed questions successfully.
Failed End Condition	The user sees a blank page.
Primary Actors	A new or an existing user.
Main Flow	1. Open the website. 2. Open Feedback section.

Use Case Name	**Write a Feedback**
Related Requirements	Login
Goal in Context	A user wants to write a Feedback
Preconditions	The user has already logged in.
Successful End	The user wrote a Feedback successfully.
Failed End Condition	The process failed.
Primary Actors	User
Main Flow	1. Open the doctor profile. 2. Click on place your Feedback. 3. Write the feedback. 4. Click Submit your Feedback.

Use Case Name	**Open the Chat Section**
Related Requirements	Login, Book an appointment.
Goal in Context	A user wants to chat with the doctor
Preconditions	The time of the appointment has started
Successful End	The user chatted with the doctor successfully
Failed End Condition	The process failed.
Primary Actors	User/ Doctor
Main Flow	1. Open the chat section. 2. Select the doctor chat page. 3. Wait till the slot starts. 4. Chat with the doctor. 5. The appointment ends.

Use Case Name	**Show Doctor Dashboard**
Related Requirements	Login
Goal in Context	A doctor wants to open his dashboard
Preconditions	-
Successful End	The doctor opens his dashboard page successfully.
Failed End Condition	The process failed.
Primary Actors	Doctor
Main Flow	1. Login. 2. Select the doctor dashboard page.

Use Case Name	**View Doctor Profile**
Related Requirements	Show Doctor Dashboard.
Goal in Context	A doctor wants to open his profile and edit it.
Preconditions	Admin should verify this doctor.
Successful End	The doctor opens his profile page and edits it successfully
Failed End Condition	Admin should verify this doctor.
Primary Actors	Doctor
Main Flow	1. Login. 2. Select the doctor dashboard page. 3. Select the profile page. 4. Include:: Verification. 5. Include:: Adding his clinic information.

Use Case Name	**Verification**
Related Requirements	View doctor dashboard.
Goal in Context	Verify doctor account.
Preconditions	Doctor has appropriate information.
Successful End	The admin verifies the doctor's account.
Failed End Condition	The admin rejects this doctor account.
Primary Actors	Admin
Main Flow	1. Login. 2. Select admin dashboard page. 3. Select doctor management page. 4. Verify doctor’s account.

Use Case Name	**View Doctor Schedule**
Related Requirements	Show Doctor Dashboard.
Goal in Context	A doctor wants to open his schedule and edit it.
Preconditions	Admin should verify this doctor.
Successful End	The doctor opens his schedule page and edit it successfully.
Failed End Condition	The admin rejects this doctor.
Primary Actors	Doctor
Main Flow	1. Login. 2. Select the doctor dashboard page. 3. Select the schedule page. 4. Include:: Verification. 5. Check his schedule.

Use Case Name	**Adding New Appointments**
Related Requirements	Show Doctor's schedule.
Goal in Context	A doctor wants to add a new appointment and edit it.
Preconditions	Admin should verify this doctor.
Successful End	The doctor adds new appointment and edit it successfully.
Failed End Condition	The appointment hasn't been added to the database.
Primary Actors	Doctor
Main Flow	1. Login. 2. Select the doctor dashboard page. 3. Select the schedule page. 4. Include:: Verification. 5. Click new appointment. 6. Select time slot (hour, minute and AM/PM). 7. Select the appointment duration, fees, type 8. Click add now. 9. Include: The system adds the appointment to the database. 10. View schedule page.

Use Case Name	**Show Doctor Appointments**
Related Requirements	Adding Appointment.
Goal in Context	A doctor wants to show his appointments.
Preconditions	The doctor has already added an appointment.
Successful End	The doctor sees the details of his appointments.
Failed End Condition	The doctor sees a blank page on his appointment
Primary Actors	Doctor
Main Flow	1. Open my appointment page. 2. Include::Add an Appointment: Check if the doctor has added an appointment by checking the doctor database. 3. See the appointment details in the total appointment tab (status, day, date, appointment time, remaining time, and patient rating).

Use Case Name	**Open the Chat Section**
Related Requirements	Login, Add an appointment.
Goal in Context	A doctor wants to chat with the patient.
Preconditions	The time of the appointment has started.
Successful End	The doctor chatted with the patient successfully.
Failed End Condition	The process failed.
Primary Actors	Doctor
Main Flow	1. Open the chat section. 2. Select the patient chat page. 3. Wait till the slot starts. 4. Chat with the patient.

Use Case Name	**Show Admin Dashboard**
Related Requirements	Login
Goal in Context	An admin wants to open his dashboard.
Preconditions	-
Successful End	The admin opens his dashboard page successfully.
Failed End Condition	The process failed.
Primary Actors	Admin
Main Flow	1. Login. 2. Include: Check the admin's email and password in the database. 3. Select the admin dashboard page.

Use Case Name	**Show Doctor Management Page**
Related Requirements	Login, Show admin dashboard.
Goal in Context	An admin wants to open doctor management page.
Preconditions	-
Successful End	The admin opens doctor management page successfully
Failed End Condition	The process failed.
Primary Actors	Admin
Main Flow	1. Login. 2. Include: Check the admin's email and password in the database. 3. Select the admin dashboard page. 4. Select the doctor management page.

Use Case Name	**View Questions**
Related Requirements	Ask question.
Goal in Context	A new, an existing user, or a doctor wants to view
Preconditions	A new, an existing user, or a doctor wants to view.
Successful End	The user viewed questions successfully.
Failed End Condition	The user sees a blank page.
Primary Actors	User/ Doctor
Main Flow	1. Open login Screen. 2. Enter username and password. 3. Press submit. 4. Include::Sign up: Check if the user has signed up by checking the username and password in the database. 5. Home screen is viewed.

Use Case Name	**Show Admin Profile**
Related Requirements	Login, Show admin dashboard.
Goal in Context	An admin wants to open his profile page.
Preconditions	-
Successful End	The admin opens his profile page successfully.
Failed End Condition	The process failed.
Primary Actors	Admin
Main Flow	1. Login. 2. Include: Check the admin's email and password in the database. 3. Select the admin dashboard page. 4. Select the profile page.

Use Case Name	**Show User Management Page**
Related Requirements	Login, Show admin dashboard.
Goal in Context	An admin wants to open user management page.
Preconditions	-
Successful End	The admin opens user management
Failed End Condition	The process failed.
Primary Actors	Admin
Main Flow	1. Login. 2. Include: Check the admin's username and password in the database. 3. Select the admin dashboard page. 4. Select the user management page.

7.4.5) Class Model

CRC

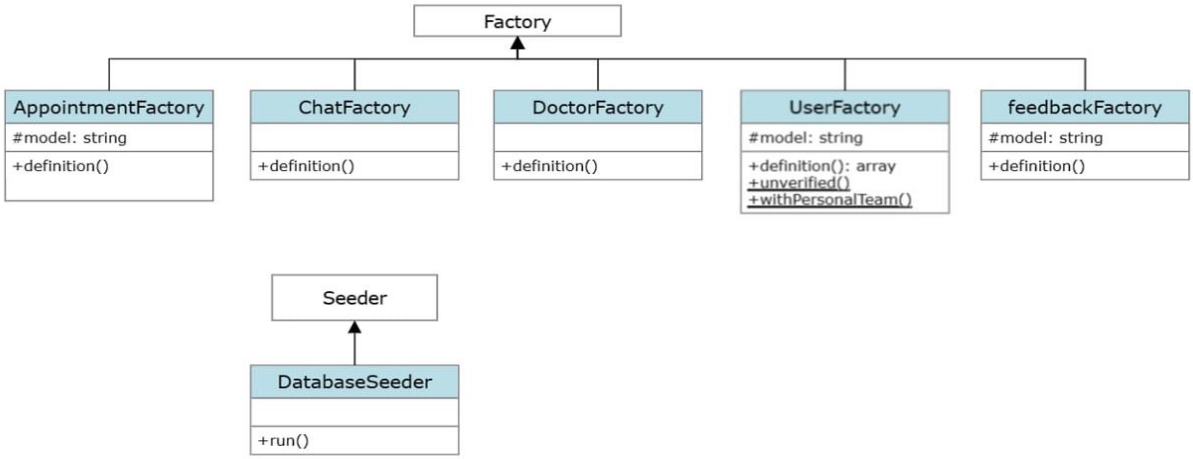
Class **patient**
Responsibility 1. Create new account 2. Send login data to class system controller to check it 3. Update personal data 4. Show personal info 5. Make posts 6. reply to post 7. Do feedback on doctor 8. Book appointment 9. Update booking 10. Delete booking 11. Chat with doctor 12. Search doctors 13. Filter doctors 14. Delete account
Collaboration 1. System Controller class 2. Admin class 3. Doctor class

Class Admin
Responsibility 1. Verify doctor 2. Block doctor 3. Delete doctor or patient 4. Open chat for doctor
Collaboraton 1. patient class 2. Doctor class 3. System Controller class

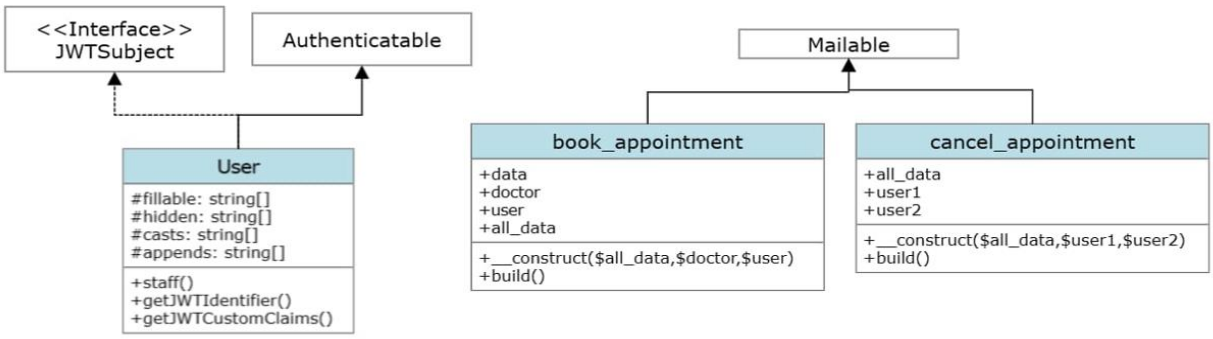
Class doctor
Responsibility 1. Create new account 2. Send login data to class system controller to check it 3. Update personal data 4. Show personal info 5. Make posts 6. Reply to post 7. Make appointment 8. Update appointment 9. Delete appointment 10. Chat with patient 11. Do feedback on another doctor 12. Search doctors 13. Filter doctors
Collaboraton 1. patient class 2. Admin class 3. System Controller class

Class System Controller
Responsibility 1. Verify doctor or patient sign up 2. Verify doctor or patient login 3. Show doctors to patient 4. Manage time slot with doctor 5. Show time slot to user 6. Manage appointment register with patient 7. Show appointment to user 8. Open chat between doctor and user 9. Show doctor info 10. Show patient info 11. Show patient history 12. Show doctor feedbacks 13. Show posts and reply 14. Delete doctor or patient account
Collaboraton 1. patient class 2. Admin class 3. doctor class

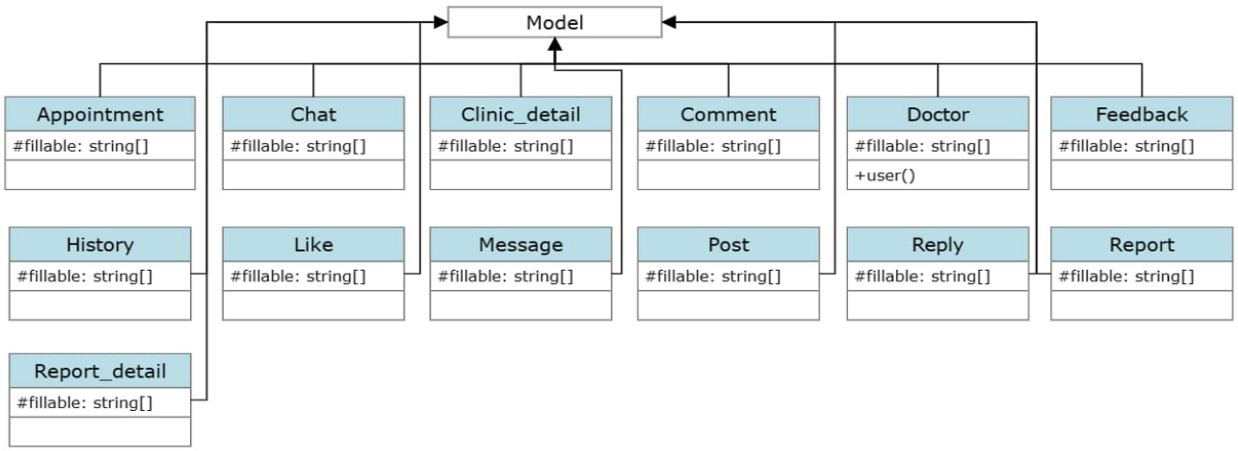
Factory Classes



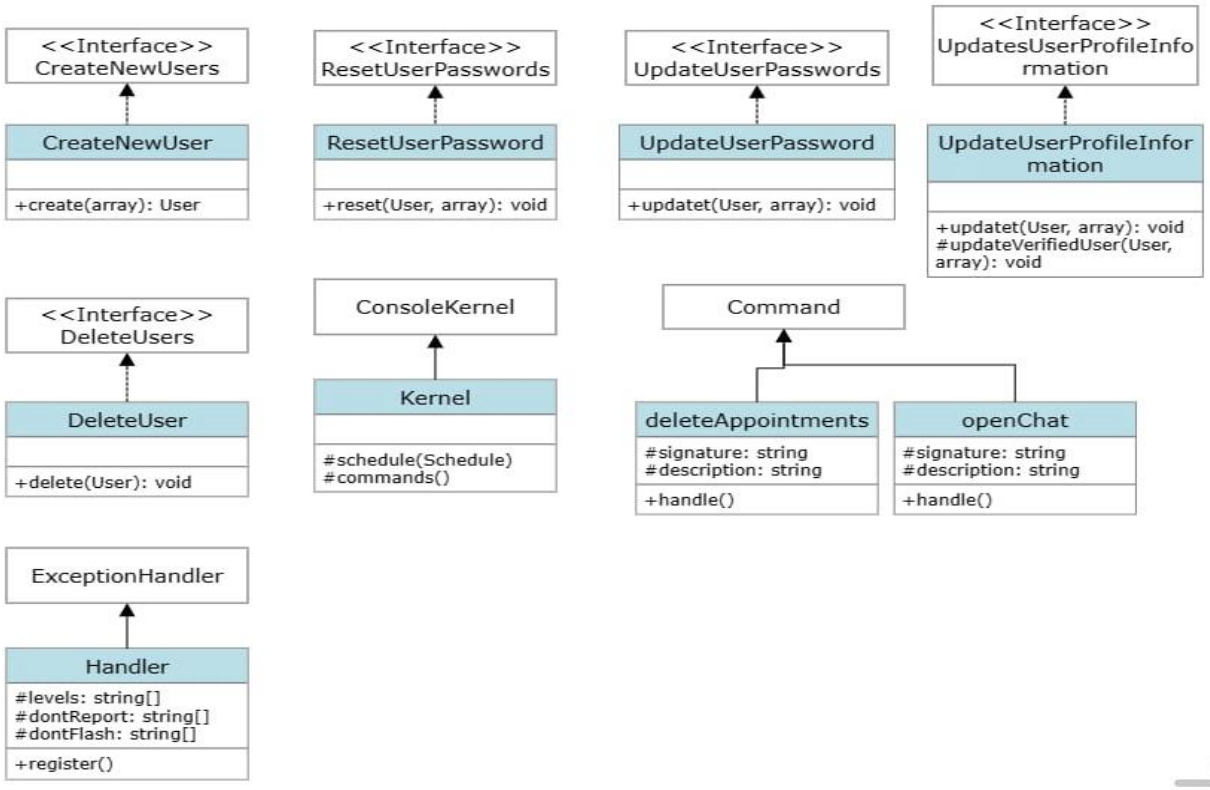
Appointment and user authenticator



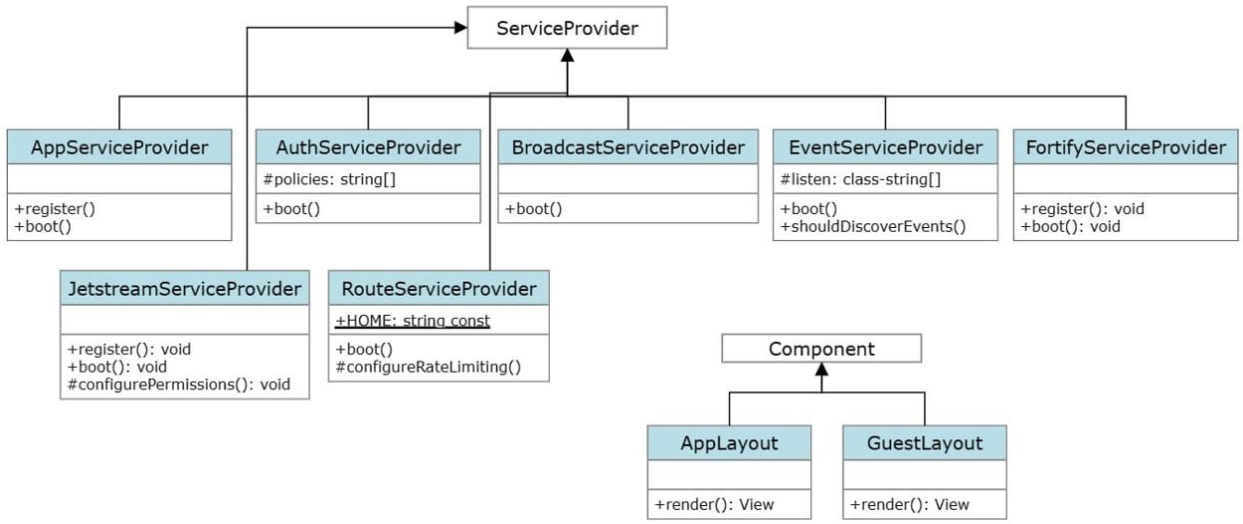
Model Class



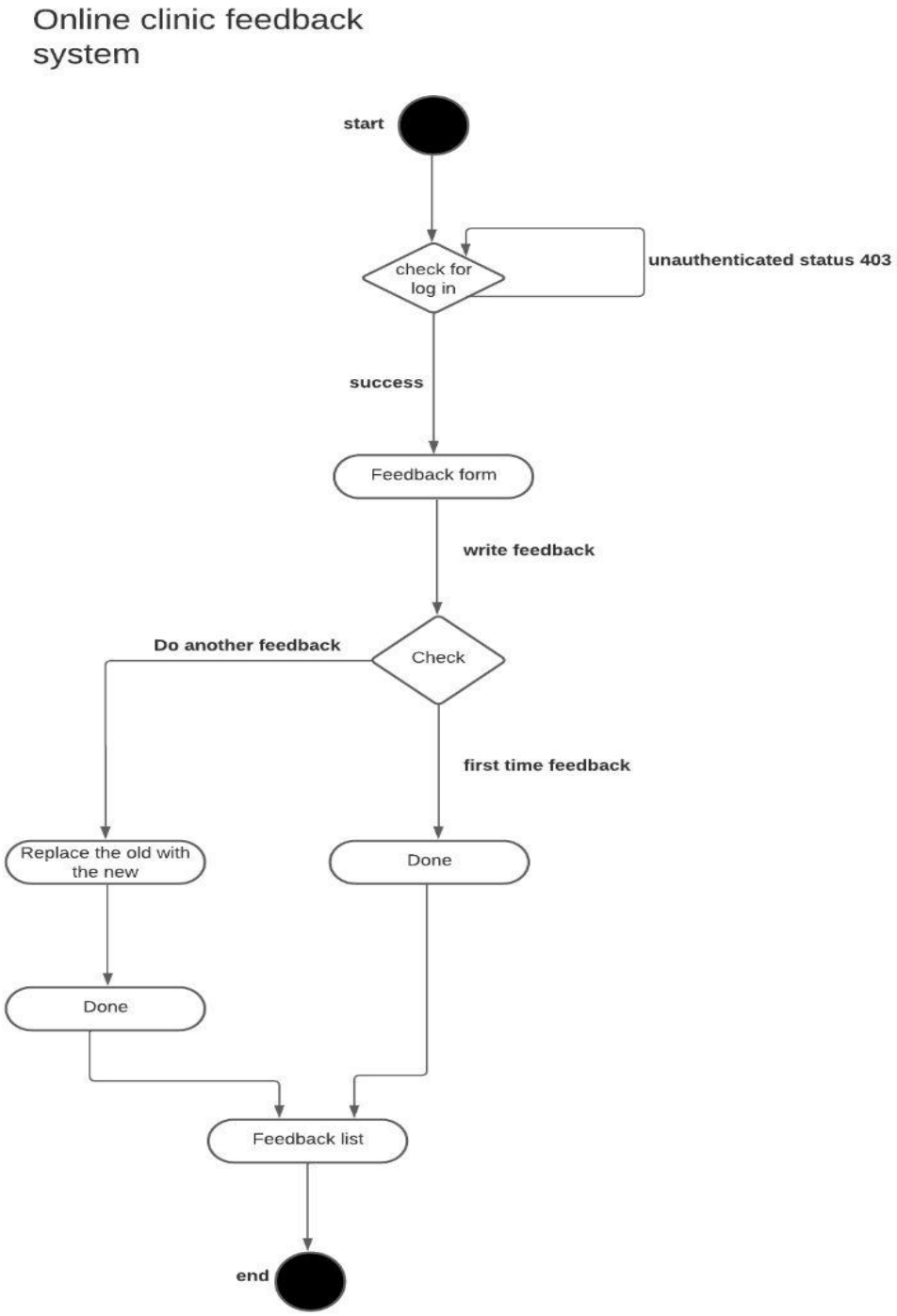
Commands



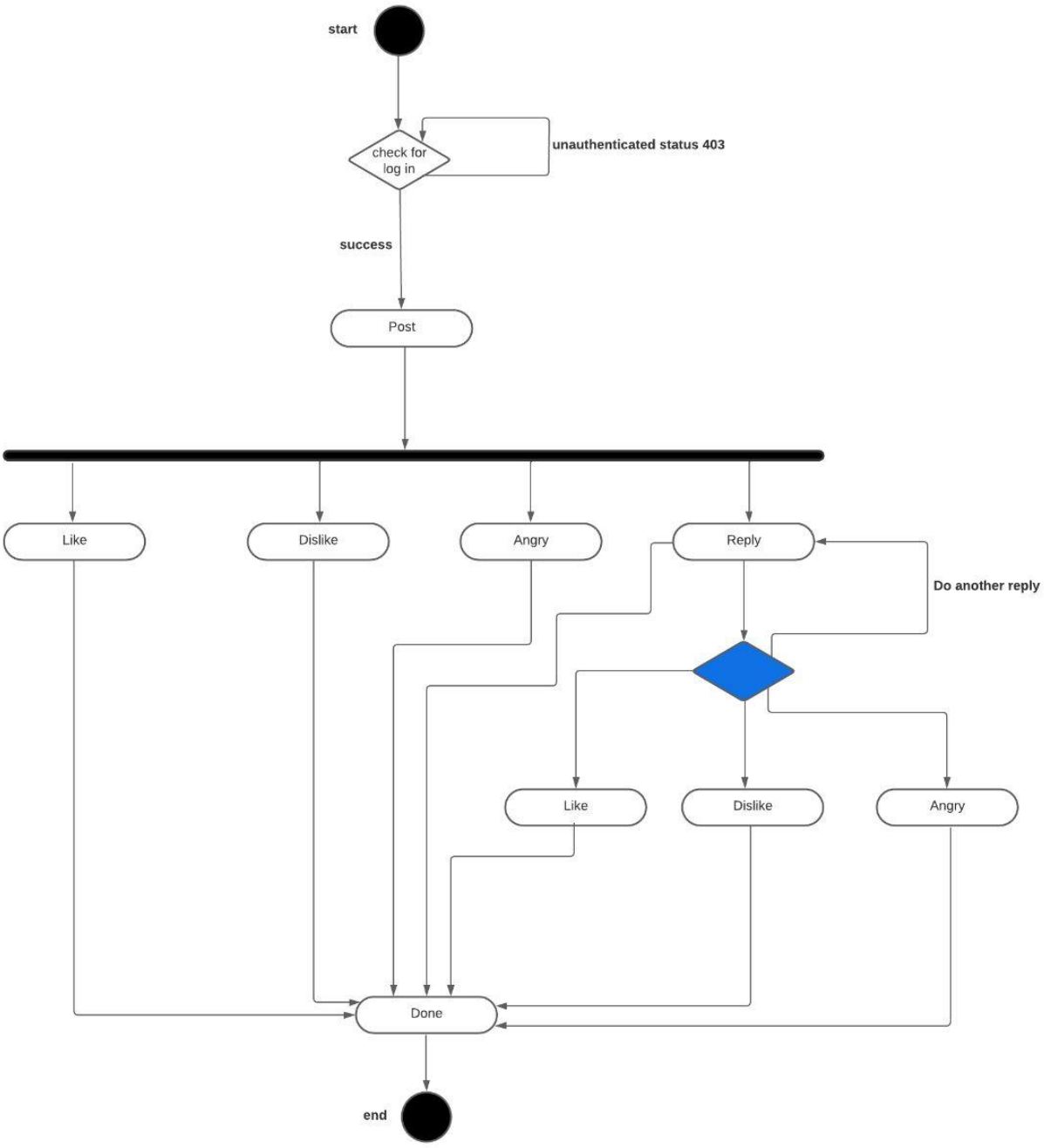
Service provider



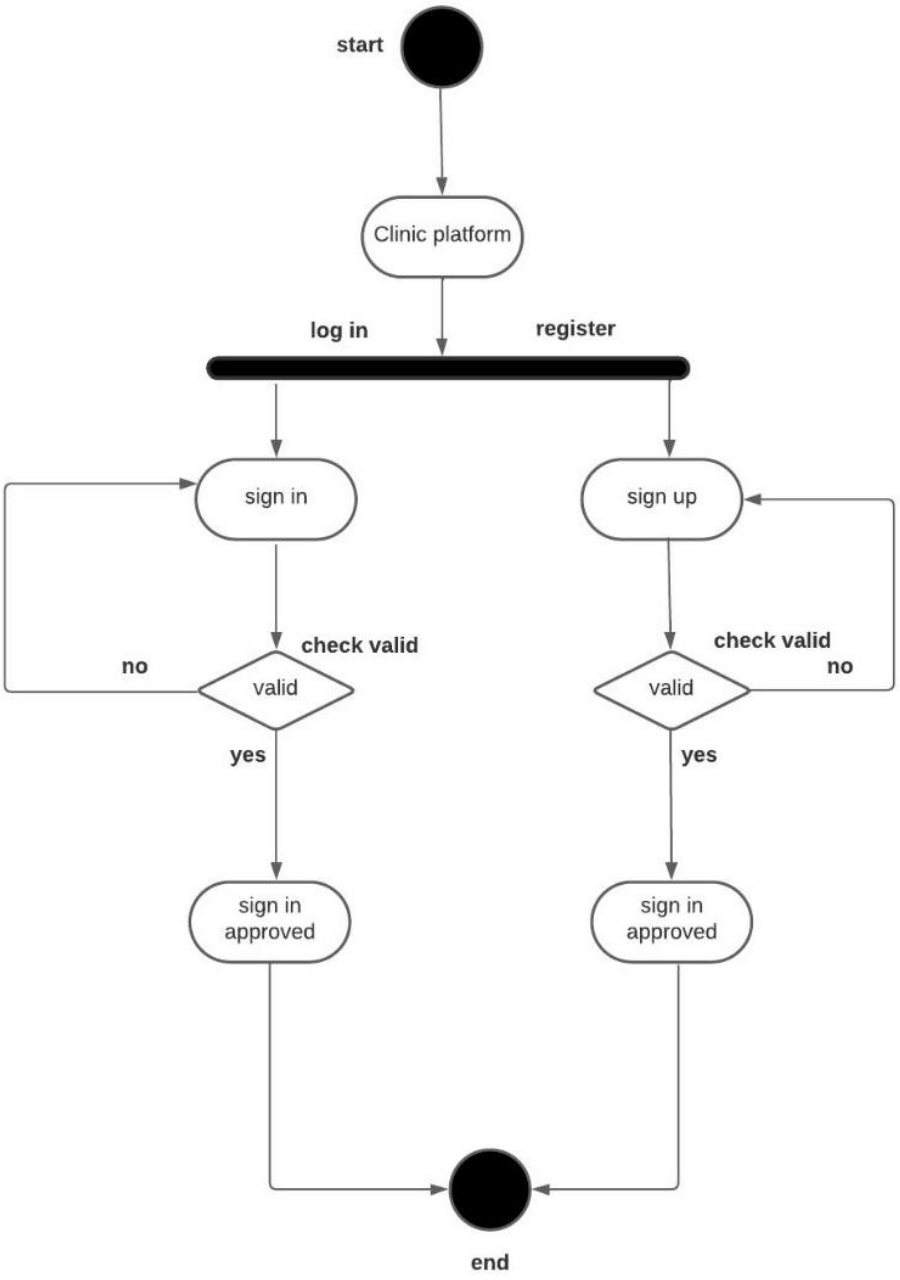
7.4.6) State Diagram



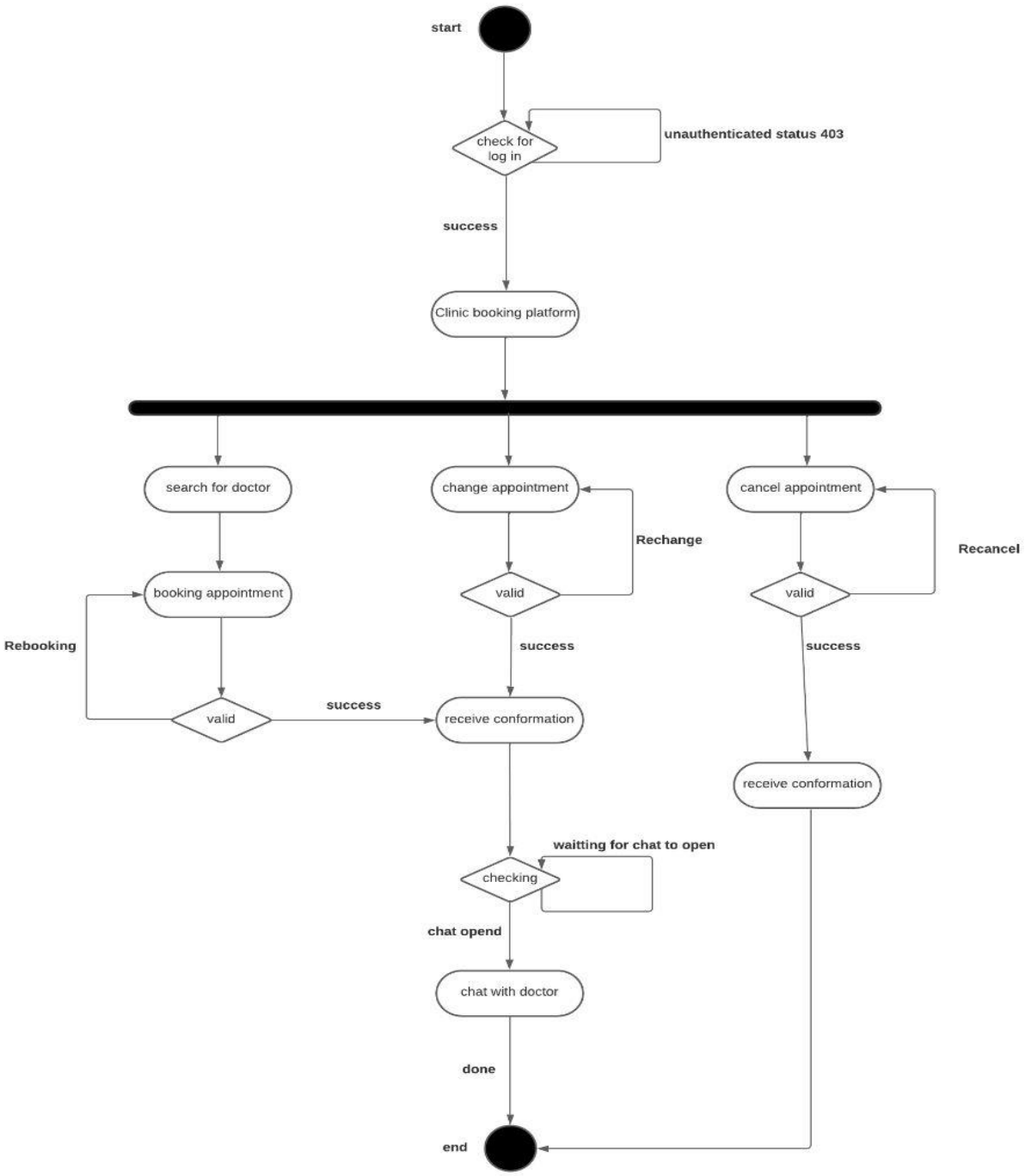
Online clinic posts system



clinic sign in and up

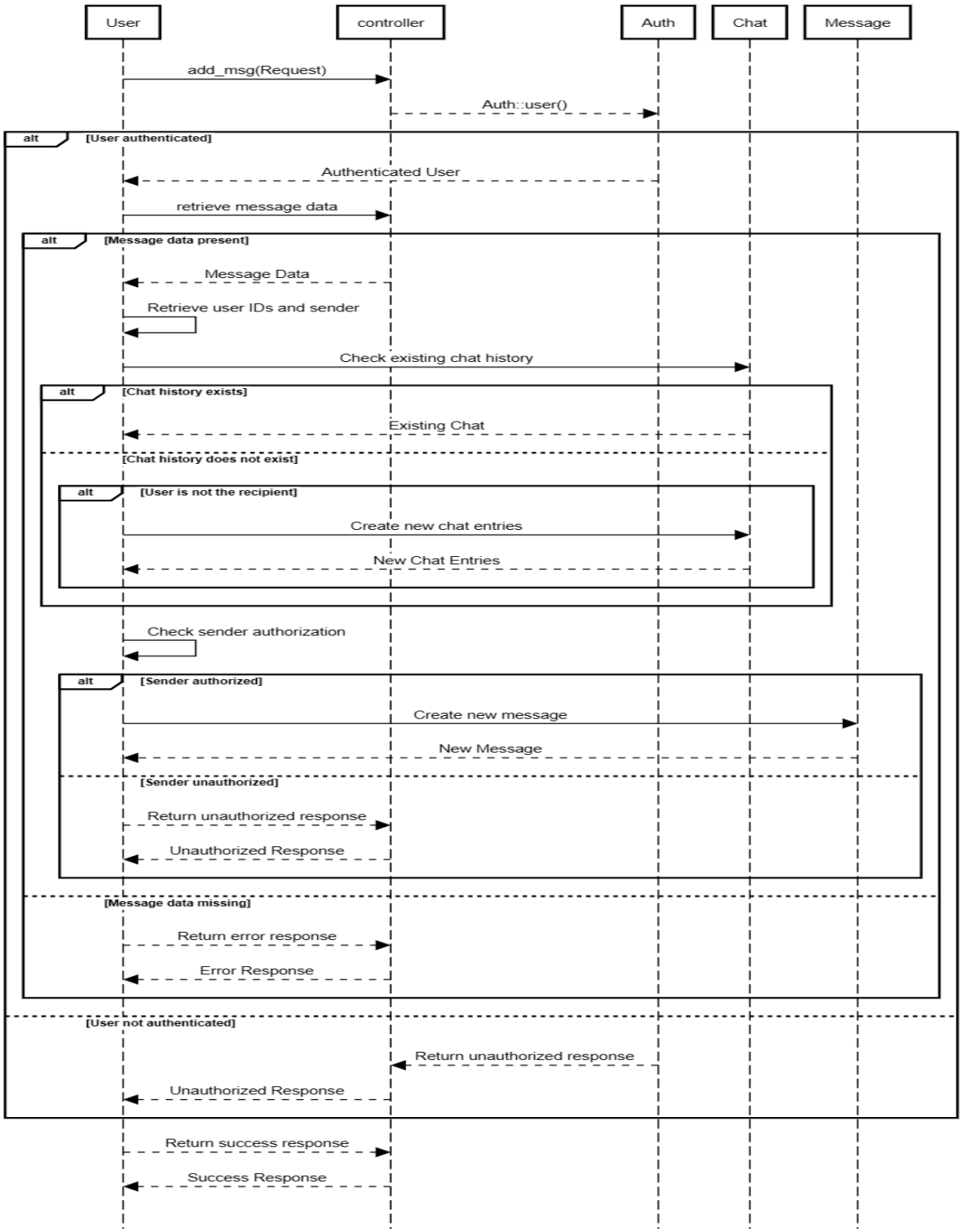


Online clinic booking and chatting system

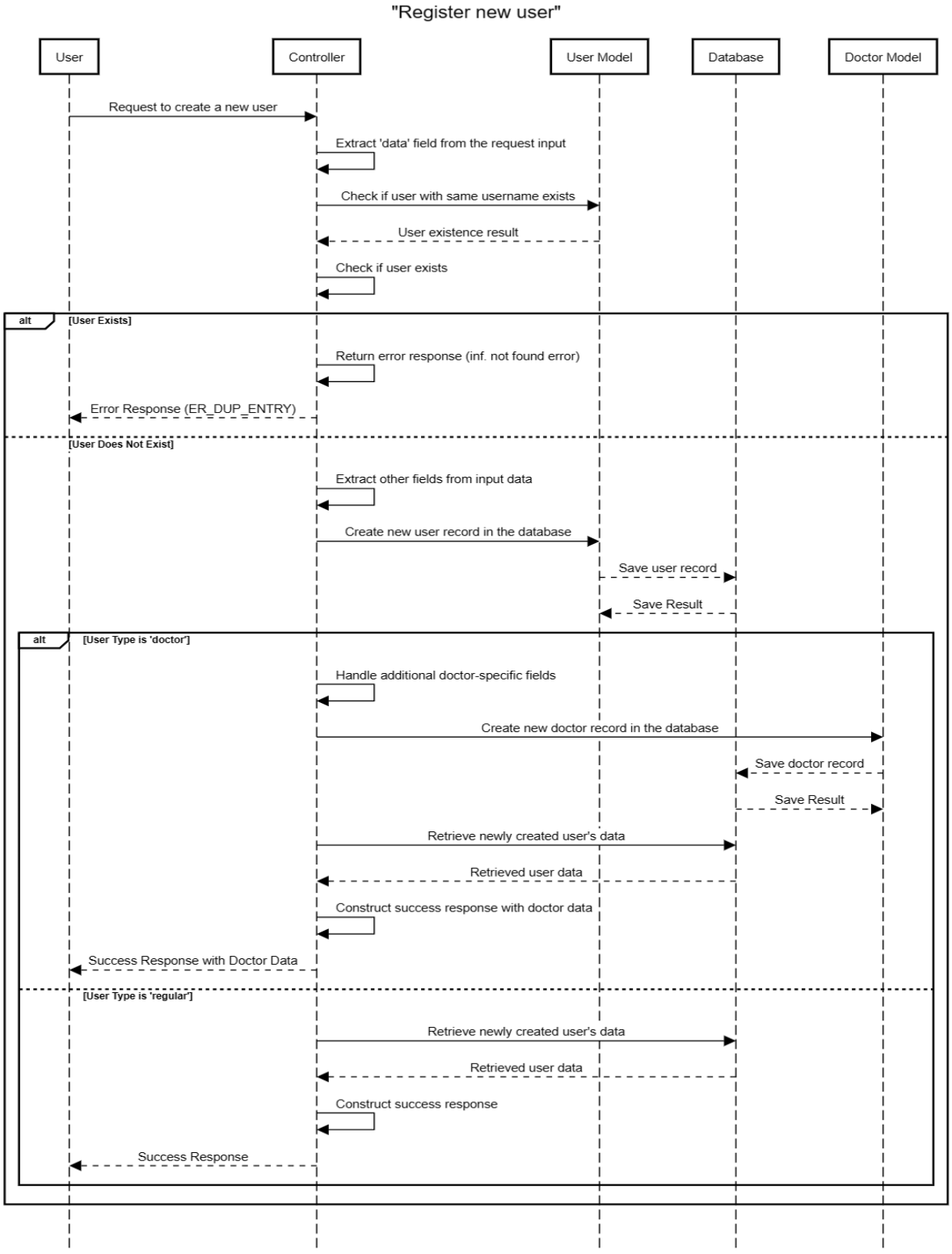


7.4.7) sequence diagram

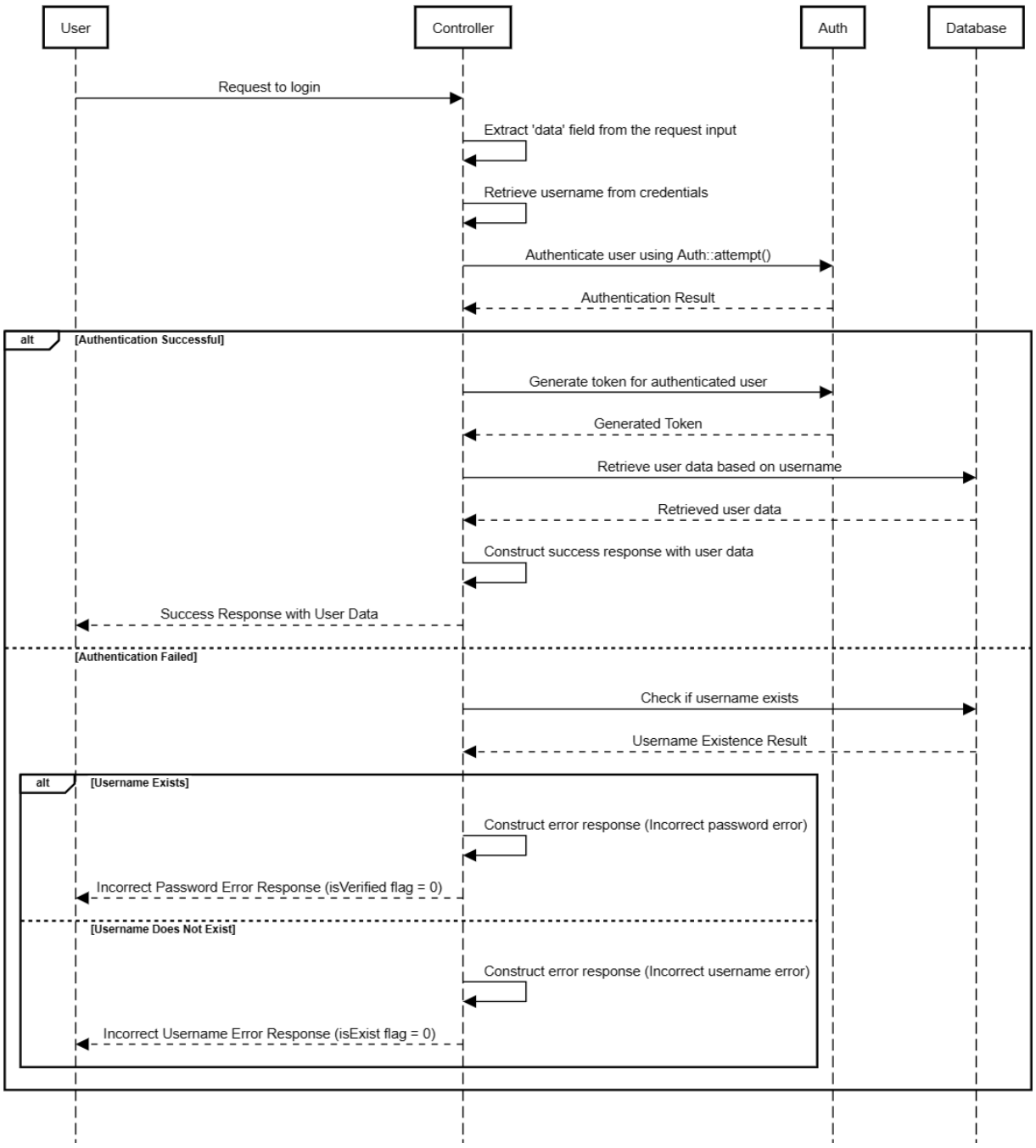
System



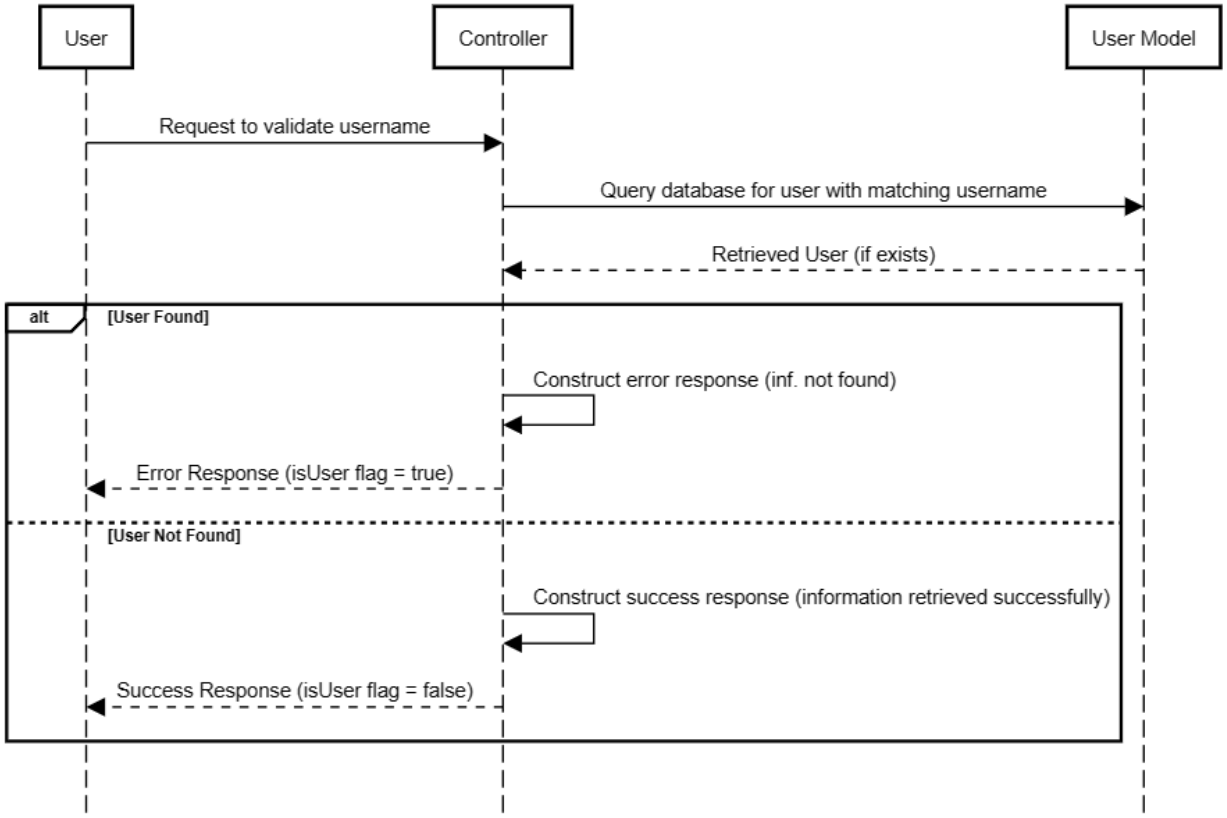
Register new user



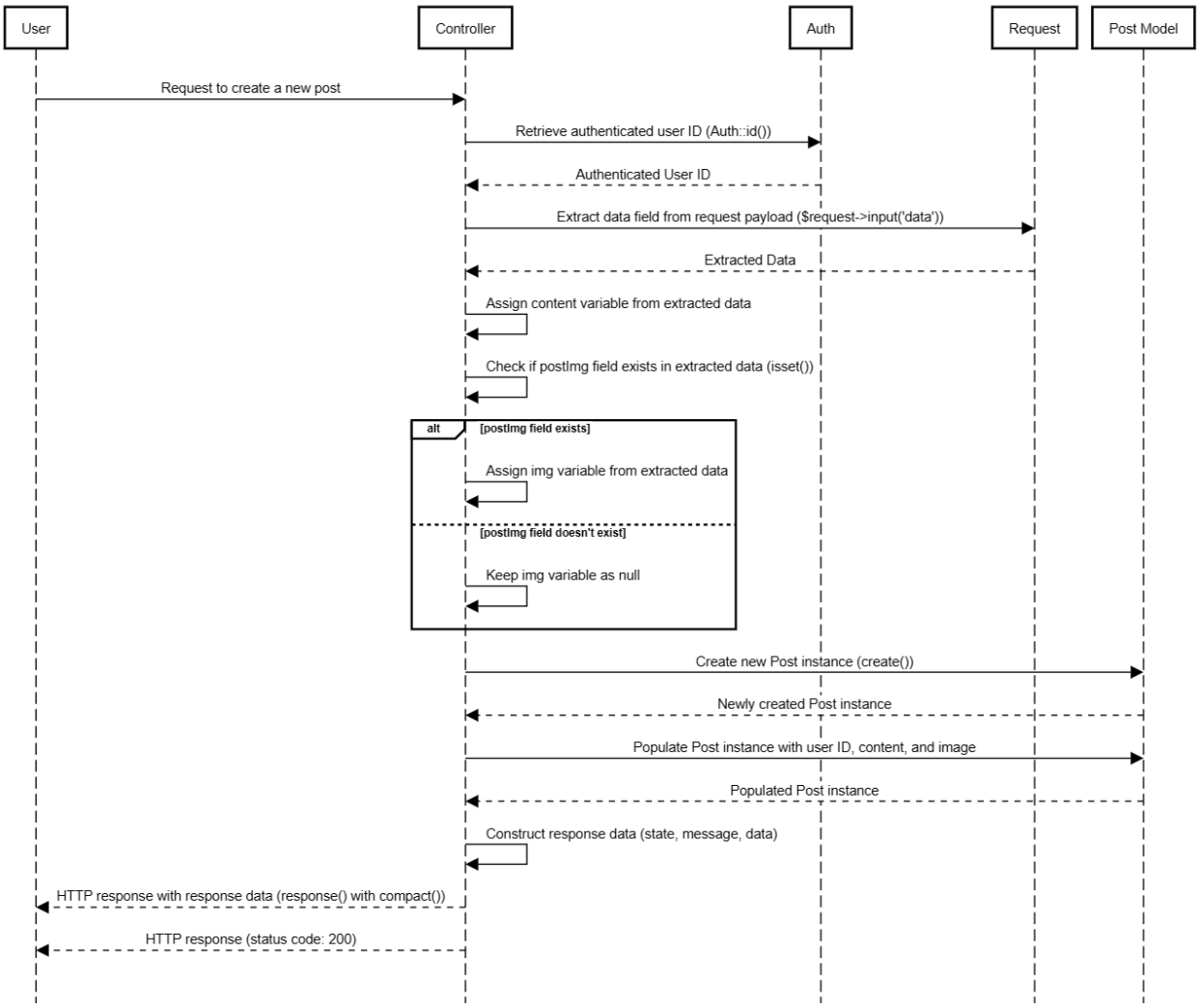
"User Login "


User
Controller
Auth
Database
Request to login
Extract 'data' field from the request input
Retrieve username from credentials
Authenticate user using Auth::attempt()
Authentication Result
alt
[Authentication Successful]
Generate token for authenticated user
Generated Token
Retrieve user data based on username
Retrieved user data
Construct success response with user data
Success Response with User Data
[Authentication Failed]
Check if username exists
Username Existence Result
alt
[Username Exists]
Construct error response (Incorrect password error)
Incorrect Password Error Response (isVerified flag = 0)
[Username Does Not Exist]
Construct error response (Incorrect username error)
Incorrect Username Error Response (isExist flag = 0)

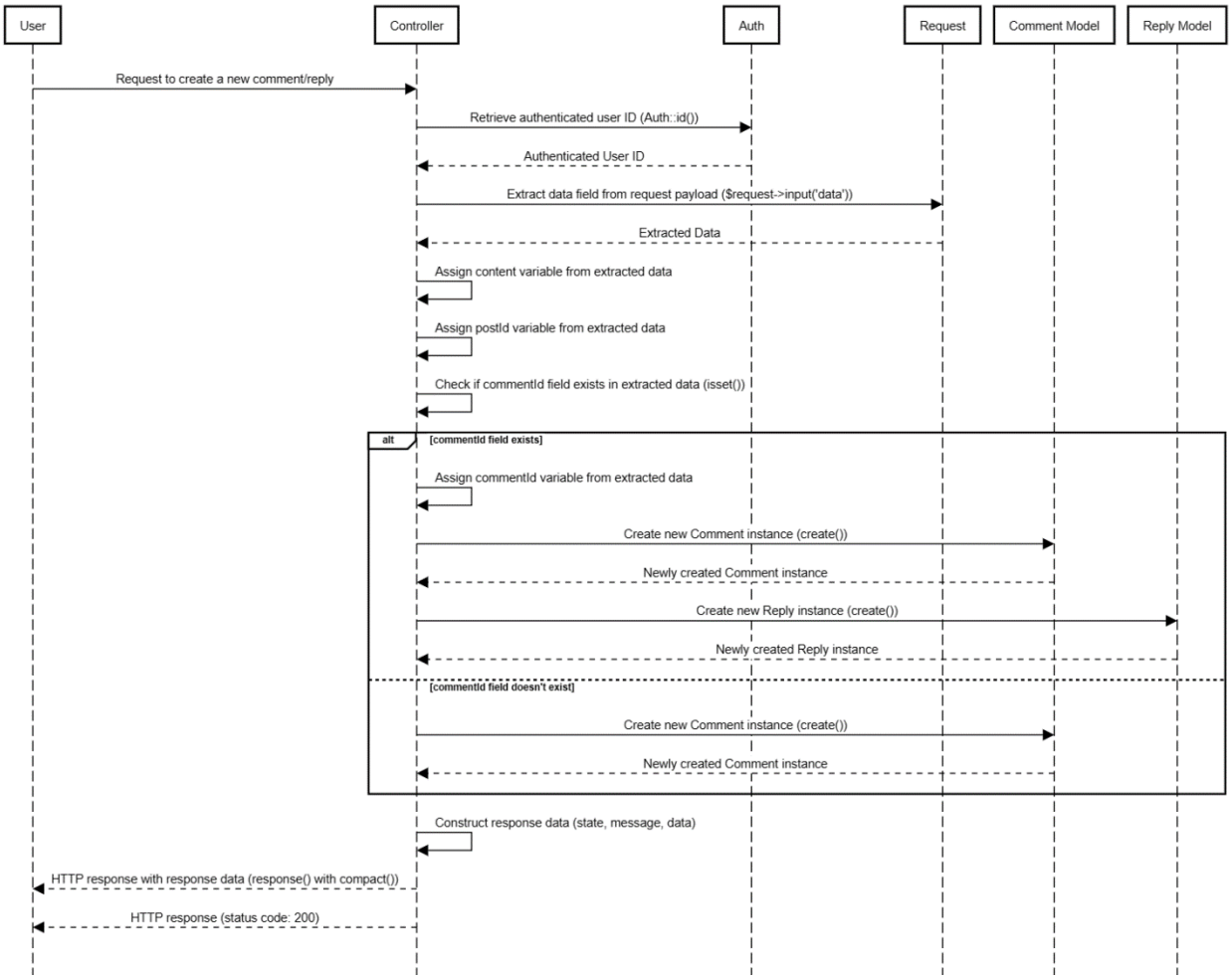
Username Validation



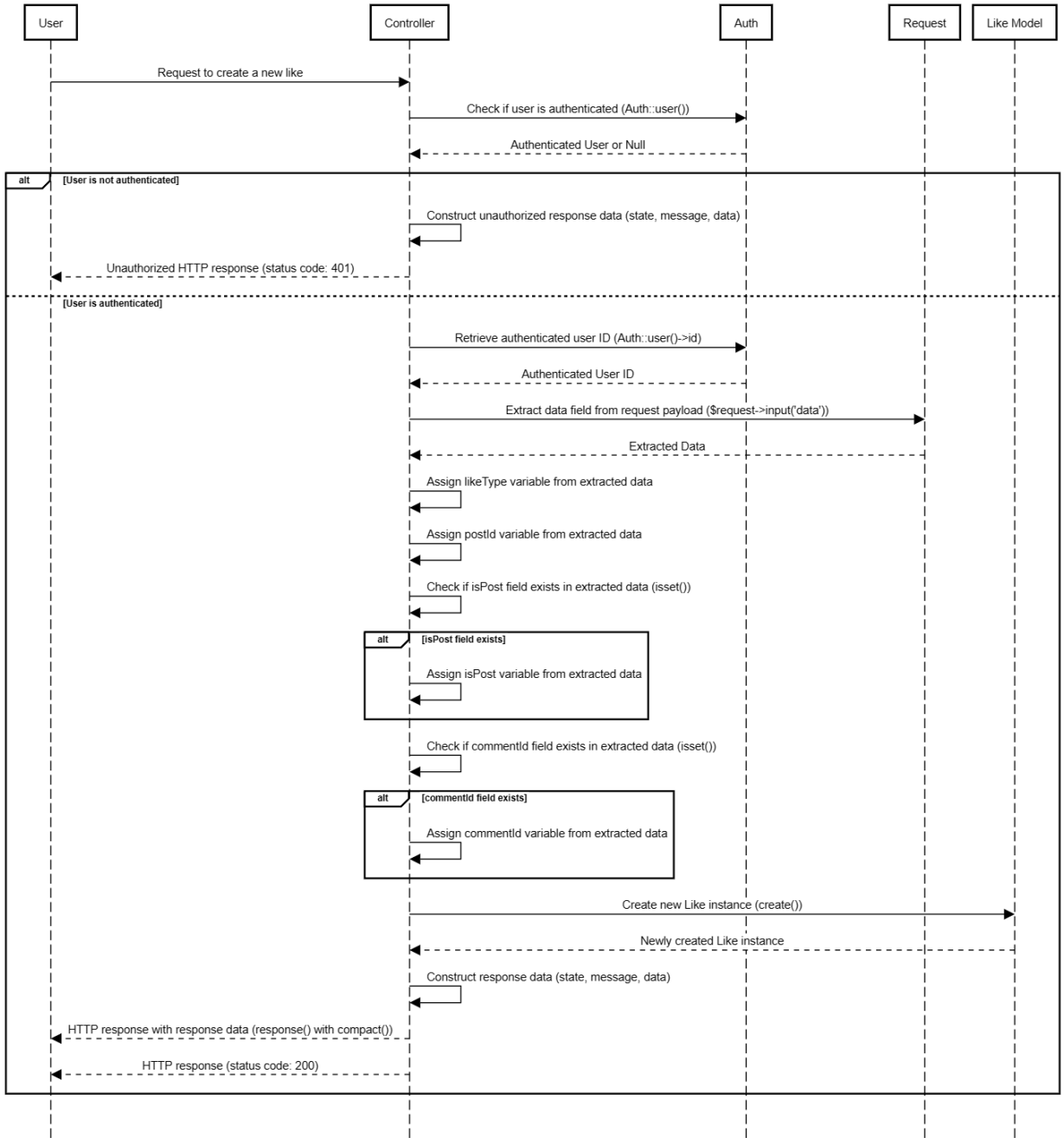
Submit Post


User
Controller
Auth
Request
Post Model
Request to create a new post
Retrieve authenticated user ID (Auth::id())
Authenticated User ID
Extract data field from request payload ($request->input('data'))
Extracted Data
Assign content variable from extracted data
Check if postImg field exists in extracted data (isset())
alt
[postImg field exists]
Assign img variable from extracted data
[postImg field doesn't exist]
Keep img variable as null
Create new Post instance (create())
Newly created Post instance
Populate Post instance with user ID, content, and image
Populated Post instance
Construct response data (state, message, data)
HTTP response with response data (response() with compact())
HTTP response (status code: 200)

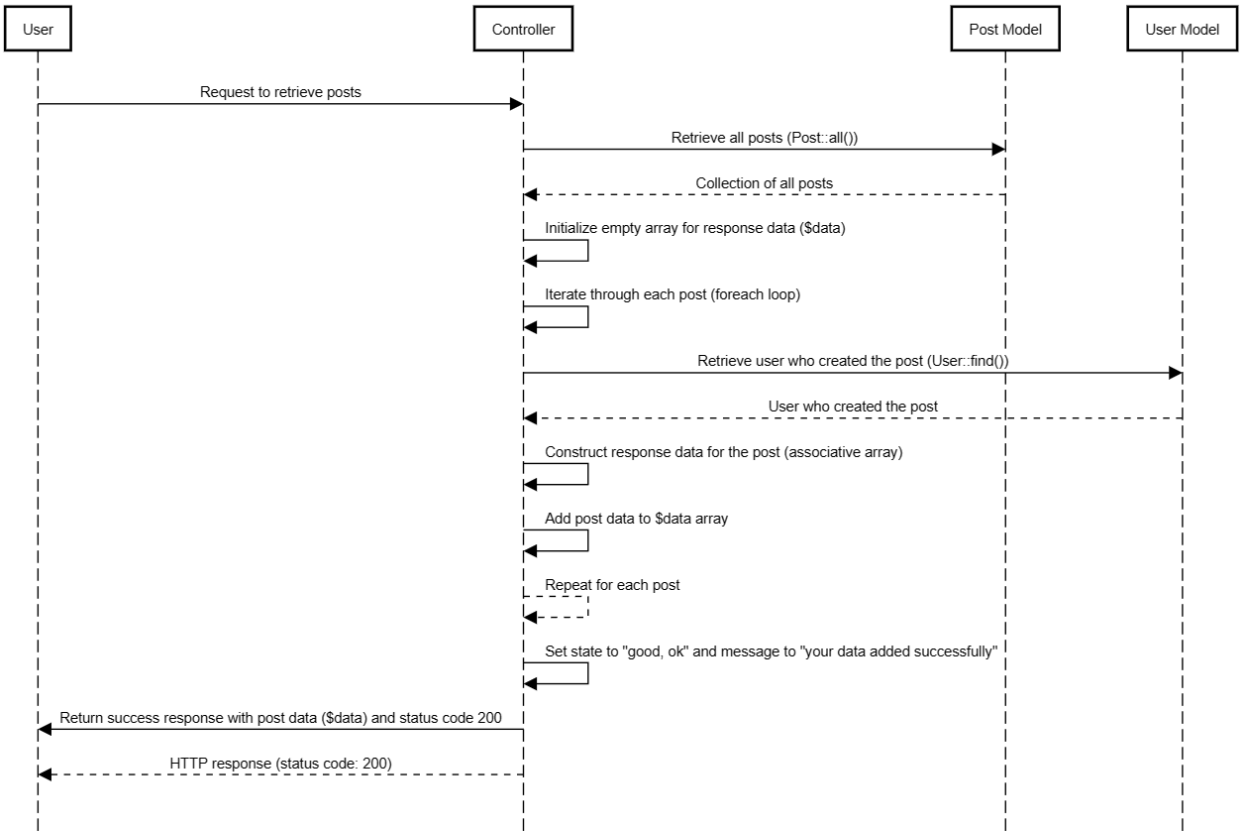
Submit comment and replay



Submit Like

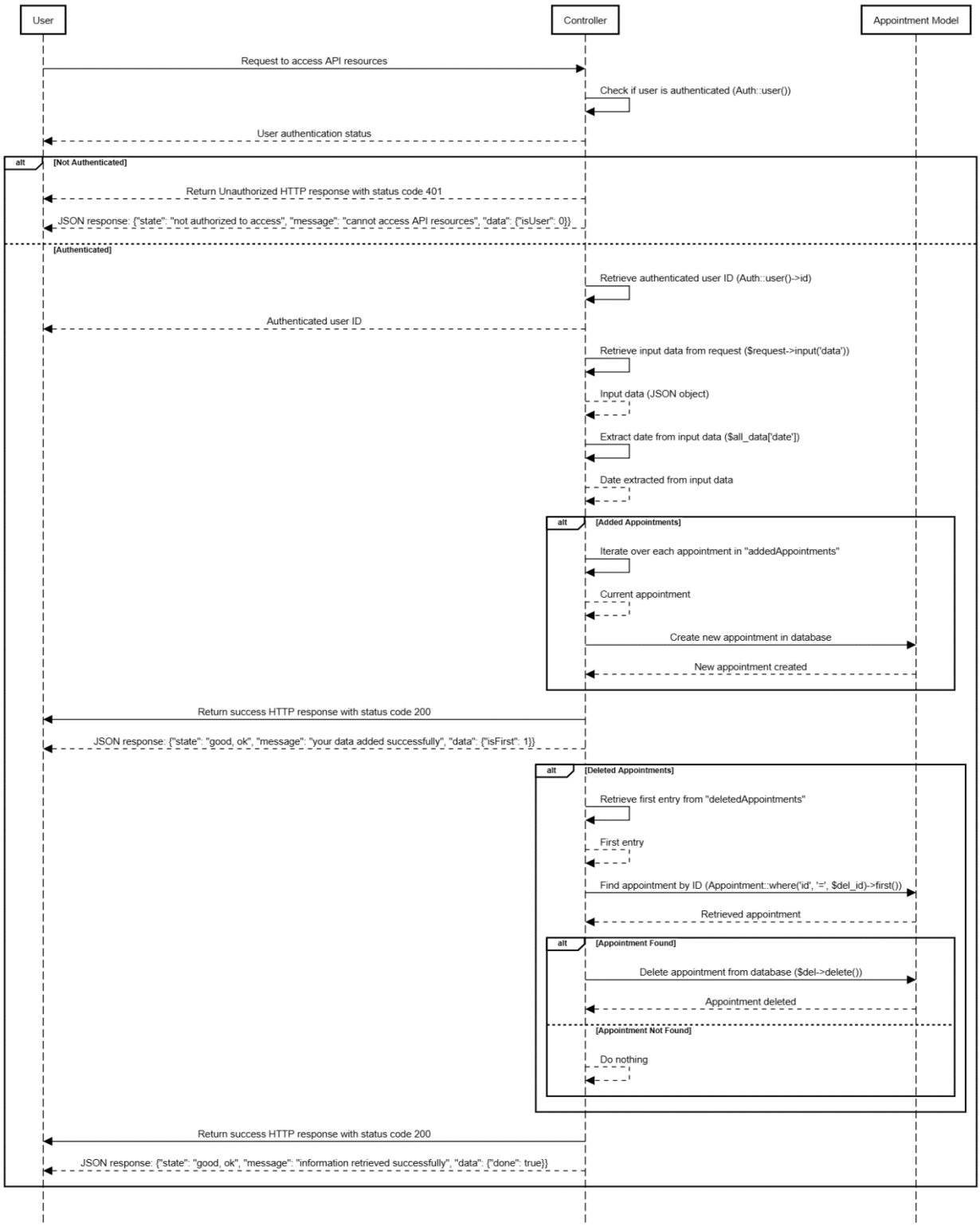

User
Controller
Auth
Request
Like Model
Request to create a new like
Check if user is authenticated (Auth::user())
Authenticated User or Null
alt
[User is not authenticated]
Construct unauthorized response data (state, message, data)
Unauthorized HTTP response (status code: 401)
[User is authenticated]
Retrieve authenticated user ID (Auth::user()->id)
Authenticated User ID
Extract data field from request payload ($request->input('data'))
Extracted Data
Assign likeType variable from extracted data
Assign postId variable from extracted data
Check if isPost field exists in extracted data (isset())
alt
[isPost field exists]
Assign isPost variable from extracted data
Check if commentId field exists in extracted data (isset())
alt
[commentId field exists]
Assign commentId variable from extracted data
Create new Like instance (create())
Newly created Like instance
Construct response data (state, message, data)
HTTP response with response data (response() with compact())
HTTP response (status code: 200)

Retrieve post



Retrieve comment and replay

Schedule appointment



7.5) Additional implementation details

```
CREATE TABLE `user` (
`user_id` int NOT NULL AUTO_INCREMENT,
`nick_name` varchar(22) NOT NULL,
`user_name` varchar(22) NOT NULL,
`password` text,
`user_type` char(7) DEFAULT NULL,
`bdate` date DEFAULT NULL,
`gender` char(6) NOT NULL,
`pnumber` varchar(16) DEFAULT NULL,
`prefix` varchar(5) DEFAULT NULL,
`email` varchar(30) DEFAULT NULL,
`city` varchar(22) DEFAULT NULL,
`street` longtext,
`img_url` longtext,
PRIMARY KEY (`user_id`),
UNIQUE KEY `user_name` (`user_name`)
);
```

```
CREATE TABLE `doctor` (
`doctor_id` int NOT NULL,
`specialty` varchar(30) DEFAULT NULL,
`rate` double NOT NULL DEFAULT '0',
`current_hospital` varchar(30) DEFAULT NULL,
`graduation_year` int DEFAULT NULL,
`experience_years` int DEFAULT NULL,
`experiences` varchar(200) DEFAULT NULL,
`about` varchar(200) DEFAULT NULL,
`salary` decimal(10,2) DEFAULT NULL,
`certificate_count` int DEFAULT NULL,
`num_rate` int NOT NULL DEFAULT '0',
`fees` decimal(8,3) DEFAULT NULL,
`is_verified` tinyint DEFAULT NULL,
PRIMARY KEY (`doctor_id`),
CONSTRAINT `doctor_ibfk_1` FOREIGN KEY (`doctor_id`)
REFERENCES `user` (`user_id`)
ON DELETE CASCADE ON UPDATE CASCADE,
CONSTRAINT `doctor_chk_1` CHECK ((`rate` between 0 and 5))
);
```

```
CREATE TABLE `appointment` (
`appointment_id` int NOT NULL AUTO_INCREMENT,
`schedule_from` int NOT NULL,
`booked_from` int DEFAULT NULL,
`slot_time` varchar(30) DEFAULT NULL,
`schedule_date` date NOT NULL,
`appointment_duration` int DEFAULT NULL,
`appointment_type` varchar(15) DEFAULT NULL,
`appointment_state` varchar(30) DEFAULT 'free',
`appointment_fees` float DEFAULT NULL,
`booked_time` timestamp NULL DEFAULT NULL,
`rate` double DEFAULT NULL,
`feedback` varchar(200) DEFAULT NULL,
`expired_at` timestamp NULL DEFAULT CURRENT_TIMESTAMP,
PRIMARY KEY (`appointment_id`),
KEY `appointment_ibfk_1` (`schedule_from`),
KEY `appointment_ibfk_2` (`booked_from`),
CONSTRAINT `appointment_ibfk_1`
FOREIGN KEY (`schedule_from`)
REFERENCES `doctor` (`doctor_id`)
ON DELETE CASCADE ON UPDATE CASCADE,
CONSTRAINT `appointment_ibfk_2`
FOREIGN KEY (`booked_from`)
REFERENCES `user` (`user_id`) ON
DELETE CASCADE ON UPDATE CASCADE
```

```
CREATE TABLE `patient_records` (
`patient_id` int NOT NULL,
`test_results` longtext,
`current_issue` varchar(250) DEFAULT NULL,
`allergies` varchar(250) DEFAULT NULL,
`immunizations` varchar(250) DEFAULT NULL,
`surgeries` varchar(250) DEFAULT NULL,
`illnesses_history` varchar(200) DEFAULT NULL,
PRIMARY KEY (`patient_id`),
KEY `patientrecords_ibfk_1_idx` (`patient_id`),
CONSTRAINT `patient_records_ibfk_1`
FOREIGN KEY (`patient_id`)
REFERENCES `user` (`user_id`)
ON DELETE CASCADE ON UPDATE CASCADE
);
```

```
CREATE TABLE `clinic_details` (
`doctor_id` int NOT NULL,
`clinic_name` varchar(22) DEFAULT NULL,
`city` varchar(22) DEFAULT NULL,
`street` longtext,
`pnumber` varchar(16) DEFAULT NULL,
`tnumber` varchar(16) DEFAULT NULL,
`prefix` varchar(5) DEFAULT NULL,
PRIMARY KEY (`doctor_id`),
CONSTRAINT `clinic_details_ibfk_1`
FOREIGN KEY (`doctor_id`)
REFERENCES `doctor` (`doctor_id`)
ON DELETE CASCADE ON UPDATE CASCADE
);
```

```
CREATE TABLE `feedback` (
`feedback_from` int NOT NULL,
`feedback_to` int NOT NULL,
`rate` double DEFAULT NULL,
`feedback` text,
`issued_time` timestamp NULL DEFAULT
CURRENT_TIMESTAMP,
PRIMARY KEY (`feedback_from`,`feedback_to`),
KEY `feedback_to` (`feedback_to`),
CONSTRAINT `feedback_ibfk_1`
FOREIGN KEY (`feedback_from`)
REFERENCES `user` (`user_id`)
ON DELETE CASCADE ON UPDATE CASCADE,
CONSTRAINT `feedback_ibfk_2` FOREIGN KEY
(`feedback_to`) REFERENCES `doctor` (`doctor_id`)
ON DELETE CASCADE ON UPDATE CASCADE
);
```

```
CREATE TABLE `chat` (
`chat_from` int NOT NULL,
`chat_to` int NOT NULL,
`is_open` tinyint DEFAULT NULL,
PRIMARY KEY (`chat_from`,`chat_to`),
KEY `chat_to` (`chat_to`),
CONSTRAINT `chat_ibfk_1`
FOREIGN KEY (`chat_from`)
REFERENCES `user` (`user_id`)
ON DELETE CASCADE ON UPDATE CASCADE,
CONSTRAINT `chat_ibfk_2`
FOREIGN KEY (`chat_to`)
REFERENCES `user` (`user_id`)
ON DELETE CASCADE ON UPDATE CASCADE
```

```
CREATE TABLE `message` (
`message_id` int NOT NULL AUTO_INCREMENT,
`message_from` int NOT NULL,
`message_to` int NOT NULL,
`content` text CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci,
`issued_date` date DEFAULT (curdate()),
`issued_time` time DEFAULT (curtime()),
PRIMARY KEY (`message_id`),
KEY `message_from` (`message_from`),
KEY `message_to` (`message_to`),
CONSTRAINT `message_ibfk_1`
FOREIGN KEY (`message_from`)
REFERENCES `chat` (`chat_from`)
ON DELETE CASCADE ON UPDATE CASCADE,
CONSTRAINT `message_ibfk_2` FOREIGN KEY (`message_to`)
REFERENCES `chat` (`chat_to`)
ON DELETE CASCADE ON UPDATE CASCADE
);
```

```
CREATE TABLE `video_call` (
`appointment_id` int NOT NULL,
`session_id` text,
`created_at` timestamp NULL DEFAULT CURRENT_TIMESTAMP,
PRIMARY KEY (`appointment_id`),
CONSTRAINT `video_call_ibfk_1`
FOREIGN KEY (`appointment_id`)
REFERENCES `appointment` (`appointment_id`)
ON DELETE CASCADE ON UPDATE CASCADE
);
```

```
CREATE TABLE `report` (
`report_from` int NOT NULL,
`num_reports` int DEFAULT '0',
`latest_report_time` timestamp NULL DEFAULT CURRENT_TIMESTAMP,
PRIMARY KEY (`report_from`),
CONSTRAINT `report_ibfk_1`
FOREIGN KEY (`report_from`) REFERENCES `user` (`user_id`)
);
```

```
CREATE TABLE `post` (
`post_id` int NOT NULL AUTO_INCREMENT,
`user_id` int NOT NULL,
`content` text CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci,
`issued_time` timestamp NULL DEFAULT CURRENT_TIMESTAMP,
`num_comments` int DEFAULT '0',
`like_emoji` int DEFAULT '0',
`dislike` int DEFAULT '0',
`angry` int DEFAULT '0',
`post_img` longtext,
PRIMARY KEY (`post_id`),
KEY `user_id` (`user_id`),
CONSTRAINT `post_ibfk_1`
FOREIGN KEY (`user_id`)
REFERENCES `user` (`user_id`)
ON DELETE CASCADE
ON UPDATE CASCADE
);
```

```
CREATE TABLE `comment` (
`comment_id` int NOT NULL AUTO_INCREMENT,
`post_id` int NOT NULL,
`user_id` int NOT NULL,
`content` text,
`issued_time` timestamp NULL DEFAULT CURRENT_TIMESTAMP,
`num_replies` int DEFAULT '0',
`like_emoji` int DEFAULT '0',
`dislike` int DEFAULT '0',
`angry` int DEFAULT '0',
PRIMARY KEY (`comment_id`),
KEY `post_id` (`post_id`),
KEY `user_id` (`user_id`),
CONSTRAINT `comment_ibfk_1`
FOREIGN KEY (`post_id`)
REFERENCES `post` (`post_id`)
ON DELETE CASCADE ON UPDATE CASCADE,
CONSTRAINT `comment_ibfk_2`
FOREIGN KEY (`user_id`)
REFERENCES `user` (`user_id`)
ON DELETE CASCADE
ON UPDATE CASCADE
);
```

```
CREATE TABLE `reply` (
`reply_id` int NOT NULL,
`post_id` int NOT NULL,
`reply_on` int NOT NULL,
PRIMARY KEY (`reply_id`),
KEY `post_id` (`post_id`),
KEY `reply_on` (`reply_on`),
CONSTRAINT `reply_ibfk_1`
FOREIGN KEY (`post_id`)
REFERENCES `comment` (`post_id`)
ON DELETE CASCADE ON UPDATE CASCADE,
CONSTRAINT `reply_ibfk_2`
FOREIGN KEY (`reply_id`)
REFERENCES `comment` (`comment_id`)
ON DELETE CASCADE ON UPDATE CASCADE,
CONSTRAINT `reply_ibfk_3`
FOREIGN KEY (`reply_on`)
REFERENCES `comment` (`comment_id`)
ON DELETE CASCADE ON UPDATE CASCADE
)
```

```
CREATE TABLE `likes` (
`like_id` int NOT NULL AUTO_INCREMENT,
`post_id` int DEFAULT NULL,
`comment_id` int DEFAULT NULL,
`user_id` int DEFAULT NULL,
`is_post` tinyint(1) DEFAULT NULL,
`like_type` varchar(10) DEFAULT NULL,
PRIMARY KEY (`like_id`),
UNIQUE KEY `post_id` (`post_id`,`comment_id`,`user_id`),
KEY `comment_id` (`comment_id`),
KEY `user_id` (`user_id`),
CONSTRAINT `likes_ibfk_1` FOREIGN KEY (`post_id`)
REFERENCES `post` (`post_id`) ON DELETE
CASCADE ON UPDATE CASCADE,
CONSTRAINT `likes_ibfk_2`
FOREIGN KEY (`comment_id`)
REFERENCES `comment` (`comment_id`) ON
DELETE CASCADE ON UPDATE CASCADE,
CONSTRAINT `likes_ibfk_3` FOREIGN KEY (`user_id`)
REFERENCES `user` (`user_id`)
ON DELETE CASCADE
ON UPDATE CASCADE
);
```

```
CREATE TABLE `report_details` (
`report_id` int NOT NULL AUTO_INCREMENT,
`user_id` int NOT NULL,
`issue` text,
`report_type` varchar(30) DEFAULT NULL,
`issued_time` timestamp NULL DEFAULT
CURRENT_TIMESTAMP,
PRIMARY KEY (`report_id`),
CONSTRAINT `report_details_ibfk_1`
FOREIGN KEY (`user_id`)
REFERENCES `report` (`report_from`)
);
```

8) Backend Management

Laravel

8.1) Introduction

Laravel is a popular open-source PHP web framework that follows the model-view-controller (MVC) architectural pattern. It provides developers with a robust and elegant toolkit for building web applications efficiently and securely. This report explores how Laravel works and highlights its key advantages.

8.2) Key Advantages of Laravel

1- **Rapid Application Development (RAD)**: Laravel simplifies and accelerates the development process through its expressive syntax, extensive libraries, and built-in functionalities. Developers can leverage these features to build applications quickly and efficiently.

2- **MVC Architecture**: Laravel follows the MVC architectural pattern, which promotes a clear separation between business logic (Model), user interface (View), and application flow (Controller). This separation enhances code organization, maintainability, and scalability.

3- **Rich Ecosystem**: Laravel benefits from a vibrant ecosystem with a vast community and numerous third-party packages. These packages extend Laravel's functionality, providing solutions for various tasks like authentication, payment gateways, API integrations, and more.

4- **Robust Security**: Laravel incorporates security features such as CSRF protection, input sanitization, encryption, and password hashing out-of-the-box. Additionally, Laravel's ORM mitigates common security risks associated with raw SQL queries by offering parameter binding and query builders.

5- **Testing Support**: Laravel provides robust testing capabilities, including unit testing and integration testing. The framework offers convenient tools for writing and executing tests, enabling developers to ensure code reliability and maintainability.

6- **Community Support and Documentation**: Laravel enjoys a vast and active community, offering support, tutorials, and resources. The official Laravel documentation is comprehensive and well-maintained, making it easy for developers to get started and find solutions to common problems.

7- **Task Scheduling**: Task scheduling is a crucial feature provided by Laravel, allowing developers to automate the execution of recurring tasks within their web applications. This feature is particularly useful for performing routine maintenance, generating reports, and handling background processes. Laravel's task scheduling is built on top of the underlying Cron system, providing a simple and expressive syntax for defining scheduled tasks.

8.3) Laravel's Core Components

1- Routing: Laravel offers a concise and expressive syntax for defining web routes, making it easy to handle HTTP requests and define endpoints.

2- Middleware: Middleware enables developers to filter and modify HTTP requests and responses. Laravel includes a rich set of middleware for tasks such as authentication, CSRF protection, and caching.

3- Views: Laravel's templating engine, called Blade, allows developers to create dynamic and reusable views easily. Blade provides features like template inheritance, control structures, and reusable components.

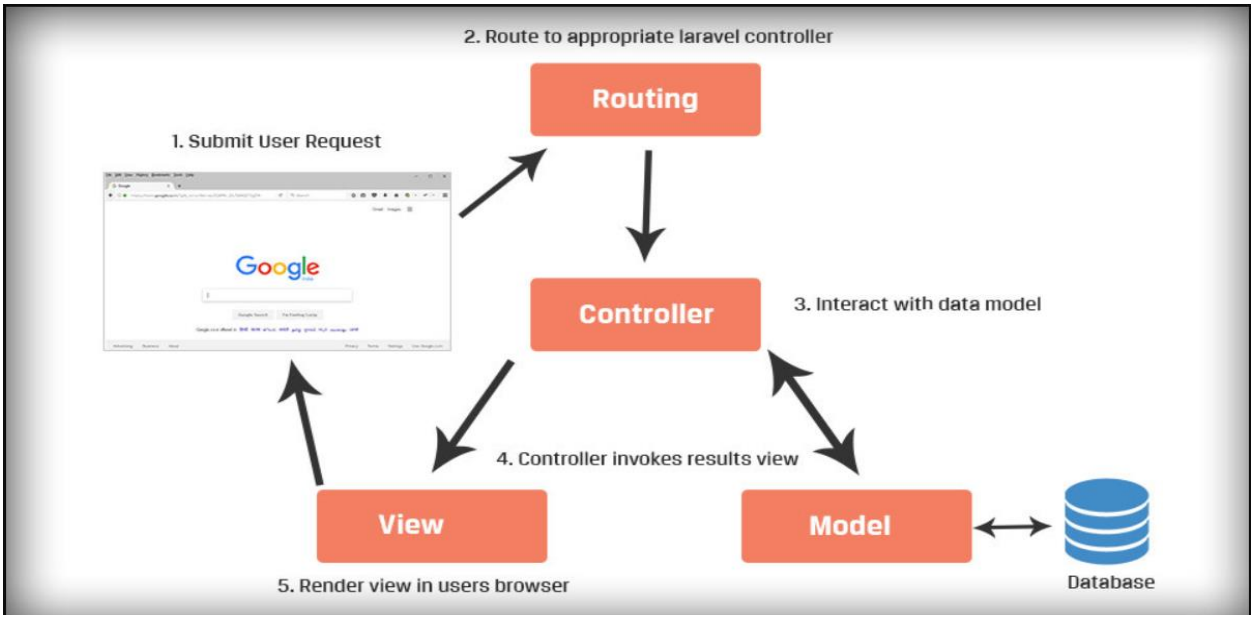
4- Database Abstraction: Laravel's ORM (Object-Relational Mapping) called Eloquent simplifies database operations by providing an intuitive and expressive syntax for interacting with databases. It supports various database systems, including MySQL, PostgreSQL, SQLite, and SQL Server.

5- Eloquent Relationships: Laravel's Eloquent ORM supports defining relationships between database tables, such as one-to-one, one-to-many, and many-to-many relationships, making it convenient to work with related data.

6- Migrations: Laravel offers a database migration system that allows developers to version and manage database changes effortlessly. Migrations provide a structured approach for creating and modifying database tables and columns.

8.4) About the Backend Server

- ✓ This project is implemented using Laravel framework which uses **MVC design pattern.**



- ✓ JWT token is used for Authentication.
- ✓ Eloquent model is used to Insert, delete and Update database.
- ✓ Using Gmail SMTP for notification.
- ✓ Stripe is used for payment.
- ✓ Laravel Migration is used to insert and edit our tables.
- ✓ Laravel Task Scheduling is used to check some models in database periodically.
- ✓ React Server represents the View.
- ✓ PHP Unit to apply testing.

8.5) Authentication part

8.5.1) What is JWT (JSON Web Tokens)?

A JWT is a token that is generated by the authentication server and contains the end-user's information (like their user id, user Type for permissions, email, etc.). The information is in JSON format and can be efficiently verified by the client application using cryptography.

8.5.2) JWT vs Other Authentication standards:

- JWT is suitable for stateless applications, as it allows the application to authenticate users and authorize access to resources without maintaining a session state on the server.
- OAuth, on the other hand, using access tokens, a client application can verify the identity of the user that authenticated themselves. When you implement
    - There's many plug-and-play OAuth options. Including services like "Sign in Google" and "Sign in with Facebook" that are already set up to be consumed within your application.
    - OAuth has well tested client libraries in almost all languages and web frameworks.
    - This means that your language of choice can be used with OAuth
    - It allows for decoupling of code.
    - Your client application code is not affected by the authentication code.

- **Cons. of it:**

1. maintains a session state on the server and uses a opaque token to grant access to the user's resources.
2. It has lower end user privacy. The auth server knows all the sites that the end user has logged in into. For example, when a site uses Sign in with Google, Google would be able to keep track of when that site's users are signing in or are active.

- In session-based authentication (also known as cookie-based authentication), the server is responsible for creating and maintaining a record of the user's authentication and providing a way for the client to reference that record in each subsequent request.

- **Cons. of it**: While session-based authentication is very reliable, at scale it can begin to introduce latency and performance issues. This information session will be stored in the database, and the session identifier will be sent back to the client to be stored as a cookie in the user's web browser.

8.5.3) Why Using JWT

- They are self-contained. The JWT can contain the user's details. So, you don't need to query a database / auth server for that information on each request.
- They offer strong security guarantees. JWTs are digitally signed which safeguards them from being modified by the client or an attacker.
- The flexibility of the claims allows you to communicate other important information to these external applications within the token itself. This can be very useful when exposing APIs to external applications.

8.5.4) Vulnerabilities of JWT:

You can't revoke them without putting in a lot of extra engineering effort. So, we can handle revoking by implementing JWT blacklist maybe using database or other tool, or we can minimize expiration time for each JWT token.

8.6) Using Gmail SMTP in Laravel

Prerequisites

Before proceeding with the configuration, make sure you have the following:

- Laravel installed on your local development environment or server.
- A Gmail account. If you don't have one, you can create a new account at https://accounts.google.com/signup.
- Composer installed on your system.

Step 1: Install Laravel's Mail Package

First, you need to ensure that Laravel's Mail package is installed in your project. If you haven't done so already, open your terminal or command prompt, navigate to your Laravel project directory, and run the following command:

```
composer require illuminate/mail
```

This command will install the required Mail package and its dependencies.

Step 2: Configure Gmail SMTP Credentials

Open your Laravel application and navigate to the .env file located in the root directory. Add the following lines to the file:

```
MAIL_MAILER=smtp
MAIL_HOST=smtp.gmail.com
MAIL_PORT=587
MAIL_USERNAME=online.clinic.c1@gmail.com
MAIL_PASSWORD=afqivpkfmmepzebj
MAIL_ENCRYPTION=null
MAIL_FROM_ADDRESS=online.clinic.c1@gmail.com
MAIL_FROM_NAME=online_clinic_c1
```

Step 3: Configure Laravel Mailer

Next, open the config/mail.php file in your Laravel application. Locate the mailers array and add or modify the smtp configuration as follows:

```
'mailers' => [
    'smtp' => [
        'transport' => 'smtp',
        'host' => env( key: 'MAIL_HOST',  default: 'smtp.mailgun.org'),
        'port' => env( key: 'MAIL_PORT',  default: 587),
        'encryption' => env( key: 'MAIL_ENCRYPTION',  default: 'tls'),
        'username' => env( key: 'MAIL_USERNAME'),
        'password' => env( key: 'MAIL_PASSWORD'),
        'timeout' => null,
        'local_domain' => env( key: 'MAIL_EHLO_DOMAIN'),
    ],
```

Step 4: Update Laravel Application Configuration

Open the config/app.php file in your Laravel application. Locate the providers array and ensure that the following service provider is present:

Illuminate\Mail\MailServiceProvider::class,

Now we are able to send emails.

Step 5: create email and make views

php artisan make:mail book_appointment

php artisan make:mail cancel_appointment

then we make to views and send data from it to the build function

book_appointment email

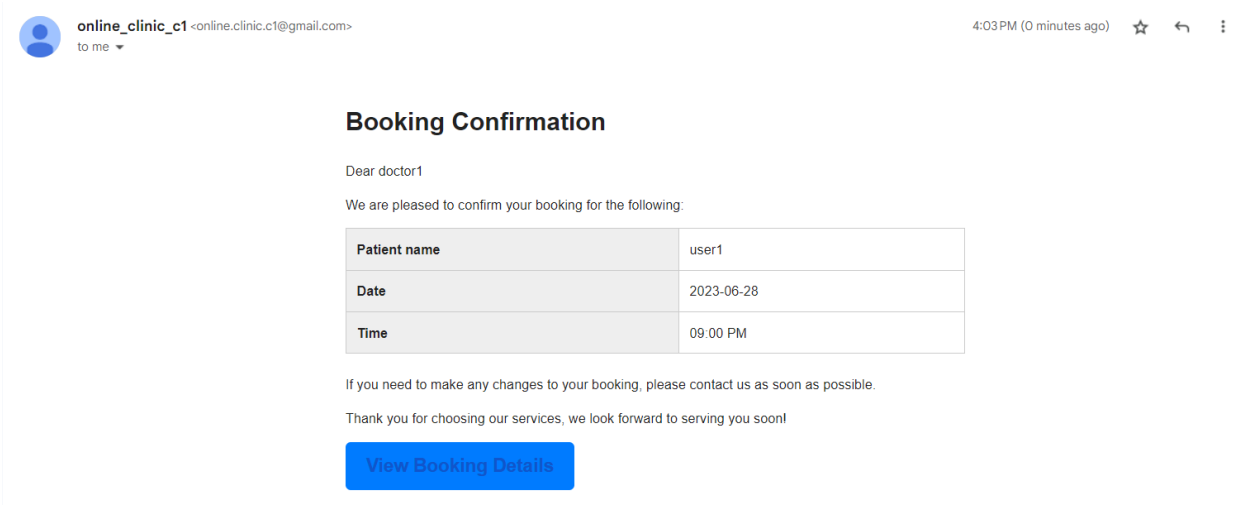
This email is sent to both doctors and patients and contain information about the booked appointment.

```php
public function __construct($all_data,$doctor,$user)
{
    $this->all_data=$all_data;
    $this->doctor=$doctor;
    $this->user=$user;
}

public function build()
{
    return $this->subject( subject: 'Your appointment is booked')
        ->view( view: 'emails.booked')
        ->with([['all_data'=>$this->all_data],['user'=>$this->user],['doctor'=>$this->doctor]]);
}
```

```php
Mail::to($user2->email)->send(new cancel_appointment($all_data,$user1,$user2));
```

Example



cancel_appointment email

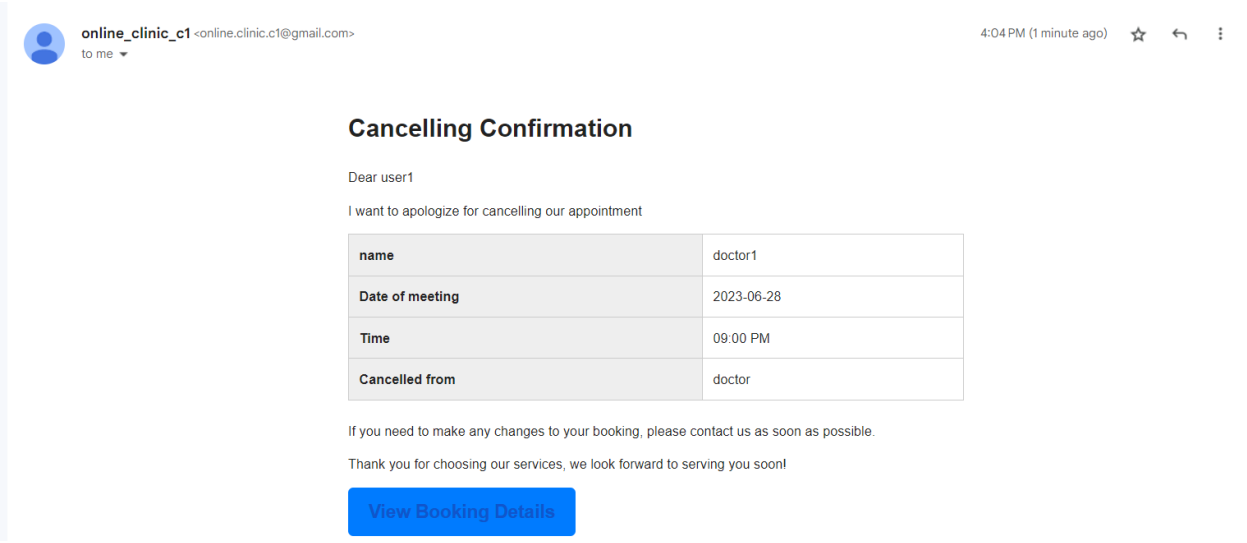
this email is sent as follows:

- If the email is cancelled from a patient an email is sent to the doctor to notify him.
- And if the email is cancelled from a doctor an email is sent to the patient to notify him.

```
public $all_data;
public $user1;
public $user2;
public function __construct($all_data,$user1,$user2)
{
    $this->all_data=$all_data;
    $this->user1=$user1;
    $this->user2=$user2;
}
public function build()
{
    return $this->subject( subject: 'Cancelling an appointment')
        ->view( view: 'emails.cancel')
        ->with([['all_data'=>$this->all_data],['user1'=>$this->user1],['user2'=>$this->user2]]);
}
```

```
Mail::to($doctor->email)->send(new book_appointment($all_data,$doctor,$user));
```

Example



8.7) Using Laravel Scheduler

The Laravel Scheduler is a component of Laravel's task scheduling system. It allows you to define tasks and specify when and how frequently they should run. The tasks are defined using the schedule method provided by the Illuminate\Console\Scheduling\Schedule class.

Functionality

- Deleting past free appointments from database.
- Open and close chat to be available only at booked time.

Step 1: Defining Scheduled Tasks

To define a scheduled task, open the app/Console/Kernel.php file in your Laravel application. In the schedule method, you can define your tasks using various methods provided by the Schedule class.

```php
protected function schedule(Schedule $schedule)
{
    // $schedule->command('inspire')->hourly();
     $schedule->command( command: 'open_chat')->everyMinute();
     $schedule->command( command: 'delete_past_appointments')->everyMinute();
}
```

As seen, we run our commands every minute

Step 2: Defining Our Commands

1- “Delete Appointments”

This command is responsible for deleting past free appointments.

```php
public function handle()
{
    $appointments=Appointment::where('appointment_state','=','free')->get();
    foreach ($appointments as $appoint){
        if ($appoint){
            $currentTime = Carbon::now()->addHours( value: 1)->addMilliseconds( value: 120000);
            $scheduledTime = Carbon::parse( time: $appoint->schedule_date . ' ' . $appoint->slot_time);
        if($currentTime->greaterThanOrEqualTo($scheduledTime)){
            $appoint->delete();
        }
    }
    }
```

2- “Open Chat”

This command is responsible for opening chat at the start of the appointment time and close it after the end of the appointment duration.

```php
        if($currentTime->greaterThanOrEqualTo($scheduledTime) and  $currentTime->lessThanOrEqualTo($endTime)){
            $openChats = Chat::where([['chat_from', $appoint->schedule_from], ['chat_to', $appoint->booked_from]])
                ->orWhere([['chat_from', $appoint->booked_from], ['chat_to', $appoint->schedule_from]])->get();
            $appoint->appointment_state="running";
            $appoint->save();
            foreach ($openChats as $openChat) {
                $openChat->is_open = 1;
                $openChat->save();}}
        elseif ($currentTime->greaterThanOrEqualTo($scheduledTime) and  $currentTime->greaterThanOrEqualTo($endTime)){
            $openChats = Chat::where([['chat_from', $appoint->schedule_from], ['chat_to', $appoint->booked_from]])
                ->orWhere([['chat_from', $appoint->booked_from], ['chat_to', $appoint->schedule_from]])->get();
            $appoint->appointment_state="done";
            $appoint->save();
            foreach ($openChats as $openChat) {
                $openChat->is_open = 0;
                $openChat->save();
            }
```

Step 3: Run Task scheduling

Use this command:

php artisan schedule: work

```
2023-06-28 14:47:00 Running ["artisan" open_chat] ..... 422ms DONE
↳ "C:\xampp\php\php.exe" "artisan" open_chat > "NUL" 2>&1
2023-06-28 14:47:00 Running ["artisan" delete_past_appointments] ..... 415ms DONE
```

8.8) Testing

Step 1: Setting up Testing Environment

Before you start writing tests, you need to ensure that your testing environment is properly set up. Laravel provides a dedicated directory called tests where you can store all your test files.

To run tests, Laravel uses PHPUnit, a popular testing framework for PHP. Laravel comes with PHPUnit pre-installed, so you don't need to install it separately.

To create a new test file, you can run the following command in your terminal:

```
PS C:\xampp\htdocs\online_clinic> php artisan make:test test1
```

This command will create a new test file named test1 in the test's directory.

Step 2: Writing Tests

Laravel tests are written as classes that extend the Testcase class provided by Laravel. Each test method within a test class should start with the word "test". Here's an example of a test class:

```
    // Make a POST request to the add_feedback endpoint
    $response = $this->postJson( uri: '/api/schedule/appointments', ['data' => $AppointmentData]);

    // Assert the response
    $response->assertStatus( status: 200);

    // Assert that the feedback was stored in the database
    $this->assertDatabaseHas( table: 'appointments', [
        'schedule_from' => 1,
        'schedule_date'=>"2024-02-26",
        'duration'=>3000000,
        'appointment_type'=>"chat",
        'slot_time' => '12:00 AM',
        'appointment_state' => 'free',

    ]);
}
/** @test */
```

Step 3: Running Tests

```
PS C:\xampp\htdocs\online_clinic> php artisan test
```

Result

```
  PASS  Tests\Feature\AppointmentTest
  ✓ it adds new appointments successfully when user is authenticated
  ✓ it returns all slots
  ✓ it returns unauthorized error when user is not authenticated

  PASS  Tests\Feature\AuthTest
  ✓ add user register a new user patient or doctor
  ✓ doctors  show  all verified doctors

  PASS  Tests\Feature\ChatTest
  ✓ it  create new chat successfully between two authenticated users
  ✓ it returns unauthorized error when user is not authenticated

  PASS  Tests\Feature\FeedbackTest
  ✓ it adds feedback successfully when user is authenticated
  ✓ it returns unauthorized error when user is not authenticated

  PASS  Tests\Feature\LikesTest
  ✓ it enable authenticated users to react posts
  ✓ it returns unauthorized error when user is not authenticated

  PASS  Tests\Feature\PostTest
  ✓ it adds posts successfully when user is authenticated
  ✓ it returns unauthorized error when user is not authenticated
  ✓ it shows all posts

  Tests:  14 passed
  Time:   5.86s
```

This test cases cover some of our main features like, schedule appointments, book appointments, submit post, likes and feedbacks

And we user the following assertions:

1- assertStatus

2- assertJsonStructure

3- assertDatabaseHas

8.9) API requirements

8.9.1) Authentication APIs

8.9.1.1) Register new user

http://127.0.0.1:8000/api/adduser

Functionality: The function (adduser) is responsible for handling the logic to add a new user based on the provided request data. It creates a new user record in the database and returns a response with the user data and a success message. If the request data is invalid or the user already exists, it returns an error response.

Type: post

Explanation:

1- At first, we check the username from the user
   a. If it is existed, return error message and state 400
   b. Else it is not used before, we heck the user type.
      i. If user is a patient create new record at user table
      ii. Else if the user is a doctor, create new record at user and doctors table then return status 200.

Function Logic

1. The function receives a request object containing the data for creating a new user.
2. It extracts the 'data' field from the request input and assigns it to the variable $u.
3. It checks if a user with the same username already exists by querying the User model with the username.

4. If a user with the same username is found, the function returns an error response indicating that the information is not found (inf. not found error) and includes an error code (ER_DUP_ENTRY).
5. If no user with the same username is found, the function proceeds to handle other fields from the input data.

6. It extracts the values for fields such as email, user type, birth date, prefix, phone, images, address, city, street, and province from the $u array.
7. It creates a new user record in the database using the User::create() method, passing the extracted values and other necessary fields such as name, password, gender, and image URL.
8. If the user type is "doctor", the function handles additional fields specific to doctors such as moreInf, chospital, gyear, eyears, achievement, about, and salary.
9. It creates a new doctor record in the database using the Doctor::create() method, passing the relevant fields.
10. The function retrieves the newly created user's data from the database.
11. It constructs a response with a success state (good, ok), a success message (information retrieved successfully), and the retrieved user's ID and name as data.
12. If the user type is "doctor", it includes the doctor-specific data in the response.
13. Finally, it returns the response with the appropriate HTTP status code.

Request Content:


```
{
"data":
{
"about": "goood",
"achievement":"nothing",
"address": {"province": "giza", "city": "elsk", "street":"wahet-g"},
"birth": "1990-02-07",
"chospital": "Dar-tawfeeq",
"email": "ahmedmohamed@gmail.com",
"eyears": 6,
"fees": 200,
"gender": "male",
"gyear": 1963,
"images": "…Z6UIhG+nuzvxnerPm9m/kXOF4uyIbIQAAAABJRU5ErkJggg==",
"moreInf": true,
"nickname": "ahmedmohamed",
"password": "AhmedAhmed22",
"phone": "01024352774",
"prefix": "20",
"salary": 3000,
"specialty": "Neurology",
"userType": "doctor",
"username": "ahmed2222"
}
}
```


Response:

- Case 1) Successful signup


```
{
    "state": "good, ok",
    "message": "information retreived successfully",
    "data": [
        {
            "user_id": 50
        }
    ]
}
```


- Case 2) User exist


```
{
    "state": "bad request",
    "message": "inf. not found",
    "data": {
        "err": {
            "code": "ER_DUP_ENTRY"
        }
    }
}
```

8.9.1.2) Login

http://127.0.0.1:8000/api/login

Functionality: The login() function handles the user login process by authenticating the user with the provided credentials.

Type: post

Explanation:

1- At first, we check if the user is authenticated or not, in case of unauthenticated we return status 401.

2- Check the needed appointment.

   i. If it is free, take the payment from the user, update database for this appointment to be booked, send email to both doctor and patient containing the appointment details and return status 200.

   ii. Else if the appointment is not free(booked), or data is not found return status 400.

   iii. Else if the appointment is not found in database(deleted from doctor), return appointment is cancelled and status 400

Function Logic

1. The function receives a request object containing the login credentials.
2. It extracts the 'data' field from the request input and assigns it to the variable $credentials.
3. It retrieves the username from the credentials.
4. The function attempts to authenticate the user using the Auth::attempt() method, passing the credentials. This method checks if the provided credentials are valid and returns a Boolean value.

5. If the authentication attempt is successful (valid credentials), the function generates a token for the authenticated user using Auth::attempt() again, and assigns it to the variable $token.
6. It retrieves the user data from the database based on the username.
7. The function constructs a response with a success state (good, ok), a success message (information retrieved successfully), and the user's ID, verification status, token, and token type as data.
8. It returns the response with the appropriate HTTP status code (200).
9. If the authentication attempt fails, the function checks if the username exists in the database.
10. If the username does not exist, it constructs an error response indicating that the username is incorrect (Incorrect username error) and includes a flag indicating that the user does not exist (isExist flag set to 0).
11. It returns the error response with the appropriate HTTP status code (400).
12. If the username exists but the password is incorrect, it constructs an error response indicating that the password is incorrect (Incorrect password error) and includes a flag indicating that the user is not verified (isVerified flag set to 0).
13. It returns the error response with the appropriate HTTP status code (400).

Request Content:

```
{
    "data":
        {
        "username": "ahmed22",
        "password": "AhmedAhmed22"
        }
}
```

Response:

- Case 1) Successful login


```
{
    "state": "good, ok",
    "message": "information retreived successfully",
    "data": {
        "isVerified": 1,
        "user_id": 5,
        "token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.
            eyJpc3MiOiJodHRwOi8vMTI3LjAuMC4xOjgwMDAvYXBpL2xvZ2luIiwiaWF0IjoxNjc5MTgwNDg0LCJleHAiOjE2NzkxODQwODQsIm5iZiI6MTY3OTE4MDQ4NCwianRpI
            joiV3d4WVluanBqakVWWFdEZyIsInN1YiI6IjUiLCJwcnYiOiIyM2JkNWM4OTQ5ZjYwMGFkYjM5ZTcwMWM0MDA4NzJkYjdhNTk3NmY3In0.
            e30pjAm3f-IPewVeDctYacXOwhzW-k9xjdiFNz2vlH4",
        "type": "bearer"
    }
}
```


- Case 2) Invalid username


```
{
    "state": "bad request",
    "message": "Incorrect username",
    "data": {
        "isExist": 0
    }
}
```


- Case 3) Incorrect password


```
{
    "state": "bad request",
    "message": "Incorrect password",
    "data": {
        "isVerified": 0
    }
}
```

8.9.1.3) Check user exist

http://127.0.0.1:8000/api/ chkuname/{username}

Functionality: The (get_slots) function is used for retrieving the available slots for a specific date and doctor. It retrieves the total slots, booked slots, free slots, and running slots for the given date and doctor. Type: post

Type: get

Explanation:

1- Take the username from the request and check if this user is existing or not

I. If user is existing return status 400.

II. Else return status 200

Function Logic

1. The function receives a string parameter $username representing the username to check.
2. It queries the database using the User model to find a user with the matching username. The where('username', $username)->first() method is used to retrieve the first matching user.
3. If a user is found (the username exists in the database), the function constructs an error response indicating that the information is not found. The state is set to "bad request", the message is set to "inf. not found", and the isUser flag is set to true.
4. If the username does not exist in the database, the function constructs a success response indicating that the information is retrieved successfully. The state is set to "good, ok", the message is set to "information retrieved successfully", and the isUser flag is set to false.
5. The function returns the appropriate response based on whether the username exists or not.

Response:

- Case 1) User is exist.

```
{
    "state": "bad request",
    "message": "inf. not found",
    "data": {
        "isUser": true
    }
}
```

- Case 2) user is not exist.

```
{
    "state": "good, ok",
    "message": "information retreived successfully",
    "data": {
        "isUser": false
    }
}
```

8.9.1.4) User page

http://127.0.0.1:8000/api/user

Functionality: The “user_after_login()” function retrieves user information after a successful login. Here's how the function works:

Type: get

Authentications: All users after login

Explanation:

1- The function first checks if the user is authenticated or not, If the user is not authenticated, it means the user is not authorized to access the API resources. And return status 401.

2- If the user is authenticated, we retrieve this user with status 200.

Function Logic

1. The function first checks if the user is authenticated by calling Auth::user(). If the user is not authenticated, it means the user is not authorized to access the API resources.

2. If the user is not authenticated, the function constructs an error response with the state set to "not authorized to access", the message set to "cannot access API resources", and the data containing the error details, including the name and message of the "JsonWebTokenError".
3. The function returns the error response with an HTTP status code of 401 (Unauthorized).
4. If the user is authenticated, the function sets the state to "good, ok" and the message to "information retrieved successfully".
5. It retrieves the authenticated user using Auth::user().
6. The function constructs the response data with the user's information, including the user ID, username, nickname, user type, birth date, gender, and image URL.
7. Finally, the function returns a success response with the state, message, and user data, along with an HTTP status code of 200 (OK).

Response:

- Case 1) Successful login


```
{
    "state": "good, ok",
    "message": "information retreived successfully",
    "data": {
        "user_id": 56,
        "user_name": "user44",
        "nick_name": "waleed22",
        "user_type": "doctor",
        "bdate": "2000-02-07T22:00:00.000Z",
        "gender": "male",
        "img_url": null
    }
}
```

- Case 2) Invalid token or token expired.


```
{
    "state": "not authorized to access",
    "message": "cannot access to api resources",
    "data": {
        "name": "JsonWebTokenError",
        "message": "invalid token"
    }
}
```


8.9.2 Appointment APIs

8.9.2.1) Schedule appointments

http://127.0.0.1:8000/api/schedule/appointments

Functionality: The (sched_appointment) function is used for scheduling and managing appointments. It handles the creation and deletion of appointments based on the input data provided in the request. Authentications: All doctors after login

Explanation:

2- At first, we check if the user is authenticated or not, in case of unauthenticated we return status 401.

3- We check the request as follows

- If it contains (“addedAppointments”) we add this appointment to database and return status 200.
- And if the request contains (“deletedAppointments”) we check as follows
    - If the appointment is free, we delete it and return status 200
    - And if it is already booked, we don’t delete it and return status 400.

Function Logic

1- The function first checks if the user is authenticated by calling Auth::user(). If the user is not authenticated, it returns an HTTP response with a 401 status code (Unauthorized). The response JSON object contains the following fields:

- state: "not authorized to access"
- message: "cannot access API resources"
- data: {"isUser": 0}

2- If the user is authenticated, the function retrieves the ID of the authenticated user (referred to as the doctor) using Auth::user()->id.

3- The function then retrieves the input data from the request using $request->input('data'). The input data is expected to be in the form of a JSON object.

4- The function extracts the date from the input data by accessing $all_data['date'].

5- Next, the function checks if the input data contains any "addedAppointments" entries. If so, it iterates over each appointment in the "addedAppointments" array and creates a new appointment record in the database using the Appointment model. The appointment details are extracted from each appointment object within the array.

6- After creating the appointments, the function returns an HTTP response with a 200-status code (OK) indicating success. The response JSON object contains the following fields:

- state: "good, ok"
- message: "your data added successfully"
- data: {"isFirst": 1}

7- If the input data contains any "deletedAppointments" entries, the function retrieves the first entry from the "deletedAppointments" array.

8- It queries the Appointment model to find the appointment with the given ID using Appointment::where('id', '=', $del_id)->first().

9- If the appointment exists, it deletes the appointment record from the database using $del->delete().

10- The function then returns an HTTP response with a 200-status code (OK) indicating success. The response JSON object contains the following fields:

- state: "good, ok"
- message: "information retrieved successfully"
- data: ` {"done": true} `

Request Content:

```
{
    "data":{
        "date":"2023-3-17"
        ,
        "addedAppointments":[
            {
            "slotTime":"10:50 am",
            "appointmentType":"chat",
            "appointmentDuration":20000
            },
            {
            "slotTime":"11:50 am",
            "appointmentType":"chat",
            "appointmentDuration":20000
            }
        ]
    }
}
```

Or

```
{
    "data":{
        "date":"2023-3-17"
        ,
        "deletedAppointments":[
            "11:50 am"
        ]
    }
}
```

Response:

- Case 1) Added successfully.


```
{
    "state": "good, ok",
    "message": "your data added successfully",
    "data": {
        "isFirst": 1
    }
}
```


- Case 2) appointment is already booked so it cannot be deleted.


```
{
    "state": "bad request",
    "message": "inf. not found",
    "data": {
        "timeSlots": [
            "9:00 AM"
        ]
    }
}
```


- Case 3) appointment is deleted successfully.


```
{
    "state": "good, ok",
    "message": "information retreived successfully",
    "data": {
        "done": true
    }
}
```

8.9.2.2) Book appointments

http://127.0.0.1:8000/api/ book /appointment

Functionality: The (book_appointment) function is a PHP function used for booking appointments and handling payment-related operations. It allows authenticated users to book appointments with doctors and process payments using the Stripe payment gateway.

Type: post

Authentications: All users after login

Explanation:

3- At first, we check if the user is authenticated or not, in case of unauthenticated we return status 401.

4- Check the needed appointment.

   i. If it is free, take the payment from the user, update database for this appointment to be booked, send email to both doctor and patient containing the appointment details and return status 200.

   ii. Else if the appointment is not free(booked), or data is not found return status 400.

   iii. Else if the appointment is not found in database(deleted from doctor), return appointment is cancelled and status 400

Function Logic

1- The function first checks if the user is authenticated by calling Auth::user(). If the user is not authenticated, it returns an HTTP response with a 401 status code (Unauthorized). The response JSON object contains the following fields:

- state: "not authorized to access"
- message: "cannot access API resources"
- data: { "isUser": 0 }

2- The function sets the Stripe secret key using the provided value.

3- If the query parameter pi is present in the request URL, the function retrieves the payment intent using the Stripe PHP SDK and the provided payment intent ID ($pi). If the payment intent contains a description, it is assumed to contain appointment data, which is then extracted and stored in the $appointment_data variable.

4- If the query parameter check is present in the request URL, it indicates that the user is checking the availability of an appointment. In this case, the function returns an HTTP response with a 200 status code (OK) indicating success. The response JSON object contains the following fields:

   - state: "good, ok"
   - message: "your data added successfully"
   - data: { "done": 1 }

   Additionally, an email is sent to the doctor providing the appointment details using the book_appointment Mailable.

5- If none of the above conditions are met, the function assumes that the user is trying to book an appointment. It retrieves the authenticated user's ID and the input data from the request.

6- The function extracts the necessary data from the input data, such as the appointment date, booked time slot, and doctor ID.

7- It queries the Appointment model to find the appointment matching the doctor ID, date, and time slot.

8- If the appointment does not exist, the function returns an HTTP response with a 400 status code (Bad Request) indicating an error. The response JSON object contains the following fields:

   - state: "bad request"
   - message: "inf. not found"
   - data: { "isCanceled": true }

9- If the appointment is already booked, the function returns an HTTP response

Request Content:

```
{
    "data":{
        "date":"2023-2-27",
        "bookedSlot":"9:00 AM",
        "doctorId":26
    }
}
```

Response:

- Case 1) The appointment is booked successfully.

```
{
    "state": "good, ok",
    "message": "your data added successfully",
    "data": {
        "isFirst": 0
    }
}
```

- Case 2) The appointment is already booked.

```
{
    "state": "bad request",
    "message": "inf. not found",
    "data": {
        "isBooked": true
    }
}
```

- Case 3) Unauthorized

```
{
    "state": "not authorized to access",
    "message": "cannot access to api resources",
    "data": {
        "isUser": 0
    }
}
```

- Case 4) The appointment is cancelled by the doctor before.

```
{
    "state": "bad request",
    "message": "inf. not found",
    "data": {
        "isCanceled": true
    }
}
```

8.9.2.3) Get Slots

http://127.0.0.1:8000/api/get/slots

Functionality: The (get_slots) function is used for retrieving the available slots for a specific date and doctor. It retrieves the total slots, booked slots, free slots, and running slots for the given date and doctor. Type: post

Authentications: All users after login

Explanation:

1- At first, we check if the user is authenticated or not, in case of unauthenticated we return status 401.

2- Check date and doctor id and get all appointments according to that and classify them as free, booked and running.

Function Logic

1- The function retrieves the input data from the request using $request->input('data'). The input data is expected to be in the form of a JSON object.

2- It extracts the necessary data from the input data, such as the appointment date and doctor ID.

3- The function initializes empty arrays for storing the total slots, booked slots, free slots, and running slots.

4- It queries the Appointment model to retrieve all appointments for the given doctor and date using $ts = Appointment::where([['schedule_from', '=', $doctor_id], ['schedule_date', '=', $date]])->get();.

5- The function iterates over each appointment in the retrieved total slots ($ts) and adds the necessary information to the $slot array. The $slot array is then pushed to the $totalSlots array.

6- Similarly, it retrieves the booked slots, free slots, and running slots using separate queries to the Appointment model ($bs, $fs, $rs).

7- For each booked slot, free slot, and running slot, the function adds the necessary information to the $slot array and pushes it to the respective arrays ($bookedSlots, $freeSlots, $runningSlots).

8- If there are no free slots available, the $freeSlots array is set to null.

9- The function returns an HTTP response

Request Content:

```
{
    "data":{
        "date":"2023-03-17",
        "doctorId":53
    }
}
```

Response:


```
{
    "state": "good, ok",
    "message": "information retreived successfully",
    "data": {
        "totalSlots": [
            {
                "appointmentState": "free",
                "appointmentType": "chat",
                "slotTime": "10:50 am",
                "appointmentDuration": 20000
            }
        ],
        "bookedSlots": [],
        "freeSlots": [
            {
                "appointmentState": "free",
                "appointmentType": "chat",
                "slotTime": "10:50 am",
                "appointmentDuration": 20000
            }
        ]
    }
}
```

8.9.2.4) Cancel appointments

http://127.0.0.1:8000/api/cancel/appointment?userid=6

Functionality: The (cancel_appointment) function is used for canceling an appointment. It allows either the doctor or the patient to cancel the appointment based on the provided data.

Type: post

Authentications: All users after login

Explanation:

1- At first, we check if the user is authenticated or not, in case of unauthenticated we return status 401.

2- Check the needed appointment.

i. If the appointment is cancelled from user, The appointment state is updated to "cancelled" in the database and send email to the doctor.

ii. Else if the appointment is cancelled from doctor, The appointment state is updated to "cancelled" in the database and send email to the patient.

Function Logic

1- The function checks if the user is authenticated. If not, it returns an unauthorized access response.

2- The function retrieves the input data from the request using $request->input('data'). The input data is expected to be in the form of a JSON object.

3- It extracts the necessary data from the input data, such as the appointment date, cancellation initiator (doctor or patient), and booked slot.

4- If the cancellation initiator is a doctor, the function retrieves the doctor's ID from the authenticated user and queries the Appointment model to find the appointment based on the doctor's ID, date, and booked slot.

5- Once the appointment is found, the function retrieves the ID of the patient who booked the appointment and sends a cancellation email notification to the patient using the cancel_appointment mail class. The appointment state is updated to "cancelled" in the database.

6- If the cancellation initiator is a patient, the function retrieves the patient's ID from the authenticated user and queries the Appointment model to find the appointment based on the patient's ID, doctor's ID, date, and booked slot.

7- Once the appointment is found, the function sends a cancellation email notification to the doctor using the cancel_appointment mail class. The appointment state is updated to "cancelled" in the database.

8- The function constructs the response JSON object with the appropriate state, message, and data fields.

9- The function returns an HTTP response with the constructed JSON object

Request Content:

```
{
    "state": "good, ok",
    "message": "information retreived successfully",
    "data": {
        "fromDoctor": 1
    }
}
```

Or

```
{
    "data":{
        "date":"2023-3-17"
        ,
        "deletedAppointments":[
            "11:50 am"
        ]
    }
}
```

Response:

- Case 1) cancelled by doctors.


```
{
    "state": "good, ok",
    "message": "information retreived successfully",
    "data": {
        "fromDoctor": 1
    }
}
```


- Case 2) cancelled by patients.


```
{
    "state": "good, ok",
    "message": "information retreived successfully",
    "data": {
        "fromPatient": 1
    }
}
```


8.9.2.5) Get appointments

http://127.0.0.1:8000/api/get/appointments?date=2023-3-17

http://127.0.0.1:8000/api/get/appointments?date=2023-3-17&doctor=true

Functionality: The (get_appointments) function is used for retrieving appointments. It retrieves either the doctor's appointments or the patient's appointments based on the provided data.

Type: get

Authentications: All users after login

Explanation:

1- At first, we check if the user is authenticated or not, in case of unauthenticated we return status 401.

2- We check the request:

   i. If the doctor query parameter is present, this means that we return the appointments of the user as a doctor.

   ii. Else this means that we return the appointments of the user as a patient.

Function Logic

1- First, it checks if the user is authenticated. If not, it returns a response indicating that the user is not authorized to access the API resources.

2- The function retrieves the date parameter from the query string.

3- It gets the user_id of the authenticated user.

4- If the doctor query parameter is present, it means the user is a doctor. The function queries the Appointment model to fetch all appointments scheduled by the doctor (schedule_from) on the specified date.

5- For each appointment, the function constructs an array (doctorData) containing the relevant appointment details, including patient and doctor information. It also populates additional fields such as profile image URLs, usernames, doctor's name, rating, fees, specialty, and verification status.

6- The doctorData array is then appended to the data array.

7- If the doctor query parameter is not present, it means the user is a patient. The function queries the Appointment model to fetch all appointments booked by the patient (booked_from) on the specified date.

8- Similar to the previous case, for each appointment, the function constructs an array (data_push) with the relevant appointment details, including patient and doctor information. It also populates additional fields such as profile image URLs, usernames, doctor's name, rating, fees, specialty, and verification status.

9- The data_push array is then appended to the data array.

10- Finally, the function returns an HTTP response with the state, message, and data fields, containing the fetched appointment information.

Response:

- Case 1) unauthorized or the token is expired


```
{
    "state": "not authorized to access",
    "message": "cannot access to api resources",
    "data": {
        "name": "TokenExpiredError",
        "message": "jwt expired",
        "expiredAt": "2023-03-17T20:39:58.000Z"
    }
}
```


- Case 2) When the authenticated is a doctor(doctor=true).


```
{
    "state": "good, ok",
    "message": "information retreived successfully",
    "data": [
        {
            "patientId": 53,
            "doctorId": 53,
            "schedule_date": "2023-03-17",
            "booked_time": "2023-03-17T18:22:23.000Z",
            "slot_time": "10:50 am",
            "appointment_state": "done",
            "appointment_type": "chat",
            "uimgUrl": null,
            "dimgUrl": null,
            "username": "user22",
            "doctorName": "user22",
            "rate": 0,
            "fees": 100,
            "specialty": "Neurology",
            "test_results": null,
            "current_issue": null,
            "allergies": null,
            "immunizations": null,
            "surgeries": null,
            "illnesses_history": null
        }
    ]
```

- Case 2) When the authenticated is a patient.


```
{
    "state": "good, ok",
    "message": "information retreived successfully",
    "data": [
        {
            "patientId": 53,
            "doctorId": 53,
            "schedule_date": "2023-03-17",
            "booked_time": "2023-03-17T18:22:23.000Z",
            "slot_time": "10:50 am",
            "appointment_state": "done",
            "appointment_type": "chat",
            "uimgUrl": null,
            "dimgUrl": null,
            "username": "user22",
            "doctorName": "user22",
            "rate": 0,
            "fees": 100,
            "specialty": "Neurology"
        },
        {
            "patientId": 53,
            "doctorId": 53,
            "schedule_date": "2023-03-17",
            "booked_time": "2023-03-17T18:51:06.000Z",
            "slot_time": "11:50 am",
            "appointment_state": "done",
```


8.9.2.6) Edit appointment

http://127.0.0.1:8000/api/edit/appointment

Functionality: The (edit_appointments) function is responsible for handling the editing of appointments based on the provided request data. It allows authenticated doctors to modify the details of existing appointments in the system.

Type: post

Authentications: All users after login

Explanation:

1- At first, we check if the user is authenticated or not, in case of unauthenticated we return status 401.

2- Given a specific appointment we update database with new appointment data and return status 200.

Function Logic

1- It starts with a check to see if the user is authenticated using Auth::user(). If the user is not authenticated, the function returns a response indicating that the user is not authorized to access the API resources.

2- The function retrieves the authenticated user's ID using Auth::user()->id.

3- The function expects a request object ($request) to be passed as a parameter. It retrieves the input data from the request using $request->input('data').

4- It extracts the necessary data from the input data, such as the date and the ID of the appointment to be edited.

5- It queries the appointments table to retrieve the appointment that matches the provided ID using Appointment::where('id', '=', $edit_id)->first(). The retrieved appointment is stored in the variable $appointment_edit.

6- If the key "addedAppointments" exists in the input data, it means that new appointment data has been provided. The code checks if the addedAppointments array is not null.

7- If the addedAppointments array is not null, it retrieves the first element of the array ($temp) and updates the corresponding fields of the $appointment_edit object with the new data.

8- The changes are saved by calling $appointment_edit->save().

9- The function returns a response indicating that the data has been successfully edited, along with a status code of 200.

Request Content:

```
{
    "data":{

    "date":"2023-04-08",
    "addedAppointments":[
        {
        "slotTime":"05:00 PM",
        "appointmentDuration":60000,
        "appointmentType":"chat"
        }
    ],
    "editSlot":"02:00 PM"
    }
}
```

Response:

```
{
    "state": "good, ok",
    "message": "your data added successfully",
    "data": {
        "done": true
    }
}
```

8.9.3 Feedback APIs

8.9.3.1Submit Feedback

http://127.0.0.1:8000/api/submit/feedback

Functionality: The (add_feedback) function is used to enable users to insert feedback to doctors.

Type: Post

Authentications: All users after login

Explanation:

4- At first, we check if the user is authenticated or not, in case of unauthenticated we return status 401.

5- We check the request as follows

- If there is feedback from this patient to a certain doctor, we delete it and add the new one to feedback table.
- And if this is the first feedback, add it directly to feedback table.

Function Logic

1. First, the function checks if the user is authenticated by calling Auth::user(). If the user is not authenticated (!Auth::user()), the function returns a response with a status code of 401 (Unauthorized) and includes an error state, message, and data indicating that the user is not authorized to access the API resources.
2. If the user is authenticated, the function retrieves the user ID from the authenticated user object (Auth::user()->id).

3. The function then extracts the input data from the request payload using $request->input('data'). It assumes that the input data is structured as an array with keys 'rate', 'feedback', and 'feedback_to'.
4. The code attempts to find existing feedback records for the current user ($user_id) and the specified feedback recipient ($feedback_to) using the Feedback model. The where clause is used to specify the conditions for the query. If no existing feedback records are found (empty($all_feed)), it means that this is the first feedback from the user to the specified recipient.
5. If it is the first feedback, the code creates a new Feedback record using the create method of the model. It assigns the user ID ($user_id), feedback recipient ID ($feedback_to), rating ($rate), and feedback message ($feedback) to the corresponding fields of the Feedback model.
6. If there are existing feedback records, the code deletes the existing feedback using $all_feed->delete() and then creates a new Feedback record with the same values as in step 5.
7. After saving the feedback record, the code sets the $first variable based on whether it was the first feedback or not. This variable will be included in the response payload.
8. Finally, the function returns a response with a status code of 200 (OK). It includes a success state, a message indicating that the data was added successfully, and data containing the value of the $first variable

Request Content:


```
{
   "data":{
   "rate":4,
   "feedback":"helllo",
   "feedback_to":59
   }
}
```

Response:

- Case 1) first feedback.


```
{
    "state": "good, ok",
    "message": "information retreived successfully",
    "data": {
        "isFirst": true
    }
}
```


- **Case 2) appointment is already booked so it cannot be deleted.**


```
{
    "state": "good, ok",
    "message": "information retreived successfully",
    "data": {
        "isFirst": false
    }
}
```

8.9.3.2) Get all feedbacks

http://127.0.0.1:8000/api/get/feedback?limit=5

http://127.0.0.1:8000/api/get/feedback?username=ahmed

Functionality: to show all feedbacks for all doctors or show feedbacks for a specific doctor

Type: Get

Explanation:

1. We check the request
    i. If it contains “username”, get all feedbacks for this doctor only from Feedback table.
    ii. Else get all feedbacks for all doctors from Feedback table.

Function Logic

1. The function starts by initializing the variables $state and $message with the values "good, ok" and "information retrieved successfully" respectively.
2. The function checks if the username parameter is set in the GET request (isset($_GET['username'])). If it is set, it retrieves the value of username from the GET parameters and searches for a Doctor record with a matching username using the Doctor model. If a matching record is found, it retrieves the corresponding User record with the same username using the User model. It then fetches the feedback records for the specified doctor ($doctor->user_id) and limits the results to the first 5 records using skip(0)->take(5)->get().
3. Inside a loop, the function retrieves additional information related to each feedback record. It fetches the corresponding User record of the feedback sender using the User model, and builds an array of data containing various feedback details and URLs to user and doctor images. Each data array is then appended to the $data array using array_push().
4. After the loop completes, the function returns a response with a JSON payload containing the success state, message, and the populated $data array.
5. If the username parameter is not set or the doctor record is not found in the previous step, the function assumes that the parameter value is an ID instead. It searches for a Doctor record with a matching user_id and retrieves the corresponding User record using the User model. The rest of the logic remains the same as in step 3.
6. If the limit parameter is set in the GET request (isset($_GET['limit'])), the function retrieves the value of limit from the GET parameters and fetches

7. the feedback records limited to the specified limit using feedback::limit($limit)->get(). The remaining logic is similar to step 3, where additional information is fetched and added to the $data array.
8. If neither the username nor the limit parameters are set, the function assumes a default limit of 10 and retrieves the feedback records accordingly. The subsequent logic remains the same as in step 3.
9. Finally, the function returns a response with a JSON payload containing the success state, message, and the populated $data array.

Response:

Case 1) all feedbacks for all doctors


```
{
    "state": "good, ok",
    "message": "information retreived successfully",
    "data": [
        {
            "feedback_from": 2,
            "feedback_to": 20,
            "rate": 3.5,
            "issued_time": "2023-02-18T17:14:41.000Z",
            "feedback": null,
            "uimgUrl": null,
            "dimgUrl": null,
            "username": "waleed",
            "doctorName": "wad"
        },
        {
            "feedback_from": 4,
```


Case 2) all feedbacks for a special doctor


```
{
    "state": "good, ok",
    "message": "information retreived successfully",
    "data": [
        {
            "feedback_from": 18,
            "feedback_to": 21,
            "rate": 3,
            "issued_time": "2023-02-18T19:46:30.000Z",
            "feedback": null,
            "uimgUrl": null,
            "dimgUrl": null,
            "username": "wad",
            "doctorName": "waleedsaber33"
        }
    ]
}
```

8.9.4 Posts, Comments and Likes APIs

8.9.4.1) Get all posts

http://127.0.0.1:8000/api/get/posts

Functionality: The get_posts() function retrieves all posts from the database and constructs a response containing information about each post and the corresponding user who created the post.

Type: get

Explanation: Get all posts from post table and data about this post from user table then return status 200.

Function Logic

1. The function initializes an empty array, $data, which will hold the response data.
2. It retrieves all posts from the Post model using Post::all().
3. The function then iterates through each post using a foreach loop and retrieves the user who created the post by querying the User model based on the user's ID.
4. Inside the loop, it constructs the response data for each post by creating an associative array ($res) with the relevant information such as the post ID, user ID, content, issued time, number of comments, like emoji, dislike count, angry count, post image, nickname of the user, and image URL of the user.
5. The constructed response data for each post is added to the $data array.
6. After iterating through all posts, the function sets the state to "good, ok" and the message to "your data added successfully".
7. Finally, the function returns a success response with the post data ($data) in the response body and an HTTP status code of 200 (OK).

Response:

```
{
    "state": "good, ok",
    "message": "information retreived successfully",
    "data": [
        {
            "post_id": 80,
            "user_id": 61,
            "content": "ds",
            "issued_time": "2023-04-14T01:33:45.000Z",
            "num_comments": 0,
            "like_emoji": 0,
            "dislike": 0,
            "angry": 0,
            "post_img": null,
            "nick_name": "waleed22",
            "img_url": null
        },
        {
            "post_id": 79,
            "user_id": 61,
            "content": "ff",
            "issued_time": "2023-04-14T01:33:15.000Z",
            "num_comments": 0,
            "like_emoji": 0,
            "dislike": 0,
            "angry": 0,
            "post_img": null,
```


8.9.4.2) Get comments and replies

http://127.0.0.1:8000/api/get/comments?postId=78

Functionality: The get_comments() function you provided retrieves comments and their corresponding replies for a specific post based on the 'postId' query parameter. It constructs a response containing information about each comment and its associated user. Here's the modified code with some improvements:

Type: get

Explanation:

The function get comments and replies to data from comment, reply and user tables based on the postId

Function Logic

1. It checks if the 'postId' query parameter is set. If it is, the function updates the $post_id variable with the provided value.

2. The function retrieves all comments for the specified post from the Comment model using Comment::where([['post_id', '=', $post_id]])->get().
3. It also retrieves all replies for the specified post from the Reply model using Reply::where([['post_id', '=', $post_id]])->get().
4. The function initializes the $state variable with the value "good, ok" and the $message variable with the value "your data added successfully". It also initializes an empty array, $data, which will hold the response data.
5. It then iterates through each comment using a foreach loop and retrieves the user who posted the comment by querying the User model based on the user's ID.
6. Inside the loop, it constructs the response data for each comment by creating an associative array ($res) with the relevant information such as the comment ID, post ID, user ID, content, issued time, number of replies, like emoji, dislike count, angry count, nickname of the user, and image URL of the user.
7. The constructed response data for each comment is added to the $data array.
8. After iterating through all comments, the function iterates through each reply using another foreach loop. It retrieves the comment on which the reply was made and the user who posted the comment in a similar manner as in step 6.
9. Inside this loop, it constructs the response data for each reply in the same format as for comments.
10. The constructed response data for each reply is added to the $data array.
11. Finally, if the 'postId' query parameter is set and the retrieval of comments and replies is successful, the function returns a success response with the comment data ($data) in the response body and an HTTP status code of 200 (OK). Otherwise, it returns null or an appropriate response based on your application logic.

Response:


```
{
    "state": "good, ok",
    "message": "information retreived successfully",
    "data": [
        {
            "reply_on": 152,
            "comment_id": 153,
            "post_id": 78,
            "user_id": 61,
            "content": "goood goood",
            "issued_time": "2023-04-14T01:19:09.000Z",
            "num_replies": 0,
            "like_emoji": 0,
            "dislike": 0,
            "angry": 0,
            "nick_name": "waleed22",
            "img_url": null
        },
        {
            "reply_on": null,
            "comment_id": 152,
            "post_id": 78,
            "user_id": 61,
            "content": "gooooooooood job",
            "issued_time": "2023-04-14T01:16:03.000Z",
            "num_replies": 1,
            "like_emoji": 0,
```


8.9.4.3) Submit new post

http://127.0.0.1:8000/api/submit/post

Functionality: The post function you provided is used to create a new post. It retrieves the necessary data from the request and creates a new Post object in the database. Here's the modified code with some improvements:

Type: post

Authentications: All users after login

Explanation:

1- At first, we check if the user is authenticated or not, in case of unauthenticated we return status 401.

2- Add post data to post table and return status 200.

Function Logic

1. It retrieves the user ID of the authenticated user using Auth::id(). The Auth facade provides convenient methods for working with authentication in Laravel.
2. The function extracts the data field from the request payload using $request->input('data'). This assumes that the request payload contains a data field that holds the post information.
3. The content variable is assigned the value of the content field from the extracted data.
4. The img variable is set to null initially. It checks if the postImg field is present in the extracted data using the isset() function. If the field exists, its value is assigned to the img variable; otherwise, it remains null.
5. A new Post model instance is created using the create() method. The create() method allows for mass assignment of attributes, where the provided array keys correspond to the model's attributes.
6. The Post model is populated with the user ID, content, and image (if available).
7. After the post is successfully created, the response data is constructed. The state variable is set to "good" to indicate a successful operation. The message variable contains a success message. The data array includes the post_id key, which holds the ID of the newly created post.
8. The function returns an HTTP response using the response() helper function. The compact() function is used to create an array of the state, message, and data variables.
9. The HTTP response has a status code of 200, indicating a successful request.

Request Content:

```
{
    "data":{
        "content":"hello, world",
        "postImg":null
    }
}
```

Response:

```
{
    "state": "good, ok",
    "message": "your data added successfully",
    "data": {
        "post_id": 78
    }
}
```

8.9.4.4) Submit comment and reply

http://127.0.0.1:8000/api/submit/comment

Functionality: The (comment) function enables users to make comments and replies

Type: post

Authentications: All users after login

Explanation:

1- At first, we check if the user is authenticated or not, in case of unauthenticated we return status 401.

2- Check the request for (commentId).

i. If we have commentId add new record to reply table.

ii. Else add new record to comment table

Function Logic

1. The function expects an instance of the Illuminate\Http\Request class as a parameter. This object represents the HTTP request made to the server.
2. It retrieves the user ID of the authenticated user using Auth::id().
3. The function extracts the data field from the request payload using $request->input('data'). This assumes that the request payload contains a data field that holds the comment information.
4. The content variable is assigned the value of the content field from the extracted data.
5. The postId variable is assigned the value of the postId field from the extracted data.
6. The commentId variable is set to null initially. It checks if the commentId field is present in the extracted data using the isset() function. If the field exists, its value is assigned to the commentId variable; otherwise, it remains null.
7. If commentId is set, it means that the comment is a reply to an existing comment. In this case, a new Comment model instance is created using the create() method. The create() method allows for mass assignment of attributes, where the provided array keys correspond to the model's attributes. The user ID, content, and post ID are assigned to the comment.
8. A new Reply model instance is created using the create() method. The Reply model represents a reply to a comment. The post_id field is assigned the post ID, the reply_on field is assigned the ID of the commented-on comment, and the reply_id field is assigned the ID of the newly created comment.
9. After the comment and reply are successfully created, the response data is constructed. The state variable is set to "good" to indicate a successful operation. The message variable contains a success message. The data array includes the comment_id key, which holds the ID of the newly created comment or reply.

10. If commentId is not set, it means that the comment is a top-level comment on the post. In this case, a new Comment model instance is created with the user ID, content, and post ID.
11. After the comment is successfully created, the response data is constructed similar to the previous case.
12. The function returns an HTTP response using the response() helper function. The compact() function is used to create an array of the state, message, and data variables.
13. The HTTP response has a status code of 200, indicating a successful request.

Request Content:

```
{
    "data":{

    "content":"goooooooooood job",
    "postId":78
    }
}
```

Or

```
{
    "data":{
        "content":"goood goood",
        "postId":78,
        "commentId":152
    }
}
```

Response:

```
{
    "state": "good, ok",
    "message": "your data added successfully",
    "data": {
        "comment_id": 152
    }
}
```

8.9.4.5) Submit likes

http://127.0.0.1:8000/api/submit/like

Functionality: The add_like() function handles the logic for adding a like to a post or comment

Type: post

Authentications: All users after login

Explanation:

1- At first, we check if the user is authenticated or not, in case of unauthenticated we return status 401.

2- Check the request for (commentId).

   i. If we have “commentId” add new record to like table as a like for a comment.

   ii. Else add new record to like table as a like for a post.

Function Logic

1. It begins by checking if the user is authenticated (Auth::user()). If the user is not authenticated, it returns a response with an error message and HTTP status code 401 (Unauthorized).
2. If the user is authenticated, it retrieves the user's ID ($user_id) from the authenticated user object.
3. The function expects a request object ($request) as a parameter. It retrieves the input data from the request using $request->input('data').
4. It checks if the likeType and postId fields exist in the input data. If likeType is not provided, it defaults to "like". If isPost is not provided, it defaults to false.
5. If isPost is true, the function handles the like for a post. It counts the number of likes, dislikes, and angry reactions for the post using the Like model.

6. It checks if the user has already liked the post before ($before_like). If the user has already liked the post and the like type matches the current like type, it deletes the like entry, decrements the respective counters, and prepares a response with the updated data.

7. If the user has already liked the post but wants to change the like type, it updates the existing like entry with the new like type, updates the counters accordingly, and prepares a response with the updated data.

8. If the user has not liked the post before, it creates a new like entry with the provided like type, increments the respective counters, and prepares a response with the new data.

9. If isPost is false, it means the like is for a comment. It follows a similar process as above but for comment likes.

10. If the function does not find commentId in the input data, it prepares a response with an empty data field.

11. Finally, it prepares a response with the state, message, and data fields and returns it with an HTTP status code 200 (OK).

Request Content:

```
{
    "data":{
        "likeType":"like",
        "isPost":true,
        "postId":83,
        "commentId":null
    }
}
```

Or

```
{
    "data":{
        "likeType":"angry",
        "postId":83,
        "commentId":162
    }
}
```

Response:

Case 1: like for a comment


```
{
    "state": "good, ok",
    "message": "your data added successfully",
    "data": {
        "like_id": 78,
        "post_id": 83,
        "comment_id": 162,
        "user_id": 63,
        "is_post": null,
        "like_type": "angry"
    }
}
```


Case 2: like for a post


```
{
    "state": "good, ok",
    "message": "your data added successfully",
    "data": {
        "like_id": 62,
        "post_id": 83,
        "comment_id": null,
        "user_id": 63,
        "is_post": 1,
        "like_type": "like"
    }
}
```

8.9.5) Chat APIs

8.9.5.1) Chat and message

http://127.0.0.1:8000/api/submit/message

Functionality: The add_msg() function allows adding a new message to the chat between two users.

Type: post

Authentications: All users after login

Explanation:

1- At first, we check if the user is authenticated or not, in case of unauthenticated we return status 401.

2- We check the if there is a chat before or not

- If there is a chat before between the two users, we check to send message or not
    - If the user is a patient and chat ("is_open" =false) don't send the message and show unauthenticated then return status 401.
    - Else the user is admin or doctor send the message directly.
- If there is not a chat before, create new chats in chat table(2 records) and then
    - If the user is a patient and chat ("is_open" =false) don't send the message and show unauthenticated then return status 401.
    - Else the user is admin or doctor send the message directly.

Function Logic

1. The function is called add_msg and accepts a Request object as a parameter. This suggests that it is a part of a Laravel application, as Laravel uses the Request object to handle HTTP requests.

2. The function starts by checking if the user is authenticated (Auth::user()). If the user is not authenticated, it returns a response with an error message and a status code of 401 (Unauthorized). The response includes a JSON payload with the state, message, and data fields.
3. If the user is authenticated, the function continues to process the message. It retrieves the authenticated user's ID using Auth::user()->id and assigns it to the $user1_id variable. It also assigns the authenticated user object to the $sender variable.
4. The function then retrieves the message data from the request object. It expects the data to be passed as an array with a key named data. It retrieves the content and message_to values from the data array using isset() checks. If any of these values are missing, they are set to null or a default value.
5. The user2_id variable is assigned the value of the message_to field, which represents the recipient's user ID. If the message_to field is not provided, it defaults to 2.
6. The function checks if there is any existing chat history between the users. It queries the Chat model to find an chats where the chat_from field matches the authenticated user ID ($user1_id) and the chat_to field matches the user2_id. It also checks for the reverse scenario where the authenticated user ID is the chat_to field and user2_id is the chat_from field. The results are stored in the $old_chat variable.
7. If there is no existing chat history and the authenticated user ID is not the same as user2_id, the function creates new chat entries for both users. It creates a chat from the authenticated user to the recipient ($user1_id to $user2_id), and vice versa. The is_open field of the chats is set to 1 if the authenticated user is an admin, indicating that the chat is open.
8. The function then checks if the sender is a regular user ($sender->user_type == "user") and if the sender's chat ($sender_chat) is not open ($sender_chat->is_open == 0). If these conditions are met, it returns a response with an error message and a status code of 401 (Unauthorized).

9. If the sender is authorized, the function creates a new message using the Message model. It sets the message_from and message_to fields to the respective user IDs, content to the message content, and issued_date and issued_time to the current date and time plus one hour.
10. Finally, the function returns a response with a success message, a status code of 200, and a JSON payload containing the state, message, and data fields. The data field includes the ID of the newly created message ($msg->id).

Request Content:

```
{
    "data":{
        "content":"hello ahmed",
        "message_to":63,
        "isFirst":true
        }
}
```

Or

```
{
    "data":{
        "content":"it's good",
        "message_to":63
        }
}
```

Response:

```
{
    "state": "good, ok",
    "message": "your data added successfully",
    "data": {
        "message_id": 136
    }
}
```

8.9.5.2) Get chats

http://127.0.0.1:8000/api/get/chat

Functionality: The get_chat() function retrieves the chats for the authenticated user.

Type: get

Authentications: All users after login

Explanation:

1- At first, we check if the user is authenticated or not, in case of unauthenticated we return status 401.

2- Get chats from chat table for the authenticated user then return status 200.

Function Logic

1. It checks if the user is authenticated using Auth::user(). If the user is not authenticated, it returns an error response with a state of "not authorized to access," a message of "cannot access API resources," and data indicating that the user is not authorized.
2. If the user is authenticated, it retrieves the authenticated user's ID ($user1_id).
3. It queries the Chat model to retrieve all chats where the authenticated user is the sender (chat_from).
4. The function then iterates through each chat and retrieves additional information about the chat participants.
5. If the participant is a doctor, it queries the Doctor model to retrieve the doctor's specialty and rate.
6. For each chat, it constructs a response data array containing information such as the chat participants' IDs, names, usernames, user types, profile image URLs, specialty (if applicable), and the chat's open status and rate.

7. The response data for each chat is added to the $data array.
8. After iterating through all chats, the function constructs a success response with a state of "good, ok," a message indicating the successful retrieval of data, and the chat data in the response data.
9. Finally, the function returns the success response with an HTTP status code of 200 (OK).

Response:


```
{
    "state": "good, ok",
    "message": "information retreived successfully",
    "data": [
        {
            "chat_from": 63,
            "chat_to": 64,
            "is_open": 1,
            "nick_name": "Admin wa",
            "user_name": "Waleed",
            "user_id": 64,
            "img_url": null,
            "user_type": "admin",
            "specialty": null,
            "rate": null
        },
        {
            "chat_from": 63,
            "chat_to": 82,
            "is_open": 1,
            "nick_name": "waleed 1",
            "user_name": "user1",
            "user_id": 82,
            "img_url": "https://static.vecteezy.com/system/resources/previews/004/788/616/original/
                icon-islamic-man-glyph-style-simple-illustration-free-vector.jpg",
            "user_type": "user",
            "specialty": null,
```

8.9.5.3) Get messages

http://127.0.0.1:8000/api/ get/messages?message_to=63

Functionality: The “get_msg()” function retrieves the messages between the authenticated user and a specified use.

Type: get

Authentications: All users after login

Explanation:

1- At first, we check if the user is authenticated or not, in case of unauthenticated we return status 401.

2- Get all messages from the message table for the authentiated user and a specified use then return status 200.

Function Logic

1- It checks if the user is authenticated using Auth::user(). If the user is not authenticated, it returns an error response with a state of "not authorized to access," a message of "cannot access API resources," and isUser set to 0 indicating that the user is not authorized.

2- If the user is authenticated, it retrieves the authenticated user's ID from Auth::user()->id. It also checks if the message_to parameter is provided in the request ($_GET['message_to']) to determine the ID of the specified user.

3- The function then queries the Message model to retrieve all messages between the authenticated user and the specified user. It searches for messages where the message_from is equal to the authenticated user's ID and the message_to is equal to the specified user's ID, or vice versa.

4- It iterates through each message and constructs the response data for each message. If the message is from the authenticated user to the specified user, it creates a response array with the relevant information.

   If the message is from the specified user to the authenticated user, it constructs the response array accordingly.

5- The constructed response data for each message is added to the $data array.

6- After iterating through all messages, the function constructs a success response with a state of "good, ok," a message indicating the successful retrieval of data, and the message data in the response data.

Response:


```
{
    "state": "good, ok",
    "message": "information retreived successfully",
    "data": [
        {
            "message_id": 187,
            "message_from": 64,
            "message_to": 63,
            "content": "hello",
            "issued_date": "2023-05-08",
            "issued_time": "1:47\u202fAM"
        },
        {
            "message_id": 188,
            "message_from": 63,
            "message_to": 64,
            "content": "are you good",
            "issued_date": "2023-05-08",
            "issued_time": "1:47\u202fAM"
        },
```

8.9.6) Admin APIs

8.9.6.1) Add report

http://127.0.0.1:8000/api/submit/report

Functionality: Enable users to add reports and then admins are able to see them.

Type: post

Authentications: All users after login

Explanation:

1- At first, we check if the user is authenticated or not, in case of unauthenticated we return status 401

2- Every user has only one record in the report table and has many records in report details table then, When the request is sent, we check:

I. If the user has a record in the report table, we add the data only in the report details table.

II. and if the user does not have a record in report table, we add data to both report and report details tables.

Function Logic

1. The function checks if the user is authenticated using Auth::user(). If not authenticated, it returns a response with a 401 status code and an error message indicating that the user is not authorized to access the API resources.
2. If the user is authenticated, it retrieves the user's ID from the authentication system.

3. It retrieves the data from the request's data parameter. If the issue and reportType fields are present, their values are assigned to variables $issue and $reportType. If they are not present, the variables are set to the string "none".
4. It checks if the user has a past report by querying the Report model based on the user ID. If no past report is found, it creates a new report and report details.
5. If a past report exists, it updates the number of reports for the past report and creates a new report detail.
6. Finally, it sets the response variables ($state, $message, $all_data) and returns a response with a 200 status code, indicating a successful operation.

Request content

```
{
    "data":{
        "reportFrom":1853,
        "issue":"i have problem in my connection",
        "reportType":"Internet Connection issue"
    }
}
```

Response:

Case 1:Unauthorized

```
{
    "state": "not authorized to access",
    "message": "cannot access to api resources",
    "data": {
        "isUser": 0
    }
}
```

Case 2:data added successfully

```
{
    "state": "good, ok",
    "message": "your data added successfully"
}
```

8.9.6.2) Get reports of all users

http://127.0.0.1:8000/api/get/reports

Functionality: enable admin only to get reports for all users

Type: get

Authentications: users after login, Admin.

Explanation:

1- At first, we check if the user is authenticated or not, in case of unauthenticated we return status 401.

2- Get all reports from report table for the authenticated user then return status 200.

Function Logic

1. The function begins by checking if the user is authenticated using Auth::user(). If the user is not authenticated, it returns a response with a 401 status code (Unauthorized) and an error message indicating that the user is not authorized to access the API resources.
2. If the user is authenticated, it retrieves the user object and user ID from the authentication system.
3. The function checks if the user is not an admin by verifying the user_type property of the user object. If the user is not an admin, it returns a response with a 403 status code (Forbidden) and an error message indicating that the user doesn't have permission to access these resources.
4. If the user is an admin, the function continues to retrieve all reports from the Report model using Report::all().

5. It initializes an empty array, $data, which will store the report data.

6. The function iterates over each report in the $all_report collection using a foreach loop.
7. For each report, it retrieves the user details associated with the report by querying the User model based on the report_from ID.
8. It prepares the data to be returned by creating an associative array named $pushed_data. The array includes information such as the report's report_from ID, the number of reports (num_reports), the latest report time (updated_at), and various user details like user_id, user_name, img_url, nick_name, user_type, bdate, and gender. The asset function is used to generate the URL for the user's image (img_url) by appending it to the 'storage' path.
9. The $pushed_data array is then pushed into the $data array using array_push to create a collection of report data.
10. Finally, the function sets the response variables ($state, $message, $data) and returns a response with a 200 status code (OK) and the report data.

Response:


```
{
    "state": "good, ok",
    "message": "information retreived successfully",
    "data": [
        {
            "report_from": 1853,
            "num_reports": 3,
            "latest_report_time": "2023-06-27T04:05:04.000Z",
            "user_id": 1853,
            "user_name": "waleed22",
            "nick_name": "dr. waleed 22",
            "user_type": "doctor",
            "bdate": "1999-01-05T22:00:00.000Z",
            "gender": "male",
            "img_url": null
        },
        {
            "report_from": 1852,
            "num_reports": 3,
            "latest_report_time": "2023-06-27T04:05:00.000Z",
            "user_id": 1852,
            "user_name": "user22",
            "nick_name": "user 22",
            "user_type": "user",
            "bdate": "1990-02-06T22:00:00.000Z",
            "gender": "male",
            "img_url": null
```

8.9.6.3) Get all reports details of a specific user

http://127.0.0.1:8000/api/ get/details/report?reportFrom=1

Functionality: enable admin to get all report details for a specific user

Type: get

Authentications: users after login, Admin

Explanation:

1- At first, we check if the user is authenticated or not, in case of unauthenticated we return status 401.

2- Get all report details from a specific user from the report details table.

Function Logic

1. The function starts by checking if the user is authenticated using Auth::user(). If the user is not authenticated, it returns a response with a 401 status code (Unauthorized) and an error message indicating that the user is not authorized to access the API resources.
2. If the user is authenticated, it retrieves the user object and user ID from the authentication system.
3. The function then checks if the user is not an admin by verifying the user_type property of the user object. If the user is not an admin, it returns a response with a 403 status code (Forbidden) and an error message indicating that the user doesn't have permission to access these resources.
4. Next, the function retrieves the reportFrom parameter from the query parameters using $_GET['reportFrom'] and assigns it to the variable $user_report_id.

5. It checks if the $user_report_id is 0, indicating that the report ID was not provided or is invalid. In such a case, it returns a response with a 200 status code (OK), a success message indicating that the user was not found, and an empty data array.
6. If the $user_report_id is not 0, the function proceeds to retrieve the user report based on the report ID using the User model.

7. It checks if the user report is empty, indicating that the user was not found. If the user report is empty, it returns a response with a 200 status code (OK), a success message indicating that the user was not found, and an empty data array.
8. The function continues by retrieving the main report associated with the user from the Report model.
9. It retrieves all report details for the user from the Report_detail model based on the user ID.
10. The function initializes an empty array, $data, which will store the report details.
11. It iterates over each report detail in the $all_details collection using a foreach loop.
12. For each report detail, it prepares the data to be returned by creating an associative array named $pushed_data. The array includes information such as the detail's ID (report_id), the user ID (user_id), the issue, the report type, and the time the detail was issued (issued_time).
13. The $pushed_data array is then pushed into the $data array using array_push to create a collection of report details.
14. Finally, the function sets the response variables ($state, $message, $data) and returns a response with a 200 status code (OK) and the report details.

Response:


```
{
    "state": "good, ok",
    "message": "information retreived successfully",
    "data": [
        {
            "report_id": 2,
            "user_id": 1853,
            "issue": "hellp !!",
            "report_type": "Internet Connection issue",
            "issued_time": "2023-06-27T04:04:47.000Z"
        },
        {
            "report_id": 3,
            "user_id": 1853,
            "issue": "wewe",
            "report_type": "Video call issue",
            "issued_time": "2023-06-27T04:04:51.000Z"
        },
        {
            "report_id": 6,
            "user_id": 1853,
            "issue": "were",
            "report_type": "Payment issue",
```


8.9.6.4) Change doctor state

http://127.0.0.1:8000/api/change/doctor

Functionality: Enable admin to verify, pend, and reject doctor.

Type: post

Authentications: All users after login

Explanation:

1- At first, we check if the user is authenticated or not, in case of unauthenticated we return status 401.

2- Update doctor state in doctor table as needed then return status 200.

Function Logic

1- The function first checks if the user is authenticated by using Auth::user(). If the user is not authenticated, it returns a response indicating that the user is not authorized to access the API resources.

2- The function then retrieves the data from the request input using $request->input('data'). This assumes that the request contains a parameter named 'data' that holds the necessary information.

3- The function extracts the values of doctorId and type from the retrieved data and assigns them to the variables $doctor_id and $type, respectively.

4- Next, the function queries the Doctor model to find the doctor with a matching user_id equal to $doctor_id. It assumes that there is a doctor model with a corresponding database table.

5- The function updates the type property of the retrieved doctor with the value of $type and saves the changes to the database using $doctor->save().

6- After updating the doctor's type, the function sets the response variables. The $state variable is set to 'good, ok', indicating a successful operation. The $message variable is set to 'information retrieved successfully' to provide a descriptive message. The $data variable is set to an array with a single key-value pair, where the key is 'isDone' and the value is true. This indicates that the verification process is complete.

7- Finally, the function returns a response with a 200 status code (OK), containing the response variables ($state, $message, $data) in a compact format.

Request Content

```
{
    "data":{
        "doctorId":63,
        "type":"verify"
    }
}
```

Or

```
{
    "data":{
        "doctorId":63,
        "type":"reject"
    }
}
```

Or

```
{
    "data":{
        "doctorId":63,
        "type":"pend"
    }
}
```

Response

```
{
    "state": "good, ok",
    "message": "your data added successfully",
    "data": {
        "isDone": true
    }
}
```

8.9.6.5) Change user

http://127.0.0.1:8000/api/change/user

Functionality: Enable admin to delete and restrict users.

Type: post

Authentications: users after login and admin

Explanation:

1- At first, we check if the user is authenticated or not, in case of unauthenticated we return status 401.

2- Delete the user from user table as needed or restrict chat.

Function Logic

1- The function first checks if the user is authenticated using Auth::user(). If the user is not authenticated, it returns a response indicating that the user is not authorized to access the API resources.

2- The function retrieves the data from the request input using $request->input('data').

3- The function checks if the userId and type keys are present in the retrieved data using isset(). If both keys are present, it proceeds to handle user deletion.

4- If the type is 'delete', the function deletes the user from the database based on the userId using the User::where('id', '=', $user_id)->delete() statement.

5- After deleting the user or if the userId and type keys are not present, the function continues to check if the chat_from and chat_to keys are present in the data. If both keys are present and the type is 'restrict', it proceeds to update the chat entry.

6- If the chat entry is found based on the chat_from and chat_to values, the function updates the is_open property of the chat entry and saves the changes to the database.

7- Finally, the function sets the response variables ($state, $message, $data) and returns a response with a 200 status code (OK) and the response variables in a compact format.

Request Content

```
{
    "data":{
        "doctorId":63,
        "type":"verify"
    }
}
```

Or

```
{
    "data":{
        "chat_from":24,
        "chat_to":24,
        "is_open":true,
        "type":"restrict"
    }
}
```

Response

```
{
    "state": "good, ok",
    "message": "your data added successfully",
    "data": {
        "isDone": true
    }
}
```

8.9.6.6) Get all users

http://127.0.0.1:8000/api/users

Functionality: Enable admin to get all users.

Type: get

Authentications: users after login and admin

Explanation:

1- At first, we check if the user is authenticated or not, in case of unauthenticated we return status 401.

2- Get all doctors from user and doctors' table

Function Logic

1- The function initializes an empty array $data to store the user data.

2- It retrieves all users from the User model using User::all().

3- If there are users in the database (the $users collection is not empty), the function enters a loop to process each user.

4- Inside the loop, it checks the user type using $user->user_type. If the user is a doctor, it retrieves additional doctor-specific details using the Doctor and Clinic_detail models.

5- For doctors, it constructs an array $userData containing the user information along with doctor-specific details.

6- For non-doctors, it constructs an array $userData containing the user information without the doctor-specific details.

7- The constructed $userData array is then appended to the $data array using array_push($data, $userData).

8- Once all users are processed, the function sets the response variables $state and $message to indicate success.

9- Finally, it returns a response with a 200 status code (OK) along with the response variables ($state, $message, $data) in a compact format.

Response


```
{
    "state": "good, ok",
    "message": "information retreived successfully",
    "data": [
        {
            "user_id": 4,
            "nick_name": "user1",
            "user_type": "user",
            "bdate": "2000-05-29",
            "gender": "male",
            "prefix": "20",
            "pnumber": "01149071132",
            "email": "ahmed.essameldin.essa@gmail.com",
            "province": null,
            "city": "Cairo",
            "street": "Cairo,Egypt, dar elsalam",
            "img_url": "http://127.0.0.1:8000/storage/images/1687830058_649a3e2acc729.jpg",
            "age": 23,
            "img_urls": [
                {
                    "img_url": "http://127.0.0.1:8000/storage/images/1687830058_649a3e2acc729.jpg"
                }
            ]
        },
        {
            "user_id": 5,
            "nick_name": "user2",
```

8.10 Socket part

8.10.1 Introduction

A web Socket is a protocol that provides full-duplex(multiway) communication i.e allows communication in both directions simultaneously. It is a modern web technology in which there is a continuous connection between the user's browser(client) and the server. In this type of communication, between the web server and the web browser, both can send messages to each other at any point in time. Traditionally on the web, we had a request/response format where a user sends an HTTP request, and the server responds to that. This is still applicable in most cases, especially those using RESTful API. But a need was felt for the server to also communicate with the client, without getting polled (or requested) by the client. The server should be able to send information to the client or the browser. This is where the Web Socket comes into the picture.

8.10.3 Why do we use it

1- providing live chatting is Powerful feature and required for our application for real-time communication.

2- Handle all interactions in real-time like adding appointments appears directly to active users once it's added or even booking appointments directly removed from available appointments so it' handle many challenges and conflictions.

3- Users can ask questions and directly get their answers without refreshing their page or degrade performance of application with set Interval to be UpToDate.

Socket can deal with all these challenges and struggles, providing real-time communications between all users.

8.10.4 How its programmed in Node.js

In order to make use of the Socket in NodeJS, we first need to install a dependency that is socket.io. We can simply install it by running the below command in terminal and then add this dependency to your server-side JavaScript file also install an express module which is basically required for server-side application.

npm install socket.io --save

npm install express --save

```
var { Server } = require("socket.io");
const server = http.createServer(app);
const io = new Server(server, {
  cors: {
    origin: true,
    methods: ["GET", "POST"],
  },
});
io.on("connection", (socket) => {/* required code */}
```

8.10.5 Socket Implementation details

Inside the required code we'll implement the following:

1- Handling chat using sockets:

- Each communicating pair of users must join a common channel to communicate using their messages.

```
  socket.on("join_chat", (data) => {
    socket.join(data);
  });
```

- Sending messages by firing event through joined channel only to ensure that data only transferred between two participants.

```
  socket.on("send_message", (data) => {
    socket.to(data?.chatId).emit("recieve_message", data);
  });
```

2- Handling interaction between user and doctor to book appointments

- Each user joins doctor channel when he actively opens his profile or doctor card, so all users keep tracking all appointment states UpToDate.

```
socket.on("join_doctor", (data) => {
   socket.join(data);
 });
```

- Once any update occurred event fired with updated appointment details

```
socket.on("update_appointments", (data) => {
   socket.to(data.doctorId).emit("update_slots", data);
 });
```

3- Handling posts and comments using sockets:

- Because posts are public overall system, so we don't think about joining any channel rather we'll firing a broadcast event.

```
 socket.on("send_post", (data) => {
   socket.broadcast.emit("recieve_post", data);
 });
```

- Same way for sending comments as a broadcast but we'll emit event to receive comment for specific post as illustrated below for more efficient and organized way.

```
socket.on("send_comment", (data) => {
   socket.broadcast.emit(`recieve_comment_${data?.post_id}`, data);
 });
```

9) Frontend Management

React.js

9.1) Introduction

React is an open-source JavaScript library that has gained immense popularity for building dynamic and interactive web applications. Developed and maintained by Facebook, React provides developers with a powerful toolkit for creating user interfaces with reusable components. Its component-based architecture and virtual DOM (Document Object Model) approach enable efficient rendering and seamless updates, making React an ideal choice for building modern, responsive, and high-performance web applications.

One of the key features of React is its declarative nature. Instead of manipulating the DOM directly, developers can describe how the UI should look at any given time, and React takes care of efficiently updating the actual DOM to match the desired state. This approach simplifies the process of building complex UIs, as developers can focus on composing reusable components and let React handle the rendering logic.

Another significant advantage of React is its component reusability. React components are self-contained and encapsulate both the UI and the associated behavior. This modular approach allows developers to create a library of reusable components, which can be easily composed to build complex user interfaces. Moreover, the React community actively shares and maintains a vast ecosystem of reusable components, known as "React libraries" or "React UI frameworks," which further accelerates development and encourages best practices.

React also embraces a unidirectional data flow pattern. This means that data flows in a single direction, from parent components to child components. By enforcing this pattern, React makes it easier to understand how data changes and propagates throughout the application, enhancing code predictability and maintainability. Additionally, React works seamlessly with other libraries or frameworks, allowing developers to leverage the benefits of React in combination with their preferred technologies.

Furthermore, React's virtual DOM enables efficient updates and optimizations. Instead of updating the entire DOM tree when a change occurs, React uses a virtual representation of the DOM, which it compares with the previous state to determine the minimal set of changes required. This reconciliation process minimizes expensive manipulations and leads to faster rendering, resulting in highly performant web applications.

React has also gained popularity due to its vibrant and supportive community. Countless resources, tutorials, and documentation are available, making it easy for developers to get started and seek assistance when needed. The community-driven nature of React ensures that the library

evolves rapidly, with regular updates and improvements, keeping up with the latest web development trends and best practices.

In conclusion, React has revolutionized the way web applications are built, offering developers a powerful, efficient, and flexible solution for creating modern user interfaces. Its component-based architecture, declarative nature, and virtual DOM approach contribute to improved development productivity and application performance. With React, developers can build highly interactive, scalable, and maintainable web applications, making it a go-to choose for both small-scale projects and large-scale enterprise applications.

9.2) Comparison (react vs angular)

React, as a JavaScript library for building user interfaces, offers several distinct advantages that set it apart from other web application frameworks. Let's explore how React compares to some popular frameworks and highlight what makes React special.

1. Angular:
    - Angular is a comprehensive framework developed by Google, offering a complete solution for building web applications.
    - React, on the other hand, focuses primarily on the UI layer, allowing developers to integrate it seamlessly with other libraries or frameworks to create a complete application stack.
    - React's virtual DOM and efficient diffing algorithm contribute to its excellent rendering performance, particularly in large-scale applications where updates need to be optimized.
2. Vue.js:
    - Vue.js is a progressive JavaScript framework that aims to be approachable and versatile.
    - React and Vue.js share similarities in their component-based architecture and reactivity. Both allow developers to build reusable UI components.
    - React's virtual DOM and efficient diffing algorithm contribute to its excellent rendering performance, particularly in large-scale applications where updates need to be optimized.
3. Ember.js:
    - Ember.js is a full-featured framework that focuses on convention over configuration, providing developers with a structured and opinionated approach to building web applications.
    - React, on the other hand, is more flexible and allows developers to make decisions about how to structure their applications.
    - React's component-based architecture and modular approach make it easy to create reusable UI components, enabling faster development and encouraging code reusability.

What Makes React Special:

1. **Virtual DOM**: React's virtual DOM enables efficient updates by comparing the current state of the virtual DOM with the previous state and determining the minimal set of changes required. This approach reduces the number of manipulations needed in the actual DOM, resulting in faster rendering and improved performance.
2. **Component Reusability:** React's component-based architecture promotes reusability, allowing developers to build modular and self-contained UI components. These components can be easily composed and reused throughout the application, resulting in code that is more maintainable and scalable.
3. **Unidirectional Data Flow**: React enforces a unidirectional data flow pattern, making it easier to understand how data changes and propagates through the application. This pattern enhances code predictability and maintainability, reducing the chances of unexpected bugs and making it easier to debug and test the application.
4. **Vibrant Community**: React has a large and active community that contributes to its growth and development. The community provides extensive documentation, tutorials, and resources, making it easier for developers to get started and seek assistance when needed. Additionally, the community actively maintains a rich ecosystem of reusable components and libraries, further accelerating development and encouraging best practices.
5. **Integration Flexibility**: React can be easily integrated with other libraries or frameworks, allowing developers to leverage its benefits alongside their preferred technologies. This flexibility makes React suitable for both greenfield projects and incremental adoption in existing applications.

In summary, React stands out with its efficient rendering through the virtual DOM, component reusability, unidirectional data flow, vibrant community support, and flexibility in integration. These features make React a popular choice for building high-performance and interactive web applications, providing developers with a robust and efficient toolkit to create modern user interfaces.

9.3) Why do we use react? (Context to request APIs),

React's context feature provides a convenient way to manage and share state across components without passing props manually through every level of the component tree. When it comes to making API requests and managing their associated state, using React's context can offer several benefits:

1. Centralized State Management: With context, you can store the API request-related state, such as loading, error, and data, in a centralized location. This allows multiple components within the application to access and update this state without the need for prop drilling (passing props through intermediate components). As a result, the code becomes more organized, easier to maintain, and reduces boilerplate code.
2. Avoiding Prop Drilling: In a large application with deeply nested component hierarchies, passing props through every level to reach a specific component can become cumbersome and error-prone. React's context provides a solution by allowing you to define a context provider at a higher level in the component tree. The components that need access to the API-related state can simply consume the context without the need for explicit prop passing.
3. Flexibility and Reusability: By encapsulating the API request logic and state management within a context, you can create reusable components that can be used in different parts of your application. The components can consume the context and utilize the API data without needing to know the implementation details of the API request itself. This separation of concerns promotes code reusability and maintainability.
4. Efficient Updates and Rerenders: When an API request is made and its result is updated in the context state, the consuming components can react to the changes and rerender accordingly. React's diffing algorithm, along with the virtual DOM, efficiently updates only the necessary parts of the UI affected by the API response. This results in better performance as unnecessary rerenders are avoided.
5. Asynchronous Handling: Context can handle asynchronous operations, such as API requests, in a straightforward manner. You can trigger the API request from within the context and update the state accordingly when the response is received. Components consuming the context will be automatically notified of the changes, allowing them to display loading spinners, error messages, or the fetched data as needed.

However, it's important to note that using context for managing API requests and their state is just one of many approaches. Depending on the complexity of your application and specific requirements, you may also consider other state management solutions like Redux, MobX, or even React's built-in useState and useEffect hooks. It's crucial to evaluate the trade-offs and choose the approach that best suits your application's needs.

9.4) User Guide and interface

Objectives

1- To be Familiar with features and functionalities can be used in our application.

2- How to use our application and achieve your goal with different scenarios.

3- Good knowledge about UI of our application with Screenshots and sample output of our web app and explanations upon it.

4- A step-by-step description of how to use our application is also provided in this section.

Introduction

This demo illustrates various scenarios where placeholders use our clinic application to accomplish their tasks/goals utilizing services to reach their goals.

Goals

1- A Doctor has a clinic at any location, he wants to deploy his work and advertise his clinic by joining our application for scheduling appointments, better management and monitoring of his appointments.

9.4.1) Scenarios to reach doctor's goals:

1) Registration process (as a doctor)

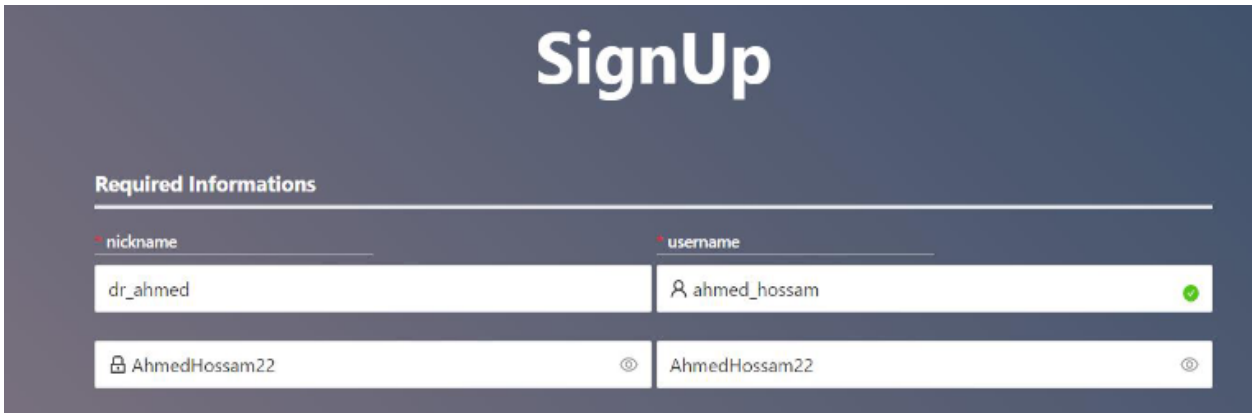


Figure 1 (account information)

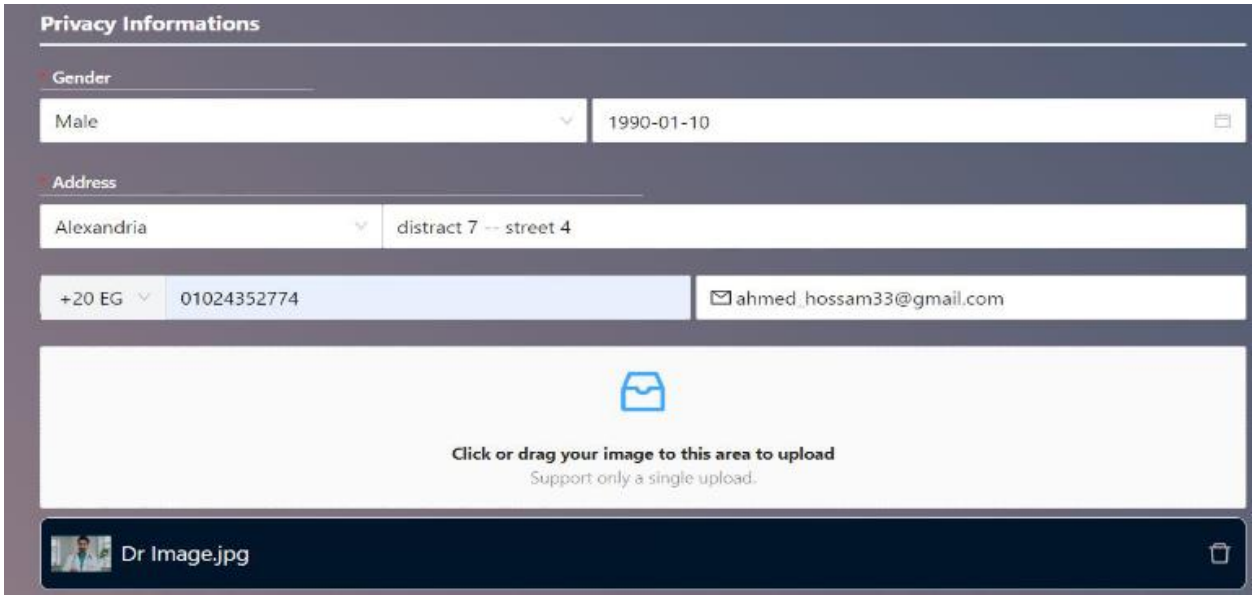


Figure 2 (additional information)

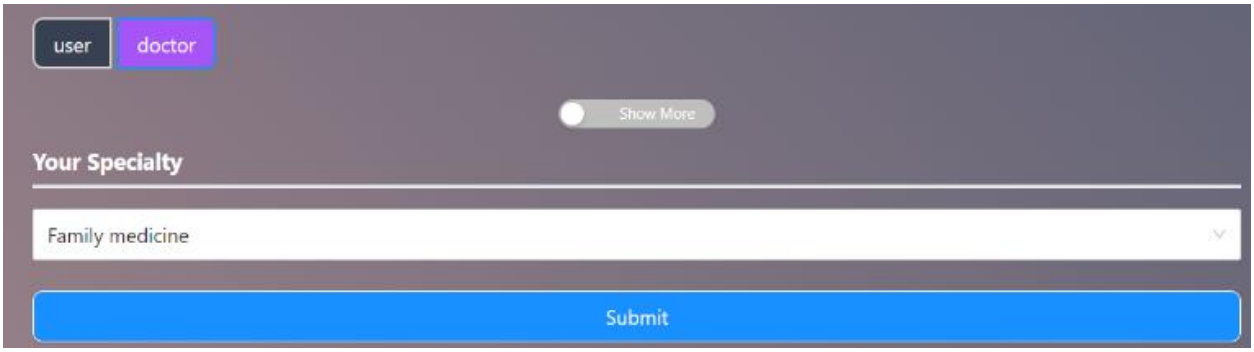
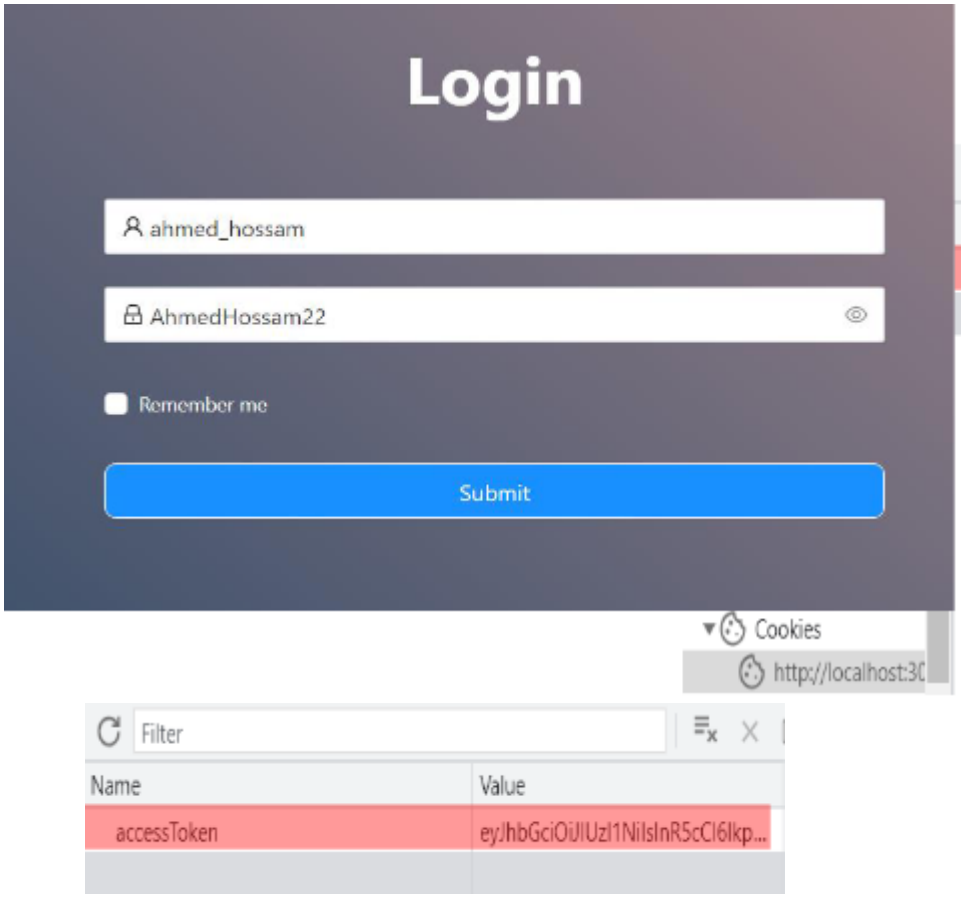


Figure 3 (doctor details)

Explanation: apply same steps for user but select only a user account.

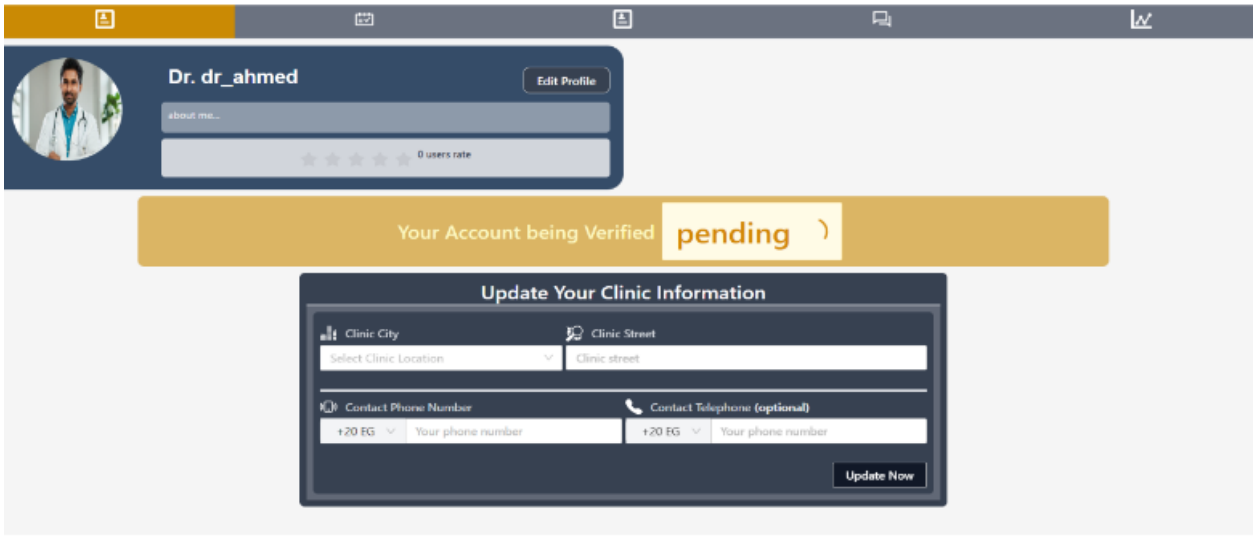
2) Login



Explanation: Authentication-token (access Token) generated once he login, so for subsequent requests that require authentication can be done using that token as user can be authenticated using JWT token.

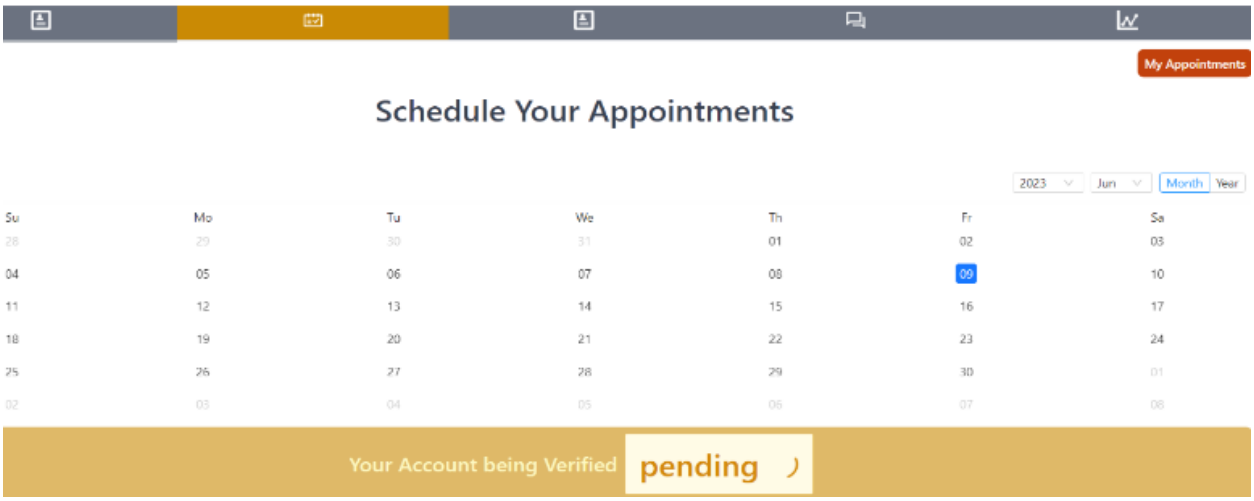
3) Doctor waiting for verification

i. Doctor profile



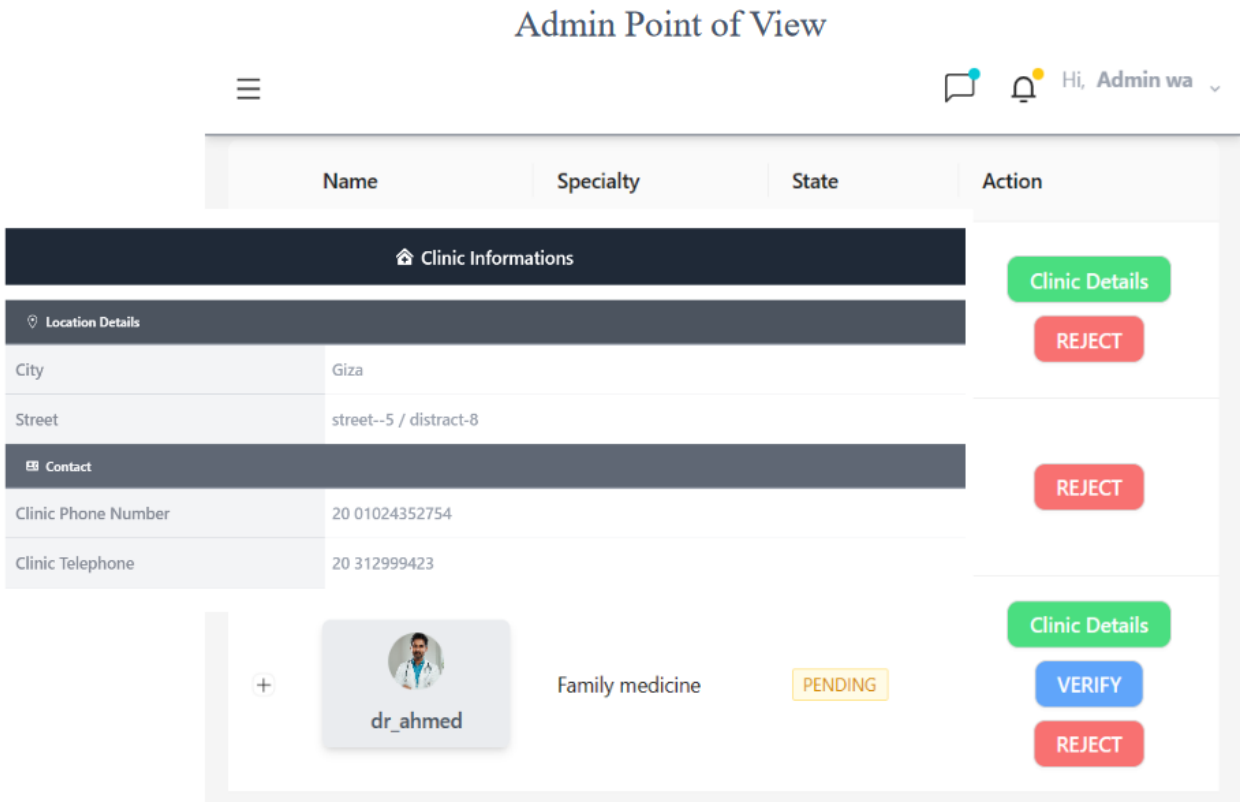
- Scheduling Appointments

ii. Scheduling Appointments



Explanation: The doctor suspended from some functionalities (managing appointments, showing up in doctors list) until he fills clinic information also, he must wait until administrator reviewing his information and verify him if he satisfies all rules and conditions.

4) Verifying Doctor

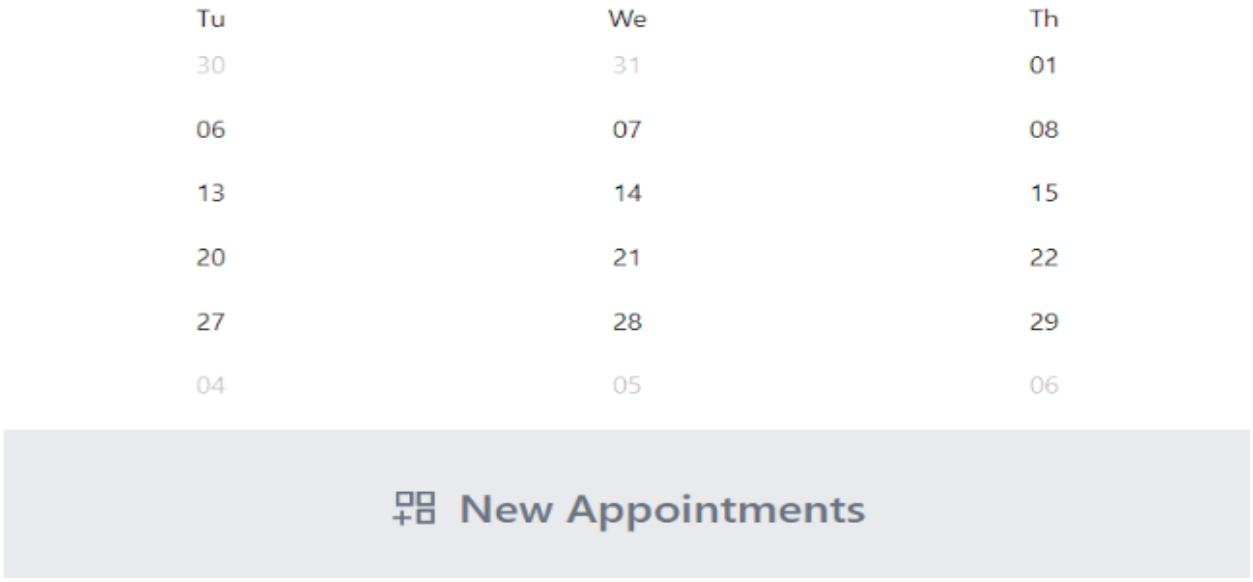


Explanation: Administrator checks doctor information to ensure that it meets all constraints. Then verify doctor if accepted.

i. Doctor verified.

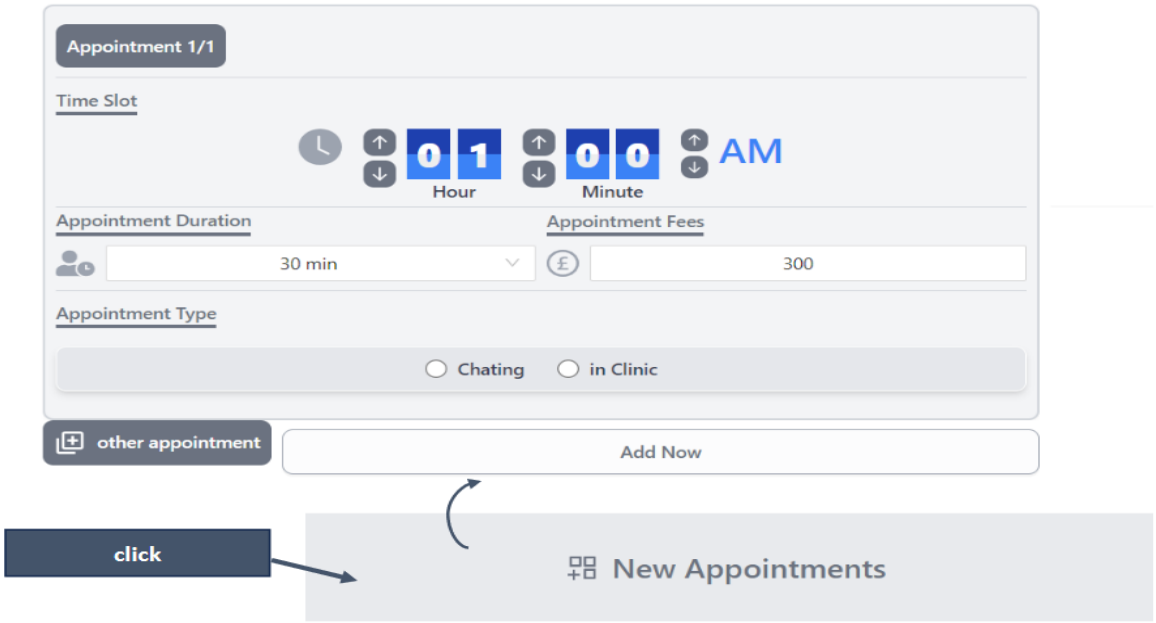
Explanation: as we notice doctor verified only using verify click from admin after revision of his information.

5) Doctor schedule, manage and monitoring his Appointments.



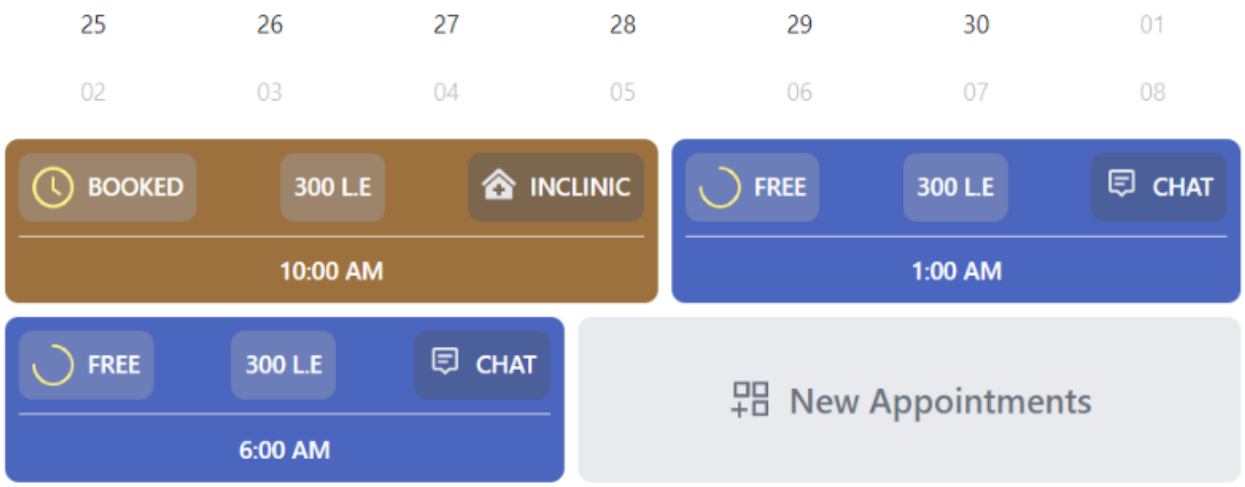
Explanation: with the new Appointments button, doctor can add new appointments.

ii. Schedule appointments



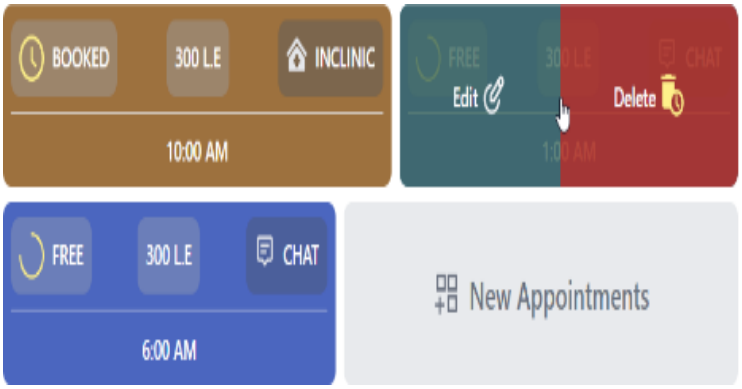
Explanation: appointment need to be filled with details, adding other appointment using other appointment button then click on add now to add it.

iii. View appointments



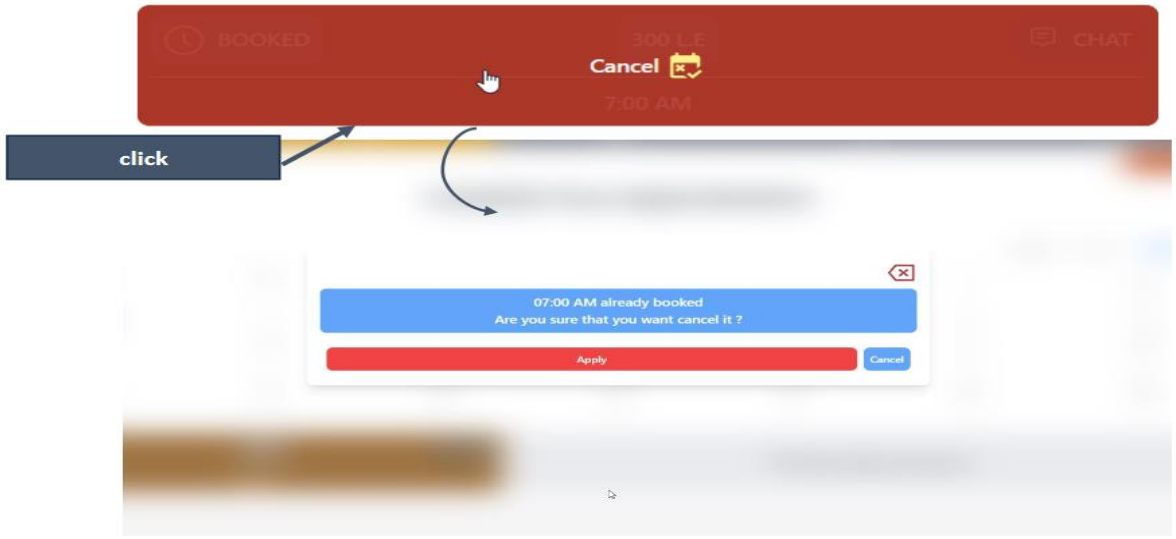
iv. Manage appointments (cancel, delete or edit).

- <u>delete or edit appointment.</u>



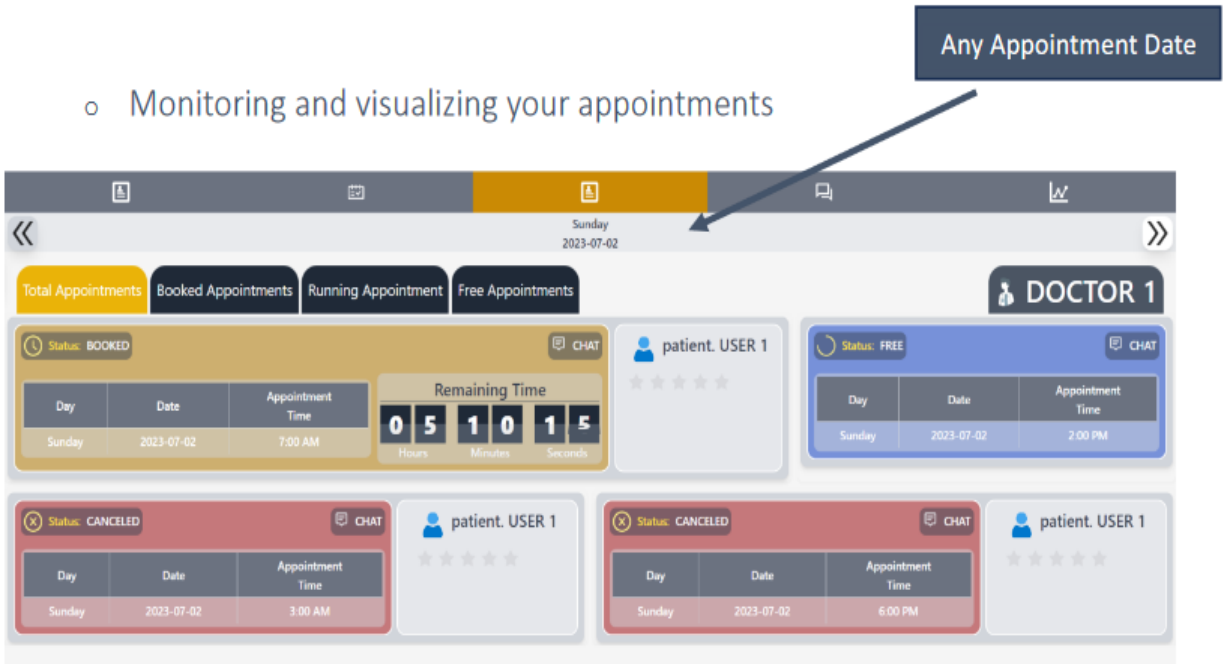
Explanation: remove or edit new appointments in a convenient way.

- cancel appointment.



Explanation: canceling any booked appointment from apply button after doctor/user click in cancel.

v. Monitoring and visualizing your appointments



Explanation: doctor can monitor his appointments for specific day in a clearly way with all details associated to it, also filter search by any appointment state (total, booked, Running, Free)

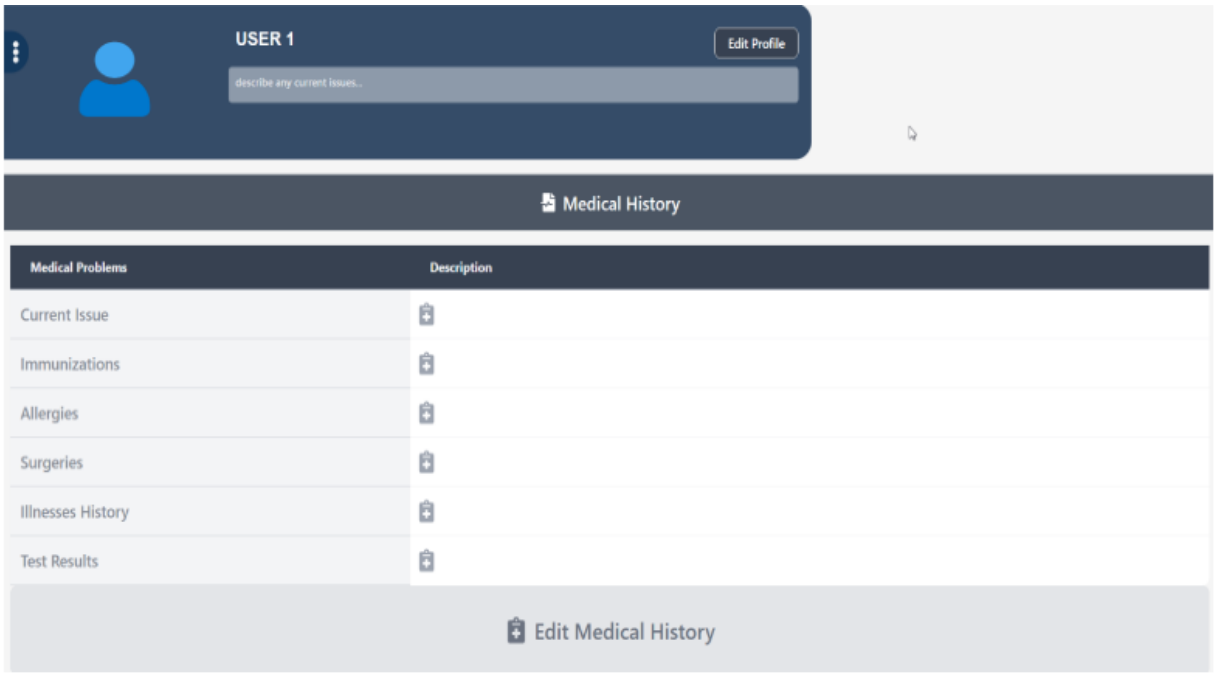
1- We are now cover some of doctor's goals let us dive into other goals and it is our patient's goals and that is why we develop our application to make patients (users) more convenient delivering to them enough care, communicating with their doctors with easy accessibility as we will see.

2- A Patient need to communicate with our doctors and reaching a right doctor for their conditions and interact with doctors over chatting, video call or offline meeting to get help or even for some cases require to ask our doctors simple questions about their conditions, store medical records to be useful for them and doctors to track or notice any diseases or medical issues, placing feedback to be helpful as reference for other patients to select a good doctor.

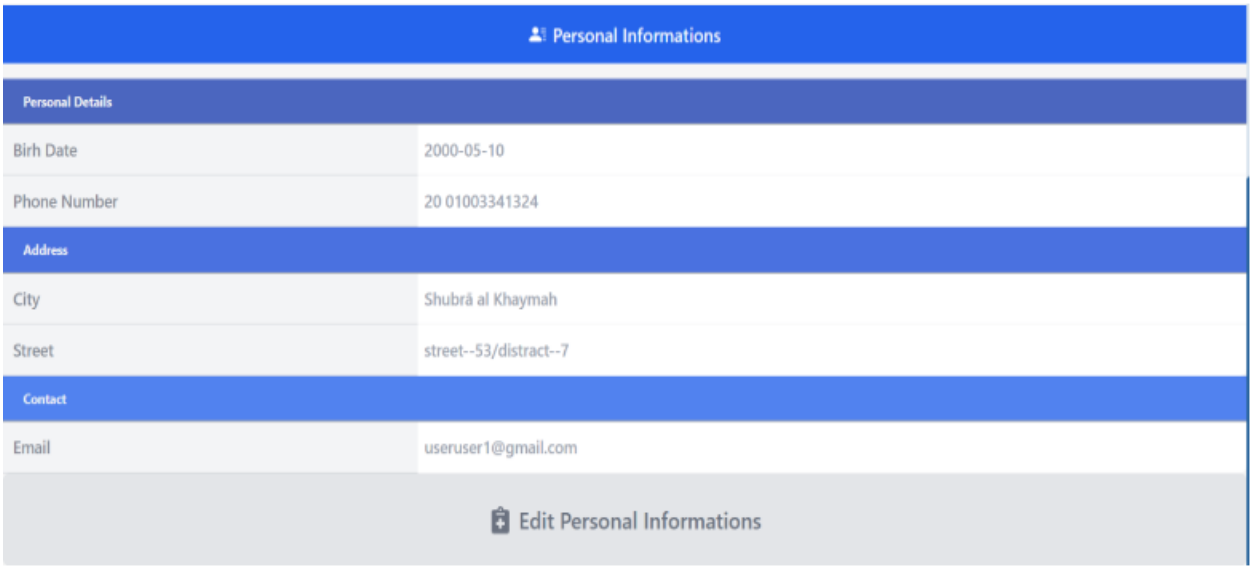
9.4.2) Scenarios to reach user's goals:

First, user repeats same process (1 and 2) as doctor to sign up and login.

1) Optionally manage his profile.



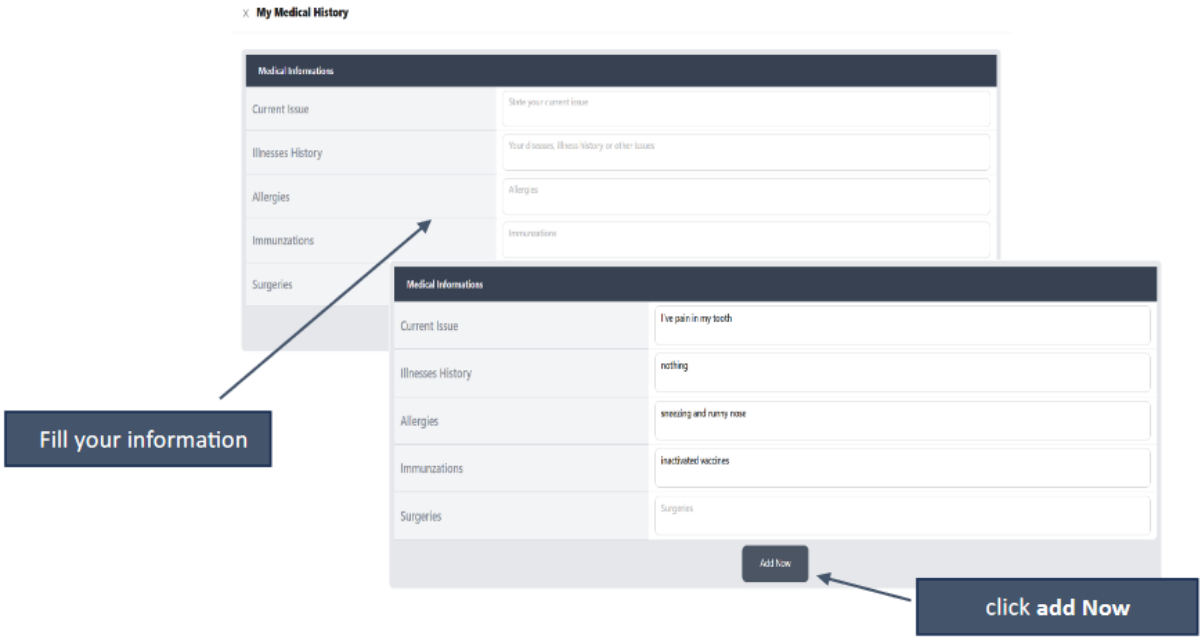
Explanation: patients store medical records to be useful for them and any doctors that they communicate with to understand their issues history, current issue, and any other medical conditions.



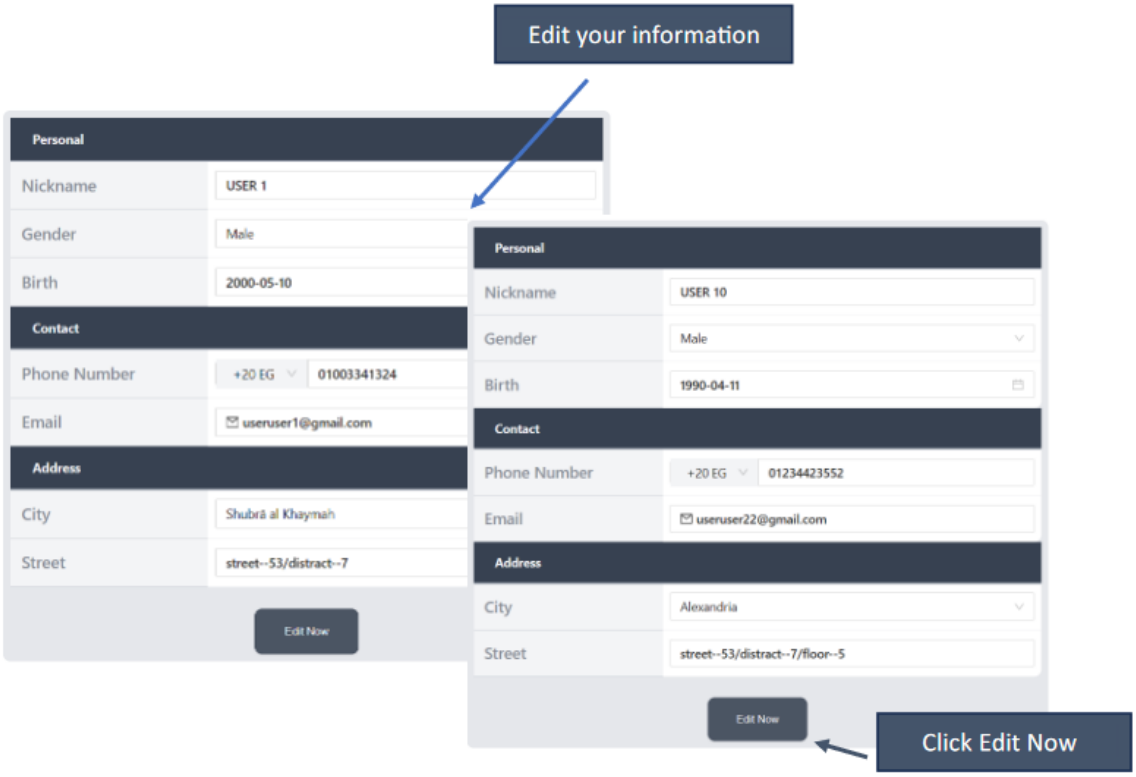
Explanation: patients can also maintain their personal information, useful for any contact so doctors can reach them quickly.

To update any information (medical or personal) just click edit Personal/Medical information.

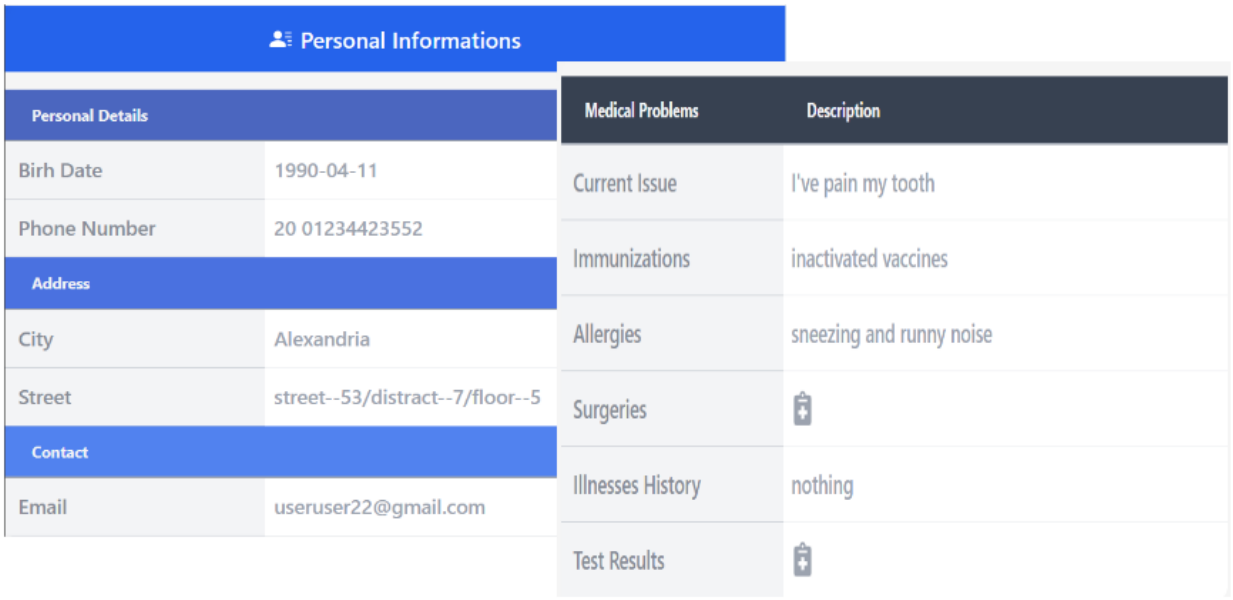
- Edit Medical information.



- Edit Personal information.

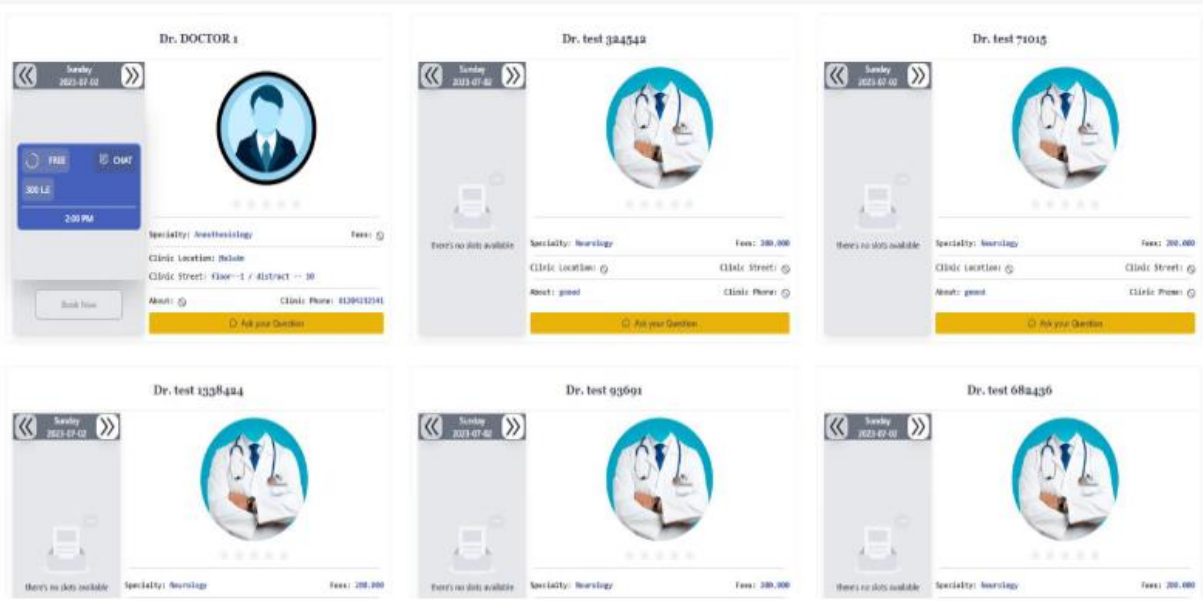


- View updated information.



3) Finding a doctor

- View Doctor Cards



Explanation: patients can find any doctor with associated information and scheduled appointments with its details too, but all these records must be ordered by a search filter make it easier and more efficient for patients to find suitable doctor.

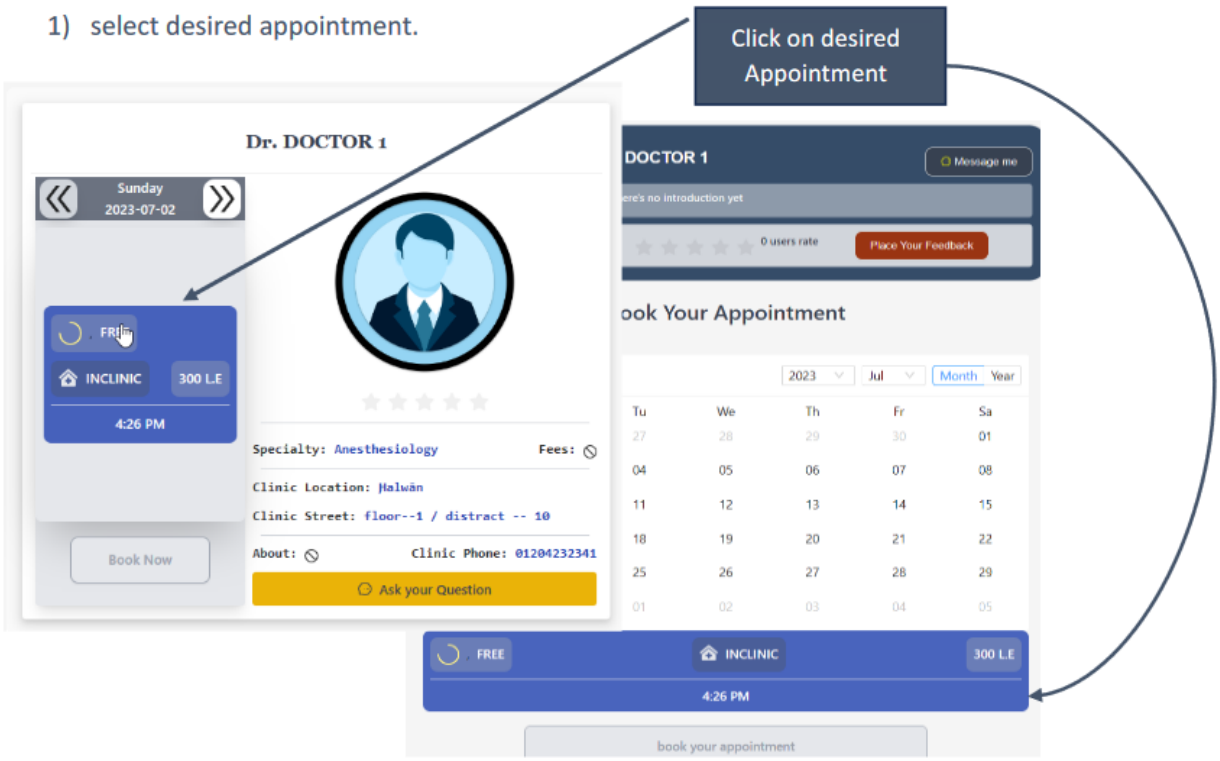
- Search Filter for all doctors

Explanation: patients can find any doctor with their specialty according to their condition, or by doctor name if they already know their doctors or clinic location and choose nearby location if they want offline meeting.

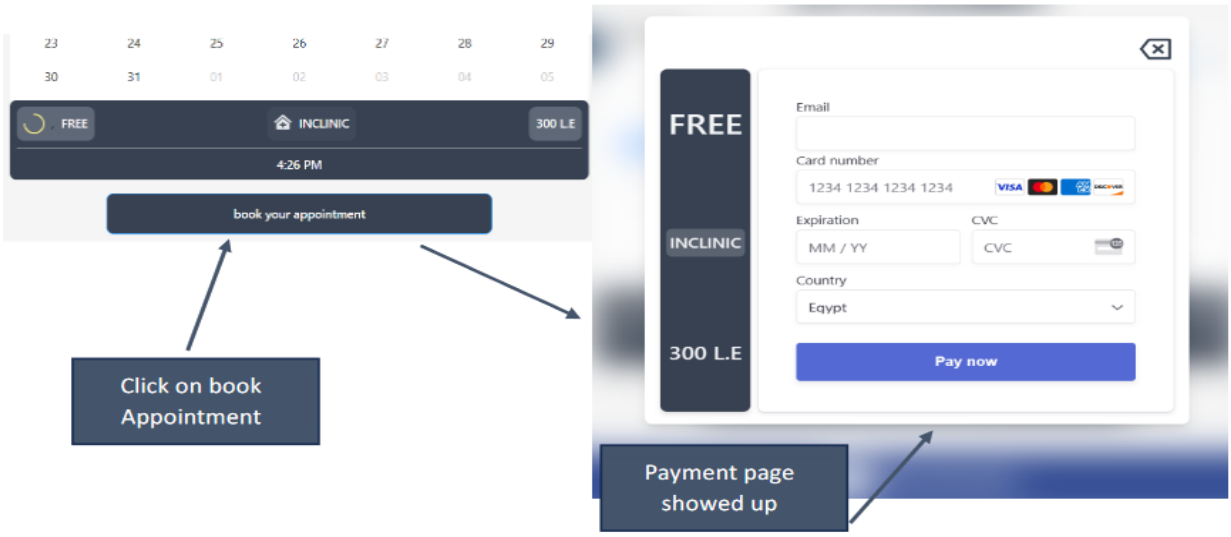
4) Booking Appointment

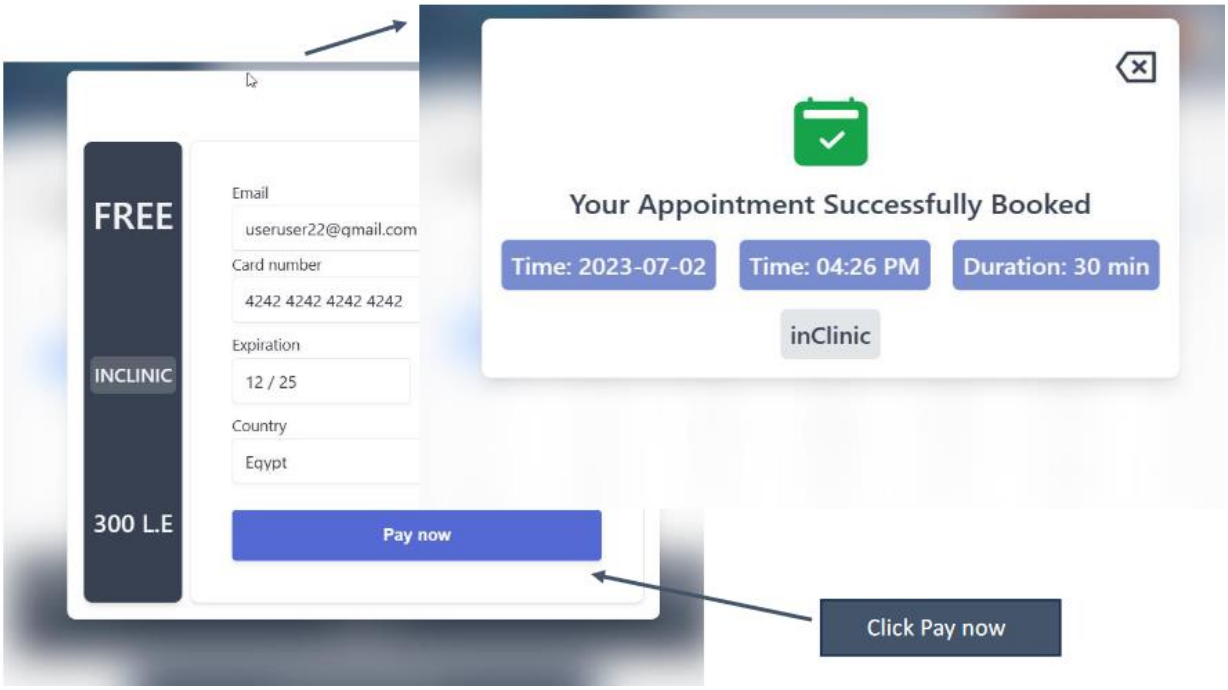
Assume patient want to book appointment for a live chatting with specialized doctor, scenario will be like following:

1) select desired appointment.



2) Book appointment



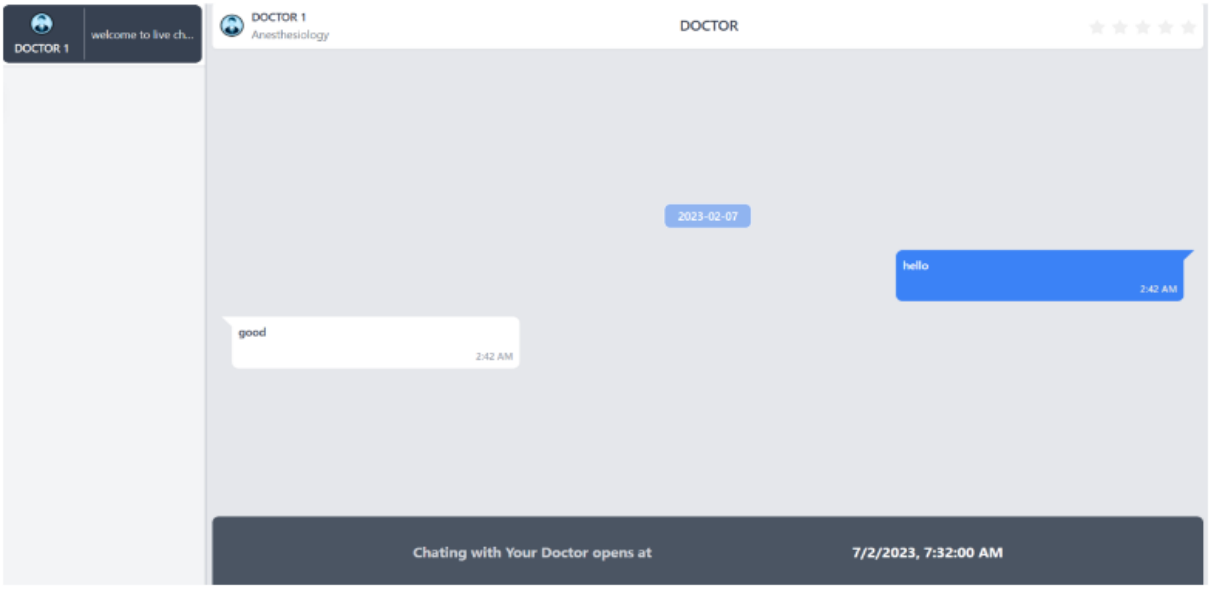


Explanation: select desired appointment according to your requirements (date, appointment type, fees, etc. …) then click book appointment will appear payment page after filled up click on Pay now after processing will navigate to a successful appointment where you can then check your appointments page.

3) Manage and visualize Your Appointment

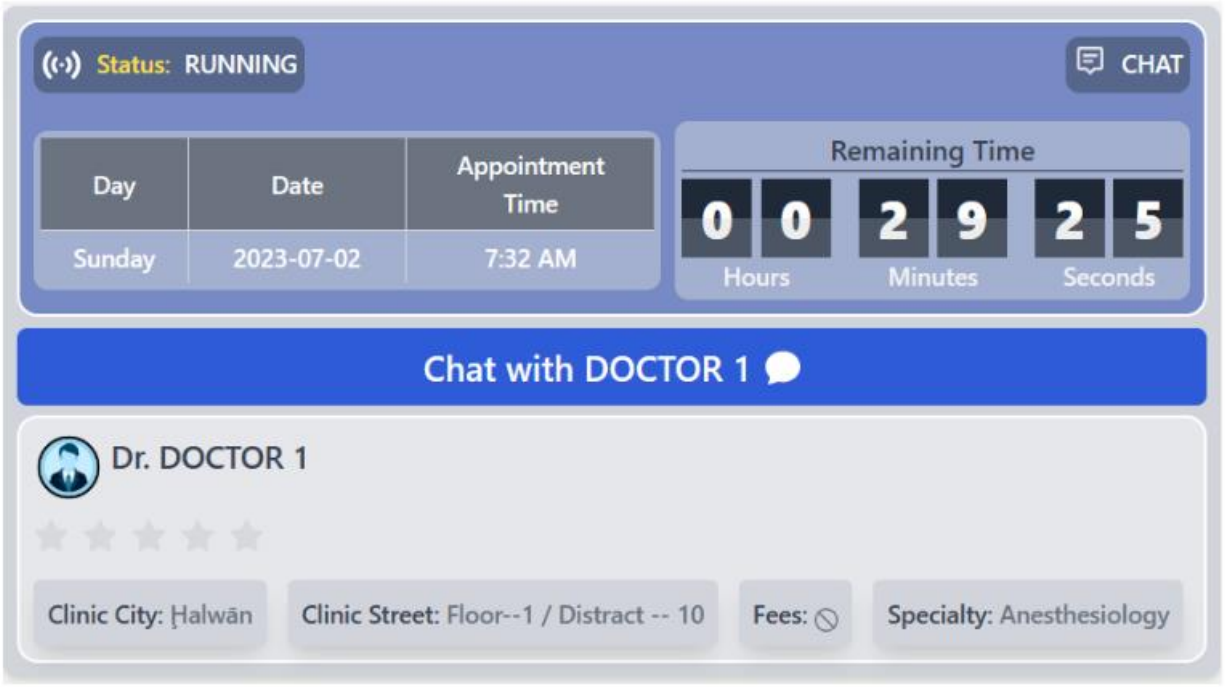
4) Starting Your Appointment

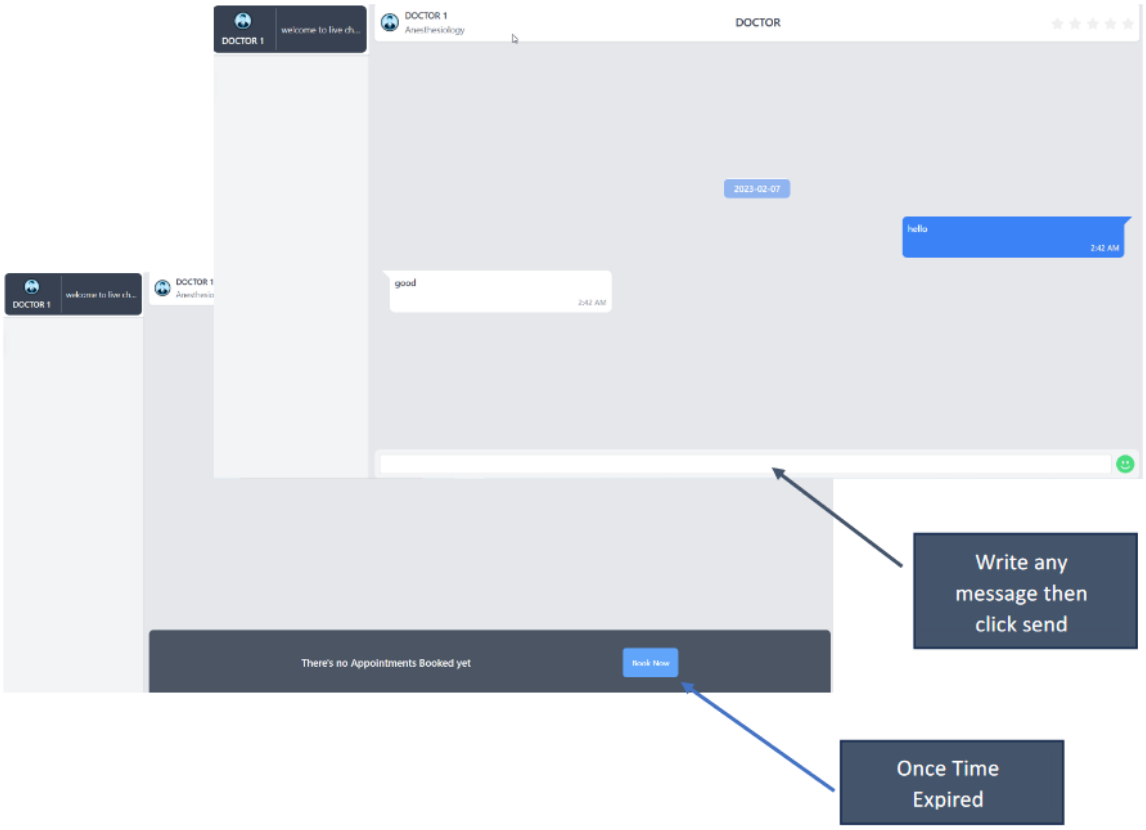
- Waiting time for appointment



Explanation: display time for your next appointment time as schedule date and time provided inside that chat.

- Running appointment



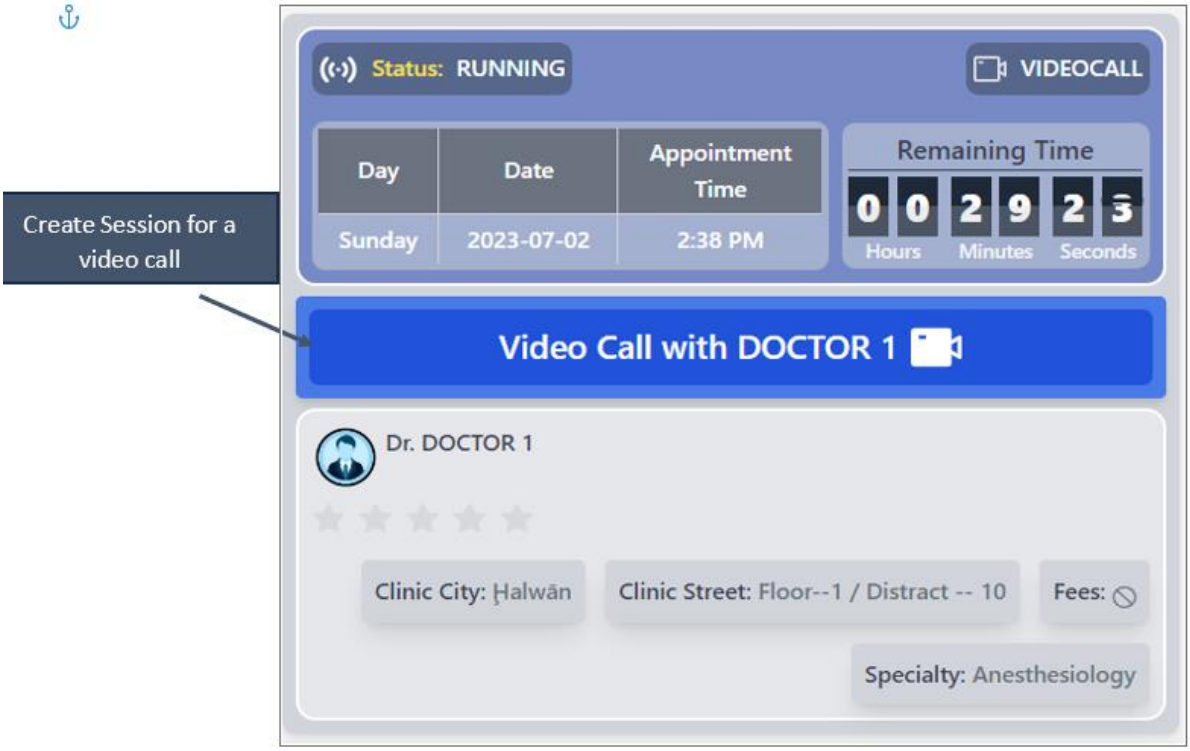


Explanation: after appointment running along with appointment duration patient will be able to interact with that appointment (live chatting) once appointment duration end, patient cannot send any message and wait for next appointment or book new one.

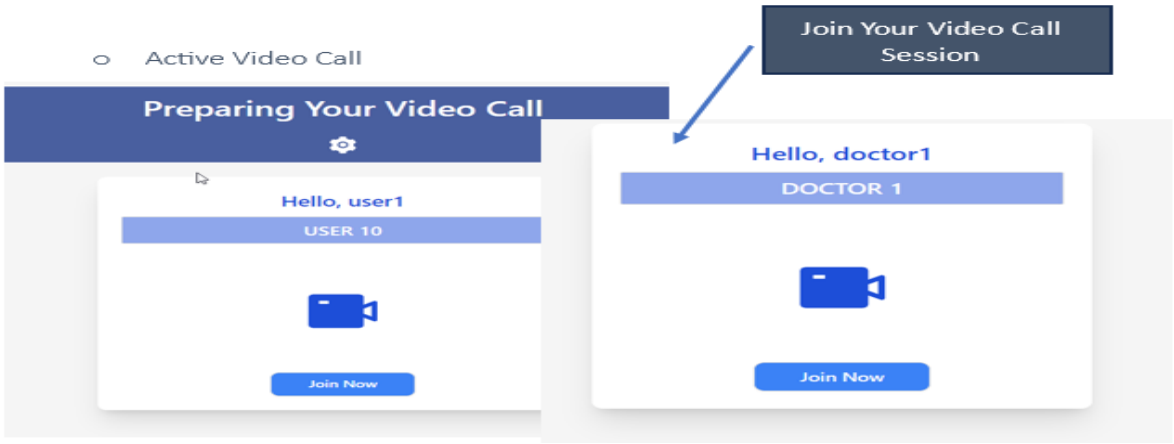
5) Video Call Appointment

1) Select desired Appointment (as illustrated in 4.1)
2) Book your Appointment (as illustrated in 4.2)
3) Monitoring and tracking your appointment (as illustrated in 4.3)
4) Starting Your Video Call

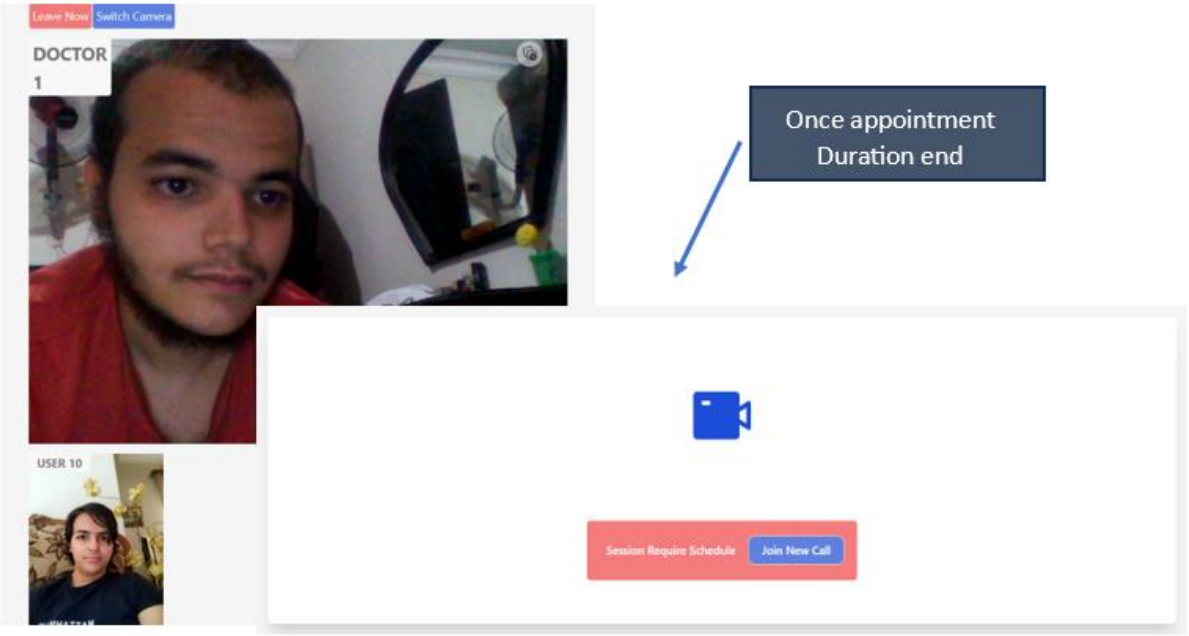
➢ Running appointment



➢ Active Video Call



- Preparing Video Call

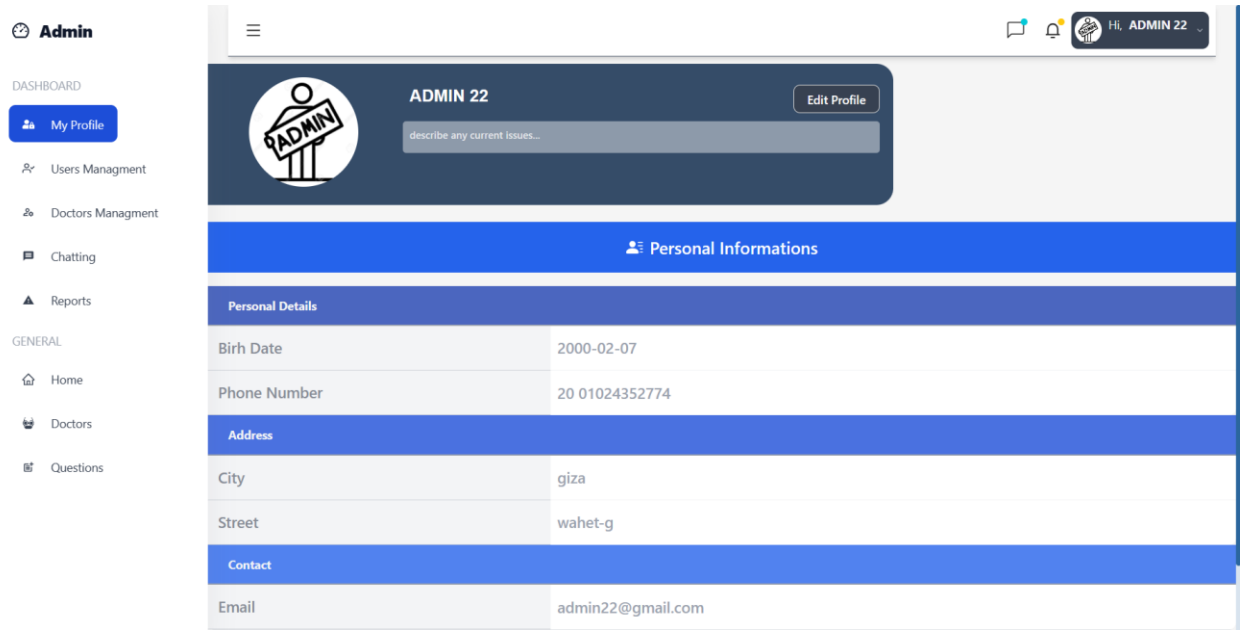


Covering patients' requirements is our main goal by delivering decent quality of communication through live chatting, video calls and appointment management according to their needs.

3- Enable Administration with an intuitive dashboard for efficient user, doctor management, system monitoring and supporting contact with users and receive reports from them for system analysis.

9.4.3) Scenarios to cover Admin goals:

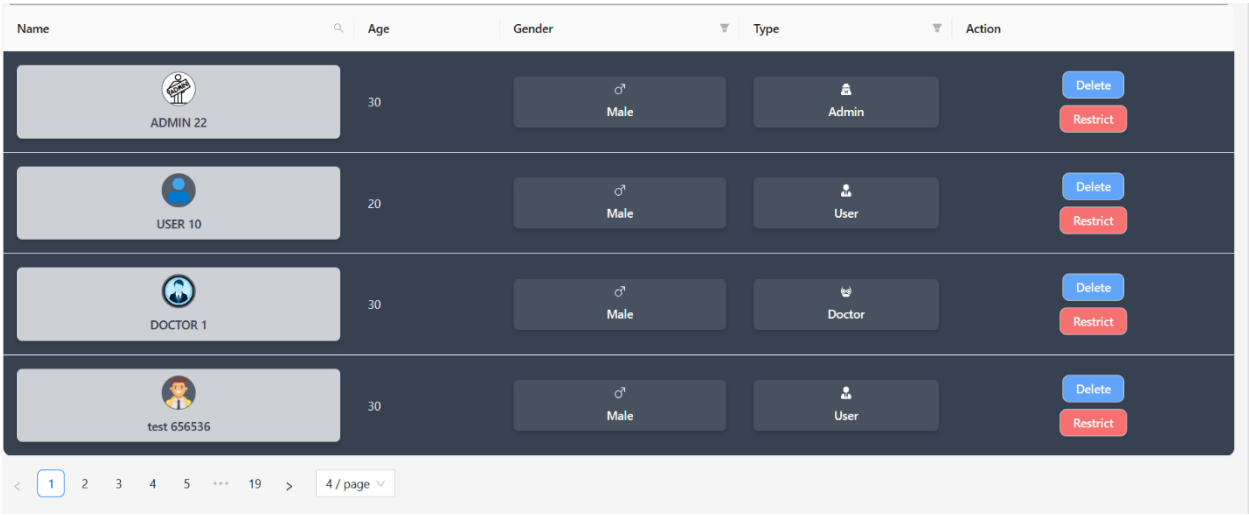
1) View of Admin Dashboard



Explanation: providing dashboard with ability to manage any user or doctor also you can receive reports for some analysis

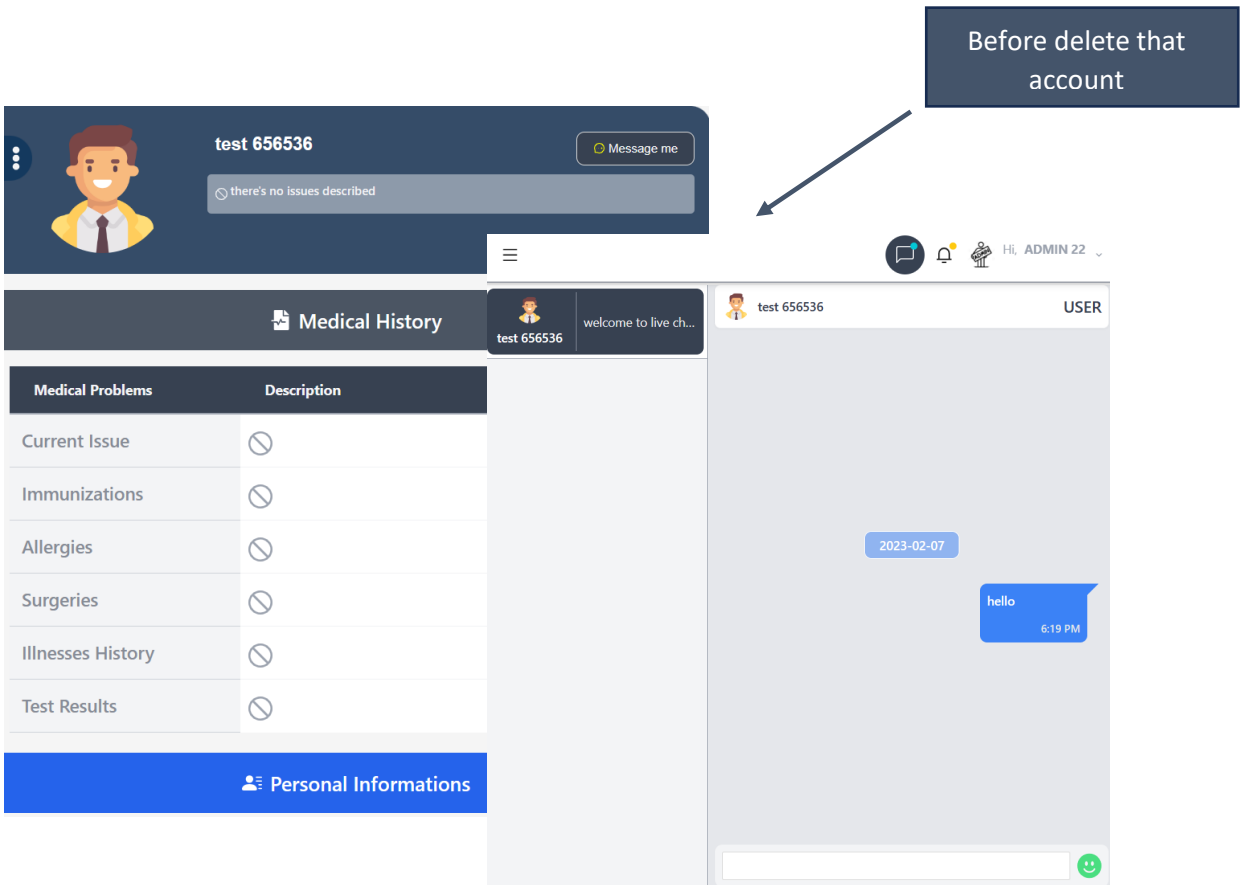
- **User management:** where you can delete user account or restrict permissions for some users (close or open chat)
- **doctor management:** where you can verify doctors according to their supported information, tracking their appointments and managing it if required.
- **Reports:** receive any problem for discovering what's cause of this problem or even find solution to it and provide communication with users have issues.

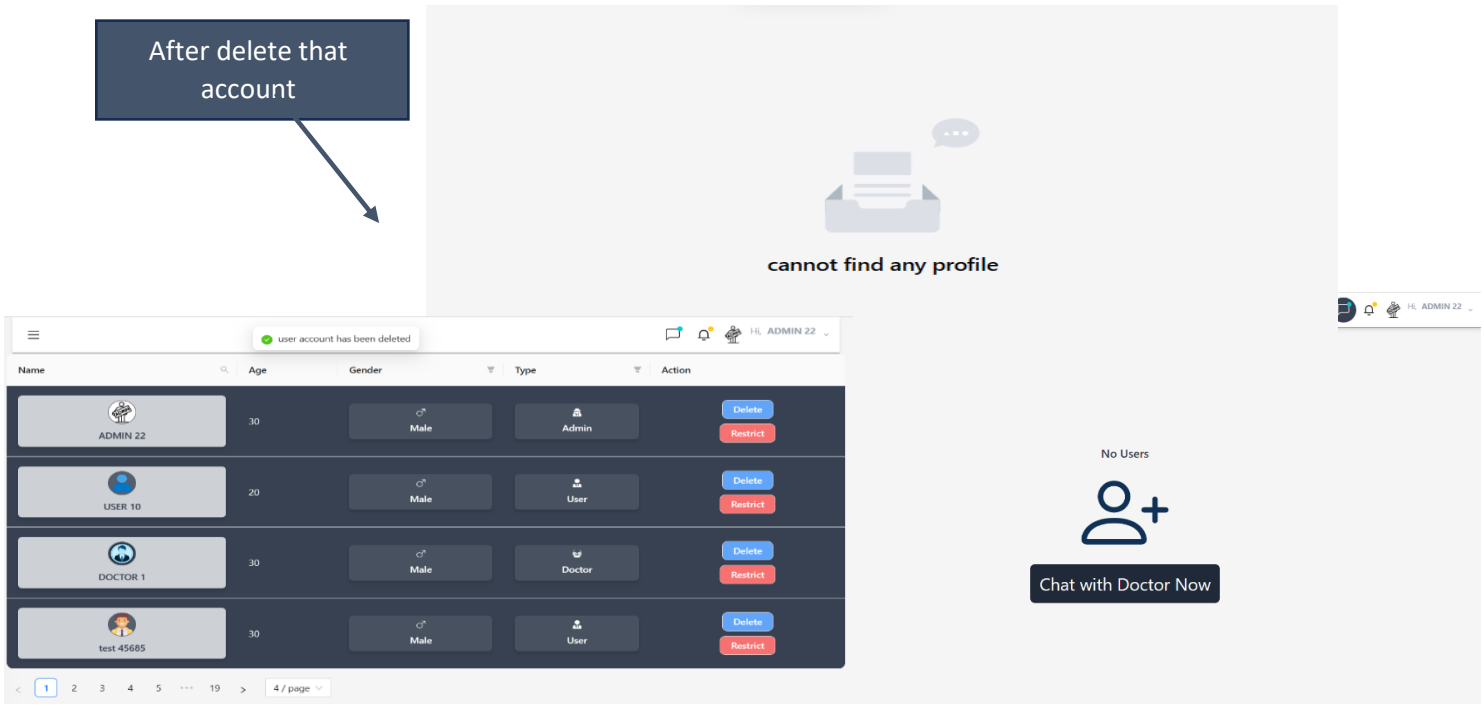
2) User Management



- **Delete user.**

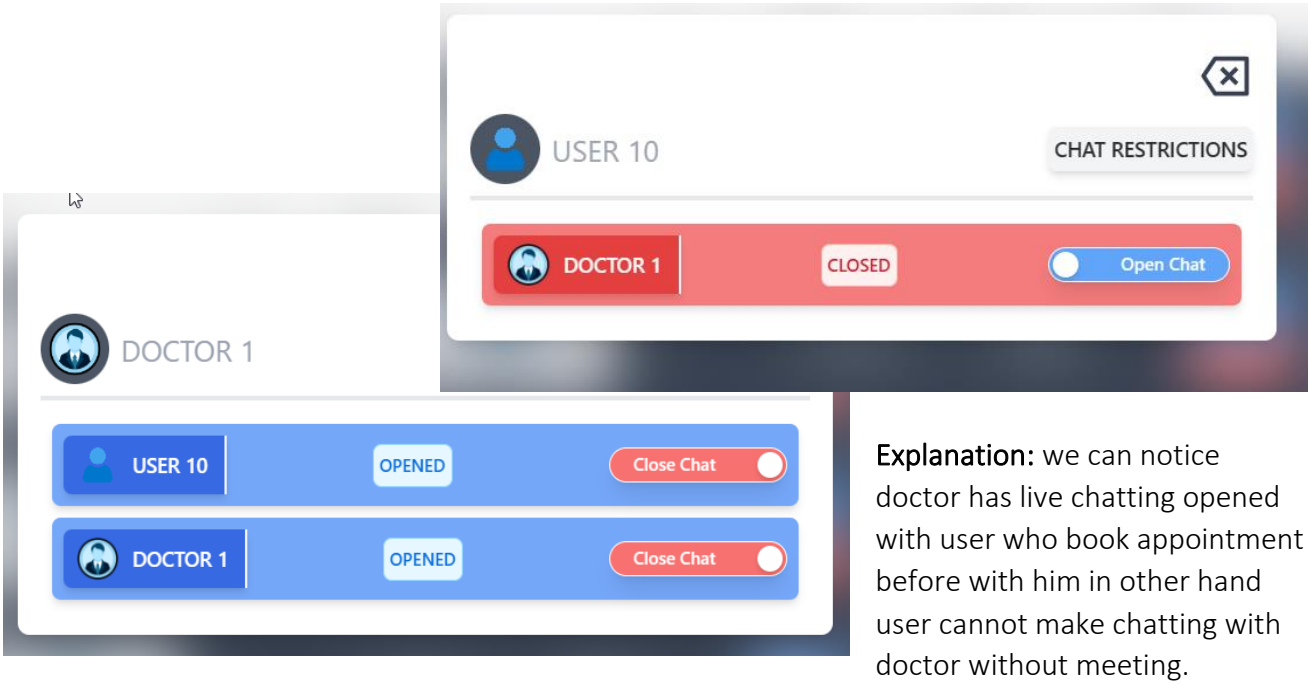
For example, let us delete **test 656536** account using delete button and visit hist profile before and after deleting his account.





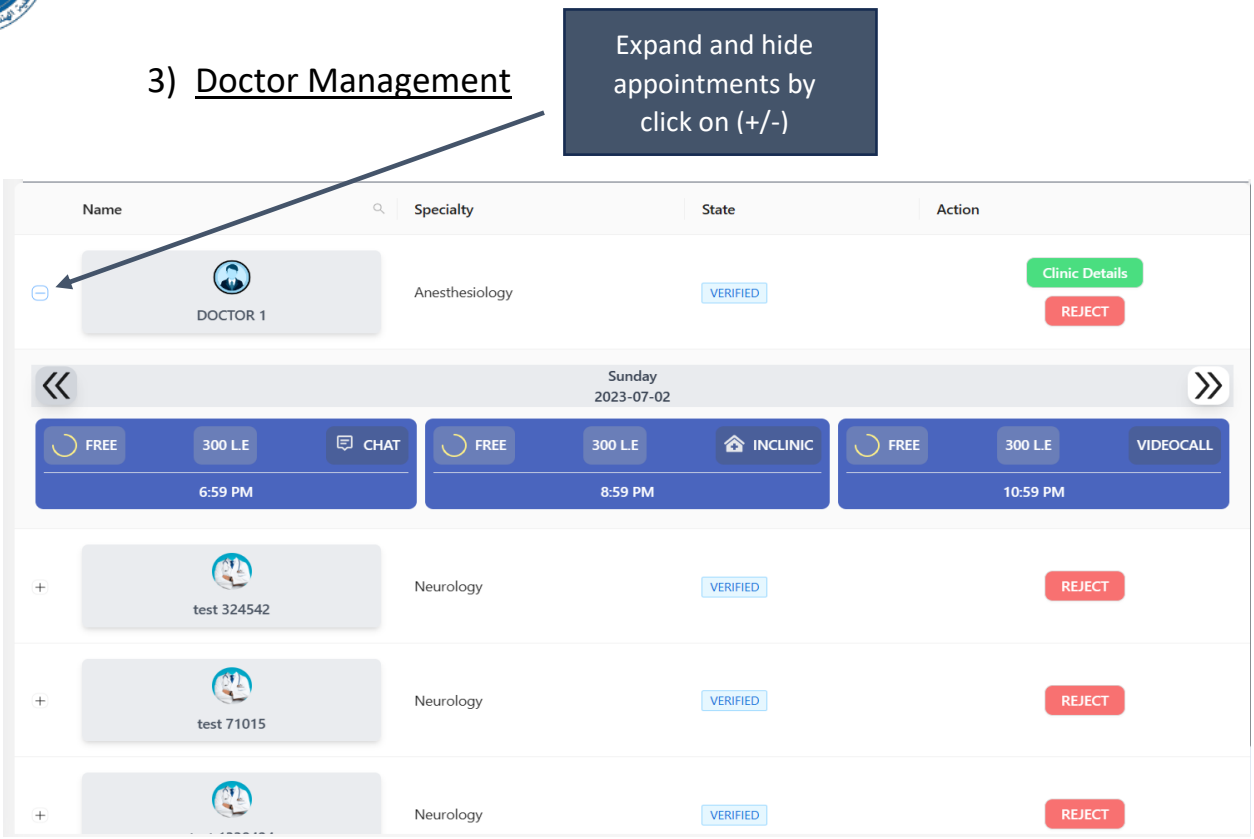
Explanation: we can notice that user no longer has profile even chat with his admin deleted due to account removed

- Restrict User



Explanation: we can notice doctor has live chatting opened with user who book appointment before with him in other hand user cannot make chatting with doctor without meeting.

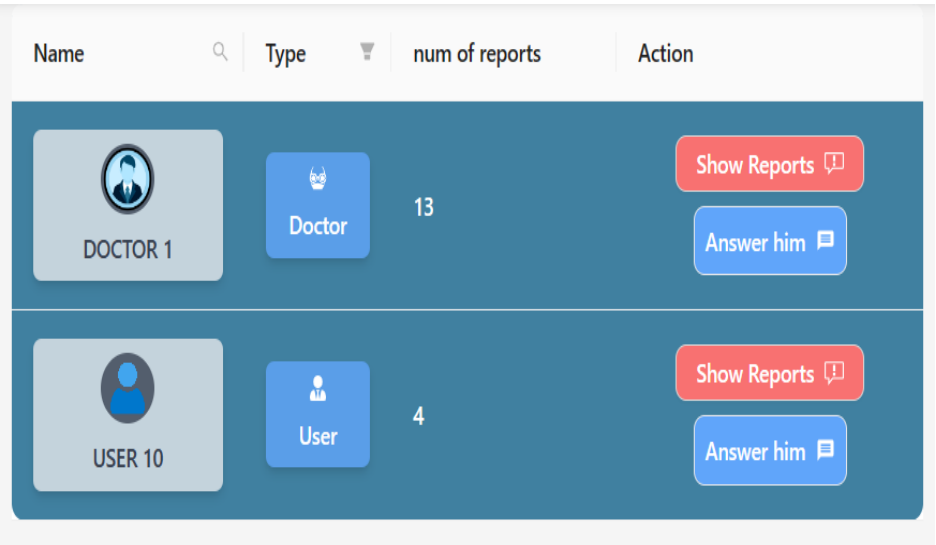
3) Doctor Management



Explanation: admin can view all doctor appointments with privileges to manage any appointment as deleting new appointments or cancelling booked appointments with same process acquired in doctor dashboard mentioned above.

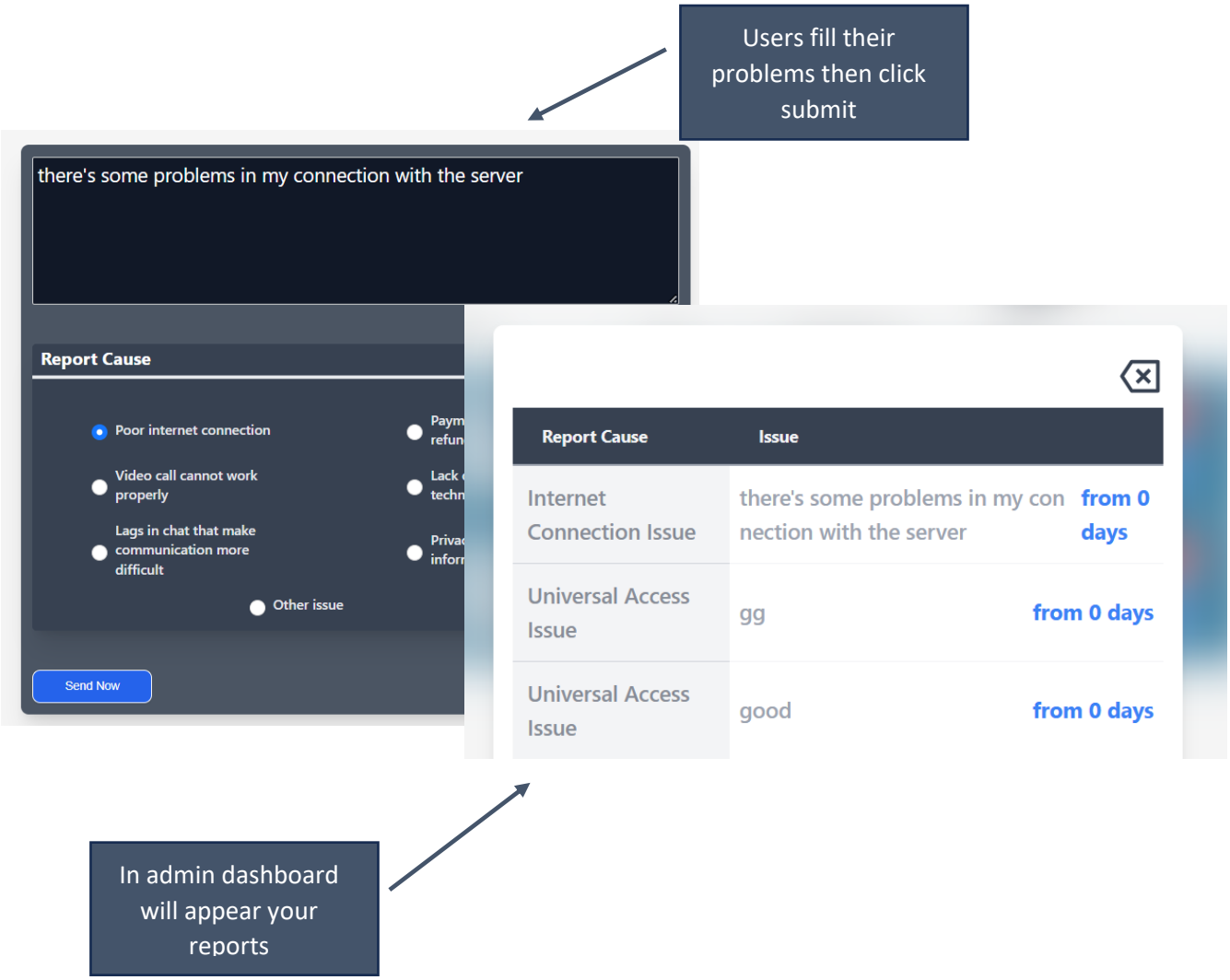
4) Report Management

- View Users with their Reports



Explanation: each user submits report has records in our system which enable admin to monitor and debug any problem or situation, also there is **answer him** button enabling admins to communicate (live Chatting) with users to solve their problems.

- Submit reports and display user reports.



Administrating our features is an important part to maintain, debug and monitor any bugs so we provide these functionalities to guarantee that our system behaves correctly.

10)Conclusion

- In conclusion, online clinic websites offer a convenient and accessible alternative to traditional in-person doctor's visits, allowing patients to receive medical consultations and treatment from the comfort of their own homes.
- However, there are several constraints and challenges that online clinics must overcome, including regulatory and legal restrictions, technical limitations, and the need for reliable internet connectivity.
- Despite these challenges, the growth of the telemedicine industry and the increasing adoption of online healthcare services demonstrate the potential of online clinics to improve access to healthcare and meet the changing needs of patients.
- As online clinics continue to evolve and advance, they have the potential to transform the way healthcare is delivered and provide patients with new and innovative options for managing their health.

11) Difficulties Faced

We are expected to face these difficulties when we start our implementation such as:

- Analysis the current workflow in the company.
- Finding information about such systems and the databases required.
- Working on the relationships between the entities.
- Time management
- Designing a suitable interface
- Sufficient testing