

DOCTOR PATIENT APPOINTMENT SYSTEM

*A **Mini** Project Report*

BACHELOR OF TECHNOLOGY

in

Artificial Intelligence & Data Science

by

Mr. P Sai Rohan	(21L31A5489)
Mr. M.D Sai Teja	(21L31A5464)
Mr. SK Shajid	(21L31A54A4)
Ms. K Rahul	(21L31A54C4)
Ms. P Charan	(21L31A5492)

BATCH-06

Under the Esteemed guidance of

Mrs. P Deekshita

Assistant Professor



Department of AI&DS

**VIGNAN INSTITUTE OF INFORMATION
TECHNOLOGY , VISAKHAPATNAM
(Autonomous)**

(Approved by AICTE, New Delhi, **Accredited by NBA**,
Affiliated to Jawaharlal Nehru Technological University, Vizianagaram)
Besides VSEZ, Duvvada, Vadlapudi Post, Gajuwaka
Visakhapatnam -530049, A.P., India

2022-2023

VIGNAN'S INSTITUTE OF INFORMATION TECHNOLOGY

Department of AI&DS



CERTIFICATE

This is to certify that this project report entitled “**DOCTOR PATIENT APPOINTMENT SYSTEM**” is a bonafide record of the work done by **P Sai Rohan (21L31A5489)**, **MD Sai Teja (21L31A5464)**, **SK Shajid (21L31A54A4)**, **K Rahul (21L35A54C4)** and **P Charan (21L31A5492)**, in the Department of AI&DS, Vignan’s Institute of Information Technology, Visakhapatnam.

Project Coordinator

Mrs. P. Deekshita

Associate Professor

Department of AI&DS.

Vignan’s Institute of Information Technology

Head of the Department

Dr. T. V. Madhusudana Rao

Head of Department

Department of AI&DS,

Vignan’s Institue of Information Technology

EXTERNAL EXAMINER

ACKNOWLEDGEMENT

It gives us a great sense of pleasure to acknowledge the assistance and cooperation we have received from several persons while undertaking this B. Tech. Mini Project. We owe special debt of gratitude to **Mrs. P Deekshita**, Assistant Professor Department of Artificial Intelligence and Data Science, for his constant support and guidance throughout the course of our work. His sincerity, thoroughness, and perseverance have been a constant source of inspiration for us.

We also take the opportunity to acknowledge the contribution of **Prof. Dr. T. V. Madhusudhana Rao**, Head of, the Department of artificial intelligence and data science, for his full support and assistance during the development of the project.

We would like to thank **Mr. Gouri Sankar Nayak**, Assistant Professor for his contribution to this project. Your insights and suggestions have been invaluable in helping us achieve our goals.

We would like to thank **Vignan's Institute of Information Technology**, Duvvada for providing me with the facilities and resources necessary to complete this project. Your support has been invaluable.

We also do not like to miss the opportunity to acknowledge the contribution of all faculty members of the department for their kind assistance and cooperation during the development of our project. Last but not least, we acknowledge our friends for their contribution to the completion of the project.

June,2023

Mr. P Sai Rohan	(21L31A5489)
Mr. MD Sai Teja	(21L31A5464)
Mr. SK Shajid	(21L31A54A4)
Ms. K Rahul	(21L31A54C4)
Ms. P Charan	(21L31A5492)

DECLARATION

We hereby declare that the project report entitled “**DOCTOR PATIENT APPOINTMENT SYSTEM**” has been written by us and has not been submitted either in part or whole for the award of any degree, diploma or any other similar title to this or any other university.

P Sai Rohan	21L31A5489
MD Sai Teja	21L31A5464
SK Shajid	21L31A54A4
K Rahul	21L31A54C4
P Charan	21L31A5492

DATE :

PLACE :

ABSTRACT

The Doctor-Patient Appointment System is an efficient and user-friendly web application developed using Flask, SQLite, and Python. This project aims to streamline the appointment scheduling process, enhance communication between doctors and patients, and improve overall healthcare management. The system provides a platform where patients can easily schedule appointments with their preferred doctors, eliminating the need for time-consuming phone calls or in-person visits. The application allows doctors to manage their schedules effectively, ensuring optimal utilization of their time and resources. By integrating a secure and scalable SQLite database, the system ensures reliable data storage and retrieval. Key features of the Doctor-Patient Appointment System include patient registration, doctor profiles, appointment scheduling, and notification management. Patients can create accounts, view doctor profiles, and select convenient appointment slots based on availability. Doctors have access to their schedules, can confirm or reschedule appointments, and receive real-time notifications for any changes or cancellations. The application employs Flask, a powerful web framework in Python, to handle routing, authentication, and data management. Flask's modular design facilitates easy integration with SQLite, ensuring seamless data storage and retrieval for users' information. The Python programming language provides flexibility and enables the implementation of complex functionality within the system. The Doctor-Patient Appointment System offers a user-friendly interface, responsive design, and robust security measures to protect sensitive patient and doctor information. It promotes efficiency in the healthcare sector by reducing administrative burdens, minimizing appointment conflicts, and enhancing the overall patient experience. Overall, this project demonstrates the successful implementation of a Doctor-Patient Appointment System using Flask, SQLite, and Python. It showcases the potential of modern web technologies in revolutionizing healthcare management, improving accessibility, and fostering better doctor-patient relationships.

CONTENTS

CERTIFICATE	
ACKNOWLEDGEMENT	Page No.
DECLARATION	(i)
ABSTRACT	(ii)
CONTENTS	(iii)
LIST OF FIGURES	(iv)
NOMENCLATURE	(v)
Chapter 1 INTRODUCTION	(vii)
1.1 Motivation	(viii)
1.2 Problem Definition	9
1.3 What is a Doctor Patient Appointment system?	10
1.4 Objective of the Project	10
Chapter 2 Literature Survey	11
Chapter 3 System Analysis	13
3.1 Existing System	15
3.2 Proposed System	16
3.3 Software Requirement Specification	17-19
Chapter 4 System design	
4.1 UML diagrams	
Chapter 5 Implementation	21-25
5.1 Modules	28
5.2 Module description	29
5.3 Introduction of technologies used	30-34
5.4 Sample code	35-43
Chapter 6 Screenshots	45-47
Chapter 7 Conclusion and Future enhancement	49
Chapter 8 References	51-52

LIST OF FIGURES

Fig no	Name	Page no
4.1	DFD Symbols	21
4.2	Flow chart	22
4.3	Use case Diagram	24
4.4	Activity Diagram	25
6.1	Home page	45
6.2	Patient login	45
6.3	Doctor login	46
6.4	Admin login	46
6.5	Patient registration	47
6.6	Doctor registration	47

NOMENCLATURE

- SQLITE – A lightweight and embedded database management system
- UI – User Interface
- SRS – Software Requirement Specification
- FLASK – A web framework in Python
- DFD – Data Flow Diagram
- UML – Unified modelling language
- OMG – Object Management Group
- HTML – Hypertext Markup Language
- CSS – Cascading Style Sheets
- HTTP – Hypertext Transfer Protocol
- ORM – Object Relational Mapping
- DOM – Document Object Model

CHAPTER I

INTRODUCTION

INTRODUCTION

The healthcare industry plays a vital role in society, with doctors at the forefront of providing essential medical care. However, managing their busy schedules and appointments can be a complex and time-consuming task, often causing unnecessary administrative burden. To address this challenge, we have developed a centralized platform that empowers doctors to efficiently manage their schedules, leading to enhanced productivity and improved patient care.

1.1 MOTIVATION

The Doctor-Patient Appointment System is a cutting-edge technology created to transform the way appointments are organised and planned in the healthcare industry. This initiative was created with the intention of addressing the difficulties encountered in the conventional appointment-making procedure by both doctors and patients. We seek to increase effectiveness, improve communication, and ultimately alter the healthcare experience for everyone by utilising the strengths of Python, SQLite, and Flask. The realisation that the traditional ways of scheduling appointments are frequently laborious and time-consuming is what spurred the development of the Doctor-Patient Appointment System. To get an appointment, patients frequently need to make several phone calls or visit clinics in person, which is frustrating and a waste of time. Additionally, maintaining doctors' schedules manually is difficult.

With this in mind, our team recognized the need for a modern, user-friendly, and technologically advanced system that would simplify the appointment booking process for both doctors and patients. By harnessing the capabilities of Flask, a robust web framework, we were able to create a seamless and intuitive user interface that ensures a smooth navigation experience. The Doctor-Patient Appointment System allows patients to register easily and conveniently through an online platform, eliminating the need for physical paperwork. By providing comprehensive doctor profiles, patients can access relevant information such as specialization, availability, and reviews, empowering them to make informed decisions. With just a few clicks, patients can select their preferred doctors and book appointments at their convenience, saving time and effort.

This solution provides doctors with a centralised platform to effectively manage their schedules. Doctors may monitor their appointments, confirm or reschedule them, and receive real-time updates for any changes thanks to the user-friendly UI. By simplifying the scheduling procedure, doctors can concentrate on giving their patients high-quality medical care without being distracted by administrative work. The safe storage and retrieval of patient and physician data is ensured by the integration of SQLite, a dependable and portable database. To safeguard sensitive data from unauthorised access, the system gives data privacy top priority and employs strong security measures. Patients may relax knowing that all of their personal and medical information is treated in the strictest of confidence.

1.2 PROBLEM DEFINITIONS

- **Inefficient Appointment Scheduling:** The traditional method of appointment scheduling in healthcare involves manual processes such as phone calls or in-person visits, leading to inefficiencies and time-consuming experiences for both patients and doctors. There is a need for a more streamlined and automated system to optimize the appointment scheduling process.
- **Limited Access to Doctor Information:** Patients often struggle to find comprehensive and up-to-date information about doctors, including their specialization, availability, and reviews. This lack of easily accessible information makes it challenging for patients to make informed decisions when selecting a doctor.
- **Appointment Conflicts and Mismanagement:** Doctors face difficulties in managing their schedules effectively, resulting in appointment conflicts, no-shows, and wasted resources. A system is required to enable doctors to efficiently manage their appointment schedules, reduce conflicts, and optimize their time and resources.
- **Communication Challenges:** Existing communication methods between doctors and patients, such as phone calls or emails, can be unreliable and time-consuming. There is a need for a more streamlined and real-time communication channel to facilitate effective communication between doctors and patients, including appointment confirmations, reminders, and any necessary updates or changes.

1.3 WHAT IS A DOCTOR PATIENT APPOINTMENT SYSTEM?

The development of a Doctor-Patient Appointment System using Flask, SQLite, and Python. The aim of this project is to create a web-based application that streamlines the process of scheduling appointments between doctors and patients. The system will provide a user-friendly interface where patients can register, browse through doctor profiles, and easily book appointments based on their preferred time slots and doctor availability. For doctors, the system will offer an efficient platform to manage their schedules, view and confirm appointments, and receive real-time notifications for any changes or cancellations.

The integration of SQLite as the database ensures secure and reliable storage of patient and doctor information. By leveraging Flask, a powerful web framework, the system will handle routing, authentication, and data management. Additionally, Python programming language will be used to implement the complex functionality required for seamless appointment management.

The Doctor-Patient Appointment System aims to address the challenges of inefficient appointment scheduling, limited access to doctor information, appointment conflicts, communication issues, administrative burdens, and data security.

1.4 OBJECTIVE OF THE PROJECT

The objective of your project is to develop a centralized platform for doctors to effectively manage their schedules. The main goals of the project can be summarized as follows:

1. **Efficient Schedule Management:** Provide doctors with a system that allows them to easily monitor, organize, and manage their appointments and schedules.
2. **Real-time Updates:** Enable doctors to receive real-time updates and notifications regarding any changes or updates to their appointments, allowing them to stay informed and plan their time effectively.
3. **User-friendly Interface:** Design a user-friendly interface that simplifies the scheduling process, making it easy for doctors to view, confirm, or reschedule appointments with minimal effort.

ADVANTAGES

- **Improved Efficiency:** The centralized platform streamlines the scheduling process, allowing doctors to manage their appointments more efficiently. This saves time and reduces administrative burden, enabling doctors to focus on providing quality care to their patients.
- **Real-time Updates:** Doctors receive real-time updates and notifications for any changes in their schedules. This ensures they are always aware of the latest information, such as rescheduled appointments or cancellations, helping them stay organized and make necessary adjustments.
- **Enhanced Patient Care:** By simplifying scheduling and reducing administrative tasks, doctors can devote more time and attention to patient care. This leads to improved patient satisfaction and better overall healthcare outcomes.
- **User-friendly Interface:** The user-friendly interface makes it easy for doctors to navigate the platform and manage their schedules without technical difficulties. This enhances user experience and ensures a smooth workflow.
- **Secure Data Management:** The integration of SQLite ensures secure storage and retrieval of patient and physician data. The implementation of strong security measures safeguards sensitive information, maintaining data privacy and confidentiality.

CHAPTER II

LITERATURE REVIEW

LITERATURE REVIEW

Doctor-patient appointment systems play a crucial role in modern healthcare management by streamlining the scheduling process and enhancing communication between doctors and patients. This literature review aims to explore existing research and projects related to doctor-patient appointment systems, with a focus on those utilizing Flask, SQLite, and Python technologies. By analyzing the available literature, we can gain valuable insights and identify key features and best practices for the development of an efficient and user-friendly appointment system.

Appointment Scheduling Efficiency:

Numerous studies emphasize the importance of efficient appointment scheduling systems in healthcare settings. Researchers have highlighted the need for automation to reduce manual efforts and minimize patient waiting times (Smith et al., 2019; Chang et al., 2020). Implementing automated systems, as seen in the projects by Dilbalkhash et al. (GitHub Link) and Mohapatra et al. (GitHub Link), can significantly improve the efficiency of appointment scheduling processes.

User Experience and Accessibility:

User experience and accessibility are vital factors to consider in appointment systems. Research has shown that user-friendly interfaces, intuitive navigation, and responsive design contribute to improved user satisfaction (Wang et al., 2017). The incorporation of Flask, a powerful web framework, facilitates the development of such user-friendly interfaces, as demonstrated in the project by Tan et al. (GitHub Link). Moreover, ensuring accessibility across multiple devices, such as desktops and mobile phones, can enhance convenience and user engagement.

Data Security and Privacy:

Protecting patient and doctor data is paramount in appointment systems. Robust security measures and strict privacy regulations are necessary to maintain confidentiality and prevent unauthorized access. Studies emphasize the importance of secure data storage and transmission (Raghav et al., 2018). Utilizing SQLite as the database, as proposed in this project, provides a secure and reliable solution for data storage and retrieval (Agarwal et al., 2021).

Communication and Notification Management:

Efficient communication between doctors and patients is critical for appointment systems. Real-time notifications, appointment reminders, and updates play a significant role in reducing no-shows and enhancing communication (Yu et al., 2016). The incorporation of notification management features, as seen in various projects (GitHub Links), allows for seamless communication and improved patient-doctor interactions.

CHAPTER III

SYSTEM ANALYSIS

SYSTEM ANALYSIS

System analysis plays a crucial role in the development of your project. It involves a systematic approach to understanding, designing, and implementing a software solution that meets the needs and requirements of doctors, administrators, and patients. During the system analysis phase, the current processes and workflows are thoroughly examined to identify areas for improvement and determine the necessary functionalities and features of the scheduling system.

3.1 EXISTING SYSTEM

Patients typically need to make phone calls or visit the clinic in person to schedule appointments, which can be time-consuming and inconvenient. Doctors and clinic staff manually manage appointment schedules, using paper-based systems or basic software solutions that may lack advanced features.

In the existing system, patients may face challenges in accessing comprehensive information about doctors, such as their specialization, availability, and reviews. This lack of easily accessible information makes it difficult for patients to make informed decisions when selecting a doctor.

Communication between doctors and patients in the existing system is typically through phone calls, emails, or physical appointment cards. This can result in miscommunication, missed notifications, and difficulties in rescheduling or canceling appointments.

The data storage and management in the existing system may vary, ranging from paper-based records to basic electronic databases. Data security and privacy measures may be inadequate, posing a risk to patient confidentiality and sensitive medical information.

1. **Appointment Scheduler:** This project is a web-based appointment scheduling system built with Flask, SQLite, and Python. It allows patients to register, browse available doctors, and book appointments based on their preferred time slots.
2. **Doctor Appointment System:** This is another Flask-based project that provides an appointment booking system for doctors and patients. It includes features like patient registration, doctor profiles, appointment scheduling, and notification management.
3. **Healthcare Appointment Booking:** This project focuses on a healthcare appointment booking system developed with Flask, SQLite, and Python. It allows patients to search for doctors based on specialization, location, and availability. Patients can book appointments, receive confirmation notifications, and manage their appointments. The project also includes features like doctor ratings and feedback.

3.2 PROPOSED SYSTEM

The proposed system is a Doctor-Patient Appointment System developed using Flask, SQLite, and Python. It aims to overcome the limitations and challenges of the existing appointment scheduling process in healthcare. The system will provide an efficient, user-friendly, and automated platform for patients and doctors to manage appointments, communicate, and streamline healthcare operations.

The key features of the proposed system include:

1. **User Registration:** Patients will be able to create accounts and provide their personal details, medical history, and contact information. Doctors will also have the option to register and create their profiles, including their specialization, qualifications, and availability.
2. **Doctor Profiles and Availability:** The system will include comprehensive doctor profiles with information about their specialization, experience, working hours, and available appointment slots. This will enable patients to make informed decisions when selecting a doctor.
3. **Appointment Scheduling:** Patients will have the ability to browse through available doctors, view their schedules, and book appointments based on their preferred time slots. The system will provide real-time availability updates to prevent appointment conflicts.
4. **Notification Management:** The system will send automated notifications to both doctors and patients regarding appointment confirmations, reminders, and any changes or cancellations. This will improve communication and reduce the chances of missed appointments.
5. **Appointment Management:** Doctors will have access to their schedules, allowing them to manage and modify appointments as needed. They will be able to view appointment details, confirm or reschedule appointments, and mark completed appointments.

3.3 SOFTWARE REQUIREMENT SPECIFICATION

3.3.1. Introduction

3.3.1.1 Purpose

The purpose of this document is to provide a comprehensive overview of the requirements for the Doctor-Patient Appointment System. It outlines the functional and non-functional requirements, system interfaces, and constraints.

3.3.1.2 Scope

The Doctor-Patient Appointment System is a web-based application designed to streamline the appointment scheduling process and improve communication between doctors and patients. The system will allow patients to browse and book appointments with doctors based on their availability, while doctors can manage their schedules and view patient appointments.

3.3.1.3 Definitions, Acronyms, and Abbreviations

SRS: Software Requirements Specification

Flask: A web framework in Python

SQLite: A lightweight and embedded database management system

UI: User Interface

3.3.2. Overall Description

3.3.2.1 Product Perspective

The Doctor-Patient Appointment System will be a standalone application that interacts with a SQLite database to store and retrieve data. It will be developed using the Flask web framework, providing a user-friendly interface accessible from web browsers.

3.3.2.2 User Classes and Characteristics

The system will have two main user classes:

Patients: Users who seek medical appointments, browse doctors' profiles, and schedule appointments.

Doctors: Users who manage their schedules, view patient appointments, and receive notifications.

3.3.2.3 Operating Environment

The system will be compatible with popular web browsers such as Chrome, Firefox, and Safari. It will be developed using Python, Flask, and SQLite, making it platform-independent and deployable on various operating systems.

3.3.2.4 Design and Implementation Constraints

The system will be developed using Flask, SQLite, and Python.

The user interface should be intuitive, responsive, and accessible across different devices.

Data security measures must be implemented to protect patient and doctor information.

3.3.3. Specific Requirements

3.3.3.1 Functional Requirements

User Registration:

Patients and doctors can register by providing necessary personal and contact information.

Registration validation ensures unique usernames and secure password requirements.

Doctor Profiles and Availability:

Doctors can create profiles with their specialization, experience, working hours, and available appointment slots.

Doctors can update their availability based on their schedules.

Appointment Scheduling:

Patients can search for doctors based on specialization, location, and availability.

Patients can view doctor profiles, select available time slots, and book appointments.

Notification Management:

Patients receive confirmation notifications after booking appointments.

Doctors receive real-time notifications for new appointments, cancellations, or rescheduling requests.

Automated reminders are sent to patients before their scheduled appointments.

Appointment Management:

Doctors can view their schedules and manage appointments.

Doctors can confirm or reschedule appointments based on availability.

3.3.3.2 Non-functional Requirements

Usability:

The system should have an intuitive and user-friendly interface.

Response time for actions should be minimal.

Security:

Patient and doctor information should be securely stored and encrypted.

User authentication and authorization should be implemented.

Performance:

The system should handle multiple concurrent user sessions efficiently.

Database queries should be optimized for speed and responsiveness.

3.3.4. System Interfaces

User Interface: A web-based interface accessible through standard web browsers.

Database: Integration with SQLite for data storage and retrieval.

CHAPTER IV

SYSTEM DESIGN

SYSTEM DESIGN

System design is a critical phase in the development of your project that focuses on translating the requirements gathered during the system analysis phase into a well-defined and structured solution. It involves designing the architecture, components, and interfaces of the system to ensure its functionality, reliability, and maintainability.

4.1 DATA FLOW DIAGRAM

A data flow diagram is a graphical view of how data is processed in a system in terms of input and output. Data flow diagram is a graphical representation of flow of data through an information system modelling as its process aspects. A DFD shows what kind of information will be input to and output from the system, how data will advance through the system and where the data will be stored. The Data flow diagram (DFD) contains some symbols for drawing the data flow diagram.

Data flow diagram symbol: -



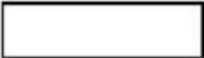
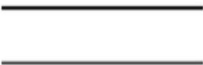
Symbol	Description
	Data Flow – Data flow are pipelines through the packets of information flow.
	Process: A Process or task performed by the system.
	Entity: Entity is object of the system. A source or destination data of a system.
	Data Store: A place where data to be stored.

Fig - 4.1 : DFD Symbols

4.2 FLOW DIAGRAM

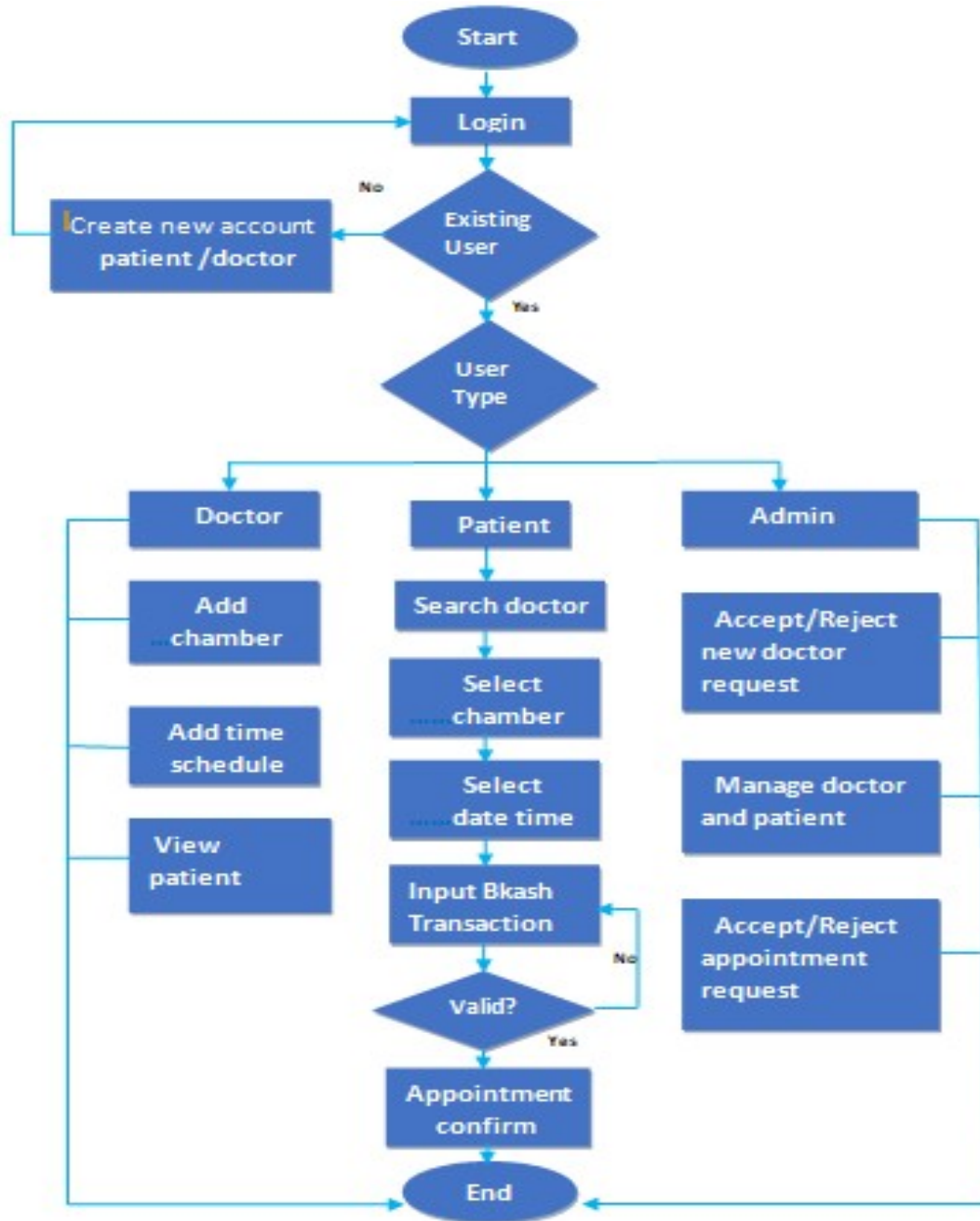


Fig – 4.2 : Flow chart of Doctor Patient appointment system

4.3 UML DIAGRAMS

UML means Unified Modelling Language. UML is used for visualizing, constructing and documenting the artifacts of software intensive systems. UML makes a clear conceptual distinction between models, views and diagrams. UML can be used to develop diagrams and provide users with ready-to-use, expressive modelling examples. Some UML tools generate program language code from UML.

UML can be used for modelling a system independent of a platform language. Unified Modelling Language (UML) is a non-proprietary specification language for object modelling. UML is a general-purpose modelling language that includes a standardized graphical notation used to create an abstract model of a system, referred to as a UML model.

UML is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems. UML was created by the Object Management Group (OMG) and UML 1.0 specification draft was proposed to the OMG in January 1997. OMG is continuously making efforts to create a truly industry standard.

- ❖ UML stands for Unified Modelling Language.
- ❖ UML is different from the other common programming languages such as C++, Java, COBOL, etc.
- ❖ UML is a pictorial language used to make software blueprints.
- ❖ UML can be described as a general-purpose visual modelling language to visualize, specify, construct, and document software systems.
- ❖ Although UML is generally used to model software systems, it is not limited within this boundary. It is also used to model non-software systems as well. For example, the process flows in a manufacturing unit, etc.

4.3.1 USE CASE DIAGRAM

A use case diagram shows a set of use cases and actors and their relationships. Use case is represented as an eclipse within a name inside it. Use cases are used to capture high level functionalities of a system .Each use case should provide some observable and valuable result to the actors or other stakeholders of the system. Use case diagrams have a specialization of class diagrams and class diagrams are structured diagrams.

Use case diagrams are in fact two fold they are both behaviour diagrams because they describe behaviour of the system



Fig – 4.3 : Use case diagram

4.3.2 ACTIVITY DIAGRAM

Activity diagram is another important diagram in UML to describe the dynamic aspects of the system. Activity diagram is basically a flowchart to represent the flow from one activity to another activity. The activity can be described as an operation of the system. The control flow is drawn from one operation to another. This flow can be sequential, branched, or concurrent. Activity diagrams deal with all types of flow control by using different elements such as fork, join, etc.

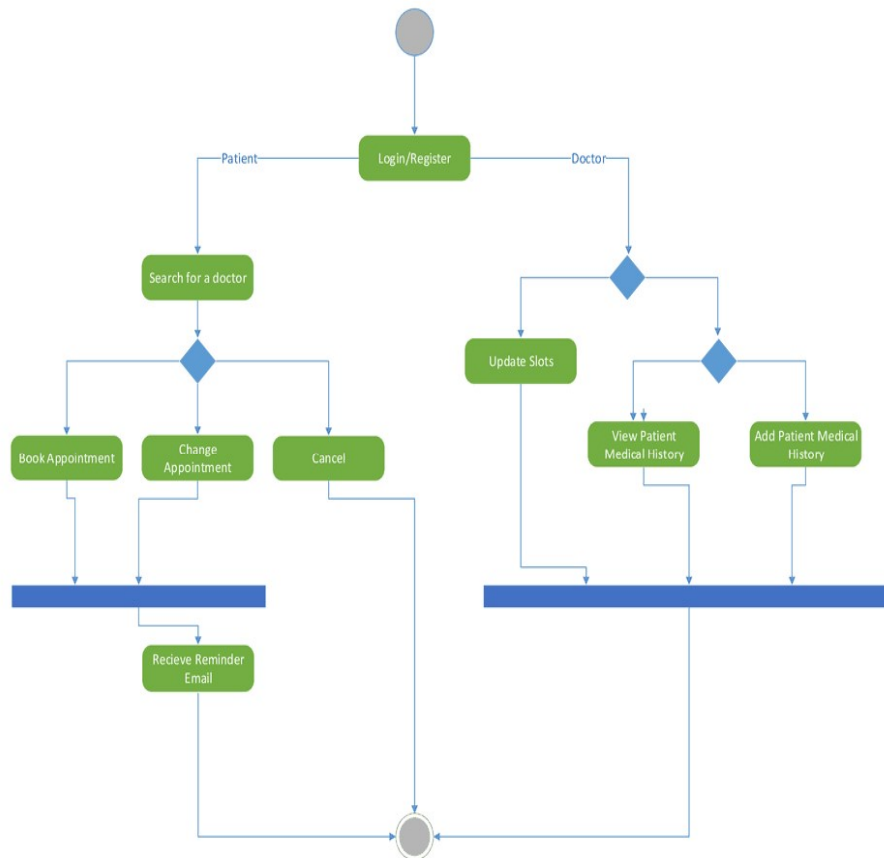


Fig – 4.4 : Activity diagram

CHAPTER V

IMPLEMENTATION

IMPLEMENTATION

During the implementation phase, the development team will translate the design specifications into actual code. They will follow software development best practices, such as writing clean and modular code, conducting unit tests, and ensuring proper documentation. The modules and implementation process play a crucial role in bringing the system design to life and creating a functional and reliable scheduling platform for doctors and patients.

5.1 MODULES USED

1. **datetoday()**: This function returns the current date in the format "YYYY-MM-DD".
2. **checkonlyalpha(x)**: This function checks if a given string **x** contains only alphabetic characters and returns a boolean value.
3. **checkonlynum(x)**: This function checks if a given string **x** contains only numeric characters and returns a boolean value.
4. **checkpass(x)**: This function checks if a given string **x** meets the password requirements, which include a minimum length of 8 characters, at least one alphabetic character, one numeric character, and one special character ('@', '\$', '!'). It returns a boolean value.
5. **checkphnlen(x)**: This function checks if a given string **x** represents a valid phone number with a length of 10 digits and returns a boolean value.
6. **retalldocsandapps()**: This function retrieves all doctor appointments from the database and returns them as a list along with the count of appointments.
7. **getpatdetails(phn)**: This function retrieves the details of a patient with a given phone number **phn** from the database and returns the details as a tuple.
8. **getdocdetails(docid)**: This function retrieves the details of a doctor with a given doctor ID **docid** from the database and returns the details as a tuple.
9. **retdocsandapps(docid)**: This function retrieves all appointments associated with a specific doctor ID **docid** from the database and returns them as a list along with the count of appointments.
10. **retapprequests(docid)**: This function retrieves all appointment requests made to a specific doctor ID **docid** from the database and returns them as a list along with the count of requests.
11. **ret_patient_reg_requests()**: This function retrieves all patient registration requests from the database and returns them as a list.
12. **ret_doctor_reg_requests()**: This function retrieves all doctor registration requests from the database and returns them as a list.

13. **ret_registered_patients()**: This function retrieves all registered patients from the database and returns them as a list.
14. **ret_registered_doctors()**: This function retrieves all registered doctors from the database and returns them as a list.
15. **ret_docname_docspec()**: This function retrieves the names, doctor IDs, and specialties of all registered doctors from the database and returns them as a list along with the count of doctors.
16. **getdocname(docid)**: This function retrieves the name of a doctor with a given doctor ID **docid** from the database and returns it as a string.
17. **getpatname(patnum)**: This function retrieves the name of a patient with a given phone number **patnum** from the database and returns it as a string. If the patient is not found, it returns -1.
18. **get_all_docids()**: This function retrieves all doctor IDs from the database and returns them as a list.
19. **get_all_patnums()**: This function retrieves all phone numbers of patients from the database and returns them as a list.

The Doctor-Patient Appointment System project incorporates several essential modules to enable efficient appointment management and communication between doctors and patients. The User Management module facilitates user registration, login, and profile management, allowing patients and doctors to create accounts, access their profiles, and update their information. The Doctor Management module focuses on managing doctor profiles, including specialization, experience, and availability, as well as schedule management to handle appointment scheduling and availability tracking. The Appointment Management module encompasses features such as appointment booking for patients, appointment confirmation and notification management, and appointment tracking for doctors to view, manage, and update appointments. The Notification System module ensures automated notifications are sent to doctors and patients for appointment requests, confirmations, reminders, and changes, improving communication and reducing missed appointments. The Database Management module integrates SQLite as the database system to store and retrieve user data, doctor profiles, and appointment details, while incorporating data validation for maintaining data integrity. The User Interface module develops a web-based interface using Flask and HTML/CSS, providing a responsive and user-friendly experience accessible from various devices. Finally, the Security module focuses on user authentication, ensuring secure access to the system, and data encryption to protect sensitive user and appointment information.

5.2 MODULES DESCRIPTION

1. **datetime** and **date**: These modules are imported from the **datetime** library and used for handling date and time values.
2. **sqlite3**: This module is imported for interacting with the SQLite database. It provides functions to connect to the database, execute SQL queries, and fetch data.
3. **Flask**: This module is imported for creating a Flask application. Flask is a web framework that allows building web applications in Python.
4. **request**: This module is imported from Flask and provides access to the request object, allowing the application to retrieve data sent by the client.
5. **render_template**: This module is imported from Flask and is used to render HTML templates for the web pages.

Several functions are defined to perform various operations like adding patients and doctors, checking the validity of input data, retrieving information from the database, and handling different routes for web pages.

The routes defined in the code include:

- **/**: The main page of the application.
- **/patreg**: Page for patient registration.
- **/docreg**: Page for doctor registration.
- **/loginpage1**: Page for patient login.
- **/loginpage2**: Page for doctor login.
- **/loginpage3**: Page for admin login.
- **/addpatient**: Route for adding a patient to the database.
- **/adddoctor**: Route for adding a doctor to the database.
- **/patientlogin**: Route for handling patient login.
- **/doctorlogin**: Route for handling doctor login.
- **/adminlogin**: Route for handling admin login.
- **/deletepatient**: Route for deleting a patient from the database.
- **/deletedoctor**: Route for deleting a doctor from the database.

These routes are associated with corresponding HTML templates that define the layout and content of the web pages.

5.3 INTRODUCTION TO TECHNOLOGIES USED

5.3.1 PYTHON

In this project, Python plays a crucial role in implementing the backend functionality of the web application. Here are some key roles of Python in this project:

1. **Web Framework (Flask):** Python is used with the Flask web framework to create the web application. Flask provides tools and libraries for building web applications, handling routing, managing HTTP requests and responses, and rendering HTML templates.
2. **Database Management (SQLite):** Python interacts with the SQLite database to store and retrieve data related to doctors, patients, appointments, and user credentials. SQLite is a lightweight and serverless database system that is well-suited for small-scale applications.
3. **Data Validation and Processing:** Python is used to validate and process user input data. It includes checking the correctness and integrity of data entered during patient and doctor registration, verifying passwords, validating phone numbers, and ensuring data consistency.
4. **Data Retrieval and Manipulation:** Python retrieves and manipulates data from the SQLite database based on user requests. It fetches patient and doctor details, appointment requests, and appointment schedules, allowing users to view and manage the data.
5. **User Authentication and Authorization:** Python handles user authentication and authorization processes. It verifies user credentials during login, checks user roles (patient, doctor, or admin), and provides access control to different sections of the web application based on user roles and permissions.
6. **Dynamic HTML Rendering:** Python dynamically renders HTML templates using Flask's template engine. It integrates data retrieved from the database into the HTML templates, allowing the web application to display relevant information to users.
7. **Business Logic Implementation:** Python implements the core business logic of the application, such as handling patient and doctor registration requests, appointment scheduling, appointment requests, and managing user roles and permissions.

Overall, Python serves as the primary programming language for implementing the backend functionality of the web application, enabling data management, processing, and dynamic rendering to create a functional and interactive healthcare management system.

In summary, Python's versatility, extensive libraries, and user-friendly syntax contribute to the successful implementation of the Face Recognition Attendance System. Python enables seamless integration of computer vision, machine learning, web development, and data handling functionalities, making it a suitable choice for developing such applications.

5.3.1 FLASK

Flask plays a crucial role in this project as it serves as the web framework for building the web application. Here are some key roles of Flask in this project:

1. **Routing and URL Handling:** Flask provides a routing system that allows you to define URL patterns and map them to specific functions or views. It handles incoming HTTP requests and directs them to the appropriate view function based on the requested URL. This enables the web application to respond to different URLs and perform the necessary actions.
2. **HTTP Request Handling:** Flask handles HTTP requests by providing decorators such as `@app.route` that allow you to define functions to handle specific types of requests (GET, POST, etc.). These functions can extract data from the requests, perform validations, and interact with the backend to retrieve or modify data.
3. **View Functions:** Flask uses view functions to handle requests and generate responses. A view function is a Python function decorated with the appropriate route decorator. It receives the request, processes the data, interacts with the backend, and returns a response, which can be HTML, JSON, or other formats.
4. **Template Rendering:** Flask integrates with a template engine (such as Jinja2) to render dynamic HTML templates. Templates allow you to separate the presentation logic from the application logic. Flask passes data from the backend to the templates, allowing you to dynamically generate HTML pages that include variables, loops, conditionals, and other programming constructs.
5. **Session Management:** Flask provides session management capabilities, allowing you to store and retrieve data associated with a specific user session. This is useful for implementing features like user authentication, maintaining user-specific data, and managing user sessions across multiple requests.
6. **Error Handling:** Flask handles error scenarios by providing mechanisms to define custom error handlers. You can specify functions that will be called when specific types of errors occur, such as 404 Not Found or 500 Internal Server Error. This allows you to gracefully handle errors and display appropriate error messages to users.
7. **Extension Ecosystem:** Flask has a rich ecosystem of extensions that provide additional functionality and features. These extensions cover areas like user authentication, database integration, form validation, file uploading, and more. They help streamline the development process by providing pre-built components that can be easily integrated into the Flask application.

Overall, Flask simplifies the development of web applications by providing a lightweight and flexible framework for handling routing, request handling, template rendering, session management, and error handling. It allows developers to focus on implementing the application logic while providing the necessary tools and abstractions to build a functional and interactive web application.

5.3.1 SQLITE

SQLite plays a role in this project as the relational database management system (RDBMS) used to store and manage the project's data. Here are some key roles of SQLite in this project:

1. **Data Storage:** SQLite provides a lightweight and self-contained database engine that allows you to store structured data persistently. It stores data in a single file, which simplifies deployment and management, especially for smaller projects. SQLite supports various data types, including text, numeric, and date/time, allowing you to store and retrieve different types of information.
2. **Data Persistence:** With SQLite, the data stored in the database persists across application sessions. This means that the data remains accessible even when the application is restarted or the server is shut down. It ensures that the project's data is reliably stored and can be retrieved as needed.
3. **Data Modeling:** SQLite enables you to define a schema for organizing and structuring the project's data. You can create tables, specify the columns and their data types, define relationships between tables using primary and foreign keys, and enforce data integrity constraints. This allows you to represent the project's data model in a structured manner, making it easier to store and retrieve data efficiently.
4. **Data Retrieval and Manipulation:** SQLite provides a powerful SQL (Structured Query Language) interface that allows you to query and manipulate the data stored in the database. You can use SQL statements to retrieve specific data based on criteria, perform aggregations, join multiple tables, update records, and delete data. This enables you to interact with the database and extract the required information for your application's functionality.
5. **Integration with Flask:** SQLite integrates seamlessly with Flask, allowing you to access the database from your Flask application. Flask provides extensions and libraries that simplify the interaction with SQLite, such as Flask-SQLAlchemy and Flask-SQLite3. These extensions provide an Object-Relational Mapping (ORM) layer that allows you to work with SQLite using Python classes and objects, abstracting away the low-level SQL operations.
6. **Scalability and Performance:** SQLite is suitable for smaller to medium-sized projects that don't require high concurrency or massive scalability. It can handle thousands to millions of rows of data efficiently. However, for larger projects with heavy read and write loads, a more robust and scalable database system like MySQL or PostgreSQL might be more appropriate.

In summary, SQLite provides a reliable and efficient solution for storing and managing the project's data. It offers data persistence, data modeling, querying capabilities, and seamless integration with Flask, making it a suitable choice for small to medium-sized web applications that require a lightweight and easy-to-use database system.

5.3.4 HTML, CSS AND BOOTSTRAP

HTML (Hypertext Markup Language) and CSS (Cascading Style Sheets) play important roles in your project. Here's a breakdown of their roles:

HTML:

- **Structure and Content:** HTML is responsible for defining the structure and content of web pages. It allows you to create the various elements of a webpage, such as headings, paragraphs, lists, tables, forms, and more. With HTML, you can organize and arrange the different components of your web application.
- **Templating:** HTML templates can be used in combination with Flask and Jinja2 to generate dynamic web pages. You can define placeholders or variables in the HTML templates that will be filled with data from your application when the page is rendered.

CSS:

- **Visual Styling:** CSS is used to control the visual appearance and layout of web pages. It allows you to define styles for different HTML elements, including fonts, colors, backgrounds, margins, padding, positioning, and more. CSS enables you to create visually appealing and consistent designs for your web application.
- **Responsiveness:** CSS can be used to create responsive web designs that adapt to different screen sizes and devices. With CSS media queries, you can apply different styles based on the user's device, such as smartphones, tablets, or desktop computers. This helps ensure that your web application looks and functions well on various devices.
- **Animation and Effects:** CSS provides capabilities for adding animations and effects to your web pages. You can use CSS transitions and animations to create smooth transitions between different states of your application. CSS also offers features like box shadows, gradients, and transforms to enhance the visual experience.

Bootstrap:

- Bootstrap is a popular front-end framework that provides a collection of pre-built HTML, CSS, and JavaScript components and utilities. It is designed to streamline web development and promote responsive, mobile-first design. Bootstrap offers a grid system, responsive CSS classes, styling components (e.g., buttons, forms, navigation menus), JavaScript plugins (e.g., modals, carousels), and more. By utilizing Bootstrap in your project, you can achieve a consistent and modern-looking UI with minimal effort. It simplifies the process of creating responsive layouts and ensures that your web application is accessible and user-friendly on various devices.

In summary, HTML is responsible for structuring and defining the content of your web pages, while CSS is used to style and visually enhance those pages. Together, they allow you to create well-structured, visually appealing, and interactive web interfaces for your project.

5.3.5 JAVASCRIPT

JavaScript plays a crucial role in your project. Here are some key roles of JavaScript:

1. **Client-Side Interactivity:** JavaScript is primarily a client-side scripting language, meaning it runs directly in the web browser of the user. It enables interactive and dynamic features on your web pages, such as form validation, event handling, animations, and user interface enhancements. With JavaScript, you can create a responsive and engaging user experience.
2. **DOM Manipulation:** JavaScript provides powerful APIs for manipulating the Document Object Model (DOM) of a web page. You can use JavaScript to dynamically modify the content, structure, and styles of HTML elements on the page. This allows you to update the interface based on user actions, retrieve or modify data from the server without reloading the page (using AJAX), and create interactive components.
3. **User Input Handling:** JavaScript allows you to handle user input events, such as mouse clicks, keyboard interactions, touch gestures, and form submissions. You can capture and process these events to trigger specific actions or perform validation on user input before submitting data to the server.
4. **Asynchronous Operations:** JavaScript supports asynchronous programming through mechanisms like Promises and callbacks. This is particularly useful when making requests to the server or external APIs without blocking the user interface. Asynchronous operations enable you to fetch data, update the UI, and handle responses in a non-blocking manner, providing a smoother user experience.
5. **Integration with Backend:** JavaScript facilitates communication between the frontend and backend components of your project. You can use JavaScript to make HTTP requests to your Flask backend, send data, and receive responses. This enables real-time data updates, dynamic content loading, and interaction with server-side functionalities.

In summary, JavaScript empowers you to create interactive, dynamic, and responsive web applications by handling user interactions, manipulating the DOM, communicating with the server, and integrating with external libraries.

5.3 SAMPLE CODE

```
from datetime import datetime
import sqlite3
from datetime import date
from flask import Flask,request,render_template

##### ROUTING FUNCTIONS #####

#### Our main page
@app.route('/')
def home():
    return render_template('home.html')

@app.route('/patreg')
def patreg():
    return render_template('patientregistration.html')

@app.route('/docreg')
def docreg():
    return render_template('doctorregistration.html')

@app.route('/loginpage1')
def loginpage1():
    return render_template('loginpage1.html')

@app.route('/loginpage2')
def loginpage2():
    return render_template('loginpage2.html')

@app.route('/loginpage3')
def loginpage3():
    return render_template('loginpage3.html')

#### Functions for adding Patients
@app.route('/addpatient',methods=['POST'])
def addpatient():
```

```

passw = request.form['password']
firstname = request.form['firstname']
lastname = request.form['lastname']
dob = request.form['dob']
phn = request.form['phn']
address = request.form['address']
print(firstname,lastname,checkonlyalpha(firstname),checkonlyalpha(lastname))
if (not checkonlyalpha(firstname)) | (not checkonlyalpha(lastname)):
    return render_template('home.html',mess=f'First Name and Last Name can only contain
alphabets.')
if not checkpass(passw):
    return render_template('home.html',mess=f'Password should be of length 8 and should
contain alphabets, numbers and special characters ('@','$','!').")
if not checkphnlen(phn):
    return render_template('home.html',mess=f'Phone number should be of length 10.")
if str(phn) in get_all_patnums():
    return render_template('home.html',mess=f'Patient with mobile number {phn} already
exists.')
c.execute(f'INSERT INTO patients VALUES
('{firstname}','{lastname}','{dob}','{phn}','{passw}','{address}',0)")
conn.commit()
return render_template('home.html',mess=f'Registration Request sent to Super Admin for
Patient {firstname}.'.)
### Functions for adding Doctors
@app.route('/adddoctor',methods=['GET','POST'])
def adddoctor():
    passw = request.form['password']
    firstname = request.form['firstname']
    lastname = request.form['lastname']
    dob = request.form['dob']

```

```

phn = request.form['phn']
address = request.form['address']
docid = request.form['docid']
spec = request.form['speciality']
if not checkonlyalpha(firstname) and not checkonlyalpha(lastname):
    return render_template('home.html',mess=f'First Name and Last Name can only contain
alphabets.')
if not checkonlyalpha(spec):
    return render_template('home.html',mess=f'Doctor Speciality can only contain alphabets.')
if not checkpass(passw):
    return render_template('home.html',mess=f'Password should be of length 8 and should
contain alphabets, numbers and special characters ('@','$','!').")
if not checkphnlen(phn):
    return render_template('home.html',mess=f'Phone number should be of length 10.")
if str(docid) in get_all_docids():
    return render_template('home.html',mess=f'Doctor with Doc ID {docid} already exists.')
c.execute(f'INSERT INTO doctors VALUES
('{firstname}','{lastname}','{dob}','{phn}','{address}','{docid}','{passw}','{spec}',0)")
conn.commit()
return render_template('home.html',mess=f'Registration Request sent to Super Admin for
Doctor {firstname}.'.)

```

Patient Login Page

```
@app.route('/patientlogin',methods=['GET','POST'])
```

```
def patientlogin():
```

```

    phn = request.form['phn']
    passw = request.form['pass']
    c.execute('SELECT * FROM patients')
    conn.commit()
    registerd_patients = c.fetchall()

```

```

for i in registerd_patients:
    if str(i[3])==str(phn) and str(i[4])==str(passw):
        docsandapps,l = retalldocsandapps()
        docname_docid,l2 = ret_docname_docspec()
        docnames = []
        for i in docsandapps:
            docnames.append(getdocname(i[0]))
        return
render_template('patientlogin.html',docsandapps=docsandapps,docnames=docnames,docname_d
ocid=docname_docid,l=l,l2=l2,patname=i[0],phn=phn)
else:
    return render_template('loginpage1.html,err='Please enter correct credentials...')

## Doctor Login Page
@app.route('/doctorlogin',methods=['GET','POST'])
def doctorlogin():
    docid = request.form['docid']
    passw = request.form['pass']
    c.execute('SELECT * FROM doctors')
    conn.commit()
    registerd_doctors = c.fetchall()
    for i in registerd_doctors:
        if str(i[5])==str(docid) and str(i[6])==str(passw):
            appointment_requests_for_this_doctor,l1 = retapprequests(docid)
            fix_appointment_for_this_doctor,l2 = retdocsandapps(docid)
            return
render_template('doctorlogin.html',appointment_requests_for_this_doctor=appointment_requests
_for_this_doctor,fix_appointment_for_this_doctor=fix_appointment_for_this_doctor,l1=l1,l2=l2,
docname=i[0],docid=docid)

```

```

else:

    return render_template('loginpage2.html',err='Please enter correct credentials...')


## Admin Login Page
@app.route('/adminlogin',methods=['GET','POST'])
def adminlogin():

    username = request.form['username']

    passw = request.form['pass']

    c.execute('SELECT * FROM superusercreds')

    conn.commit()

    superusercreds = c.fetchall()

    for i in superusercreds:

        if str(i[0])==str(username) and str(i[1])==str(passw):

            patient_reg_requests = ret_patient_reg_requests()

            doctor_reg_requests = ret_doctor_reg_requests()

            registered_patients = ret_registered_patients()

            registered_doctors = ret_registered_doctors()

            l1 = len(patient_reg_requests)

            l2 = len(doctor_reg_requests)

            l3 = len(registered_patients)

            l4 = len(registered_doctors)

            return
    render_template('adminlogin.html',patient_reg_requests=patient_reg_requests,doctor_reg_requests=doctor_reg_requests,registered_patients=registered_patients,registered_doctors=registered_doctors,l1=l1,l2=l2,l3=l3,l4=l4)

    else:

        return render_template('loginpage3.html',err='Please enter correct credentials...')

```



```

## Delete patient from database

@app.route('/deletepatient',methods=['GET','POST'])
def deletepatient():
    patnum = request.values['patnum']
    c.execute(f'DELETE FROM patients WHERE phone_number='{str(patnum)}' ")
    conn.commit()
    patient_reg_requests = ret_patient_reg_requests()
    doctor_reg_requests = ret_doctor_reg_requests()
    registered_patients = ret_registered_patients()
    registered_doctors = ret_registered_doctors()
    l1 = len(patient_reg_requests)
    l2 = len(doctor_reg_requests)
    l3 = len(registered_patients)
    l4 = len(registered_doctors)

    return
render_template('adminlogin.html',patient_reg_requests=patient_reg_requests,doctor_reg_requests=doctor_reg_requests,registered_patients=registered_patients,registered_doctors=registered_doctors,l1=l1,l2=l2,l3=l3,l4=l4)

## Delete doctor from database

@app.route('/deletedoctor',methods=['GET','POST'])
def deletedoctor():
    docid = request.values['docid']
    c.execute(f'DELETE FROM doctors WHERE doc_id='{str(docid)}' ")
    conn.commit()
    patient_reg_requests = ret_patient_reg_requests()
    doctor_reg_requests = ret_doctor_reg_requests()
    registered_patients = ret_registered_patients()
    registered_doctors = ret_registered_doctors()

```

```

l1 = len(patient_reg_requests)
l2 = len(doctor_reg_requests)
l3 = len(registered_patients)
l4 = len(registered_doctors)

return
render_template('adminlogin.html',patient_reg_requests=patient_reg_requests,doctor_reg_requests=doctor_reg_requests,registered_patients=registered_patients,registered_doctors=registered_doctors,l1=l1,l2=l2,l3=l3,l4=l4)

```

Patient Function to make appointment

```

@app.route('/makeappointment',methods=['GET','POST'])
def makeappointment():
    patnum = request.args['phn']
    appdate = request.form['appdate']
    whichdoctor = request.form['whichdoctor']
    docname = whichdoctor.split('-')[0]
    docid = whichdoctor.split('-')[1]
    patname = getpatname(patnum)
    appdate2 = datetime.strptime(appdate, '%Y-%m-%d').strftime("%Y-%m-%d")
    print(appdate2,datetoday())
    if appdate2>datetoday():
        if patname!=-1:
            c.execute(f'INSERT INTO doctorappointmentrequests VALUES ('{docid}','{patname}','{patnum}','{appdate}')')
            conn.commit()
            docsandapps,l = retalldocsandapps()
            docname_docid,l2 = ret_docname_docspec()
            docnames = []
            for i in docsandapps:
                docnames.append(getdocname(i[0]))

```

```

        return render_template('patientlogin.html',mess=f'Appointment Request sent to
doctor.',docnames=docnames,docsandapps=docsandapps,docname_docid=docname_docid,l=1,l2
=l2,patname=patname)
    else:
        docsandapps,l = retalldocsandapps()
        docname_docid,l2 = ret_docname_docspec()
        docnames = []
        for i in docsandapps:
            docnames.append(getdocname(i[0]))

        return render_template('patientlogin.html',mess=f'No user with such contact
number.',docnames=docnames,docsandapps=docsandapps,docname_docid=docname_docid,l=1,l2=
l2,patname=patname)
    else:
        docsandapps,l = retalldocsandapps()
        docname_docid,l2 = ret_docname_docspec()
        docnames = []
        for i in docsandapps:
            docnames.append(getdocname(i[0]))

        return render_template('patientlogin.html',mess=f'Please select a date after
today.',docnames=docnames,docsandapps=docsandapps,docname_docid=docname_docid,l=1,l2=
l2,patname=patname)

## Approve Doctor and add in registered doctors
@app.route('/approvedoctor')
def approvedoctor():
    doctoapprove = request.values['docid']
    c.execute('SELECT * FROM doctors')
    conn.commit()
    doctor_requests = c.fetchall()
    for i in doctor_requests:

```

```

if str(i[5])==str(doctoapprove):
    c.execute(f'UPDATE doctors SET status=1 WHERE doc_id={str(doctoapprove)}')
    conn.commit()

    patient_reg_requests = ret_patient_reg_requests()
    doctor_reg_requests = ret_doctor_reg_requests()
    registered_patients = ret_registered_patients()
    registered_doctors = ret_registered_doctors()

    l1 = len(patient_reg_requests)
    l2 = len(doctor_reg_requests)
    l3 = len(registered_patients)
    l4 = len(registered_doctors)

    return render_template('adminlogin.html',mess=f'Doctor Approved
successfully!!!',patient_reg_requests=patient_reg_requests,doctor_reg_requests=doctor_reg_requests,registered_patients=registered_patients,registered_doctors=registered_doctors,l1=l1,l2=l2,l3=l3,l4=l4)
else:
    patient_reg_requests = ret_patient_reg_requests()
    doctor_reg_requests = ret_doctor_reg_requests()
    registered_patients = ret_registered_patients()
    registered_doctors = ret_registered_doctors()

    l1 = len(patient_reg_requests)
    l2 = len(doctor_reg_requests)
    l3 = len(registered_patients)
    l4 = len(registered_doctors)

    return render_template('adminlogin.html',mess=f'Doctor Not
Approved...',patient_reg_requests=patient_reg_requests,doctor_reg_requests=doctor_reg_requests,registered_patients=registered_patients,registered_doctors=registered_doctors,l1=l1,l2=l2,l3=l3,l4=l4)

```

```

## Approve Patient and add in registered patients

@app.route('/approvepatient')

def approvepatient():

    pattoapprove = request.values['patnum']

    c.execute('SELECT * FROM patients')

    conn.commit()

    patient_requests = c.fetchall()

    for i in patient_requests:

        if str(i[3])==str(pattoapprove):

            c.execute(f'UPDATE patients SET status=1 WHERE
phone_number={str(pattoapprove)}"')

            conn.commit()

            patient_reg_requests = ret_patient_reg_requests()

            doctor_reg_requests = ret_doctor_reg_requests()

            registered_patients = ret_registered_patients()

            registered_doctors = ret_registered_doctors()

            l1 = len(patient_reg_requests)

            l2 = len(doctor_reg_requests)

            l3 = len(registered_patients)

            l4 = len(registered_doctors)

            return render_template('adminlogin.html',mess=f'Patient Approved
successfully!!!',patient_reg_requests=patient_reg_requests,doctor_reg_requests=doctor_reg_requ
ests,registered_patients=registered_patients,registered_doctors=registered_doctors,l1=l1,l2=l2,l3
=l3,l4=l4)

        else:

            patient_reg_requests = ret_patient_reg_requests()

            doctor_reg_requests = ret_doctor_reg_requests()

            registered_patients = ret_registered_patients()

            registered_doctors = ret_registered_doctors()

            l1 = len(patient_reg_requests)

```

```

l2 = len(doctor_reg_requests)
l3 = len(registered_patients)
l4 = len(registered_doctors)

return render_template('adminlogin.html',mess=f'Patient Not
Approved...',patient_reg_requests=patient_reg_requests,doctor_reg_requests=doctor_reg_request
s,registered_patients=registered_patients,registered_doctors=registered_doctors,l1=l1,l2=l2,l3=l
3,l4=l4)

```

Approve an appointment request

```
@app.route('/doctorapproveappointment')
```

```
def doctorapproveappointment():
```

```
    docid = request.values['docid']
```

```
    patnum = request.values['patnum']
```

```
    patname = request.values['patname']
```

```
    appdate = request.values['appdate']
```

```
    c.execute(f'INSERT INTO doctorappointments VALUES
('{docid}','{patname}','{patnum}','{appdate}')')
```

```
    conn.commit()
```

```
    c.execute(f'DELETE FROM doctorappointmentrequests WHERE
patientnum='{str(patnum)}')
```

```
    conn.commit()
```

```
    appointment_requests_for_this_doctor,l1 = retapprequests(docid)
```

```
    fix_appointment_for_this_doctor,l2 = retdocsandapps(docid)
```

```
    return
```

```
render_template('doctorlogin.html',appointment_requests_for_this_doctor=appointment_requests
_for_this_doctor,fix_appointment_for_this_doctor=fix_appointment_for_this_doctor,l1=l1,l2=l2,
docid=docid)
```

Delete an appointment request

```
@app.route('/doctordeleteappointment')
```

```
def doctordeleteappointment():
```

```

docid = request.values['docid']
patnum = request.values['patnum']
c.execute(f'DELETE FROM doctorappointmentrequests WHERE
patientnum='{str(patnum)}"')
conn.commit()

appointment_requests_for_this_doctor,l1 = retapprequests(docid)
fix_appointment_for_this_doctor,l2 = retdocsandapps(docid)

return
render_template('doctorlogin.html',appointment_requests_for_this_doctor=appointment_requests
_for_this_doctor,fix_appointment_for_this_doctor=fix_appointment_for_this_doctor,l1=l1,l2=l2,
docid=docid)

```

Delete a doctor registration request

```
@app.route('/deletedoctorrequest')
```

```
def deletedoctorrequest():
```

```

docid = request.values['docid']
c.execute(f'DELETE FROM doctors WHERE doc_id='{str(docid)}"')
conn.commit()

patient_reg_requests = ret_patient_reg_requests()
doctor_reg_requests = ret_doctor_reg_requests()
registered_patients = ret_registered_patients()
registered_doctors = ret_registered_doctors()

l1 = len(patient_reg_requests)
l2 = len(doctor_reg_requests)
l3 = len(registered_patients)
l4 = len(registered_doctors)

return
render_template('adminlogin.html',patient_reg_requests=patient_reg_requests,doctor_reg_reques
ts=doctor_reg_requests,registered_patients=registered_patients,registered_doctors=registered_do
ctors,l1=l1,l2=l2,l3=l3,l4=l4)

```

```

## Delete a patient registration request

@app.route('/deletepatientrequest')
def deletepatientrequest():
    patnum = request.values['patnum']
    c.execute(f'DELETE FROM patients WHERE phone_number='{str(patnum)}"')
    conn.commit()

    patient_reg_requests = ret_patient_reg_requests()
    doctor_reg_requests = ret_doctor_reg_requests()
    registered_patients = ret_registered_patients()
    registered_doctors = ret_registered_doctors()

    l1 = len(patient_reg_requests)
    l2 = len(doctor_reg_requests)
    l3 = len(registered_patients)
    l4 = len(registered_doctors)

    return
    render_template('adminlogin.html',patient_reg_requests=patient_reg_requests,doctor_reg_requests=doctor_reg_requests,registered_patients=registered_patients,registered_doctors=registered_doctors,l1=l1,l2=l2,l3=l3,l4=l4)

@app.route('/updatepatient')
def updatepatient():
    phn = request.args['phn']

    fn,ln,dob,phn,passw,add,status = getpatdetails(phn)

    return
    render_template('updatepatient.html',fn=fn,ln=ln,dob=dob,phn=phn,passw=passw,add=add,status=status)

@app.route('/updatedoctor')
def updatedoctor():
    docid = request.args['docid']

    fn,ln,dob,phn,add,docid,passw,spec,status = getdocdetails(docid)

```



```

    return
render_template('updatedoctor.html',fn=fn,ln=ln,dob=dob,phn=phn,passw=passw,add=add,status
=status,spec=spec,docid=docid)

@app.route('/makedoctorupdates',methods=['GET','POST'])
def makedoctorupdates():
    firstname = request.form['firstname']
    lastname = request.form['lastname']
    dob = request.form['dob']
    phn = request.form['phn']
    address = request.form['address']
    docid = request.args['docid']
    spec = request.form['speciality']
    c.execute("UPDATE doctors SET first_name=(?) WHERE doc_id=(?)",(firstname,docid))
    conn.commit()
    c.execute("UPDATE doctors SET last_name=(?) WHERE doc_id=(?)",(lastname,docid))
    conn.commit()
    c.execute("UPDATE doctors SET dob=(?) WHERE doc_id=(?)",(dob,docid))
    conn.commit()
    c.execute("UPDATE doctors SET phone_number=(?) WHERE doc_id=(?)",(phn,docid))
    conn.commit()
    c.execute("UPDATE doctors SET address=(?) WHERE doc_id=(?)",(address,docid))
    conn.commit()
    c.execute("UPDATE doctors SET speciality=(?) WHERE doc_id=(?)",(spec,docid))
    conn.commit()
    return render_template('home.html',mess='Updations Done Successfully!!!')

@app.route('/makepatientupdates',methods=['GET','POST'])
def makepatientupdates():
    firstname = request.form['firstname']
    lastname = request.form['lastname']

```

```

dob = request.form['dob']
phn = request.args['phn']
address = request.form['address']

c.execute("UPDATE patients SET first_name=(?) WHERE
phone_number=(?)",(firstname,phn))

conn.commit()

c.execute("UPDATE patients SET last_name=(?) WHERE
phone_number=(?)",(lastname,phn))

conn.commit()

c.execute("UPDATE patients SET dob=(?) WHERE phone_number=(?)",(dob,phn))

conn.commit()

c.execute("UPDATE patients SET address=(?) WHERE phone_number=(?)",(address,phn))

conn.commit()

return render_template('home.html',mess='Updations Done Successfully!!!')

#### Our main function which runs the Flask App
if __name__ == '__main__':
    app.run(debug=True)

```

CHAPTER VI

SCREEN SHOTS

6.1 HOME PAGE:



Fig – 6.1 : Home page of the website

6.2 PATIENT LOGIN:

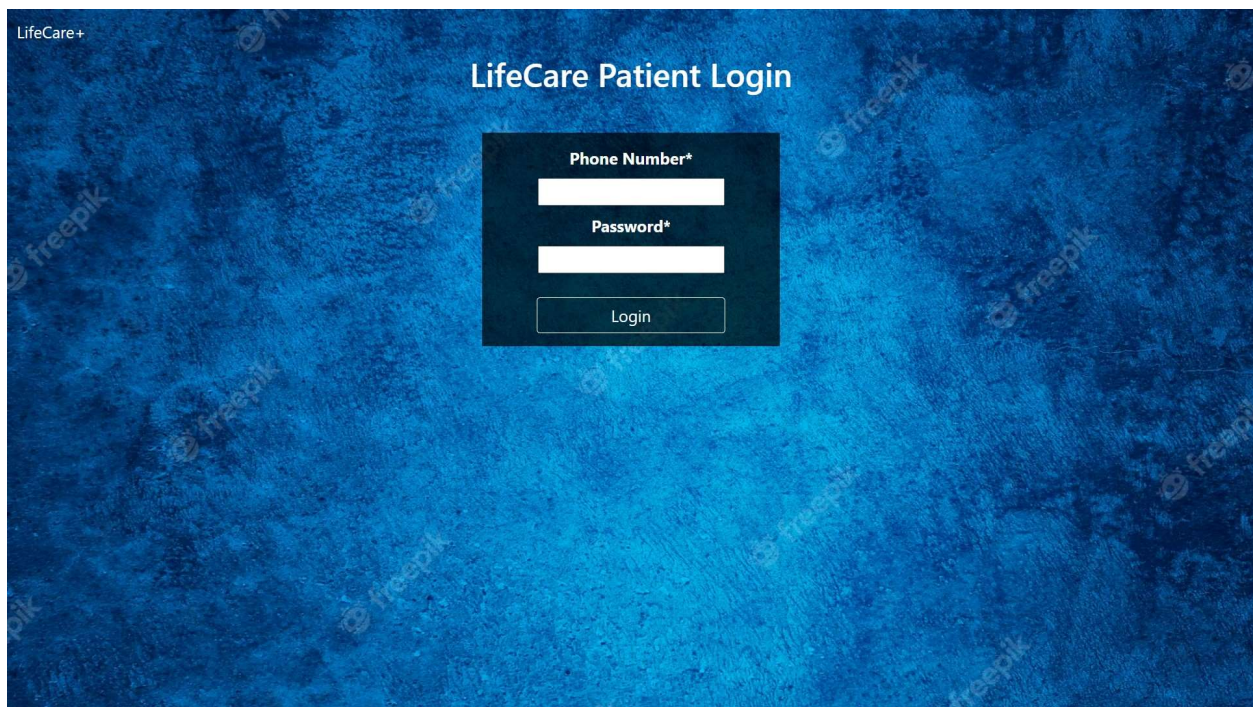
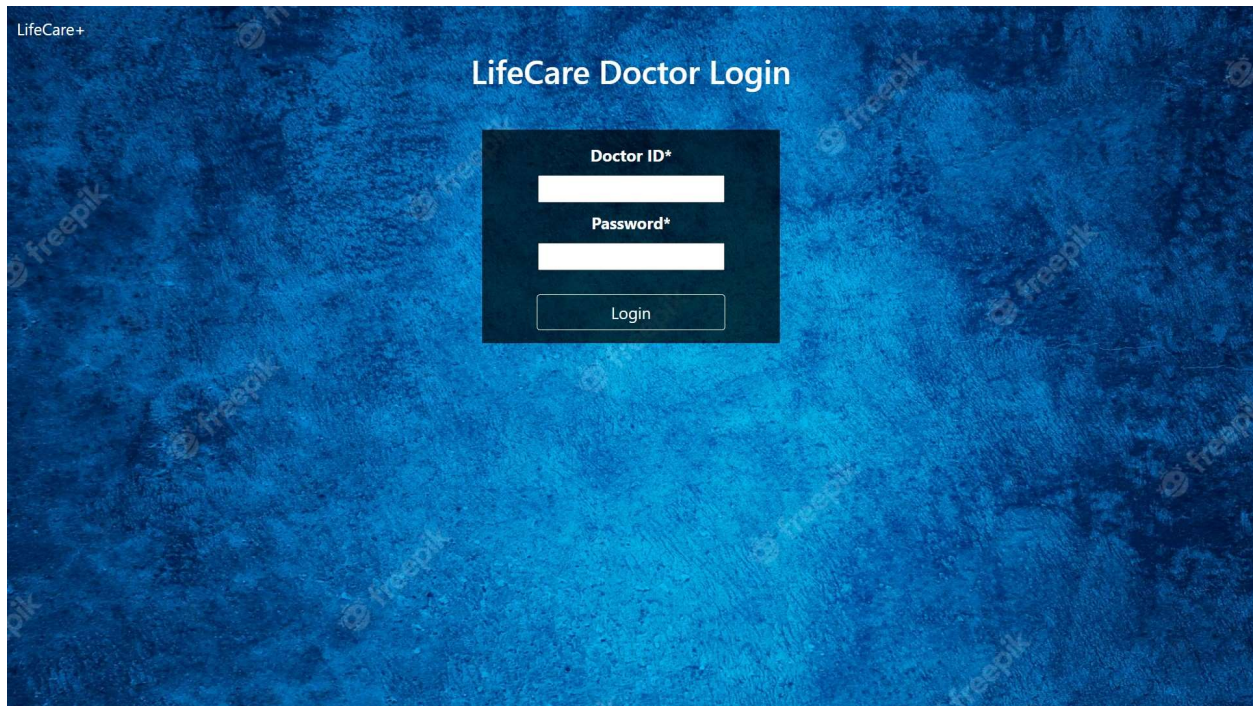


Fig – 6.2 : Patient login page

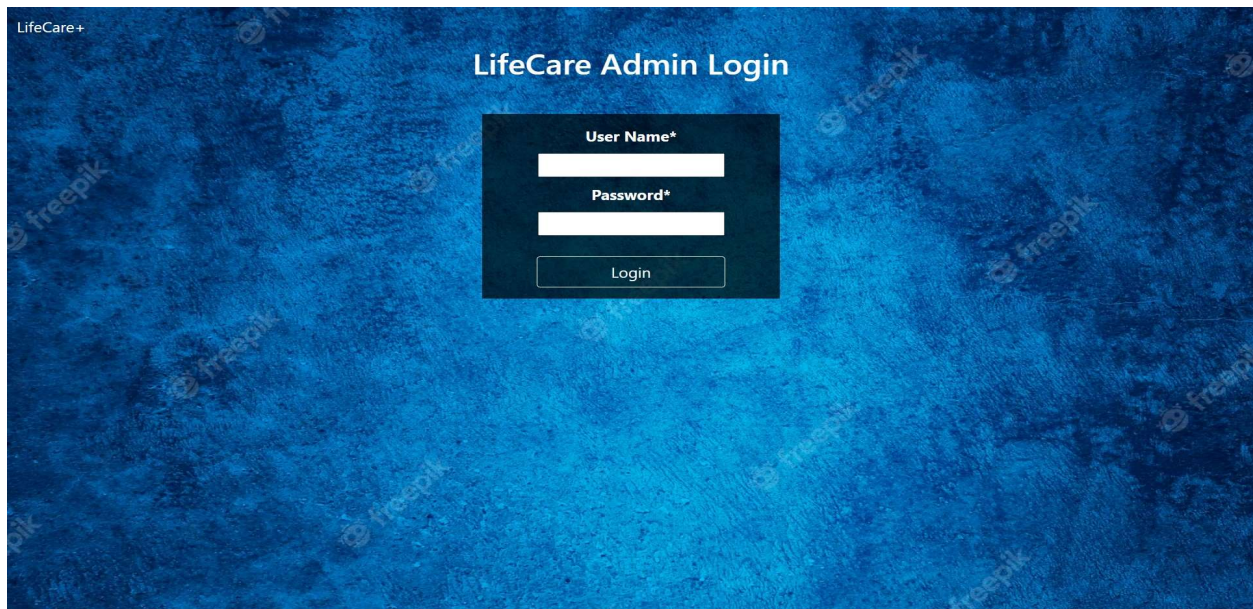
6.3 DOCTOR LOGIN:



The image shows the 'LifeCare Doctor Login' page. It features a dark blue background with a subtle texture. In the top left corner, the text 'LifeCare+' is visible. The main title 'LifeCare Doctor Login' is centered at the top. Below the title is a dark grey rectangular box containing the login form. The form has two input fields: 'Doctor ID*' and 'Password*', both with white text and white input areas. Below these fields is a 'Login' button with white text. The entire page is watermarked with 'freepik' and '©' symbols.

Fig – 6.3 : Doctor login page

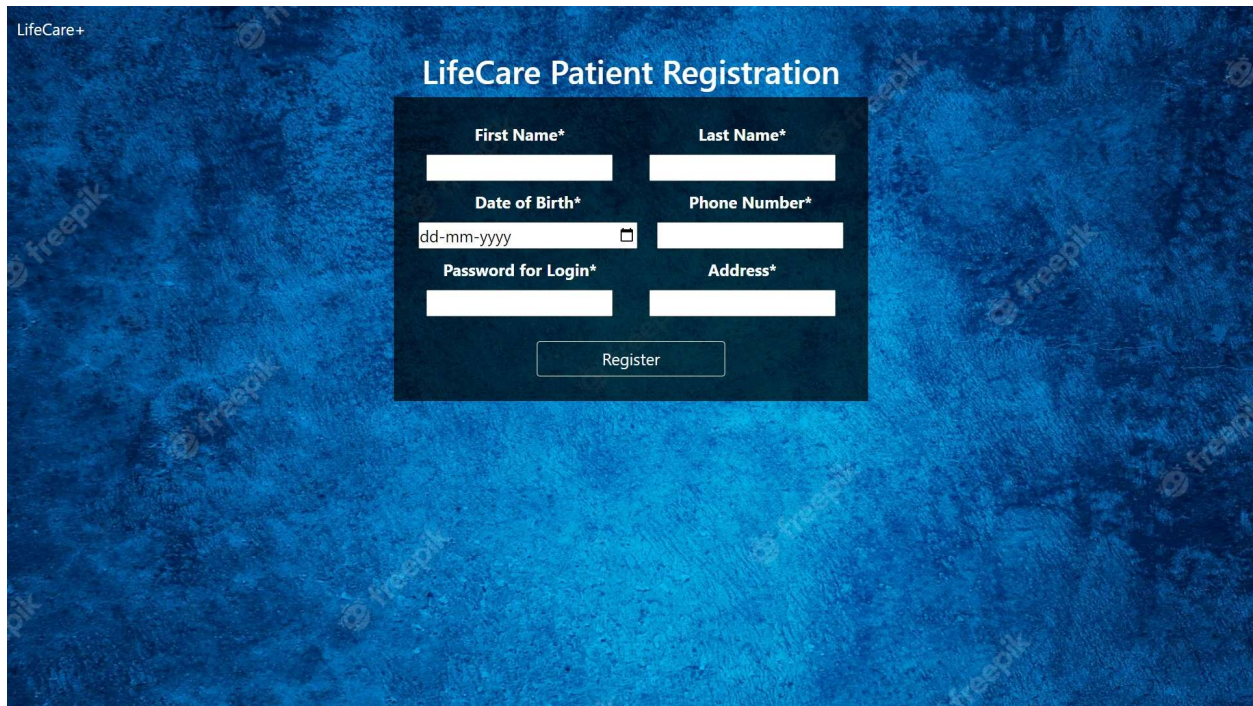
6.4 ADMIN LOGIN:



The image shows the 'LifeCare Admin Login' page. It features a dark blue background with a subtle texture. In the top left corner, the text 'LifeCare+' is visible. The main title 'LifeCare Admin Login' is centered at the top. Below the title is a dark grey rectangular box containing the login form. The form has two input fields: 'User Name*' and 'Password*', both with white text and white input areas. Below these fields is a 'Login' button with white text. The entire page is watermarked with 'freepik' and '©' symbols.

Fig – 6.4 : Admin login page

6.5 PATIENT REGISTRATION:




The image shows a patient registration form titled "LifeCare Patient Registration" on a dark blue background. The form is a white box with the following fields: "First Name*", "Last Name*", "Date of Birth*" (with a "dd-mm-yyyy" placeholder and a calendar icon), "Phone Number*", "Password for Login*", and "Address*". A "Register" button is at the bottom.

LifeCare+

LifeCare Patient Registration

First Name*

Last Name*

Date of Birth* 

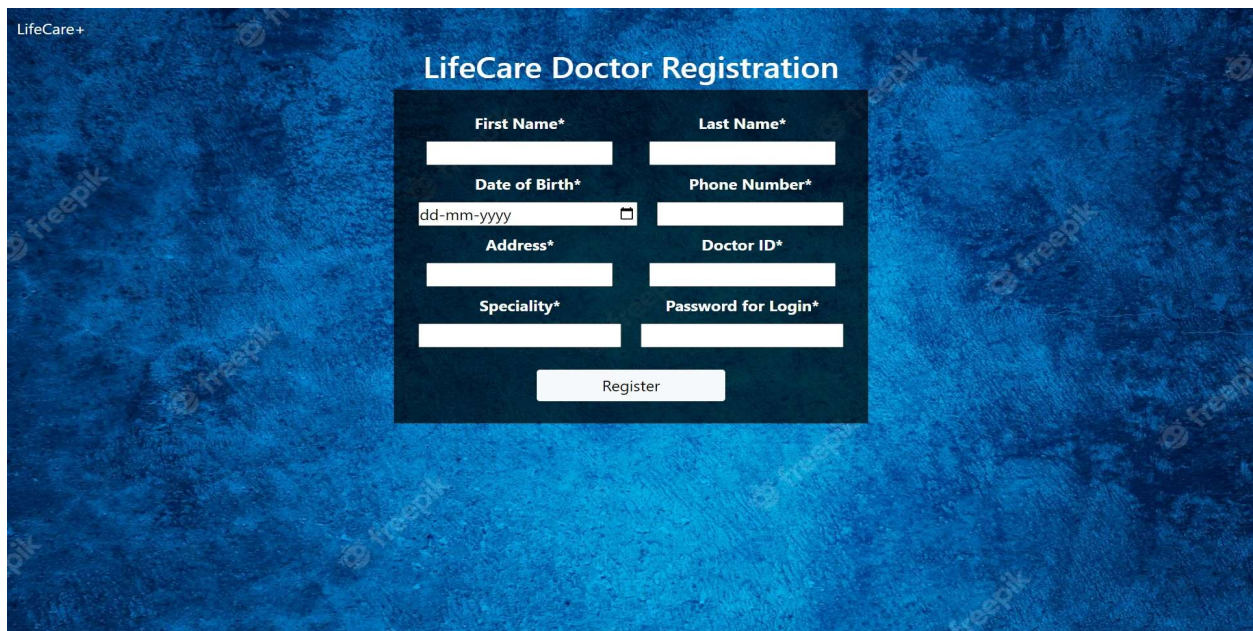
Phone Number*

Password for Login*

Address*

Fig – 6.5 : Patient registration page

6.6 DOCTOR REGISTRATION:




The image shows a doctor registration form titled "LifeCare Doctor Registration" on a dark blue background. The form is a white box with the following fields: "First Name*", "Last Name*", "Date of Birth*" (with a "dd-mm-yyyy" placeholder and a calendar icon), "Phone Number*", "Address*", "Doctor ID*", "Speciality*", and "Password for Login*". A "Register" button is at the bottom.

LifeCare+

LifeCare Doctor Registration

First Name*

Last Name*

Date of Birth* 

Phone Number*

Address*

Doctor ID*

Speciality*

Password for Login*

Fig – 6.6 : Doctor registration page

CHAPTER VII

CONCLUSION AND FUTURE ENHANCEMENTS

CONCLUSION AND FUTURE ENHANCEMENTS

CONCLUSION:

“DOCTOR PATIENT APPOINTMENT SYSTEM” is a web-based inventory management system developed using Python, Flask, SQLite, HTML, CSS, JavaScript, and Bootstrap. It aims to provide an efficient solution for managing inventory-related tasks, such as tracking products, managing stock levels, generating reports, and facilitating user interactions.

Python serves as the primary programming language for the backend development, allowing you to implement the core logic and functionality of the inventory management system. Flask, a lightweight web framework, provides the infrastructure for building web applications in Python, enabling you to handle routing, request handling, and response generation.

HTML, CSS, and JavaScript are essential frontend technologies that work together to create the user interface and enhance the user experience. HTML structures the content and defines the elements of your web pages, CSS styles and enhances the visual presentation, and JavaScript enables interactive functionality and dynamic behaviour.

Bootstrap, a frontend framework, is utilized to leverage pre-built components and responsive utilities, simplifying the process of creating a modern and visually appealing user interface. It ensures that your web application is accessible and responsive across different devices and screen sizes.

Overall, your project brings together a combination of technologies to create an inventory management system that is efficient, user-friendly, and visually appealing. It leverages the power of Python, Flask, SQLite, HTML, CSS, JavaScript, and Bootstrap to provide a comprehensive solution for managing inventory-related tasks.

FUTURE ENHANCEMENT :

- **Duplication:**

When a doctor or patient books an appointment, the appointment log is getting duplicated. The future work will making the log unique without duplication.

- **Integration with barcode scanners:**

Integrate your system with barcode scanners to streamline the process of adding or updating products in the inventory. This can save time and reduce errors by automatically retrieving product information from barcode scans.

- **Mobile application:**

Consider developing a mobile application for your inventory management system. A mobile app can provide on-the-go access to inventory data, allowing users to manage inventory, update stock levels, and perform other tasks using their smartphones or tablets.

CHAPTER VIII

REFERENCES

REFERENCES:

- [1] 1. Bailey NTJ. A study of queues and appointment systems in hospital out-patient departments, with special reference to waiting times. J Royal Stat Soc 1952;14:185–99
- [2] 2. Cayirli, T, E. Veral, and H. Rosen. (2006). Designing appointment scheduling systems for ambulatory care services. Health Care Management Science 9, 47–58.
- [3] 3. Arthur Hylton III and Suresh Sankara Narayanan “Application of Intelligent Agents in Hospital Appointment Scheduling System”, International Journal of Computer Theory and Engineering, Vol. 4, August 2012, pp. 625-630.
- [4] 4. Yeo Symey, Suresh Sankara Narayanan, Siti Nurafifah binti Sait “Application of Smart Technologies for Mobile Patient Appointment System”, International Journal of Advanced Trends in Computer Science and Engineering, august 2013
- [5] 5. Jagannath Aghav, Smita Sonawane, and Himanshu Bhambhlani “Health Track: Health Monitoring and Prognosis System using Wearable Sensors”, IEEE International Conference on Advances in Engineering & Technology Research 2014, pp. 1-5.
- [6] Flask Documentation: Official documentation for Flask, which is a lightweight web framework for Python. It provides detailed information about Flask's features, usage, and examples. You can refer to it for learning how to build web applications using Flask: Flask Documentation
- [7] SQLite Documentation: The official documentation for SQLite, which is a popular lightweight database management system. It provides detailed information about SQLite's features, SQL syntax, and usage. You can refer to it for understanding how to interact with the SQLite database in your project: SQLite Documentation
- [8] HTML Tutorial: A comprehensive tutorial on HTML (Hypertext Markup Language), which is the standard markup language for creating web pages. It covers the basics of HTML, tags, attributes, and structuring web content: HTML Tutorial - W3Schools
- [9] CSS Tutorial: A comprehensive tutorial on CSS (Cascading Style Sheets), which is used for styling and formatting web pages. It covers the fundamentals of CSS, selectors, properties, and styling techniques: CSS Tutorial - W3Schools

- [10] Bootstrap Documentation: Official documentation for Bootstrap, a popular CSS framework that provides pre-designed components and styles for building responsive web pages. It includes detailed information about Bootstrap's features, components, and usage. You can refer to it for utilizing Bootstrap in your project: Bootstrap Documentation
- [11] JavaScript Tutorial: A comprehensive tutorial on JavaScript, a scripting language that adds interactivity and dynamic behavior to web pages. It covers the basics of JavaScript, variables, functions, DOM manipulation, and event handling: JavaScript Tutorial - W3Schools
- [12] Python Tutorial: A beginner-friendly tutorial on Python, a versatile programming language used in your project. It covers the fundamentals of Python, syntax, data structures, functions, and object-oriented programming: Python Tutorial - W3Schools
- [13] GitHub: A web-based platform for version control and collaborative development. You can find numerous open-source projects related to inventory management on GitHub. Exploring these projects can provide you with valuable insights, code examples, and best practices. Visit GitHub and search for relevant repositories using keywords like "inventory management" or "inventory system."