HEALTHCARE INFORMATION SYSTEM

A PROJECT REPORT

(21CSC205P – Database Management Systems)

Submitted by

SAM FREDRICK [RA2211030010072]

CHITHRESH A [RA2211030010073]

M REDDY ROHITH [RA2211030010078]

Under the Guidance of

Dr. KAYALVIZHI R

Assistant Professor, Department of Networking and Communications

in partial fulfillment of the requirements for the degree of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING
specialization in Cyber Security

DEPARTMENT OF NETWORKING AND COMMUNICATIONS

COLLEGE OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR– 603 203

MAY 2024

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY
KATTANKULATHUR–603 203
BONAFIDE CERTIFICATE

Certified to be the bonafide work done by **Sam Fredrick (RA2211030010072)** of II year/IV Sem B. Tech Degree Course in the Project Course – **21CSC205P Database Management Systems** in **SRM INSTITUTE OF SCIENCE AND TECHNOLOGY**, Kattankulathur for the academic year 2023-2024.

Date:

Faculty in Charge
Dr. Kayalvizhi R
Assistant Professor
Department of Networking and Communications
SRM Institute of Science and Technology
Kattankulathur,
Chennai – 603 203

HEAD OF THE DEPARTMENT
Dr. Annapurani K
Professor & Head
Department of Networking and Communications
SRM Institute of Science and Technology
Kattankulathur,
Chennai – 603 203

ABSTRACT

The healthcare information system is a comprehensive web application developed using the Django framework in Python, aimed at streamlining and digitizing various processes within a healthcare facility to enhance efficiency and patient care. It encompasses functionalities like patient registration, employee management, appointment scheduling, pharmacy operations, prescription management, and laboratory test tracking. Leveraging Django's Model-View-Template architecture, models represent entities such as patients, employees, appointments, pharmacy items, prescriptions, and tests. Views handle application logic, facilitating operations like registering patients, creating appointments, generating pharmacy bills, recording prescriptions, and managing test results. The system incorporates user authentication and authorization for secure access. It provides a user-friendly interface for patients to schedule appointments, view prescription history, and access test reports, while healthcare professionals can manage patient records, create prescriptions, order tests, and track pharmacy inventory. Overall, the system streamlines processes, reduces manual effort, and promotes better coordination among departments, contributing to improved patient care and operational efficiency.

PROBLEM STATEMENT

The primary purpose of the healthcare information system is to digitize and automate various processes within a healthcare facility, thereby enhancing operational efficiency, streamlining workflows, and improving overall patient care. By incorporating modern technology into the healthcare ecosystem, the system aims to address the challenges posed by traditional paper-based or disparate systems, which often lead to inefficiencies, data inconsistencies, and delays in information sharing among different departments and healthcare professionals.

The scope of the healthcare information system encompasses a wide range of functionalities, including patient registration, employee management, appointment scheduling, pharmacy operations, prescription management, and laboratory test tracking. It serves as a centralized platform for storing and accessing patient records, facilitating communication between healthcare professionals, and enabling seamless collaboration among various departments within the healthcare facility. Additionally, the system provides patients with a convenient interface to schedule appointments, view their medical history, and access test results and prescriptions.

The primary target audience for the healthcare information system includes healthcare facilities such as hospitals, clinics, and medical centers. It caters to the needs of healthcare professionals, including doctors, nurses, pharmacists, laboratory technicians, and administrative staff. Furthermore, the system is designed to benefit patients by providing them with a user-friendly platform to access their medical information and interact with the healthcare facility.

TABLE OF CONTENTS

CHAPTER 1

PROBLEM UNDERSTANDING, IDENTIFICATION OF ENTITY AND REALATIONSHIPS, CONSTRUCTION OF DB USING ER MODEL

The healthcare industry has long grappled with the challenges posed by traditional paper-based systems or disparate software solutions. These outdated approaches have given rise to a myriad of inefficiencies, data inconsistencies, and delays in the exchange of crucial information among different departments and healthcare professionals. The ramifications of these shortcomings are far-reaching and can potentially compromise the quality of patient care.

One of the most pressing issues is the difficulty in accessing and managing patient records across departments. With paper-based records or siloed systems, healthcare professionals often struggle to obtain a comprehensive view of a patient's medical history, test results, and previous treatments. This lack of centralized information can lead to uninformed decision-making, redundant tests, and potential medical errors.

Furthermore, the absence of real-time data availability hinders the ability of healthcare providers to respond promptly to evolving situations. Delayed access to critical information, such as changes in a patient's condition or updated laboratory results, can have severe consequences, particularly in emergency scenarios or time-sensitive treatments.

Manual data entry and management processes are also prone to human errors, further exacerbating the risk of inaccuracies in patient records. Transcription errors, illegible

handwriting, and inconsistent data formats can result in incorrect diagnoses, improper treatment plans, or even life-threatening situations.

Effective communication and coordination among healthcare professionals are vital for delivering seamless patient care. However, traditional systems often lack robust mechanisms for collaboration and information sharing, leading to fragmented communication and potential gaps in the continuum of care.

Moreover, inefficient scheduling and management of appointments and resources can result in lengthy wait times, underutilized resources, and decreased patient satisfaction. Manual scheduling processes are time-consuming and susceptible to errors, further compounding the challenges faced by healthcare facilities.

To address these multifaceted challenges, the healthcare information system aims to provide a comprehensive, digitized solution that streamlines processes, enhances data accuracy, and facilitates seamless collaboration among various stakeholders within the healthcare facility. By leveraging modern technology and embracing a centralized, integrated approach, the system seeks to revolutionize the way healthcare services are delivered, ultimately leading to improved patient outcomes and operational efficiency.

The development of a robust and effective healthcare information system hinges on accurately identifying the key entities and their relationships within the healthcare domain. This process serves as the foundation for designing an efficient and scalable database structure that can effectively model and store the relevant data.
Through a comprehensive analysis of the problem domain and the requirements of the healthcare information system, the following critical entities were identified:

a. **Patient:** This entity represents the individuals seeking medical care and serves as the cornerstone of the entire system. It encapsulates essential attributes such as name, date of birth, contact information, medical history, and other relevant personal and medical details.

b. **Employee:** This entity represents the healthcare professionals who provide medical services and support within the facility. It includes doctors, nurses, pharmacists, laboratory technicians, and administrative staff. Attributes associated with this entity typically include name, profession, department, contact details, and other relevant professional information.

c. **Department:** This entity represents the various specialized units or divisions within the healthcare facility, such as cardiology, radiology, pediatrics, or oncology. It helps organize and classify employees based on their areas of expertise and responsibilities.

d. **Appointment:** This entity serves as a bridge between patients and healthcare professionals, representing scheduled consultations, check-ups, or procedures. It links the Patient and Employee entities, capturing essential details such as date, time, and purpose of the appointment.

e. **Prescription:** This entity models the prescriptions issued by healthcare professionals, typically doctors, to patients. It links the Patient, Employee, and Medication entities, capturing information such as medication details, dosage instructions, and duration.

f. **Medication:** This entity represents the various medications available within

the healthcare facility, including their names, dosages, forms (e.g., tablets, capsules), and other relevant information.

g. **Laboratory Test:** This entity encompasses the different types of laboratory tests that can be ordered for patients, such as blood tests, imaging studies, or pathology examinations. It captures test names, reference ranges, and ultimately, the test results.

h. **Pharmacy Item:** This entity represents the products and supplies available in the facility's pharmacy, including medications, medical equipment, and other healthcare-related items. It typically contains attributes such as name, price, category, and current stock levels.

i. **Pharmacy Bill:** This entity models the bills generated for patients after purchasing items from the pharmacy. It links the Patient and Pharmacy Item entities, capturing details such as the items purchased, quantities, and total cost.

By identifying these entities and their relationships, the healthcare information system can effectively model and store the relevant data, enabling efficient data management, retrieval, and analysis. This structured approach lays the groundwork for building a robust database that can support the various functionalities and workflows of the healthcare facility.

To effectively design and implement the database for the healthcare information system, an Entity-Relationship (ER) model was constructed. The ER model serves as a visual representation of the entities, their attributes, and the relationships

between them, providing a conceptual blueprint for the database structure.

The ER model for this project was developed based on the identified entities and their relationships, as discussed in the previous section. It illustrates the relationships between entities, such as one-to-one, one-to-many, or many-to-many, and specifies the cardinalities and constraints associated with these relationships.

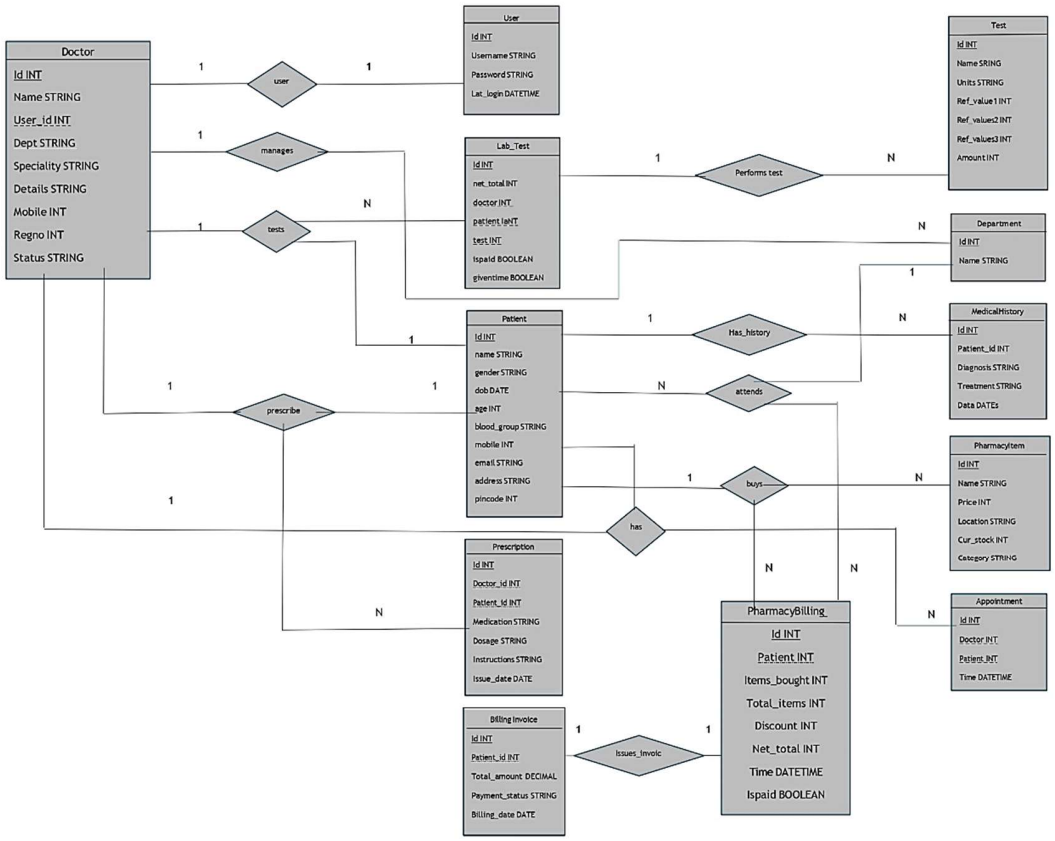


Fig.1.1 ER diagram of the project

For example, the ER model would depict the Patient entity with attributes like name, date of birth, contact information, and medical history. It would also show the relationship between the Patient entity and the Appointment entity, indicating that

one patient can have multiple appointments, but each appointment is associated with only one patient (a one-to-many relationship).

Similarly, the ER model would represent the relationship between the Employee entity and the Department entity, where one department can have multiple employees, but each employee belongs to only one department (another one-to-many relationship).

The ER model also captures the relationships between entities like Prescription, Medication, and Laboratory Test, which may involve many-to-many relationships. For instance, one prescription can include multiple medications, and one medication can be present in multiple prescriptions.

By constructing the ER model, the database design process becomes more structured and organized. It helps identify potential data redundancies, ensures data integrity, and facilitates the implementation of appropriate constraints and normalization techniques.

The ER model served as a blueprint for the subsequent implementation of the database using Django's Object-Relational Mapping (ORM) approach. The identified entities were mapped to Django models, and the relationships were translated into appropriate model fields and relationships (e.g., ForeignKey, ManyToManyField).

The ER model played a crucial role in ensuring a well-designed and efficient database structure for the healthcare information system, enabling effective data management, querying, and reporting capabilities.

CHAPTER 2

DESIGN OF REATIONAL SCHEMAS, CREATION OF DATABASE TABLES FOR THE PROJECT

The design of relational schemas is a crucial step in developing a robust and efficient database system. It involves analyzing the requirements and identifying the entities, attributes, and relationships within the system. The goal is to create a well-structured and normalized database schema that minimizes redundancy, ensures data integrity, and facilitates efficient data retrieval and manipulation.

In the context of the healthcare information system, the relational schema design process involved identifying the main entities, such as patients, employees, departments, appointments, prescriptions, lab tests, and pharmacy items. Each entity was carefully analyzed to determine its attributes, such as patient name, date of birth, address, employee profession, department name, and test details.

Relationships between entities were also established. For instance, a patient can have multiple appointments with different doctors, and an employee can work in a specific department. These relationships were modeled using primary and foreign keys to maintain referential integrity and ensure data consistency.

Normalization techniques were applied to the initial schema design to eliminate redundancies and ensure data integrity. This process involved breaking down the entities into smaller, more manageable tables and establishing relationships between them. By normalizing the schema, we minimized the risk of update anomalies and ensured that data was stored in a structured and efficient manner.

Throughout the design process, careful consideration was given to business rules and requirements to ensure that the schema accurately represented the real-world scenarios and supported the necessary operations and queries. Regular consultations with stakeholders and domain experts were conducted to validate the schema design and ensure its alignment with the project's objectives.

The relational schema design process involved iterative refinement and optimization to ensure that the final schema was efficient, scalable, and capable of handling future requirements and growth. By following industry-standard practices and leveraging database normalization techniques, we created a robust and reliable foundation for the healthcare information system.

Patient:

Attributes	Data Type	Constraints
Id	int	Primary key
name	Varchar(50)	NOT NULL
Gender	Varchar(50)	NOT NULL
DOB	Date	NOT NULL
age	int	NOT NULL
Blood_group	Varchar(50)	NOT NULL
Mobile	Int	NOT NULL
email	Varchar(50)	NOT NULL
Address	Varchar(50)	NOT NULL
Reddate	Date	NOT NULL

Patient (id, name, gender, dob, age, blood_group, mobile, email, address, pincode, regdate)

Fig.2.1 Relationship Schema of Patient

DOCTOR:

Attributes	Data Type	Constraints
Id	int	Primary key
name	Varchar(50)	NOT NULL
user_id	Int	Foreign key
Dept	Int	Foreign key
Specilality	Varchar(50)	NOT NULL
Details	Varchar(50)	NOT NULL
Mobile	Int	NOT NULL
Status	bool	NOT NULL

Doctor (id, name, user_id, dept, specilatity, details, mobile, regno, status)

Fig.2.2 Relationship Schema of Doctor

User:

Attributes	Data Type	Constraints
Id	int	Primary key
Username	Varchar(50)	NOT NULL
Password	Varchar(50)	NOT NULL
Last_login	datetime	Defaut = NOW()

User (id, username, password, last_login)

Fig.2.3 Relationship Schema of User

Appointment:

Attributes	Data Type	Constraints
Id	int	Primary key
doctor	int	Foreign key
Patient	Int	Foreign key
Time	datetime	Defaut = NOW()

Appointment (id, doctor, patient, time)

Fig.2.4 Relationship Schema of Appointment

Lab Test:

Attributes	Data Type	Constraints
Id	int	Primary key
doctor	int	Foreign key
Test	Int	Foreign key
Net total	Int	NOT NULL
Patient	Int	Foreign key
Ispaid	Bool	NOT NULL
Time	datetime	Defaut = NOW()

Lab_Test (id, net_total, doctor, patient, test, ispaid, time)

Fig.2.5 Relationship Schema of Lab Test

Department:

Attributes	Data Type	Constraints
Id	int	Primary key
Name	Varchar(50)	NOT NULL

Department (id, name)

Fig.2.6 Relationship Schema of Department

Test:

Attributes	Data Type	Constraints
Id	int	Primary key
Name	Varchar(50)	NOT NULL
Units	Varchar(50)	NOT NULL
Ref_values1	Int	NOT NULL
Ref_valeus2	Int	NOT NULL
Ref_values3	Int	NOT NULL
Amount	Int	NOT NULL

Test (id, name, units, ref_value1, ref_value2, ref_value3, amount)

Fig.2.7 Relationship Schema of Tests

Pharmacy Billing:

Attributes	Data Type	Constraints
Id	int	Primary key
Items_bought	int	Foreign key
Total_items	Int	NOT NULL
Net total	Int	NOT NULL
Patient	Int	Foreign key
Total_items	Int	NOT NULL
Ispaid	Bool	NOT NULL
Time	datetime	Defaut = NOW()

PharmacyBilling (id, patient, items_bought, total_items, discount, net_total, time, ispaid)

Fig.2.8 Relationship Schema of Pharmacy Billing

PharmacyItem:

Attributes	Data Type	Constraints
Id	int	Primary key
Name	Varchar(50)	NOT NULL
Price	Int	NOT NULL
Location	Varchar(50)	NOT NULL
Curr_stock	Int	NOT NULL
Category	Varchar(50)	NOT NULL

PharmacyItem (id, name, price, location, cur_stock, category)

Fig2.9 Relationship Schema of PharmacyItem

Prescription:

Attributes	Data Type	Constraints
Id	int	Primary key
doctor	int	Foreign key
Medication	Varchar(50)	NOT NULL
Dosage	Varchar(50)	NOT NULL
Patient	Int	Foreign key
Instruction	Varchar(50)	NOT NULL
Time	datetime	Defaut = NOW()

Prescription (id, doctor_id, patient_id, medication, dosage, instructions, issue_date)

Fig.2.10 Relationship Schema of Prescription

After designing the relational schemas, the next step was to create the actual database tables that would store and manage the data for the healthcare information system. This process involved translating the conceptual schema design into a physical implementation within the chosen database management system (DBMS).

Using Django's Object-Relational Mapping (ORM) functionality, we defined Python classes representing the database models. Each class corresponded to a table in the database, and its attributes represented the columns or fields within that table. Django's ORM provided a powerful abstraction layer, allowing us to work with Python code while handling the underlying database operations seamlessly.

The Patient model was created to store patient-related information, such as name,

date of birth, gender, contact details, address, age, guardian, email, marital status, and profession as shown in Figure 2.1. This model served as the central entity for managing patient data within the system.

The Employee model was designed to capture details about healthcare professionals, including their user account information, entry number, profession (e.g., doctor, nurse, lab technician), joining date, personal details (name, gender, blood group, date of birth, religion, nationality), contact information, department association, education, and experience details as shown in Figure 2. 2..

The Dept model represented the various departments within the healthcare facility, such as radiology, cardiology, and pediatrics. This model facilitated the organization and management of employees based on their respective departments as shown in Figure 2.6.

To handle appointments, the Appointment model was created, establishing a relationship between patients and doctors, along with the scheduled date and time for the appointment as shown in Figure 2.4.

For the pharmacy component, the PharmacyItem model stored information about available pharmaceutical items, including their names, prices, locations, current stock levels, and categories as shown in Figure 2.9. The PharmacyBill model managed billing information, linking patients with the purchased items and tracking the total cost and purchase time as shown in Figure 2.8.

The Prescription model captured prescription details, such as the patient, prescribing doctor, medications, diagnosis, remarks, and the time the prescription was issued.

For laboratory tests, the Test model defined the available tests, their names, units of measurement, reference ranges, and associated costs as shown in Figure 2.5.

The LabTestResult model linked patients, doctors, and the specific tests performed, while the LabTestResultItem model stored the individual test results and their values.

Throughout the table creation process, careful attention was paid to defining appropriate data types, constraints, and relationships between tables. Foreign key relationships were established to maintain referential integrity and ensure data consistency across related tables.

Django's built-in migration system was utilized to manage and apply database schema changes systematically. Migrations allowed us to track and version-control the database schema, enabling seamless updates and deployments without risking data loss or corruption.

Additionally, indexing strategies were implemented to optimize query performance and ensure efficient data retrieval. Indexes were created on frequently queried columns or combinations of columns to enhance the overall system's responsiveness and scalability.

By leveraging Django's ORM and following best practices for database design and implementation, we successfully created the necessary database tables to support the various functionalities of the healthcare information system, ensuring data integrity, consistency, and efficient management of the system's critical information.

CHAPTER 3

COMPLEX QUERIES BASED ON THE CONCEPTS OF CONTRAINTS, SETS, JOINS, VIEWS, TRIGGERS AND CURSORS

In the healthcare information system, various complex querying techniques were employed to ensure efficient data retrieval, manipulation, and maintenance across multiple tables. These techniques not only enhance the system's performance but also contribute to maintaining data integrity, security, and adherence to business rules and requirements.

Constraints:

Constraints play a crucial role in maintaining data integrity and ensuring the accuracy and consistency of the stored information. Throughout the system, various constraints were implemented to enforce data validity and business rules. For instance, the Patient model incorporated a NOT NULL constraint on the Name field, ensuring that every patient record must have a name assigned. Additionally, unique constraints were applied to fields like Email to prevent duplicate entries, maintaining data uniqueness.

Foreign key constraints were also employed to maintain referential integrity between related tables. For example, the relationship between the Patient and Appointment tables was established through a foreign key constraint, ensuring that each appointment is associated with a valid patient record. These constraints not only maintain data consistency but also simplify querying by eliminating the need for additional checks and validations.

Set Operations:

Set operations, such as UNION, INTERSECT, and EXCEPT, were leveraged to combine and manipulate data from multiple tables or queries efficiently. These operations enabled the retrieval of comprehensive and customized results, catering to diverse reporting and analytical requirements within the healthcare domain.

For instance, the UNION operator was used to combine the results of two queries, one retrieving patients with upcoming appointments and another retrieving patients with past appointments. This operation provided a consolidated view of all patients with scheduled appointments, facilitating better resource allocation and planning.

The INTERSECT operator proved invaluable in identifying patients who had both upcoming appointments and pending lab test results, enabling proactive communication and coordination between healthcare professionals for efficient patient management.

Joins:

Joins are fundamental to querying relational databases, and they played a pivotal role in combining data from multiple tables based on related columns or foreign key relationships. Various types of joins, including inner, outer (left, right, and full), and cross joins, were employed to retrieve and present comprehensive information within the healthcare system.

For example, an inner join between the Patient, Appointment, and Employee tables enabled the retrieval of patient names, appointment dates, and the names of the attending doctors in a single query. This join operation facilitated the generation of

comprehensive appointment reports, providing healthcare professionals with a holistic view of scheduled consultations.

Outer joins were particularly useful in scenarios where records from one table did not have matching records in the other table. For instance, a left outer join between the Patient and Appointment tables ensured that all patient records were retrieved, regardless of whether they had scheduled appointments or not. This approach allowed for comprehensive patient data analysis and reporting.

Views:

Views, which are virtual tables derived from one or more base tables or other views, played a crucial role in simplifying complex queries, providing logical representations of data, and enforcing data security and access control within the system.

For instance, a view was created to display only the relevant patient information accessible to nurses, without exposing sensitive data like contact details or medical history. This view ensured that nurses could access the necessary patient information while maintaining data privacy and adhering to access control policies.

Another view was implemented to consolidate appointment data from multiple tables, including patient details, doctor information, and appointment timestamps. This view enabled the creation of reusable query structures, making it easier to maintain and optimize complex queries over time, ultimately improving the system's performance and maintainability.

Triggers:

Triggers, which are special types of stored procedures that automatically execute when specific events occur, were implemented to enforce business rules, maintain data integrity, and automate certain tasks within the healthcare system. One example of a trigger implementation was the automatic update of the current stock level of a pharmacy item whenever a new bill was generated. This trigger ensured accurate inventory tracking by decrementing the stock levels of the purchased items, preventing potential discrepancies and providing real-time stock information for efficient inventory management.

Another trigger was employed to log changes made to patient records, providing an audit trail for monitoring and compliance purposes. This trigger captured essential details such as the user who made the change, the timestamp of the change, and the specific fields that were modified. This audit trail not only facilitated effective record-keeping but also enabled compliance with regulatory requirements and data governance policies.

Cursors:

Cursors, which are database objects that enable row-by-row processing of query results, were employed for tasks that required iterating through large result sets or performing complex data manipulations that could not be efficiently accomplished using set-based operations alone.

One notable use case for cursors was the generation of personalized reports or summaries based on specific criteria, such as medical conditions or treatment history. By iterating through a set of patient records using a cursor, the system could

perform complex calculations, apply intricate filtering criteria, and generate customized reports tailored to individual patient needs. Additionally, cursors were utilized in scenarios where data manipulation or transformation required processing large datasets that exceeded the available system memory. By processing the data in a row-by-row manner, cursors minimized memory consumption and enabled efficient handling of large datasets, ensuring optimal system performance and scalability.

By leveraging these complex query concepts and advanced querying techniques, the healthcare information system achieved efficient data retrieval, manipulation, and maintenance across multiple tables. These techniques not only enhanced the system's performance but also contributed to maintaining data integrity, security, and adherence to business rules and requirements, ultimately supporting the delivery of high-quality healthcare services.

CHAPTER 4

ANALYZING THE PITFALLS, IDENTIFYING THE DEPENDENCIES AND APPLYING NORMALIZATIONS

In the design and development of the healthcare information system, careful consideration was given to analyzing potential pitfalls, identifying dependencies, and applying normalization techniques. These processes are crucial for creating a well-structured and efficient database schema that supports data integrity, minimizes redundancy, and ensures the overall reliability and performance of the system.

Analyzing the Pitfalls:

One of the critical steps in the database design process was to analyze potential pitfalls that could arise due to improper data modeling or inadequate normalization. These pitfalls, if left unaddressed, can lead to various issues that can undermine the system's reliability and performance.

Repeating groups, where a single entity contains multiple instances of the same type of data, were identified as a common pitfall. For example, storing multiple phone numbers or email addresses for a single patient within the same table can lead to data redundancy, making updates and deletions cumbersome and error-prone. To mitigate this issue, separate tables were introduced to store repeating groups, such as a "Patient Contact Details" table, which maintained a one-to-many relationship with the main "Patient" table.

Another pitfall that was analyzed was the presence of derived data, where certain attribute values could be calculated or derived from other attributes within the same

table. For instance, storing a patient's age in the database, when it can be easily calculated from their date of birth and the current date, can lead to potential inconsistencies and data integrity issues. To address this, derived data was eliminated from the database schema, and calculations were instead performed dynamically when needed, either within the application logic or through database functions or views.

Identifying Dependencies:

The identification and understanding of dependencies played a crucial role in the design and normalization of the healthcare information system's database schema. By recognizing these dependencies, the design team was able to create a well-structured and efficient database that minimized data redundancy and update anomalies.

Functional dependencies, where the value of an attribute (or a set of attributes) uniquely determines the value of another attribute within the same table, were identified and addressed. For example, in the healthcare system, a patient's name and date of birth may uniquely determine their age, creating a functional dependency. To eliminate this dependency, the age attribute was removed from the "Patient" table, and the age calculation was performed dynamically based on the patient's date of birth.

Multivalued dependencies, which arise when a single attribute value can have multiple corresponding values for another attribute, were also identified and addressed. For instance, a patient may have multiple phone numbers or a doctor may treat multiple patients, leading to multivalued dependencies. To handle these dependencies, separate tables were introduced, such as a "Patient Contact Details" table for storing multiple phone numbers and a "Doctor-Patient Relationship" table

for maintaining the many-to-many relationship between doctors and patients.

Transitive dependencies, where a functional dependency exists between two attributes, and one of those attributes is functionally dependent on a third attribute, were carefully analyzed and addressed. For example, if a patient's address is dependent on their city, and the city is dependent on the state, then there is a transitive dependency between the patient's address and the state. To eliminate this dependency, separate tables were created for addresses, cities, and states, with appropriate foreign key relationships stablished between them.

Applying Normalization Techniques:

Based on the analysis of pitfalls and dependencies, normalization techniques were applied to the database schema of the healthcare information system. Normalization involves organizing data into tables and defining relationships between them to minimize redundancy and ensure data integrity. The following normalization techniques were applied:

a. **First Normal Form (1NF):** Each table in the database was designed to satisfy the requirements of 1NF, which ensures that each column contains atomic values, and there are no repeating groups. For example, patient details like name, date of birth, and gender were stored in separate columns, avoiding repeating groups.

b. **Second Normal Form (2NF):** Tables were further normalized to satisfy 2NF, which eliminates partial dependencies by ensuring that each non-key attribute is fully functionally dependent on the primary key. For instance, in the appointment table, appointment details such as date and time were moved to a separate table linked by the appointment ID, removing any partial

dependencies.

c. **Third Normal Form (3NF):** To achieve 3NF, tables were designed to eliminate transitive dependencies, where non-key attributes depend on other non-key attributes. For example, the patient table was normalized to remove transitive dependencies such as age depending on date of birth, ensuring that each attribute directly depends only on the primary key.

d. **Boyce-Codd Normal Form (BCNF):** In some cases, tables were further normalized to BCNF, which ensures that every determinant is a candidate key. This was achieved by identifying and resolving dependencies to ensure that each non-trivial functional dependency is a superkey.

By applying these normalization techniques, the database schema of the healthcare information system was optimized for efficient data storage, retrieval, and manipulation. The normalized schema minimized data redundancy, reduced the risk of update anomalies, and ensured the overall reliability and performance of the system. Additionally, the normalization process facilitated easier maintenance and scalability of the system as it grows and evolves over time.

CHAPTER 5

IMPLEMENTATION OF CONCURRENCY CONTROL AND RECOVERY MECHANISMS

In the design and development of a healthcare information system, ensuring data consistency, reliability, and recoverability are paramount. Concurrency control mechanisms are essential for managing simultaneous access to shared data by multiple users, while recovery mechanisms are crucial for restoring the system to a consistent state in case of failures. This chapter discusses the implementation of concurrency control and recovery mechanisms in the healthcare information system, focusing on techniques to maintain data integrity and recoverability.

Concurrency Control Mechanisms:

Concurrency control ensures that transactions executed concurrently do not interfere with each other, preserving data consistency and integrity. In the healthcare information system, various concurrency control mechanisms are implemented to manage concurrent access to patient records, appointments, and other critical data.

a. **Lock-Based Concurrency Control:**

   Lock-based mechanisms are commonly used to prevent conflicts between transactions accessing shared data. In the healthcare system, different types of locks, such as exclusive locks and shared locks, are employed to control access to resources. For example, when a transaction updates a patient's medical record, it acquires an exclusive lock on the corresponding database rows to prevent other transactions from modifying the same data simultaneously.

b. **Timestamp-Based Concurrency Control:**

   Timestamp-based concurrency control uses timestamps to order transactions and resolve conflicts. Each transaction is assigned a unique timestamp, and transactions are executed based on their timestamps. In the healthcare system, transactions accessing patient records or scheduling appointments are ordered based on their timestamps to ensure sequential execution and prevent conflicts.

c. **Multi version Concurrency Control (MVCC):**

   MVCC allows concurrent transactions to access different versions of data, ensuring read consistency without blocking write operations. In the healthcare system, MVCC is implemented to support concurrent read access to patient records while ensuring that write operations do not conflict with read operations. Each transaction sees a consistent snapshot of the database at a specific point in time, eliminating the need for locking and improving concurrency.

Recovery Mechanisms:

Recovery mechanisms ensure that the healthcare information system can recover from failures and restore data to a consistent state. In the event of hardware failures, software crashes, or other disruptions, recovery mechanisms are essential for preserving data integrity and minimizing downtime.

a. **Transaction Logging:**

   Transaction logging involves recording all changes made by transactions in a log file before they are applied to the database. In the healthcare system,

transaction logging ensures that every update, insertion, or deletion operation is logged, allowing the system to recover from failures by replaying the logged transactions and restoring the database to a consistent state.

b. **Checkpoints:**

   Checkpoints are periodic markers in the transaction log that indicate a consistent state of the database. In the healthcare system, checkpoints are implemented to reduce the time required for recovery by providing a known point from which the system can start the recovery process. Checkpoints flush all modified data to disk and update the transaction log, ensuring durability and recoverability.

c. **Undo and Redo Operations:**

   Undo and redo operations are used during recovery to either rollback or reapply changes made by transactions. In the healthcare system, undo operations are performed to reverse the effects of incomplete transactions or aborted transactions, while redo operations are used to reapply committed transactions that were not saved to disk before a failure occurred.

Concurrency control and recovery mechanisms play a vital role in ensuring the reliability, consistency, and recoverability of a healthcare information system. By implementing lock-based, timestamp-based, and multi version concurrency control mechanisms, the system can manage concurrent access to shared data while maintaining data integrity. Additionally, transaction logging, checkpoints, and undo/redo operations enable the system to recover from failures and restore data to a consistent state, ensuring uninterrupted operation and minimizing downtime. Overall, the implementation of these mechanisms is essential for the successful deployment and operation of a healthcare information system in a dynamic and demanding environment.

CHAPTER 6
CODE FOR THE PROECT

In this chapter, we delve into the implementation details of the healthcare information system project. We provide an overview of the project's codebase, highlighting key modules, classes, and functionalities. Through code snippets and explanations, readers will gain insights into how the system is designed and structured to meet the requirements of healthcare data management.

Models:

The models.py file contains the database models used to represent various entities in the healthcare system, such as patients, employees, appointments, prescriptions, pharmacy items, and lab tests.

```
from django.db import models
from django.contrib.auth.models import User

# Create your models here.

Sex_types = (
    ("Male", "Male"),
    ("Female", "Female"),
)

Patient_Marital = (
    ("Unmarried","Unmarried"),
    ("Married","Married"),
    ("Divorced", "Divorced"),
)

Profession_types= (
    ("Doctor","Doctor"),
    ("Nurse","Nurse"),
    ("Reports","Reports"),
```

```
    ("Lab","Lab"),
    ("Pharamacy","Pharamacy"),
    ("Undefined","Undefined"),
)

class Patient(models.Model):
    Name = models.CharField(max_length=512, null=True, blank=True)
    Dob = models.DateField(auto_now_add=False, null=True, blank=True)
    Sex = models.CharField(max_length=10, null=True, blank=True, choices=Sex_types)
    Mobile_nos = models.CharField(max_length=13,null=True, blank=True,default="+91")
    Alt_Mobile_nos = models.CharField(max_length=13,null=True, blank=True,default="+91")
    Address = models.CharField(max_length=200, null=True, blank=True)
    Age = models.IntegerField(null=True, blank=True)
    Guardian = models.CharField(max_length=512, null=True, blank=True)
    Email = models.EmailField(null= True, blank=True)
    Marital_Status = models.CharField(max_length=100, null=True, blank=True, choices=Patient_Marital)
    Profession = models.CharField(max_length=100, null=True, blank=True)
    Created = models.DateTimeField(auto_now_add=True, null=True)


class Dept(models.Model):

    Name = models.CharField(max_length=100,blank=True,unique=True)

    def __str__(self):
        return f"{self.Name}"


class Employee(models.Model):
    User = models.ForeignKey(User, on_delete=models.CASCADE,null=True,blank=True)
    Entry_No=models.IntegerField(null=True,blank=True)
    Profession=models.CharField(max_length=20,null=True,blank=True,choices=Profession_types)
    DOJ=models.DateField(blank=True,null=True)
    First_Name = models.CharField(max_length= 20, null=True, blank=True)
    Middle_Name=models.CharField(max_length= 20, null=True, blank=True)
    Last_Name=models.CharField(max_length= 20, null=True, blank=True)
    Gender=models.CharField(max_length=7,null=True,blank=True, choices=Sex_types)
    Blood_Group=models.CharField(max_length=10,null=True,blank=True)
    DOB=models.DateField(max_length=100,null=True)
    Religion=models.CharField(max_length=10,null=True,blank=True)
    Nationality=models.CharField(max_length= 20, null=True, blank=True)
    Address=models.CharField(max_length=100,null=True,blank=True)
    Pincode=models.IntegerField(blank=True,null=True)
    Email=models.EmailField(blank=True,null=True,)
    Department=models.ForeignKey(Dept, on_delete=models.CASCADE,null=True,blank=True)
```

```
    Details = models.CharField(max_length= 100, null=True, blank=True)
    Mobile = models.IntegerField(null=True, blank=True)
    AltMobile = models.IntegerField(null=True, blank=True)
    Marital_Status=models.CharField(max_length=100,null=True, blank=True, choices=Patient_Marital)
    Education_Details=models.CharField(max_length=100,blank=True)
    Experience_Details=models.CharField(max_length=100,blank=True)

    def __str__(self):
        return f"{self.First_Name} {self.Middle_Name} {self.Last_Name}"


class Appointment(models.Model):
    Patient = models.ForeignKey(Patient,on_delete=models.CASCADE,null=True,blank=True)
    Doctor = models.ForeignKey(Employee,on_delete=models.CASCADE,null=True,blank=True)
    datetime = models.DateTimeField(null=True)


class PharmacyItem(models.Model):
    name = models.CharField(max_length=100)
    price = models.IntegerField()
    location = models.CharField(max_length=100,null=True,blank = True)
    curr_stock = models.IntegerField(null=True,blank=True,default=100)
    category = models.CharField(max_length=100,null=True,blank=True,default='Medicine')

    def __str__(self) -> str:
        return self.name


class PharmacyBill(models.Model):
    patient = models.ForeignKey(Patient,on_delete=models.CASCADE)
    items = models.ManyToManyField(PharmacyItem,related_name='items',through='PharmacyItemQuantity')
    itemcount = models.IntegerField(null=True,blank=True)
    net_total = models.IntegerField()
    time = models.DateTimeField(auto_now_add=True, null=True)

class PharmacyItemQuantity(models.Model):
    bill = models.ForeignKey(PharmacyBill,on_delete=models.CASCADE)
    item = models.ForeignKey(PharmacyItem,on_delete=models.CASCADE)
    quantity = models.IntegerField(default=1)


class Prescription(models.Model):
    patient = models.ForeignKey(Patient,on_delete=models.CASCADE)
    doctor = models.ForeignKey(Employee,on_delete=models.CASCADE)
    medications = models.TextField(blank=True, null=True)
```

```
    diagnosis = models.CharField(max_length=200)
    remarks = models.CharField(max_length=200)
    time = models.DateTimeField(auto_now_add=True, null=True)


class Test(models.Model):
    name = models.CharField(max_length=30)
    units = models.CharField(max_length=10)
    ref1 = models.CharField(max_length=10)
    ref2 = models.CharField(max_length=10,null=True)
    ref3 = models.CharField(max_length=10, null=True)
    amount = models.IntegerField()

class LabTestResult(models.Model):
    doctor = models.ForeignKey(Employee,on_delete=models.CASCADE)
    patient = models.ForeignKey(Patient,on_delete=models.CASCADE)
    result = models.ManyToManyField(Test,related_name='lab_items',through='LabTestResultItem')
    time = models.DateTimeField(auto_now_add=True, null=True)

class LabTestResultItem(models.Model):
    test = models.ForeignKey(Test,on_delete=models.CASCADE)
    result = models.ForeignKey(LabTestResult,on_delete=models.CASCADE)
    value = models.CharField(max_length=15)
```

Views:

The views.py file contains the view functions responsible for processing incoming requests, interacting with the database through models, and rendering HTML templates to generate dynamic web pages. We walk through the different views implemented for patient registration, appointment scheduling, prescription management, pharmacy operations, lab test handling, and administrative tasks.

```
from django.shortcuts import render,HttpResponse,redirect,resolve_url,get_object_or_404
from .models import Patient,
Dept,Employee,Appointment,PharmacyItem,PharmacyBill,PharmacyItemQuantity,Prescription,Test,LabTestResult,L
abTestResultItem
from django.contrib.auth.models import User
from allauth.account.decorators import login_required
from django.conf import settings
import json


from django.http import JsonResponse
from reportlab.lib import colors
from reportlab.lib.pagesizes import letter
from reportlab.platypus import SimpleDocTemplate, Paragraph, Table, TableStyle
from reportlab.lib.styles import getSampleStyleSheet


# Create your views here.
Profession_types= (
    ("Doctor","Doctor"),
    ("Nurse","Nurse"),
    ("Reports","Reports"),
    ("Lab","Lab"),
    ("Pharamacy","Pharamacy"),
    ("Undefined","Undefined"),
)

def home(request):
    a = Appointment.objects.all().count()
    amt = 0
    for i in PharmacyBill.objects.all():
        amt += i.net_total
    p = Patient.objects.all().count()
```

```
    d = Employee.objects.all().filter(Profession="Doctor").count()

    return render(request,'dashboard/index.html',context={'a':a,'amt':amt,'p':p,'d':d})


def register(request):
    if request.method == 'POST':
        pat = Patient(
                Name=request.POST["name"],
                Dob=request.POST["dob"],
                Sex=request.POST["gender"],
                Mobile_nos=request.POST["mobileno"],
                Alt_Mobile_nos=request.POST["alternateNo"],
                Address = request.POST["address"],
                Age = request.POST["age"],
                Guardian = request.POST["guardian"],
                Email = request.POST["email"],
                Marital_Status = request.POST["ismarried"],
                Profession= request.POST["profession"],
                )
        pat.save()

        return redirect(resolve_url('home'))
    return render(request,'patient/register.html')


def newDept(request):
    if request.method == 'POST':
        dept = Dept(
            Name = request.POST['name']
        )
        dept.save()
        return redirect(resolve_url('home'))
    dept = Dept.objects.all()
    return render(request,'dept/dept.html',context={'departments':dept})

def newemp(request):
    if request.method == 'POST':
        username = request.POST.get('user')
        email = request.POST.get('email')
        password1 = request.POST.get('pass')
        password2 = request.POST.get('pass2')

        # Validate form data
        if password1 != password2:
```

```
        return redirect('register')

    # Check if username is unique
    if User.objects.filter(username=username).exists():
        return redirect('register')

    # Check if email is unique
    if User.objects.filter(email=email).exists():
        return redirect('register')

    # Create user
    user = User.objects.create_user(username=username, email=email, password=password1)
    user.save()

    dept = Dept.objects.get(id=request.POST.get('Department'))

    emp = Employee(
        User = user,
        Department = dept,
        Entry_No = request.POST.get('entryno'),
        Profession = request.POST.get('profession'),
        DOJ = request.POST.get('DOJ'),
        First_Name = request.POST.get('fname'),
        Last_Name = request.POST.get('mname'),
        Middle_Name = request.POST.get('lname'),
        Gender = request.POST.get('Gender'),
        Blood_Group = request.POST.get('Blood_Group'),
        DOB= request.POST.get('DOB'),
        Religion = request.POST.get('religion'),
        Nationality = request.POST.get('Nationality'),
        Address = request.POST.get('address'),
        Pincode = request.POST.get('pincode'),
        Email = request.POST.get('email'),
        Mobile=request.POST.get('mob'),
        AltMobile = request.POST.get('altmob'),
        Marital_Status = request.POST.get('ismarried'),
        Education_Details = request.POST.get('Education'),
        Experience_Details = request.POST.get('Experience')
    )
    emp.save()

    return redirect(resolve_url('home'))
```

```
    return
render(request,'employee/new.html',context={'DepartmentList':Dept.objects.all(),'Profession_types':Profess
ion_types})



def op(request):

    if request.method == 'POST':
        appointment = Appointment(
            Patient  = Patient.objects.get(id = request.POST.get('patientid')),
            Doctor = Employee.objects.get(id = request.POST.get('doctor')),
            datetime = request.POST.get('date')
        )
        appointment.save()

        return redirect('appointments')
    patient = None
    doctors = Employee.objects.all().filter(Profession="Doctor")
    if 'id' in request.GET:
        patient_id = request.GET['id']
        try:
            patient = Patient.objects.get(id=patient_id)
        except Patient.DoesNotExist:
            pass



    return render(request, 'patient/op.html', {'patient': patient,'doctors':doctors})

@login_required
def dashboard(request):
    user = request.user
    doc = Employee.objects.get(User=user)
    if doc.Profession == 'Doctor':
        appointments = Appointment.objects.filter(Doctor = doc)

        return render(request,'employee/dashboard.html',context={'appointments':appointments})
    return redirect(resolve_url('home'))


def patients(request):
    p = Patient.objects.all()
    return render(request,'patient/patients.html',context={'patients':p})
```

```
def employees(request):
    e = Employee.objects.all()
    return render(request,'employee/employees.html',context={'employees':e})


def pharmcyitem(request):
    if request.method == 'POST':
        p = PharmacyItem(
            name = request.POST.get('name'),
            price = request.POST.get('price'),
            location = request.POST.get('loc'),
            curr_stock = request.POST.get('stock'),
            category = request.POST.get('category')

        )

        p.save()

        return redirect('pharmacyitem')
    p = PharmacyItem.objects.all()
    return render(request,'pharmacy/newitem.html',context={'items':p})


def pharmacybill(request):
    if request.method == 'POST':
        p = Patient.objects.get(id = request.POST.get('patient-id'))
        items = json.loads(request.POST.get('items'))
        bill = PharmacyBill(
            patient = p,
            itemcount = 0,
            net_total = 0
        )
        bill.save()
        amt = 0
        item_count = 0
        for i in items:
            curr_item = PharmacyItem.objects.get(id = int(i['id']))
            curr_quantity = int(i['quantity'])
            bill_item = PharmacyItemQuantity(
                bill= bill,
                item = curr_item,
                quantity = curr_quantity
            )
            bill_item.save()
            curr_item.curr_stock -= curr_quantity
```

```
            curr_item.save()
            amt += curr_quantity * curr_item.price
            item_count += curr_quantity
        bill.net_total = amt
        bill.itemcount = item_count
        bill.save()
        return redirect('pharmacyviewbill',id=bill.id)

    p = PharmacyItem.objects.all()
    patient = None
    if 'id' in request.GET:
        patient_id = request.GET['id']
        try:
            patient = Patient.objects.get(id=patient_id)
        except Patient.DoesNotExist:
            pass
    return render(request,'pharmacy/bill.html',context={'items':p,'patient':patient})



def pharmacyviewbill(request, id):
    bill = get_object_or_404(PharmacyBill, id=id)

    # Create a PDF document
    response = HttpResponse(content_type='application/pdf')
    response['Content-Disposition'] = f'attachment; filename="bill_{id}.pdf"'
    doc = SimpleDocTemplate(response, pagesize=letter)
    elements = []
    styles = getSampleStyleSheet()

    # Add bill details
    elements.append(Paragraph(f"Patient: {bill.patient.Name}", styles['Normal']))
    elements.append(Paragraph(f"Date: {bill.time}", styles['Normal']))
    elements.append(Paragraph("Items:", styles['Heading2']))

    # Add item details
    data = [['Item', 'Price', 'Quantity', 'Total']]
    total_cost = 0
    for item_quantity in bill.pharmacyitemquantity_set.all():
        item = item_quantity.item
        quantity = item_quantity.quantity
        total_item_cost = item.price * quantity
        total_cost += total_item_cost
        data.append([item.name, item.price, quantity, total_item_cost])
```

```
    # Add table to the document
    table = Table(data)
    style = TableStyle([('BACKGROUND', (0, 0), (-1, 0), colors.gray),
                        ('TEXTCOLOR', (0, 0), (-1, 0), colors.whitesmoke),
                        ('ALIGN', (0, 0), (-1, -1), 'CENTER'),
                        ('FONTNAME', (0, 0), (-1, 0), 'Courier-Bold'),
                        ('BOTTOMPADDING', (0, 0), (-1, 0), 12),
                        ('BACKGROUND', (0, 1), (-1, -1), colors.beige),
                        ('GRID', (0, 0), (-1, -1), 1, colors.black)])
    table.setStyle(style)
    elements.append(table)

    # Add total cost
    elements.append(Paragraph(f"Total Cost: {total_cost}", styles['Normal']))

    # Build the PDF document
    doc.build(elements)
    return response


def prescription(request):
    if request.method == 'POST':
        pres = Prescription(
            patient  = Patient.objects.get(id = request.POST.get('patientid')),
            doctor = Employee.objects.get(id = request.POST.get('doctor')),
            medications = request.POST.get('medications'),
            remarks = request.POST.get('remarks'),
            diagnosis = request.POST.get('diagnosis')
        )

        pres.save()

        return redirect('viewprescription',id=pres.id)


    doctors = Employee.objects.all().filter(Profession="Doctor")

    patient = None
    if 'id' in request.GET:
        patient_id = request.GET['id']
        try:
            patient = Patient.objects.get(id=patient_id)
        except Patient.DoesNotExist:
            pass
    return render(request,'prescription/new.html',context={'patient':patient,'doctors':doctors})
```

```
def view_prescription(request, id):
    prescription = get_object_or_404(Prescription, id=id)

    # Create a PDF document
    response = HttpResponse(content_type='application/pdf')
    response['Content-Disposition'] = f'attachment; filename="prescription_{id}.pdf"'
    doc = SimpleDocTemplate(response, pagesize=letter)
    elements = []
    styles = getSampleStyleSheet()

    # Add prescription details
    elements.append(Paragraph(f"Patient: {prescription.patient.Name}", styles['Normal']))
    elements.append(Paragraph(f"Doctor: {prescription.doctor}", styles['Normal']))
    elements.append(Paragraph(f"Diagnosis: {prescription.diagnosis}", styles['Normal']))
    elements.append(Paragraph(f"Remarks: {prescription.remarks}", styles['Normal']))
    elements.append(Paragraph(f"Time: {prescription.time}", styles['Normal']))
    elements.append(Paragraph("Medications:", styles['Heading2']))

    # Parse medications JSON and add to the document
    if prescription.medications:
        medications_data = json.loads(prescription.medications)
        data = [['Name', 'Quantity', 'Morning', 'Afternoon', 'Night']]
        for med_data in medications_data:
            data.append([
                med_data.get('name', ''),
                med_data.get('quantity', ''),
                'Yes' if med_data.get('morning', False) else 'No',
                'Yes' if med_data.get('afternoon', False) else 'No',
                'Yes' if med_data.get('night', False) else 'No'
            ])
        # Add table to the document
        table = Table(data)
        style = TableStyle([('BACKGROUND', (0, 0), (-1, 0), colors.gray),
                            ('TEXTCOLOR', (0, 0), (-1, 0), colors.whitesmoke),
                            ('ALIGN', (0, 0), (-1, -1), 'CENTER'),
                            ('FONTNAME', (0, 0), (-1, 0), 'Courier-Bold'),
                            ('BOTTOMPADDING', (0, 0), (-1, 0), 12),
                            ('BACKGROUND', (0, 1), (-1, -1), colors.beige),
                            ('GRID', (0, 0), (-1, -1), 1, colors.black)])
        table.setStyle(style)
        elements.append(table)
```

```
    # Build the PDF document
    doc.build(elements)
    return response


def newlabtest(request):
    if request.method == 'POST':
        name = request.POST.get('name')
        units = request.POST.get('units')
        ref1 = request.POST.get('ref1')
        ref2 = request.POST.get('ref2')
        ref3 = request.POST.get('ref3')
        amount = request.POST.get('amt')

        if name and units and ref1 and amount:
            try:
                t = Test(name=name, units=units, ref1=ref1, ref2=ref2, ref3=ref3, amount=amount)
                t.save()
                return redirect('labtests')
            except Exception as e:
                return HttpResponse(f"An error occurred: {str(e)}")
        else:
            return HttpResponse("Incomplete data. Test not saved.")

    return render(request, 'lab/new.html')


def labtests(request):
    tests = Test.objects.all()
    for i in tests:
        print(i)
    return render(request,'lab/test.html',context={'tests':tests})


def labreport(request):

    t = Test.objects.all()
    d = Employee.objects.all().filter(Profession="Doctor")

    if request.method == 'POST':
        tests = json.loads(request.POST.get('items'))
        p = Patient.objects.get(id = request.POST.get('patient-id'))
        d = Employee.objects.get(id = request.POST.get('doctor'))
        ltr = LabTestResult(
            doctor = d,
```

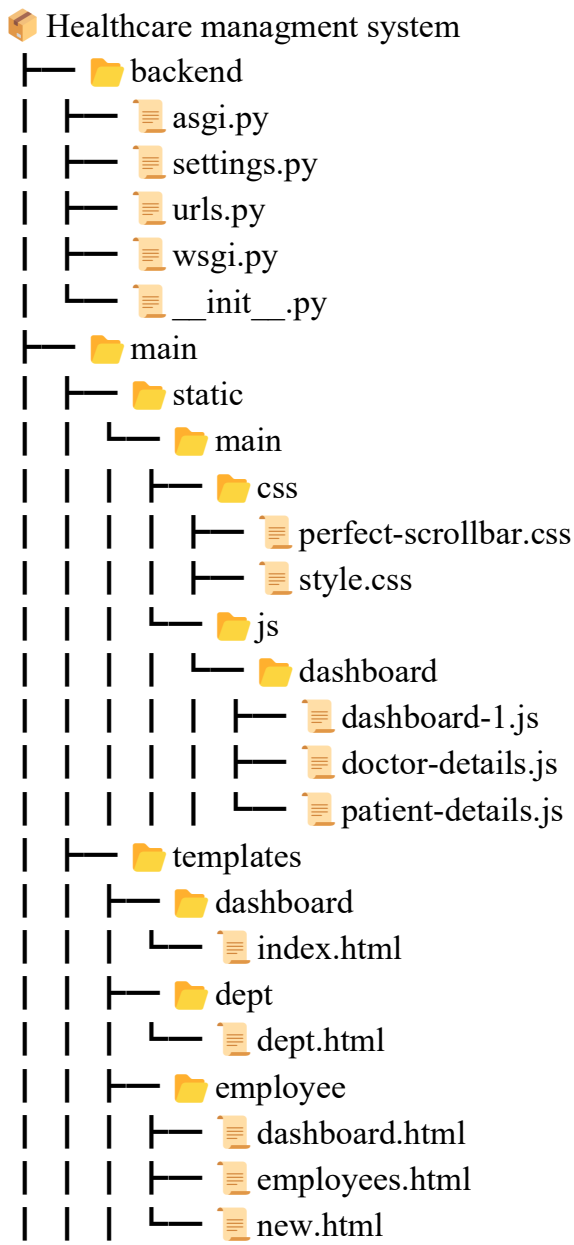
```
            patient = p,
        )
        ltr.save()
        for i in tests:

            ltri = LabTestResultItem(
                test = Test.objects.get(id= i["id"]),
                result = ltr,
                value = i["value"]
            )
            ltri.save()

        return redirect('viewlabreport',id=ltr.id)
    d = Employee.objects.all().filter(Profession="Doctor")
    patient = None
    if 'id' in request.GET:
        patient_id = request.GET['id']
        try:
            patient = Patient.objects.get(id=patient_id)
        except Patient.DoesNotExist:
            pass
    return render(request,'lab/report.html',context={'doctors':d,'tests':t,'patient':patient})



def viewlabreport(request, id):
    lab_result = get_object_or_404(LabTestResult, id=id)

    # Create a PDF document
    response = HttpResponse(content_type='application/pdf')
    response['Content-Disposition'] = f'attachment; filename="lab_report_{id}.pdf"'
    doc = SimpleDocTemplate(response, pagesize=letter)
    elements = []
    styles = getSampleStyleSheet()

    # Add lab report details
    elements.append(Paragraph(f"Patient: {lab_result.patient.Name}", styles['Normal']))
    elements.append(Paragraph(f"Doctor: {lab_result.doctor}", styles['Normal']))
    elements.append(Paragraph(f"Time: {lab_result.time}", styles['Normal']))
    elements.append(Paragraph("Test Results:", styles['Heading2']))

    # Retrieve test results for this lab report
    test_results = LabTestResultItem.objects.filter(result=lab_result)

    # Add test results to the document
```

```
    data = [['Test Name', 'Value', 'Ref 1', 'Ref 2', 'Ref 3']]
    for test_result in test_results:
        test_name = test_result.test.name
        test_value = test_result.value
        ref1 = test_result.test.ref1
        ref2 = test_result.test.ref2
        ref3 = test_result.test.ref3
        data.append([test_name, test_value, ref1, ref2, ref3])

    # Add table to the document
    table = Table(data)
    style = TableStyle([('BACKGROUND', (0, 0), (-1, 0), colors.gray),
                        ('TEXTCOLOR', (0, 0), (-1, 0), colors.whitesmoke),
                        ('ALIGN', (0, 0), (-1, -1), 'CENTER'),
                        ('FONTNAME', (0, 0), (-1, 0), 'Courier-Bold'),
                        ('BOTTOMPADDING', (0, 0), (-1, 0), 12),
                        ('BACKGROUND', (0, 1), (-1, -1), colors.beige),
                        ('GRID', (0, 0), (-1, -1), 1, colors.black)])
    table.setStyle(style)
    elements.append(table)

    # Build the PDF document
    doc.build(elements)
    return response
def prescriptionhistory(request):
    patient = None
    prescription = None
    if 'id' in request.GET:
        patient_id = request.GET['id']
        try:
            patient = Patient.objects.get(id=patient_id)
            prescription = Prescription.objects.filter(patient = patient)
        except Patient.DoesNotExist:
            pass
    return
render(request,'history/prescription.html',context={'prescriptions':prescription,'patient':patient})


def labhistory(request):
    patient = None
    labs = None
    if 'id' in request.GET:
        patient_id = request.GET['id']
        try:
            patient = Patient.objects.get(id=patient_id)
```

```
            labs = LabTestResult.objects.filter(patient = patient)
        except Patient.DoesNotExist:
            pass
    return render(request,'history/labreports.html',context={'reports':labs,'patient':patient})

def billhistory(request):
    patient = None
    bills = None
    if 'id' in request.GET:
        patient_id = request.GET['id']
        try:
            patient = Patient.objects.get(id=patient_id)
            bills = PharmacyBill.objects.filter(patient = patient)
        except Patient.DoesNotExist:
            pass
    return render(request,'history/bills.html',context={'bills':bills,'patient':patient})

def appointments(request):
    a = Appointment.objects.all()
    doctors = Employee.objects.all().filter(Profession="Doctor")
    if 'id' in request.GET:
        doc = request.GET['id']
        try:
            d = Employee.objects.get(id=doc)
            a = Appointment.objects.filter(Doctor = d)
        except Patient.DoesNotExist:
            pass

    return render(request,'patient/appointments.html',context={'appointments':a,'doctors':doctors})
def schedule(request):
    return render(request,'patient/schedule.html')

def get_item_details(request, item_id):
    item = get_object_or_404(PharmacyItem, pk=item_id)

    if request.method == 'POST':
        item.name = request.POST.get('name')
        item.price = request.POST.get('price')
        item.location = request.POST.get('loc')
        item.curr_stock = request.POST.get('stock')
        item.category = request.POST.get('category')
        item.save()
        return redirect('pharmacyitem')

    return render(request, 'edit_item_form.html', {'item':item})
```

Templates:

The templates directory houses HTML templates that define the structure and layout of web pages rendered by the Django framework. We showcase several template files used for displaying patient registration forms, appointment schedules, prescription details, pharmacy bills, lab test reports, and administrative dashboards.



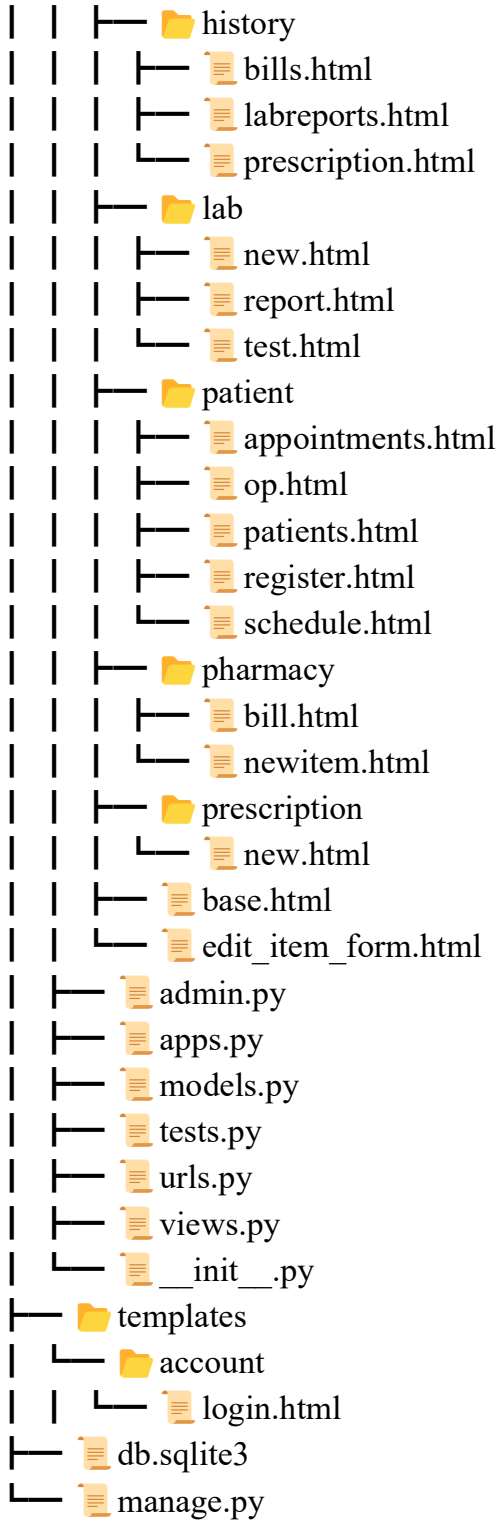


Fig.6.1 Project Directory Structure

CHAPTER 7

RESULT AND DISCUSSION

The healthcare information system project has been successfully developed and implemented, offering a comprehensive solution for managing various aspects of a healthcare facility. The system comprises several modules and features, each addressing specific requirements and streamlining processes within the healthcare domain.

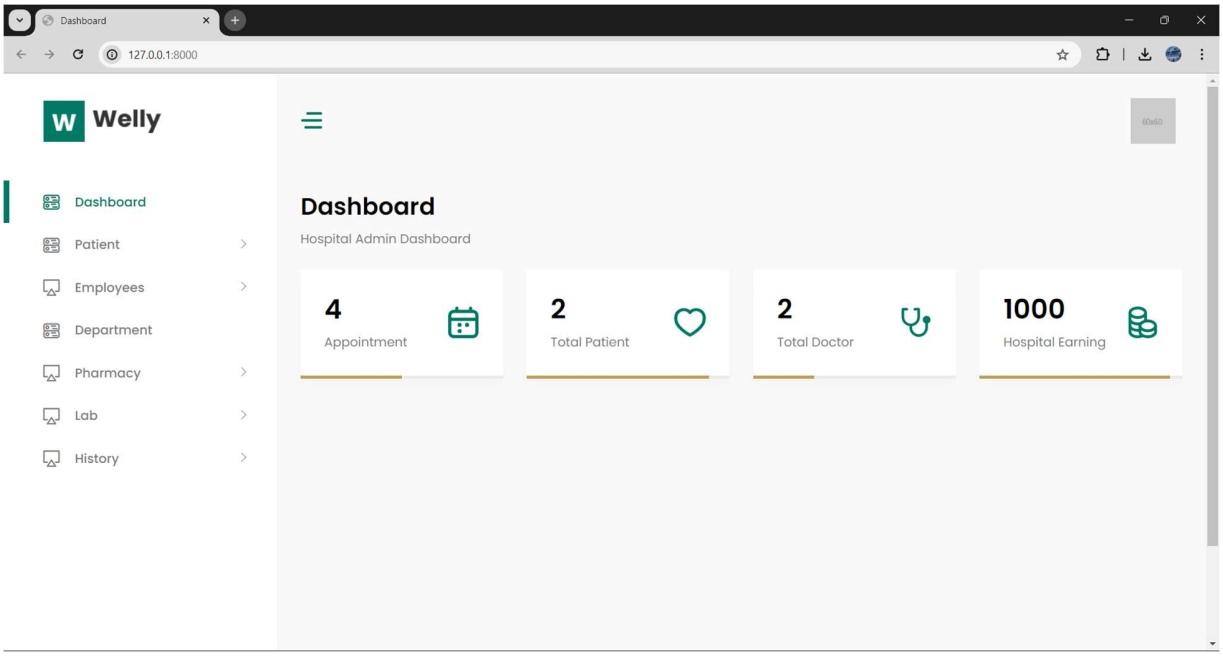


Fig.7.1 Dashboard

Patient Management

The system allows for efficient registration and management of patient information, including personal details, medical history, and contact information.

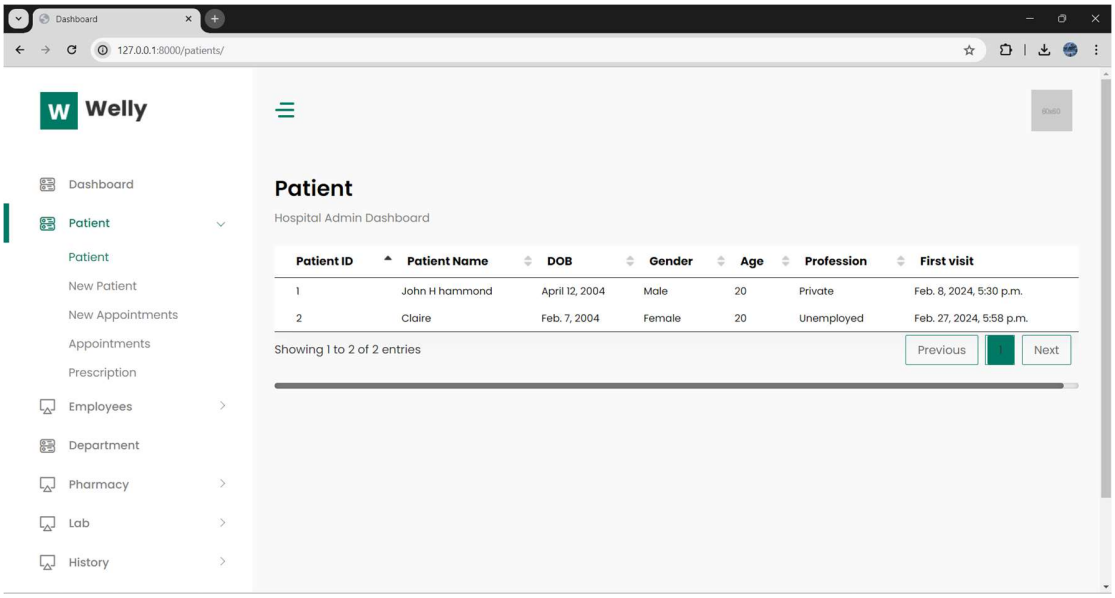


Fig.7.2 Patients

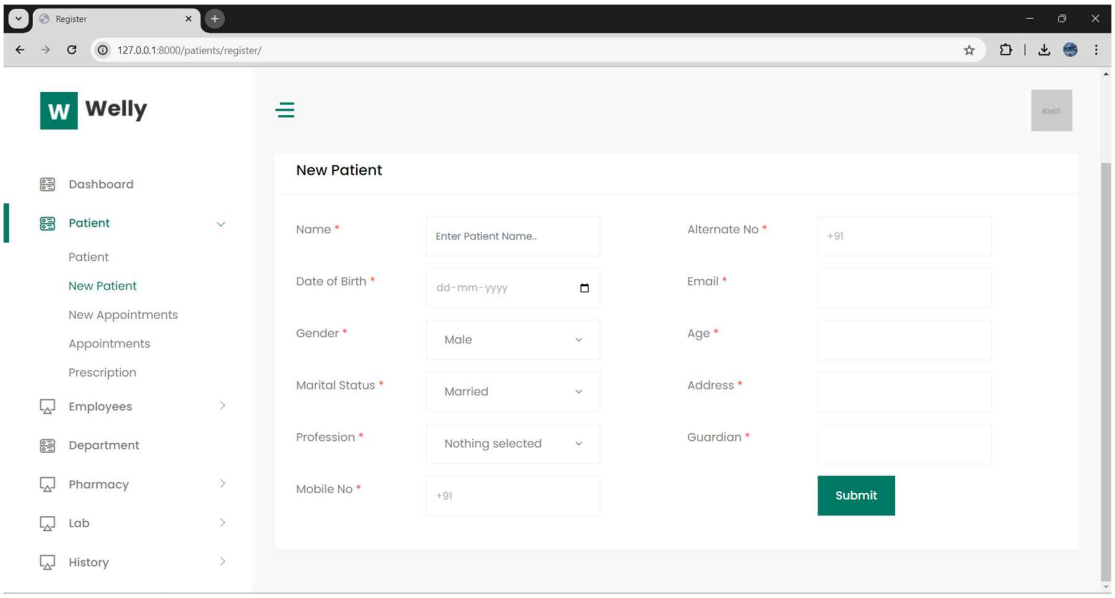


Fig.7.3 Register Patient

Employee Management

Healthcare professionals, such as doctors, nurses, and administrative staff, can be registered and managed within the system. User authentication and role-based access control ensure secure access to sensitive information based on an individual's role and responsibilities.

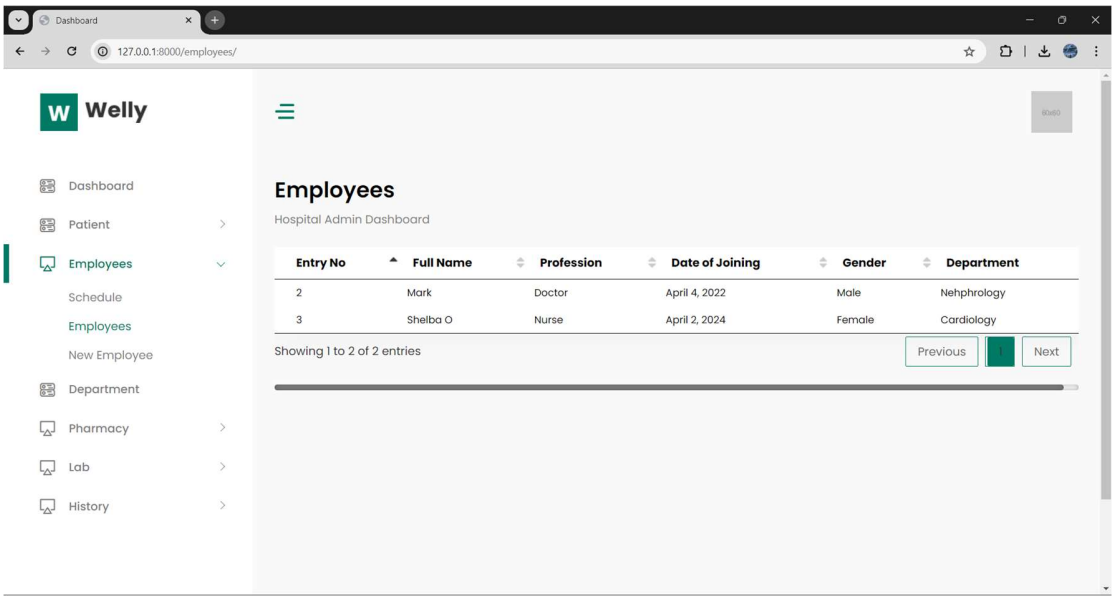


Fig.7.4 Employees

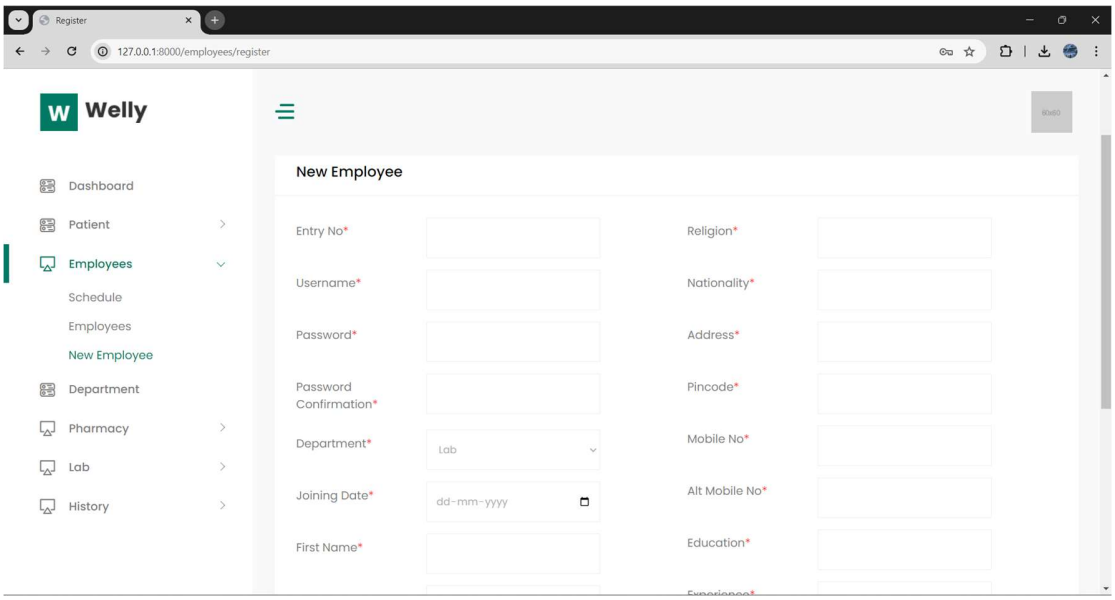


Fig.7.5 Employee Registration

Appointment Scheduling

The system facilitates efficient appointment scheduling, allowing patients to book appointments with healthcare professionals based on availability.

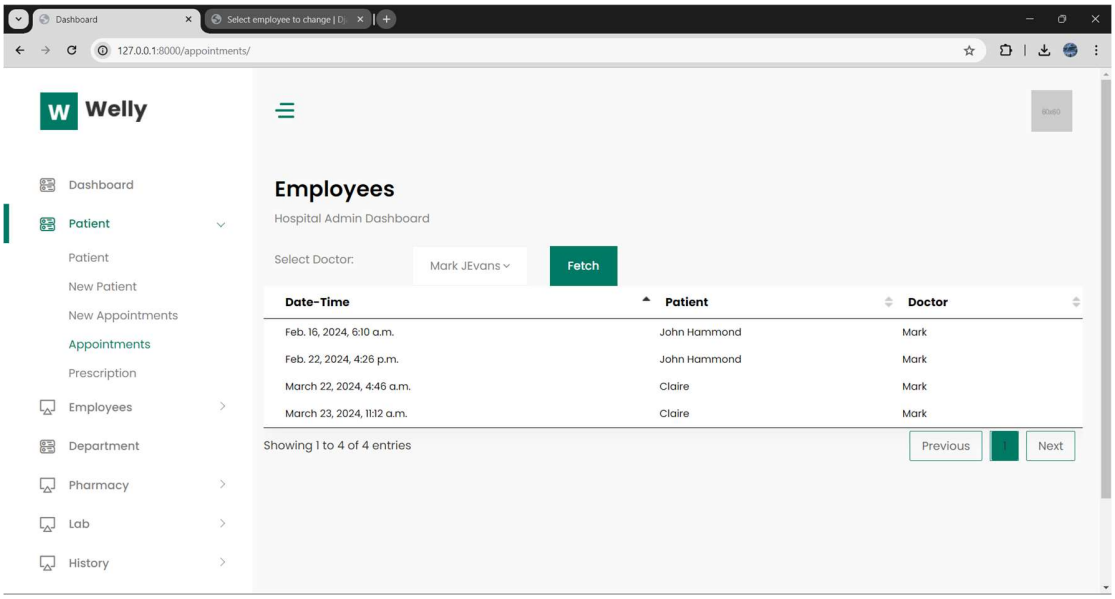


Fig.7.6 Appointments

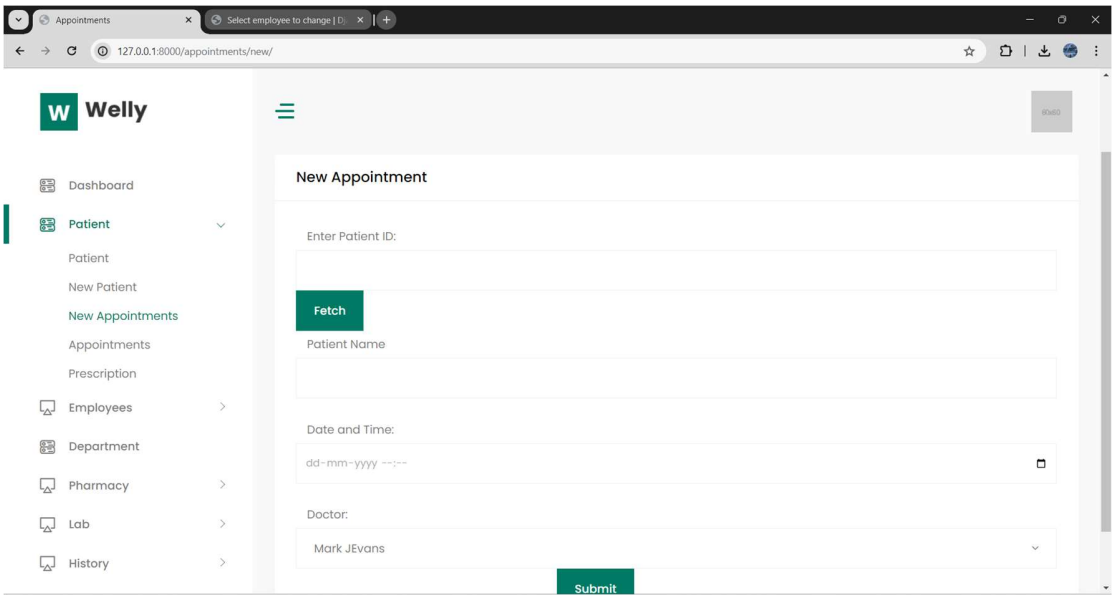


Fig.7.7 New Appointment

Pharmacy Management

The pharmacy module enables inventory management, tracking of pharmaceutical items, and generation of pharmacy bills.

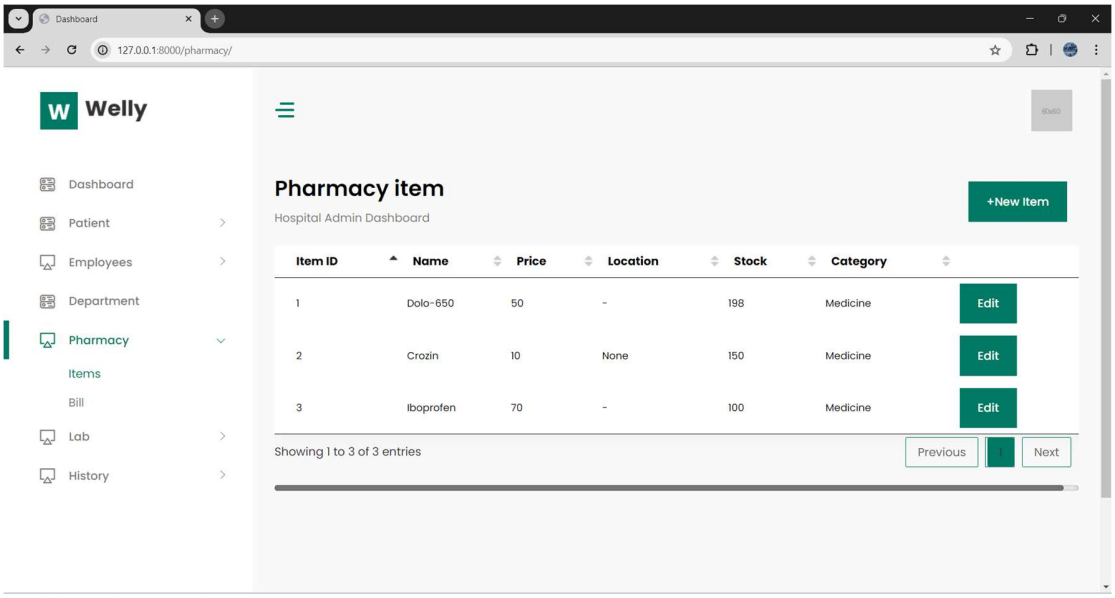


Fig.7.8 Pharmacy Items

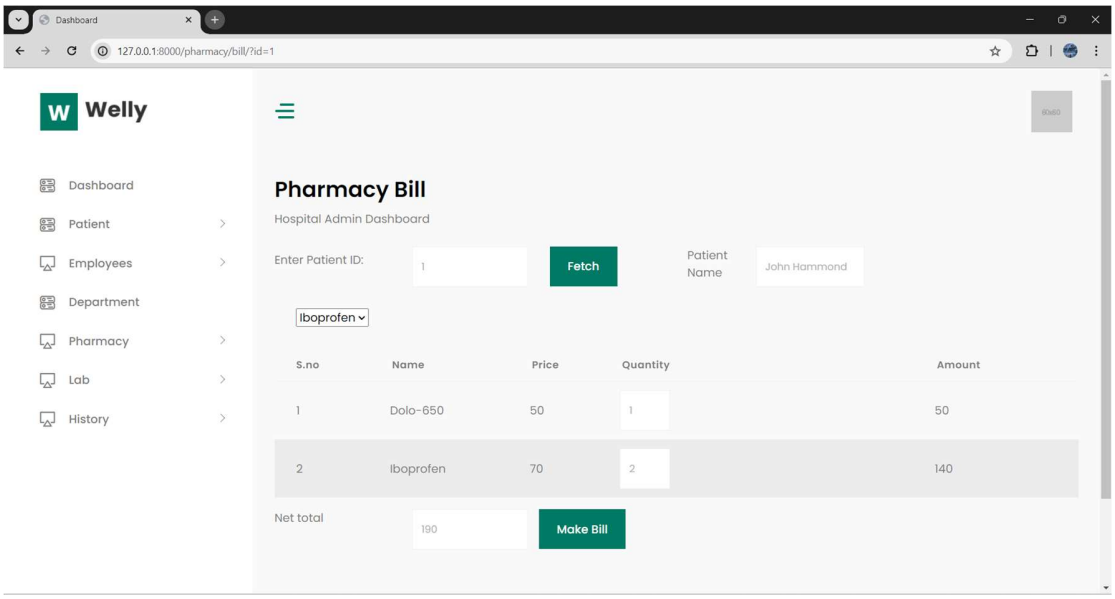


Fig.7.9 Pharmacy Billing

Laboratory Test Management

Healthcare professionals can order laboratory tests for patients and record test results within the system. Comprehensive laboratory test reports can be generated, providing detailed information about the tests conducted and their results.

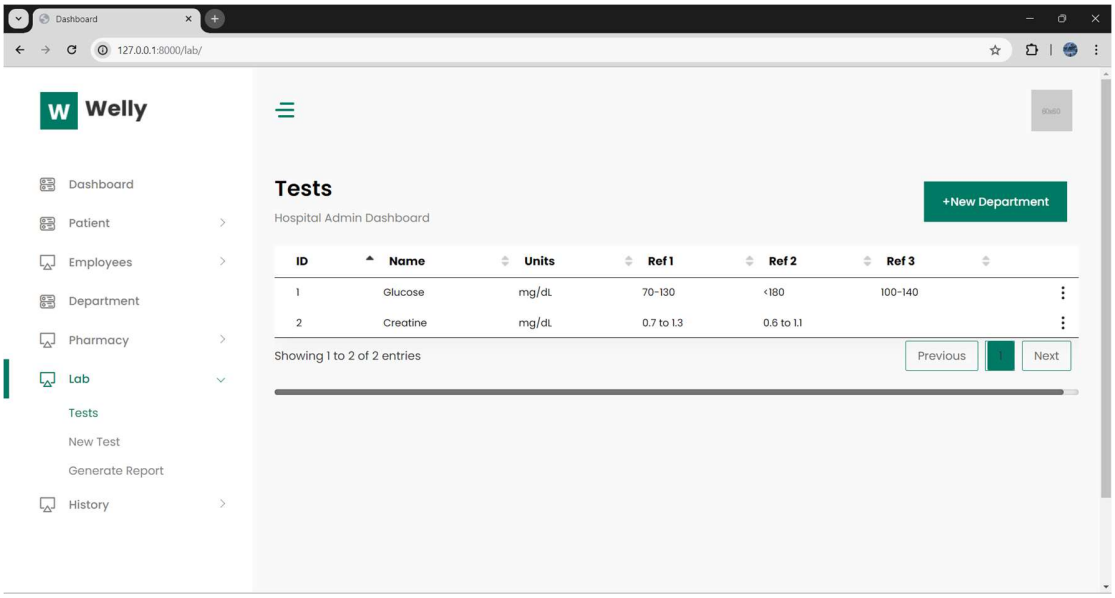


Fig.7.10 Lab tests

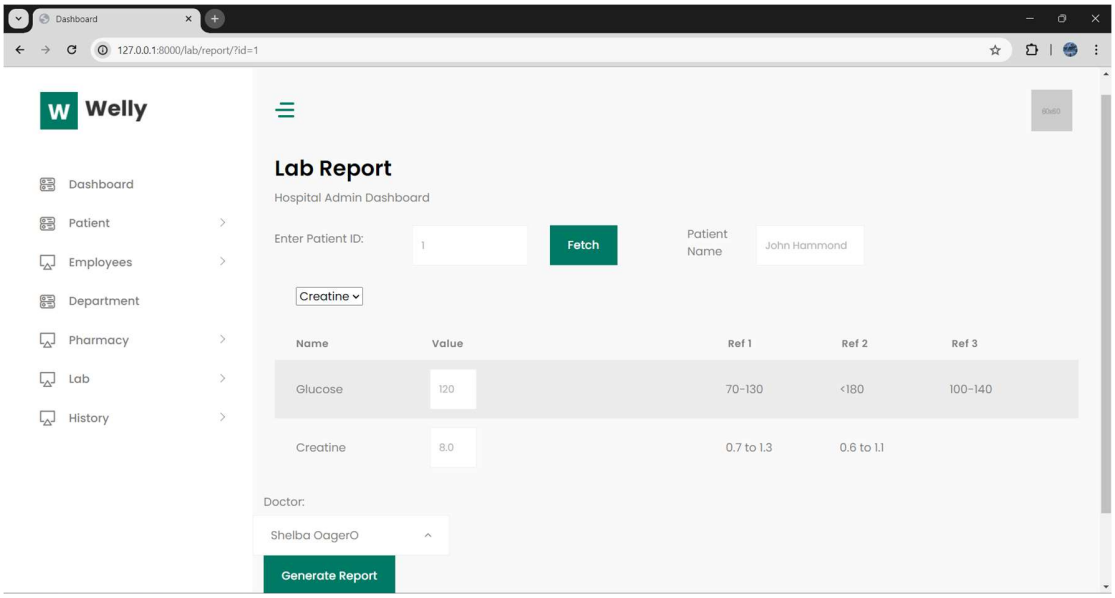


Fig.7.11 Report Generation

The healthcare information system project addresses several pain points associated with traditional paper-based or disparate systems. By providing a centralized and digitized platform, the system enhances data accuracy, improves communication and collaboration among healthcare professionals, and streamlines processes within the healthcare facility.

One of the key advantages of the system is its ability to maintain comprehensive patient records, enabling healthcare providers to access up-to-date and accurate information, reducing the risk of medical errors and ensuring continuity of care. Additionally, the system facilitates efficient appointment scheduling, minimizing wait times and optimizing resource utilization.

While the system offers numerous benefits, there are certain limitations and challenges to consider. One potential limitation is the dependence on reliable internet connectivity and hardware infrastructure, as the system relies on digital data storage and transmission. Additionally, ensuring data security and privacy remains a critical consideration, requiring robust measures to protect sensitive patient information.

In terms of scalability and extensibility, the system has been designed with a modular architecture, allowing for future enhancements and integration with other healthcare systems or third-party applications. However, as the system's userbase and data volume grow, performance optimizations and infrastructure upgrades may be necessary to maintain optimal system responsiveness.

Regarding security and privacy considerations, the system implements industry-standard security protocols and follows relevant regulations, such as HIPAA, to safeguard patient data. Access controls, data encryption, and auditing mechanisms are in place to ensure the confidentiality and integrity of sensitive information.

Throughout the development process, valuable lessons were learned, including the importance of thorough requirements gathering, effective communication with stakeholders, and the need for rigorous testing and quality assurance. Additionally, adopting agile development methodologies and embracing best practices in software engineering proved instrumental in delivering a robust and maintainable system.

Looking ahead, several potential future enhancements can be explored to further improve the healthcare information system. These may include integrating telemedicine capabilities, developing mobile applications for enhanced accessibility, and leveraging advanced data analytics and machine learning techniques to gain deeper insights and facilitate predictive healthcare.

Overall, the healthcare information system project has successfully delivered a comprehensive solution that addresses the challenges faced by traditional systems and paves the way for improved patient care, operational efficiency, and data-driven decision-making within the healthcare domain.

CHAPTER 8

ONLINE COURSE CERTIFICATE

Elite

NPTEL Online Certification

(Funded by the MoE, Govt. of India)

This certificate is awarded to

SAM FREDRICK

for successfully completing the course

Data Base Management System

with a consolidated score of **81** %

Online Assignments	21.25/25	Proctored Exam	59.25/75

Total number of candidates certified in this course: **6225**

Jan-Mar 2024

(8 week course)

Prof. Haimanti Banerji
Coordinator, NPTEL
IIT Kharagpur

Indian Institute of Technology Kharagpur

Roll No: NPTEL24CS21S544101024 To verify the certificate  No. of credits recommended: 2 or 3

GitHub Link: https://github.com/fredrick273/healthcare-managment-system/