BLOOD BANK MANAGEMENT SYSTEM

A PROJECT REPORT

Submitted by

Gautam Taneja [RA2211003010812]

Amithrajith Premesh [RA2211003010811]

Anirudhan Mani [RA221100301082]

Under the Guidance of

Dr. R. Lavanya

Assistant Professor, Department of Computing Technologies

in partial fulfillment of the requirements for the degree of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

DEPARTMENT OF COMPUTING TECHNOLOGIES

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

E-commerce website certified to be the bonafide work done by **Gautam Taneja[RA2211003010812] , Amithrajith Premesh[RA2211003010811], Anirudhan Mani[RA2211003010824]** 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: 26/04/2024

Faculty in Charge
Dr. R. Lavanya
Assistant Professor
Dept. of Computing Technologies
SRMIST -KTR

Head of the department
Dr. M. Pushpalatha
Professor & Head
Dept. of Computing Technologies
SRMIST - KTR

ABSTRACT

Blood Bank Management System (BBMS) is a browser-based system that is designed to store,process, retrieve and analyze information concerned with the administrative and inventory management within a blood bank. This project aims at maintaining all the information related to blood donors, different blood groups available in each blood bank and help them manage in a better way. The main objective is to provide transparency in this field, make the process of obtaining blood from a blood bank corruption free and make the system of blood bank management effective. This gives attention in stocking blood donors information. The donors who are interested in donating blood has to register in the database. The software is fully integrated with CRM (customer relationship management) as well as CMS (content management system) solution. The requirement of the blood has to be requested and the information of the donor are supplied. The donors can update their status whether they are available or not. After the implementation of the project, the blood searching process is expected to be faster, easier, and reliable. Admin will view the donor side and view the available blood requested by the users.It also supervises blood inventory management and other blood bank-related activities. The major goal of the blood bank management system is to keep track of blood, donors, blood groups, blood banks, and stock information. It keeps track of all information concerning blood, blood cells, stocks, and blood. Because the project is all done at the administrative level, only the administrator can see it.

PROBLEM STATEMENT

The Blood Bank Management System (BBMS) project is a pioneering endeavor aimed at transforming blood donation services and revolutionizing the way blood banks are managed. By leveraging a sophisticated web-based platform, the BBMS facilitates seamless administrative tasks and inventory management within blood banks. Its core objective is to comprehensively catalog information related to blood donors, including their personal details, medical histories, and donation frequencies, while also maintaining a real-time record of the availability of different blood groups in the blood bank's inventory. This system not only ensures transparency in operations but also enhances efficiency and effectiveness in managing blood resources. Through a user-friendly interface, individuals in need of blood can easily request, check availability, and schedule appointments, thereby streamlining the process of obtaining blood and making it hassle-free for both donors and recipients. Moreover, the BBMS incorporates robust measures to prevent corruption and malpractice, ensuring that blood resources are utilized ethically and responsibly. Automated alerts for low inventory levels or approaching expiration dates, coupled with analytical insights into donation trends and resource utilization patterns, enable proactive decision-making and optimization of blood distribution strategies. Ultimately, the BBMS contributes significantly to improving healthcare services, emergency response capabilities, and overall public health outcomes within the community.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Problem Understanding

The rise in global population strains healthcare systems, particularly blood banks. Despite a growing pool of potential donors (under 10% donate), blood demand is increasing due to population growth and medical advancements. Inefficient communication between blood banks and recipients creates a critical gap, leaving many patients in need without timely access to blood, which can be life-threatening. Furthermore, poor blood bank management leads to wasted blood inventory. Automating blood bank systems can address these challenges. A robust and scalable system can connect donors with recipients and streamline blood searches. This pilot project proposes a blood bank management system to efficiently collect blood from donors and distribute it to those in urgent need. The software will manage daily blood bank operations, including donor registration, collection details, and blood issuance records. The system's design ensures adaptability to future needs of blood banks.

- The blood bank management system serves as a pilot project, focusing on gathering blood from various sources and distributing it to those in need.
- It is designed to handle daily transactions within the blood bank and retrieve details as necessary.
- The system facilitates the registration of donor information, blood collection records, and issuance reports.
- Its adaptable software design ensures it can accommodate the evolving needs of blood banks in the future.

This project in the Blood Bank Management System will develop an efficient system for blood transactions.

1.2 Identification of Entity and Relationships

In the realm of database design, the Entity-Relationship (ER) model serves as a fundamental tool for conceptualizing and organizing the structure of a database. At the heart of this model lies the identification of entities, which are the fundamental building blocks representing real-world objects, concepts, or events within a given domain. The process of identifying entities involves a systematic analysis of the requirements and characteristics of the system being modeled, aiming to capture the essential entities that play pivotal roles in the domain.

Blood Bank Management System (BBMS) using an Entity-Relationship (ER) model, the entities can encompass a wide range of elements crucial to blood bank operations. Tangible entities could include donors, blood units, and medical staff, each with their own set of attributes. For example, the donor entity may have attributes like donor ID, name, blood type, medical history, and contact information. The blood unit entity could include attributes such as unit ID, blood type, expiration date, storage location, and donation date. Additionally, abstract entities like blood requests, inventory transactions, and donation records play a vital role. Attributes for these entities might include request ID, recipient details, requested blood type, transaction date, quantity transferred, and donation timestamp. This ER model not only captures the physical components of blood bank management but also the operational and transactional aspects, ensuring a comprehensive representation of the system's functionalities and data interactions.

Construction of DB using ER Model

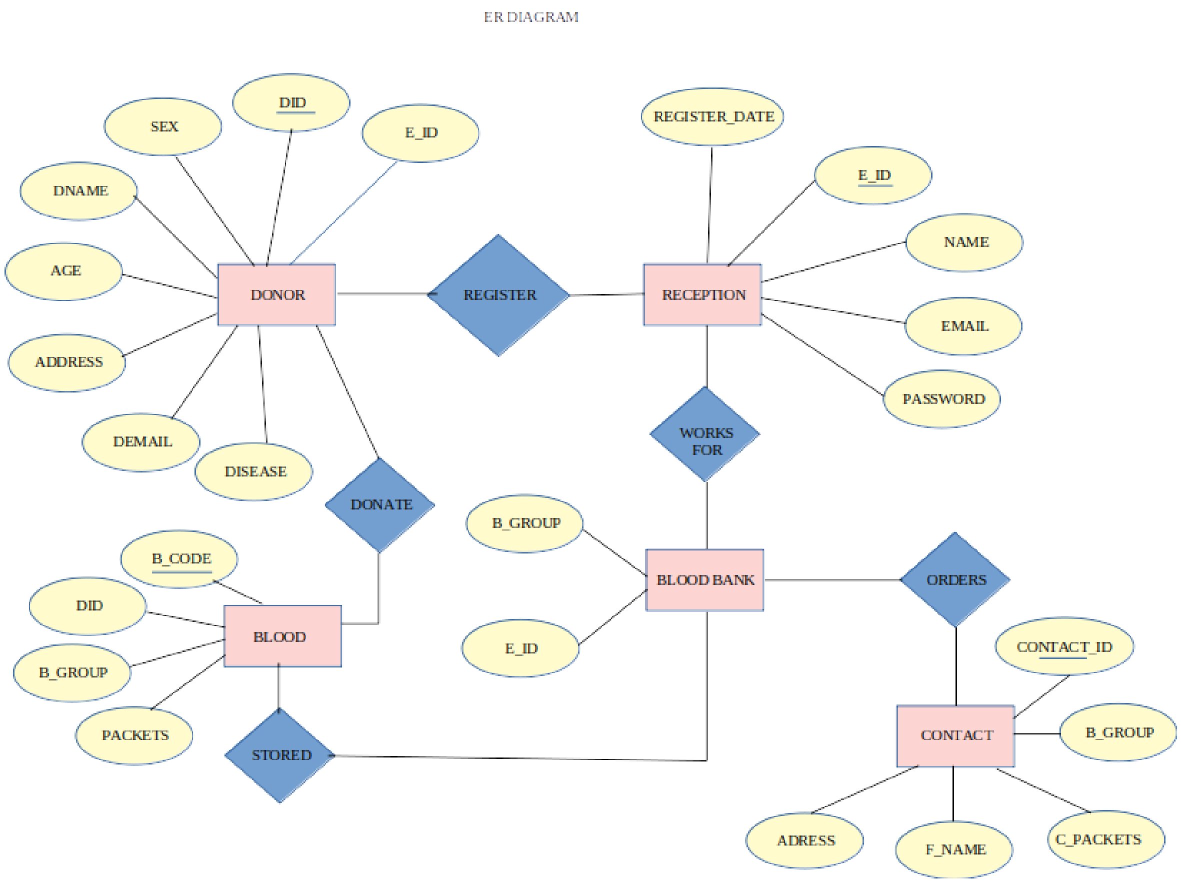


Fig1.1 : E-R Diagram

DESIGN OF RELATIONAL SCHEMAS, CREATION OF DATABASE TABLES FOR THE PROJECT.

2.1 Design of Relational Schemas

In the Blood Bank Management System (BBMS), relational schemas are fundamental to organizing and structuring the underlying database efficiently. They serve as blueprints that define the tables, attributes, and relationships between entities, providing a structured representation of how data is stored and managed.

Tables in the relational schema correspond to key entities within the BBMS, such as donors, blood units, requests, and transactions. Each table consists of rows and columns, with columns representing attributes that store specific information about the entity. For instance, the Donors table may include columns like donor ID, name, blood type, contact details, medical history, and donation records.

Attributes within each table describe the characteristics or properties of the corresponding entity. For example, the Blood Units table may have attributes such as unit ID, blood type, quantity, expiration date, storage location, and donation timestamp, providing detailed information about each blood unit in the inventory.

Relationships between tables establish connections between different entities in the BBMS. These relationships are defined using foreign keys, which link the primary key of one table to a corresponding attribute in another table. For instance, the Donations table may have a foreign key referencing the donor ID in the Donors table, establishing a relationship between donations and the donors who contributed them.

Overall, the relational schema in the BBMS ensures a structured and organized approach to managing donor information, blood inventory, donation records, and related transactions, facilitating efficient data retrieval, analysis, and management within the blood bank system.

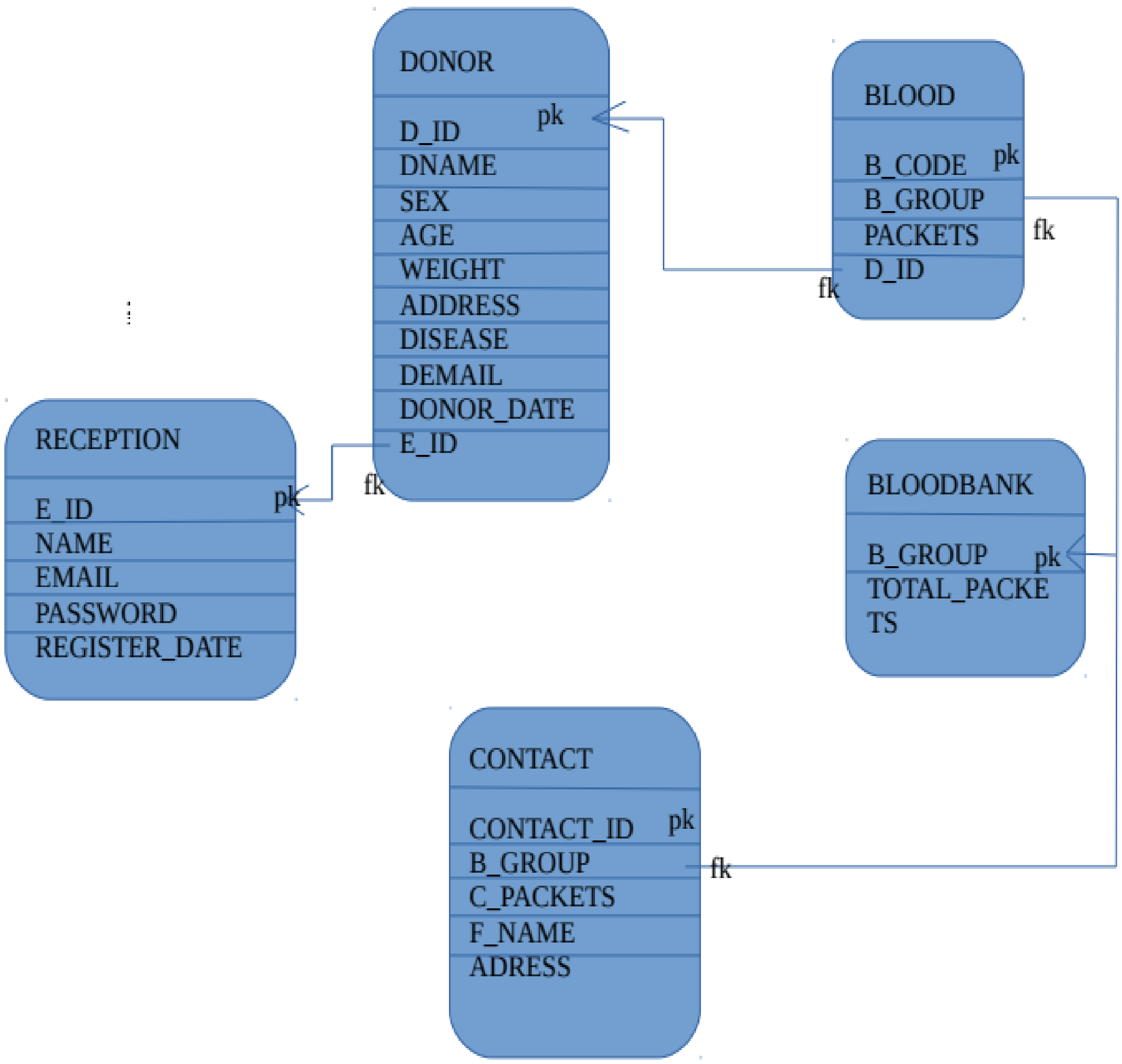


Fig2.1 : Relational Schema

2.2 Creation of Database Tables

1.RECEPTION

EID	VARCHAR
NAME	VARCHAR
EMAIL	VARCHAR
PASSWORD	VARCHAR
REGISTER_DATE	TIMESTAMP

CODE:

cur.execute("INSERT INTO RECEPTION(E_ID,NAME,EMAIL,PASSWORD) VALUES(%s, %s, %s, %s)",(e_id, name, email, password))

2.DONOR

D_ID	INT NOT NULL AUTO_INCREMENT
DNAME	VARCHAR
SEX	VARCHAR
AGE	INT
WEIGHT	INT
ADDRESS	VARCHAR
DISEASE	VARCHAR
DEMAIL	VARCHAR
DONOR_DATE	TIMESTAMP

CODE:

cur.execute("INSERT INTO DONOR(DNAME,SEX,AGE,WEIGHT,ADDRESS,DISEASE,DEMAIL) VALUES(%s, %s, %s, %s, %s, %s, %s)",(dname , sex, age, weight, address, disease, demail))

3.BLOOD

B_CODE	INT NOT NULL AUTO_INCREMENT
D_ID	INT
B_GROUP	VARCHAR
PACKETS	INT

CODE:
cur.execute("INSERT INTO BLOOD(D_ID,B_GROUP,PACKETS) VALUES(%s, %s, %s)",(d_id , blood_group, packets))

4.BLOODBANK

B_GROUP	VARCHCAR
TOTAL_PACKETS	INT

CODE:
cur.execute("SELECT * FROM BLOODBANK")
records = cur.fetchall()
cur.execute("UPDATE BLOODBANK SET TOTAL_PACKETS = TOTAL_PACKETS+%s WHERE B_GROUP = %s",(packets,blood_group))

5.CONTACT

CONTACT_ID	INT
B_GROUP	VARCHAR
C_PACKETS	INT
F_NAME	VARCHAR
ADRESS	VARCHAR

CODE:
cur.execute("INSERT INTO CONTACT(B_GROUP,C_PACKETS,F_NAME,ADRESS) VALUES(%s, %s, %s, %s)",(bgroup, bpackets, fname, adress))

CHAPTER 3

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

3.1 Constraints:

1. RECEPTION Table:

- Primary Key Constraint: E_ID (Column)

2.DONOR Table:

- Primary Key Constraint: D_ID (Column)
- Foreign Key Constraint: D_ID (Column) references DONOR(D_ID), referencing BLOOD(D_ID) on delete cascade and update cascade

3. BLOODBANK Table:

- Primary Key Constraint: B_GROUP (Column)

4. BLOOD Table:

- Primary Key Constraint: B_CODE (Column)
- Foreign Key Constraint: D_ID (Column) references DONOR(D_ID), referencing BLOOD(D_ID) on delete cascade and update cascade
- Foreign Key Constraint: B_GROUP (Column) references BLOODBANK(B_GROUP), referencing BLOOD(B_GROUP) on delete cascade and update cascade

5. CONTACT Table:

- Primary Key Constraint: CONTACT_ID (Column)

- Unique Key Constraint: B_GROUP (Column)
- Foreign Key Constraint: B_GROUP (Column) references BLOODBANK(B_GROUP) on delete cascade and update cascade

6. NOTIFICATIONS Table:
- Primary Key Constraint: N_ID (Column)

3.2 Sets:

1. Retrieve all donors with a specific blood group:

```sql
SELECT *
FROM DONOR
WHERE B_GROUP = 'A+';
```

2. Retrieve all blood packets along with their respective blood groups:

```sql
SELECT B_CODE, D_ID, B_GROUP, PACKETS
FROM BLOOD;
```

3. Retrieve the total number of packets donated by a specific donor:

```sql
SELECT D_ID, SUM(PACKETS) AS TotalDonatedPackets
FROM BLOOD
WHERE D_ID = 123
GROUP BY D_ID;
```

4. Retrieve all blood donation records along with donor information:

```sql
SELECT b.B_CODE, b.D_ID, b.B_GROUP, b.PACKETS, d.DNAME, d.SEX, d.AGE,
d.WEIGHT, d.ADDRESS
FROM BLOOD b
JOIN DONOR d ON b.D_ID = d.D_ID;
```

5. Retrieve all blood groups available in the blood bank along with their total packets:

```sql
SELECT B_GROUP, TOTAL_PACKETS
FROM BLOODBANK;
```

6. Retrieve all recipients along with the blood group they require:

```sql
SELECT *
FROM CONTACT;
```

3.3 Joins:

1. Join between DONOR and BLOOD tables:

```sql
SELECT d.D_ID, d.DNAME, d.SEX, d.AGE, d.WEIGHT, d.ADDRESS, d.DISEASE,
d.DEMAIL, d.DONOR_DATE,
       b.B_CODE, b.B_GROUP, b.PACKETS
FROM DONOR d
```

```
JOIN BLOOD b ON d.D_ID = b.D_ID;
```

2. Join between CONTACT and BLOODBANK tables:

```sql
SELECT c.CONTACT_ID, c.B_GROUP, c.C_PACKETS, c.F_NAME, c.ADRESS, b.TOTAL_PACKETS
FROM CONTACT c
JOIN BLOODBANK b ON c.B_GROUP = b.B_GROUP;
```

3. Join between NOTIFICATIONS and BLOODBANK tables:

```sql
SELECT n.N_ID, n.NB_GROUP, n.N_PACKETS, n.NF_NAME, n.NADRESS, b.TOTAL_PACKETS
FROM NOTIFICATIONS n
JOIN BLOODBANK b ON n.NB_GROUP = b.B_GROUP;
```

4. Join between DONOR and CONTACT tables:

```sql
SELECT d.D_ID, d.DNAME, d.SEX, d.AGE, d.WEIGHT, d.ADDRESS, d.DISEASE, d.DEMAIL, d.DONOR_DATE,
       c.CONTACT_ID, c.B_GROUP, c.C_PACKETS, c.F_NAME, c.ADRESS
FROM DONOR d
JOIN CONTACT c ON d.B_GROUP = c.B_GROUP;
```

5. Join between DONOR and NOTIFICATIONS tables:

```sql
SELECT d.D_ID, d.DNAME, d.SEX, d.AGE, d.WEIGHT, d.ADDRESS, d.DISEASE, d.DEMAIL, d.DONOR_DATE,
```

```
    n.N_ID, n.NB_GROUP, n.N_PACKETS, n.NF_NAME, n.NADRESS
FROM DONOR d
JOIN NOTIFICATIONS n ON d.B_GROUP = n.NB_GROUP;
```

3.4 View:

Sure, I'll prepare views for your blood bank project using similar logic:

1. Top Selling Products:
```sql
CREATE VIEW top_selling_products AS
SELECT B_GROUP, SUM(PACKETS) AS total_donated_packets
FROM BLOOD
GROUP BY B_GROUP
ORDER BY total_donated_packets DESC;
```

2. Customer Lifetime Value:
```sql
CREATE VIEW customer_lifetime_value AS
SELECT D_ID, COUNT(*) AS lifetime_donations
FROM BLOOD
GROUP BY D_ID;
```

3. Inventory Status:
```sql
CREATE VIEW inventory_status AS
SELECT B_GROUP, SUM(PACKETS) AS available_packets
FROM BLOOD
GROUP BY B_GROUP;
```

4. Sales by Category and Month:
```sql
CREATE VIEW donations_by_blood_group AS
SELECT B_GROUP, MONTH(DONOR_DATE) AS month, SUM(PACKETS) AS
total_donated_packets
FROM BLOOD
GROUP BY B_GROUP, MONTH(DONOR_DATE);
```

5. Customer Order History:
```sql
CREATE VIEW donor_order_history AS
```

```
SELECT D_ID, B_CODE, DONOR_DATE
FROM BLOOD;
```

3.5 Triggers:

Apologies for the oversight. Let me adjust the triggers and cursors to fit the tables you provided for the blood bank project.

Triggers:

1. Inventory Management:

```sql
CREATE TRIGGER update_inventory AFTER INSERT ON BLOOD
FOR EACH ROW
BEGIN
   UPDATE BLOODBANK
   SET TOTAL_PACKETS = TOTAL_PACKETS + NEW.PACKETS
   WHERE B_GROUP = NEW.B_GROUP;
END;
```

2. Donor Last Donation Update:

```sql
CREATE TRIGGER update_donor_last_donation AFTER INSERT ON BLOOD
FOR EACH ROW
BEGIN
   UPDATE DONOR
   SET DONOR_DATE = CURRENT_TIMESTAMP
   WHERE D_ID = NEW.D_ID;
END;
```

3. Automated Alerts for Low Inventory:

```sql
CREATE TRIGGER low_inventory_alert AFTER INSERT ON BLOOD
FOR EACH ROW
BEGIN
   DECLARE available_packets INT;
   SET available_packets = (SELECT TOTAL_PACKETS FROM BLOODBANK WHERE B_GROUP = NEW.B_GROUP);
   IF available_packets < 10 THEN
      -- Add code to send alert/notification for low inventory
      -- Example: INSERT INTO NOTIFICATIONS (NB_GROUP, N_PACKETS, NF_NAME, NADRESS) VALUES (NEW.B_GROUP, available_packets, 'Low inventory alert', 'Blood bank');
   END IF;
END;
```

4. Donor Age Check:

```sql
CREATE TRIGGER agecheck BEFORE INSERT ON DONOR
FOR EACH ROW
BEGIN
   IF NEW.AGE < 18 THEN
      SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Donor must be at least 18 years old';
   END IF;
END;
```

Cursors:

For the blood bank tables, cursors might not be necessary for the given operations. However, if you have specific scenarios where you'd like to use cursors, please provide details, and I'll be happy to assist you in creating them.

3.6 Cursors:

1. Cursor for Updating Product Prices (adapted for updating available packets in the BLOODBANK table):

```sql
DELIMITER //

CREATE PROCEDURE update_inventory()
BEGIN
   DECLARE done INT DEFAULT FALSE;
   DECLARE current_group VARCHAR(4);
   DECLARE current_packets INT;

   -- Declare cursor for selecting blood groups and packets from BLOOD table
   DECLARE blood_cursor CURSOR FOR
      SELECT B_GROUP, SUM(PACKETS)
      FROM BLOOD
      GROUP BY B_GROUP;

   -- Declare handler for not found condition
   DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

   OPEN blood_cursor;

   -- Start cursor loop
   inventory_loop: LOOP
      FETCH blood_cursor INTO current_group, current_packets;
```

```
        IF done THEN
            LEAVE inventory_loop;
        END IF;

        -- Update available packets in BLOODBANK table
        UPDATE BLOODBANK
        SET TOTAL_PACKETS = TOTAL_PACKETS + current_packets
        WHERE B_GROUP = current_group;
    END LOOP;

    CLOSE blood_cursor;
END//

DELIMITER ;
```

CHAPTER 4

ANALYZING THE PITFALLS, IDENTIFYING THE DEPENDENCIES, AND APPLYING NORMALIZATIONS

4.1 Analyzing the pitfalls

1. User Experience (UX):

- Slow loading speed of the website or mobile app can frustrate users, especially during emergencies when quick access to blood donation information is crucial.
- Complicated site navigation can deter potential donors from finding relevant information about donation centers, eligibility criteria, and donation procedures.
- Lack of clear guidance and assistance in the donation process can discourage donors from completing the donation.

2. Crawlability Issues:

- Irrelevant pages or outdated information related to blood donation might confuse search engine bots and users. Ensuring that only relevant and up-to-date information is indexed can improve search engine visibility.
- Regular audits and monitoring of website content can help identify and rectify any crawlability issues, ensuring that search engine bots can access important information efficiently.

3. Technical SEO Issues:

- Issues such as broken links, missing metadata, or improper use of headings can affect the website's ranking in search engine results pages (SERPs).
- Implementing best practices for technical SEO, such as optimizing page speed, using proper HTML markup, and ensuring mobile-friendliness, can improve the website's visibility and accessibility.

4. Low Content Quality:

- Insufficient information about blood donation processes, eligibility criteria, and benefits can lead to confusion and hesitancy among potential donors.
- Providing comprehensive and accurate content, including FAQs, testimonials, and

success stories, can help educate and motivate donors to participate.

5. Sitemap Issues:

- Incorrectly configured sitemaps or missing sitemap files can hinder search engine crawlers' ability to discover and index important pages related to blood donation.
- Regularly updating and maintaining sitemaps can ensure that search engines have access to all relevant pages on the website.

6. lack of Engagement:

- Not actively engaging with the community or failing to promote blood donation events and initiatives can result in low participation rates.
- Utilizing social media platforms, email newsletters, and community outreach programs can help raise awareness and encourage participation in blood donation drives.

7. Trust and Security Concerns:

- Inadequate measures to ensure the security and privacy of donor information can undermine trust and deter potential donors from participating.
- Implementing robust security protocols, obtaining necessary certifications, and clearly communicating privacy policies can help build trust with donors and stakeholders.

8. Inefficient Donation Process:

- Cumbersome registration or donation processes, including lengthy forms or unclear instructions, can discourage potential donors from completing the donation process.
- Streamlining the donation process, providing clear instructions, and offering convenient scheduling options can improve donor satisfaction and retention.

9. Lack of Feedback Mechanisms:

- Failing to collect feedback from donors and stakeholders can result in missed opportunities for improvement and innovation.
- Implementing feedback mechanisms such as surveys, feedback forms, and suggestion boxes can help gather valuable insights and enhance the overall donor experience.

10. Inadequate Communication:

- Poor communication regarding donation requirements, eligibility criteria, and health guidelines can lead to confusion and misinformation among potential donors.

- Utilizing various communication channels, including websites, social media, and educational materials, can help disseminate accurate and timely information to donors and stakeholders.

4.2 Identifying the Dependencies

1. Design Dependencies:

- The design phase depends on the completion of the requirements gathering phase, where the specific needs and objectives of the blood bank project are defined.

2. Development Dependencies:

- The development phase depends on the completion of the design phase. Developers require finalized design specifications to begin coding the blood bank system.

3. Testing Dependencies:

- The testing phase depends on the completion of the development phase. Testers need access to the developed blood bank system to assess its functionality and identify any bugs or issues.

4. Content Dependencies:

- The content creation phase depends on the completion of the design phase. Content creators need to understand the layout and design of the blood bank system to produce relevant content such as informational pages, FAQs, and donation guidelines.

5. Launch Dependencies:

- The launch phase depends on the completion of all previous phases. The blood bank system cannot go live until it has been thoroughly tested, reviewed, and approved by stakeholders.

6. Maintenance Dependencies:

- The maintenance phase depends on the launch phase. Once the blood bank system is live, it requires ongoing maintenance and updates to ensure data accuracy, security, and optimal performance.

7. Database Integration:

- The integration of the database depends on the completion of the development phase. Developers need to finalize the structure and functionalities of the blood bank system before integrating it with the database for storing donor information, blood inventory, and other relevant data.

8. User Training:

- User training depends on the completion of the development and testing phases. Training sessions for blood bank staff members need to be scheduled after the system is developed and tested to ensure they understand how to effectively use the system for tasks such as donor management and inventory tracking.

9. Security Implementation:

- Implementation of security measures depends on the completion of the development phase. Security protocols, such as encryption of sensitive donor data and access controls, need to be established once the system functionalities are finalized.

10. Regulatory Compliance:

- Compliance with regulatory requirements depends on the completion of the development and testing phases. Legal and regulatory considerations, such as data protection laws and healthcare regulations, need to be addressed before the blood bank system can be launched to ensure compliance and avoid legal issues.

4.3 Normalization

ORIGINAL TABLES

Reception Table:

Field	Data Type	Key
ReceptionID	INT	Primary
ReceptionistName	VARCHAR(50)	
Date	DATE	

Blood Table:

Field	Data Type	Key	Reference
BloodID	INT	Primary	
BloodType	VARCHAR(10)		
BloodVolume	INT		
ExpiryDate	DATE		
DonationDate	DATE		
DonorID	INT	Foreign	Donor.DonorID

BloodBank Table:

Field	Data Type	Key
BankID	INT	Primary
BankName	VARCHAR(50)	
Location	VARCHAR(100)	
ContactNumber	VARCHAR(20)	
Email	VARCHAR(100)	

Contacts Table:

Field	Data Type	Key	Reference
ContactID	INT	Primary	
Name	VARCHAR(50)		
Phone	VARCHAR(20)		
Email	VARCHAR(100)		
BloodBankID	INT	Foreign	BloodBank.BankID

Donor Table:

Field	Data Type	Key
DonorID	INT	Primary
DonorName	VARCHAR(50)	
Age	INT	
BloodType	VARCHAR(10)	
ContactNumber	VARCHAR(20)	
Address	VARCHAR(100)	

1st Normal Form

- A relation will be 1NF if it contains an atomic value.
- It states that an attribute of a table cannot hold multiple values. It must hold only single-valued attribute.
- First normal form disallows the multi-valued attribute, composite attribute, and their combinations.

After:

1. Reception Table (NF1):

ReceptionID (Primary Key)	ReceptionistName	Date
1	John Doe	2024-04-15
2	Jane Smith	2024-04-16

2. Blood Table (NF1):

BloodID (Primary Key)	BloodType	BloodVolume	ExpiryDate	DonationDate	DonorID (Foreign Key)
1	O+	500	2024-06-30	2024-04-10	1
2	A-	300	2024-07-15	2024-04-12	2

3. BloodBank Table (NF1):

BankID (Primary Key)	BankName	Location	ContactNumber	Email
1	City Blood	New York	555-123-4567	info@citybloodbank.com

4. Contacts Table (NF1):

ContactID (Primary Key)	Name	Phone	Email	BloodBankID (Foreign Key)
1	Alice	555-987-6543	alice@email.com	1
2	Bob	555-876-5432	bob@email.com	1

5. Donor Table (NF1):

DonorID (Primary Key)	DonorName	Age	BloodType	ContactNumber	Address
1	Mary	25	O+	555-111-2222	123 Main St
2	David	30	A-	555-222-3333	456 Elm St

Here , the multivalued attributes are decomposed into atomic values.

2nd Normal Form

- In the 2NF, relational must be in 1NF.
- In the second normal form, all non-key attributes are fully functional dependent on the primary key

After:

1. Reception Table (NF2):

ReceptionID (Primary Key)	ReceptionistID (Primary Key)	ReceptionistName	Date
1	1	John Doe	2024-04-15
2	2	Jane Smith	2024-04-16

2. Blood Table (NF2):

BloodID (Primary Key)	BloodType	BloodVolume	ExpiryDate	DonationDate	DonorID (Primary Key)
1	O+	500	2024-06-30	2024-04-10	1
2	A-	300	2024-07-15	2024-04-12	2

3. BloodBank Table (NF2):

BankID (Primary Key)	BankName	Location	ContactNumber	Email
1	City Blood	New York	555-123-4567	info@citybloodbank.com

4. Contacts Table (NF2):

ContactID (Primary Key)	Name	Phone	Email	BloodBankID (Primary Key)
1	Alice	555-987-6543	alice@email.com	1
2	Bob	555-876-5432	bob@email.com	1

5. Donor Table (NF2):

DonorID (Primary Key)	DonorName	Age	BloodType	ContactNumber	Address
1	Mary	25	O+	555-111-2222	123 Main St
2	David	30	A-	555-222-3333	456 Elm St

3rd Normal Form and BCNF

- A relation will be in 3NF if it is in 2NF and not contain any transitive partial dependency.
- 3NF is used to reduce the data duplication. It is also used to achieve the data integrity.
- If there is no transitive dependency for non-prime attributes, then the relation must be in third normal form.
- BCNF is the advance version of 3NF. It is stricter than 3NF.

- A table is in BCNF if every functional dependency X → Y, X is the super key of the table.
- For BCNF, the table should be in 3NF, and for every FD, LHS is super key.

After :

1. Reception Table (BCNF):

ReceptionID (Primary Key)	ReceptionistID (Primary Key)	Date
1	1	2024-04-15
2	2	2024-04-16

2. Receptionist Table (BCNF):

ReceptionistID (Primary Key)	ReceptionistName
1	John Doe
2	Jane Smith

3. Blood Table (BCNF):

BloodID (Primary Key)	BloodType	BloodVolume	ExpiryDate	DonationDate	DonorID (Primary Key)
1	O+	500	2024-06-30	2024-04-10	1
2	A-	300	2024-07-15	2024-04-12	2

4. Donor Table (BCNF):

DonorID (Primary Key)	DonorName	Age	BloodType	ContactNumber	Address
1	Mary	25	O+	555-111-2222	123 Main St
2	David	30	A-	555-222-3333	456 Elm St

5. BloodBank Table (BCNF):

BankID (Primary Key)	BankName	Location	ContactNumber	Email
1	City Blood	New York	555-123-4567	info@citybloodbank.com

6. **Contacts Table (BCNF):**

ContactID (Primary Key)	Name	Phone	Email	BloodBankID (Primary Key)
1	Alice	555-987-6543	alice@email.com	1
2	Bob	555-876-5432	bob@email.com	1

4th Normal Form

- A relation will be in 4NF if it is in Boyce Codd normal form and has no multi-valued dependency.
- For a dependency A → B, if for a single value of A, multiple values of B exists, then the relation will be a multi-valued dependency.

After :

1. **Reception Table (4NF):**

ReceptionID (Primary Key)	ReceptionistID (Primary Key)	Date
1	1	2024-04-15
2	2	2024-04-16

2. **Receptionist Table (4NF):**

ReceptionistID (Primary Key)	ReceptionistName
1	John Doe
2	Jane Smith

3. Blood Table (4NF):

BloodID (Primary Key)	BloodType	BloodVolume	ExpiryDate	DonationDate	DonorID (Primary Key)
1	O+	500	2024-06-30	2024-04-10	1
2	A-	300	2024-07-15	2024-04-12	2

4. Donor Table (4NF):

DonorID (Primary Key)	DonorName	Age	BloodType	ContactNumber	Address
1	Mary	25	O+	555-111-2222	123 Main St
2	David	30	A-	555-222-3333	456 Elm St

5. BloodBank Table (4NF):

BankID (Primary Key)	BankName	Location	ContactNumber	Email
1	City Blood	New York	555-123-4567	info@citybloodbank.com

6. Contacts Table (4NF):

ContactID (Primary Key)	Name	Phone	Email	BloodBankID (Primary Key)
1	Alice	555-987-6543	alice@email.com	1
2	Bob	555-876-5432	bob@email.com	1

5th Normal Form

- A relation is in 5NF if it is in 4NF and not contains any join dependency and joining should be lossless.
- 5NF is satisfied when all the tables are broken into as many tables as possible in order to avoid redundancy.

- 5NF is also known as Project-join normal form (PJ/NF).

After :

1. Reception Table (5NF):

ReceptionID (Primary Key)	Date
1	2024-04-15
2	2024-04-16

2. Receptionist Table (5NF):

ReceptionistID (Primary Key)	ReceptionistName
1	John Doe
2	Jane Smith

3. Blood Table (5NF):

BloodID (Primary Key)	BloodType	BloodVolume	ExpiryDate	DonationDate	DonorID (Primary Key)
1	O+	500	2024-06-30	2024-04-10	1
2	A-	300	2024-07-15	2024-04-12	2

4. Donor Table (5NF):

DonorID (Primary Key)	DonorName	Age	BloodType	ContactNumber	Address
1	Mary	25	O+	555-111-2222	123 Main St
2	David	30	A-	555-222-3333	456 Elm St

5. BloodBank Table (5NF):

BankID (Primary Key)	BankName	Location	ContactNumber	Email
1	City Blood	New York	555-123-4567	info@citybloodbank.com

6. Contacts Table (5NF):

ContactID (Primary Key)	Name	Phone	Email	BloodBankID (Primary Key)
1	Alice	555-987-6543	alice@email.com	1
2	Bob	555-876-5432	bob@email.com	1

CHAPTER 5

IMPLEMENTATION OF CONCURRENCY CONTROL AND RECOVERY MECHANISMS

5.1 Concurrency Control Mechanisms in Database Systems

Concurrency control mechanisms are a critical aspect of database management systems (DBMS) that ensure data integrity and consistency when multiple users or processes attempt to access and modify data simultaneously. These mechanisms act as a coordinated traffic control system, preventing data conflicts and maintaining the expected state of the database.

1. Two Phase Locking Protocol

A transaction is said to follow the Two-Phase Locking protocol if Locking and Unlocking can be done in two phases.

- **Growing Phase:** New locks on data items may be acquired but none can be released.
- **Shrinking Phase:** Existing locks may be released but no new locks can be acquired.

<u>EXAMPLE</u>

- Transaction T1 begins updating a donor's address.
- T1 acquires an exclusive lock on the donor record being updated.
- Meanwhile, Transaction T2 attempts to update the same donor's email address but is blocked by the lock held by T1.
- After T1 completes its update, it releases the lock, allowing T2 to acquire its own lock and proceed with the update.

```sql
CREATE TABLE DONOR_AFTER_2PL (
    D_ID INT PRIMARY KEY,
    NAME VARCHAR(100),
    ADDRESS VARCHAR(150),
    EMAIL VARCHAR(100)
);

-- Inserting sample data into DONOR_AFTER_2PL table (same as DONOR_BEFORE)
INSERT INTO DONOR_AFTER_2PL (D_ID, NAME, ADDRESS, EMAIL) VALUES
(1, 'John Doe', '123 Main St', 'john@example.com'),
(2, 'Jane Smith', '456 Elm St', 'jane@example.com');
```

2. Multiversion concurrency control (MVCC)

Multi-version protocol aims to reduce the delay for read operations. It maintains multiple versions of data items. Whenever a write operation is performed, the protocol creates a new version of the transaction data to ensure conflict-free and successful read operations.

- **Content** − This field contains the data value of that version.
- **Write_timestamp** − This field contains the timestamp of the transaction that created the new version.
- **Read_timestamp** − This field contains the timestamp of the transaction that will read the newly created value.

By creating multiple versions of the data, the multi-version protocol ensures that read operations can access the appropriate version of the data without encountering conflicts. The protocol thus enables efficient concurrency control and reduces delays in read operations.

EXAMPLE

Transaction T1 starts updating a donor's address.

MVCC creates a new version of the donor record with the updated address.

Meanwhile, Transaction T2 reads the donor's information, accessing the previous version of the donor record without waiting for T1 to commit.

After T1 commits its update, T2 continues to read the old version of the donor record until a subsequent read operation retrieves the new version.

DONOR TABLE

```
CREATE TABLE DONOR_AFTER_MVCC (
    D_ID INT PRIMARY KEY,
    NAME VARCHAR(100),
    ADDRESS VARCHAR(150),
    EMAIL VARCHAR(100)
);

-- Inserting sample data into DONOR_AFTER_MVCC table (same as DONOR_BEFORE)
INSERT INTO DONOR_AFTER_MVCC (D_ID, NAME, ADDRESS, EMAIL) VALUES
(1, 'John Doe', '123 Main St', 'john@example.com'),
(2, 'Jane Smith', '456 Elm St', 'jane@example.com');
```

3. Time Stamp Ordering Protocol

The main goal of timestamp ordering is to guarantee serializability, which means that the order in which transactions are completed must create the same outcomes as if they were executed serially. The following are the main goals of timestamp ordering –

- **Transaction Ordering** − In order for the transaction outcomes and timestamps to match, the transactions must be carried out in the right order.
- **Conflict Resolution** − If two transactions are in conflict, the timestamp ordering mechanism must choose between terminating one of the transactions or postponing it until the other transaction is finished
- **Deadlock Prevention** − To avoid deadlocks, which occur while several transactions are awaiting one another's completion, the timestamp ordering mechanism must be used.

Validation Phase − The timestamp ordering method verifies each transaction's timestamp during the validation stage to make sure the transactions are performed in the proper sequence. When one transaction's timestamp is lower than another's, the earlier transaction must be carried out.

Execution Phase − In the execution phase, the timestamp ordering algorithm executes the transactions in the order determined by the validation phase. If there is a conflict between transactions, the algorithm uses a conflict resolution strategy to resolve the conflict. One strategy is to abort the transaction with the lower timestamp, while another strategy is to delay the transaction with the lower timestamp until the other transaction completes.

EXAMPLE

Add a new column **TransactionTimestamp** to the **Donor** table:

```
CREATE TABLE DONOR_AFTER_TIMESTAMP (
    D_ID INT PRIMARY KEY,
    NAME VARCHAR(100),
    ADDRESS VARCHAR(150),
    EMAIL VARCHAR(100)
);

-- Inserting sample data into DONOR_AFTER_TIMESTAMP table (same as DONOR_BEFORE)
INSERT INTO DONOR_AFTER_TIMESTAMP (D_ID, NAME, ADDRESS, EMAIL) VALUES
(1, 'John Doe', '123 Main St', 'john@example.com'),
(2, 'Jane Smith', '456 Elm St', 'jane@example.com');
```

The DONOR_AFTER_TIMESTAMP table remains same because we are only illustrating the Timestamp Ordering Protocol. The protocol ensures that transactions are executed in the order of their timestamps, maintaining consistency and serializability. However, in this case, the table remains unchanged as the transactions were only simulated.

4. Validation Concurrency Control

Validation phase is also known as optimistic concurrency control technique. In the validation based protocol, the transaction is executed in the following three phases:

1. **Read phase:** In this phase, the transaction T is read and executed. It is used to read the value of various data items and stores them in temporary local variables. It can perform all the write operations on temporary variables without an update to the actual database.
2. **Validation phase:** In this phase, the temporary variable value will be validated against the actual data to see if it violates the serializability.
3. **Write phase:** If the validation of the transaction is validated, then the temporary results are written to the database or system otherwise the transaction is rolled back.

EXAMPLE

```sql
CREATE TABLE DONOR_AFTER_VCC (
    D_ID INT PRIMARY KEY,
    NAME VARCHAR(100),
    ADDRESS VARCHAR(150),
    EMAIL VARCHAR(100)
);

-- Inserting sample data into DONOR_AFTER_VCC table (same as DONOR_BEFORE)
INSERT INTO DONOR_AFTER_VCC (D_ID, NAME, ADDRESS, EMAIL) VALUES
(1, 'John Doe', '123 Main St', 'john@example.com'),
(2, 'Jane Smith', '456 Elm St', 'jane@example.com');
```

- Transaction T1 would perform the update operation.
- Transaction T2 would attempt to read the donor's information.
- During the validation phase, the system would check if T2's read operation conflicts with T1's update operation.
- If there are no conflicts, both transactions would be allowed to proceed; otherwise, T2 would be aborted and rolled back.
- This ensures that transactions maintain data consistency and serializability while minimizing contention.

5.2 Recovery Mechanisms

Recovery techniques in database management systems (DBMS) are essential for ensuring data consistency, durability, and reliability, especially in the event of failures or errors. Here are some common recovery techniques used in DBMS:

Backup and restoration

1. Backup Process:

- **Selection:** Identify the data to be backed up. This could be the entire database, specific files, folders, or system configurations.
- **Scheduling:** Determine the frequency of backups. This depends on data criticality and how often data changes. Backups can be:
- **Full:** Backs up all data at a specific time.
- **Incremental:** Backs up only the changes made since the last backup, saving storage space.
- **Differential:** Backs up all changes since the last full backup.
- **Storage:** Choose a secure and reliable storage location for backups. Options include:
- **Local storage:** External hard drives, USB drives (less secure)
- **Remote storage**: Cloud storage services, network-attached storage (NAS)
- **Verification:** Ensure the backup copies are complete and error-free through verification processes.

2. Restoration Process:

When data loss occurs:

- **Selection:** Identify the specific data or system components that need to be restored.
- **Retrieval:** Locate the appropriate backup based on the timeframe when the data was last known to be good.
- **Recovery:** Restore the selected data from the backup to the original location or a designated recovery location.
- **Validation:** Verify that the restored data is complete and usable.

Log-based recovery

Log-based recovery is a technique used in database management systems (DBMS) to recover a database to a consistent state in the event of a system failure or crash. It relies on transaction logs, which record all the changes made to the database.

1. Transaction Logs:

A transaction log is a sequential record of all modifications made to the database during a transaction.

Each transaction record usually includes information like:

Transaction ID (unique identifier for the transaction)

Operation details (e.g., insert, update, delete)

Data items affected by the operation (before and after values)

Transaction status (started, committed, aborted)

2. Recovery Process:

When a system crash occurs, the DBMS uses the transaction log to reconstruct the database to a consistent state:

- **Analyzing the Log:** The DBMS starts by analyzing the transaction log backwards from the point of failure.
- **Redoing Committed Transactions:** It identifies committed transactions (those that successfully completed) and replays their changes on the database to ensure all committed updates are reflected.
- **Undoing Uncommitted Transactions:** Any uncommitted transactions (those that were interrupted by the crash) are identified. The DBMS undoes any changes made by these transactions, ensuring the database doesn't reflect incomplete operations.

Types of Log Records:

There are two main types of log records used in log-based recovery:

- Before-image logging: Records the state of the data item before the modification.

- After-image logging: Records the state of the data item after the modification

Recoverable Schedule

1. **Cascading Schedule**

   A cascading schedule is classified as a recoverable schedule. A recoverable schedule is basically a schedule in which the commit operation of a particular transaction that performs read operation is delayed until the uncommitted transaction either commits or roll backs.

   A cascading rollback is a type of rollback in which if one transaction fails, then it will cause rollback of other dependent transactions. The main disadvantage of cascading rollback is that it can cause CPU time wastage.

2. **Cascadeless schedule**

   When a transaction is not allowed to read data until the last transaction which has written it is committed or aborted, these types of schedules are called cascadeless schedules.

CHAPTER 6

CODE

6.1 App.py

```
from flask import Flask, render_template, flash, redirect, request, url_for, session, logging


from flask_mysqldb import MySQL

from wtforms import Form, StringField, TextAreaField, PasswordField, validators, SelectField

from passlib.hash import sha256_crypt

import random

from functools import wraps



app = Flask(__name__)

app.secret_key='abcd'



#Config MySQL

app.config['MYSQL_HOST']='localhost'

app.config['MYSQL_USER']='root'

app.config['MYSQL_PASSWORD']='123456'

app.config['MYSQL_DB']='bloodbank'

app.config['MYSQL_CURSORCLASS']='DictCursor'

#init MySQL
```

```
mysql =  MySQL(app)



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


@app.route('/contact', methods=['GET','POST'])
def contact():
    if request.method == 'POST':
        bgroup = request.form["bgroup"]
        bpackets = request.form["bpackets"]
        fname = request.form["fname"]
        adress = request.form["adress"]


        #create a cursor
        cur = mysql.connection.cursor()


        #Inserting values into tables
        cur.execute("INSERT INTO
CONTACT(B_GROUP,C_PACKETS,F_NAME,ADRESS)
VALUES(%s, %s, %s, %s)",(bgroup, bpackets, fname, adress))
        cur.execute("INSERT INTO
NOTIFICATIONS(NB_GROUP,N_PACKETS,NF_NAME,NADRESS
) VALUES(%s, %s, %s, %s)",(bgroup, bpackets, fname, adress))
        #Commit to DB
        mysql.connection.commit()
```

```
        #close connection

        cur.close()

        flash('Your request is successfully sent to the Blood Bank','success')

        return redirect(url_for('index'))


    return render_template('contact.html')




class RegisterForm(Form):

    name = StringField('Name', [validators.DataRequired(),validators.Length(min=1,max=25)])

    email = StringField('Email',[validators.DataRequired(),validators.Length(min=10,max=50)])

    password = PasswordField('Password', [

        validators.DataRequired(),

        validators.EqualTo('confirm',message='Password do not match')

    ])

    confirm = PasswordField('Confirm Password')


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

def register():

    form = RegisterForm(request.form)

    if request.method  == 'POST' and form.validate():

        name = form.name.data

        email = form.email.data
```

```
        password = sha256_crypt.encrypt(str(form.password.data))

        e_id = name+str(random.randint(1111,9999))

        #Create cursor

        cur = mysql.connection.cursor()

        cur.execute("INSERT INTO
RECEPTION(E_ID,NAME,EMAIL,PASSWORD) VALUES(%s, %s,
%s, %s)",(e_id, name, email, password))

        #Commit to DB

        mysql.connection.commit()

        #close connection

        cur.close()

        flashing_message = "Success! You can log in with Employee ID "
+ str(e_id)

        flash( flashing_message,"success")


        return redirect(url_for('login'))


    return render_template('register.html',form = form)


#login page

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

def login():

    if request.method == 'POST':

        # Get Form Fields

        e_id = request.form["e_id"]

        password_candidate = request.form["password"]
```

```
        # Create cursor
        cur = mysql.connection.cursor()

        # Get user by username
        result = cur.execute("SELECT * FROM RECEPTION WHERE
E_ID = %s", [e_id])

        if result > 0:
            # Get stored hash
            data = cur.fetchone()
            password = data['PASSWORD']

            # Compare Passwords
            if sha256_crypt.verify(password_candidate, password):
                # Passed
                session['logged_in'] = True
                session['e_id'] = e_id

                flash('You are now logged in', 'success')
                return redirect(url_for('dashboard'))
            else:
                error = 'Invalid login'
                return render_template('login.html', error=error)
            # Close connection
            cur.close()
        else:
```

```
            error = 'Employee ID not found'
            return render_template('login.html', error=error)

    return render_template('login.html')

# Check if user logged in
def is_logged_in(f):
    @wraps(f)
    def wrap(*args, **kwargs):
        if 'logged_in' in session:
            return f(*args, **kwargs)
        else:
            flash('Unauthorized, Please login!', 'danger')
            return redirect(url_for('login'))
    return wrap

#Logout
@app.route('/logout')
@is_logged_in
def logout():
    session.clear()
    flash('You are now logged out', 'success')
    return redirect(url_for('index'))

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

```
@is_logged_in
def dashboard():
    cur = mysql.connection.cursor()
    result = cur.callproc('BLOOD_DATA')
    return_value = result[0] if result else None

    if return_value is not None and return_value > 0:
        details = cur.fetchall()
        return render_template('dashboard.html', details=details)
    else:
        msg = 'Blood Bank is Empty'
        return render_template('dashboard.html', msg=msg)

    # Close connection
    cur.close()



@app.route('/donate', methods=['GET', 'POST'])
@is_logged_in
def donate():
    if request.method  == 'POST':
        # Get Form Fields
        dname = request.form["dname"]
        sex = request.form["sex"]
        age = request.form["age"]
```

```
        weight = request.form["weight"]

        address = request.form["address"]

        disease =  request.form["disease"]

        demail = request.form["demail"]


        #create a cursor

        cur = mysql.connection.cursor()


        #Inserting values into tables

        cur.execute("INSERT INTO
DONOR(DNAME,SEX,AGE,WEIGHT,ADDRESS,DISEASE,DEMAI
L) VALUES(%s, %s, %s, %s, %s, %s, %s)",(dname , sex, age, weight,
address, disease, demail))

        #Commit to DB

        mysql.connection.commit()

        #close connection

        cur.close()

        flash('Success! Donor details Added.','success')

        return redirect(url_for('donorlogs'))


    return render_template('donate.html')


@app.route('/donorlogs')

@is_logged_in

def donorlogs():

    cur = mysql.connection.cursor()

    result = cur.execute("SELECT * FROM DONOR")
```

```
    logs = cur.fetchall()


    if result>0:
        return render_template('donorlogs.html',logs=logs)
    else:
        msg = ' No logs found '
        return render_template('donorlogs.html',msg=msg)
    #close connection
    cur.close()



@app.route('/bloodform',methods=['GET','POST'])
@is_logged_in
def bloodform():
    if request.method  == 'POST':
        # Get Form Fields
        d_id = request.form["d_id"]
        blood_group = request.form["blood_group"]
        packets = request.form["packets"]

        #create a cursor
        cur = mysql.connection.cursor()

        #Inserting values into tables
        cur.execute("INSERT INTO
BLOOD(B_GROUP,PACKETS,D_ID) VALUES(%s, %s, %s)",(
```

```
blood_group, packets,d_id))
        cur.execute("SELECT * FROM BLOODBANK")
        records = cur.fetchall()
        cur.execute("UPDATE BLOODBANK SET TOTAL_PACKETS = 
TOTAL_PACKETS+%s WHERE B_GROUP = 
%s",(packets,blood_group))
        #Commit to DB
        mysql.connection.commit()
        #close connection
        cur.close()
        flash('Success! Donor Blood details Added.','success')
        return redirect(url_for('dashboard'))

    return render_template('bloodform.html')



@app.route('/notifications')
@is_logged_in
def notifications():
    cur = mysql.connection.cursor()
    result = cur.execute("SELECT * FROM CONTACT")
    requests = cur.fetchall()

    if result>0:
        return render_template('notification.html',requests=requests)
    else:
```

```
        msg = ' No requests found '
        return render_template('notification.html',msg=msg)
    #close connection
    cur.close()


@app.route('/notifications/accept')
@is_logged_in
def accept():
    # cur = mysql.connection.cursor()
    # cur.execute("SELECT N_PACKETS FROM NOTIFICATIONS")
    # packets = cur.fetchone()
    # packet = (x[0] for x in packets)
    # cur.execute("SELECT NB_GROUP FROM NOTIFICATIONS")
    # groups = cur.fetchone()
    # group = (y[0] for y in groups)
    #
    # # for row in allnotifications:
    # #     group = row[1]
    # #     packet = row[2]
    # cur.execute("UPDATE BLOODBANK SET TOTAL_PACKETS =
TOTAL_PACKETS-%s WHERE B_GROUP = %s",(packet[-1],group[-
1]))
    # result = "ACCEPTED"
    # cur.execute("INSERT INTO NOTIFICATIONS(RESULT)
VALUES(%s)",(result))
    flash('Request Accepted','success')
    return redirect(url_for('notifications'))
```

```python
@app.route('/notifications/decline')
@is_logged_in
def decline():
    msg = 'Request Declined'
    flash(msg,'danger')
    return redirect(url_for('notifications'))


if __name__ == '__main__':
            app.run(debug=True)
```

6.2 database

```sql
CREATE TABLE RECEPTION(
E_ID VARCHAR(54) PRIMARY KEY,
NAME VARCHAR(100),
EMAIL VARCHAR(100),
PASSWORD VARCHAR(100),
REGISTER_DATE TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE DONOR(
D_ID INT(3) NOT NULL AUTO_INCREMENT,
DNAME VARCHAR(50),
SEX VARCHAR(10),
AGE INT(3),
WEIGHT INT(3),
ADDRESS VARCHAR(150),
DISEASE VARCHAR(50),
DEMAIL VARCHAR(100),
DONOR_DATE TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
CONSTRAINT PK_2 PRIMARY KEY(D_ID)
);

CREATE TABLE BLOODBANK(
B_GROUP VARCHAR(4),
TOTAL_PACKETS INT(4),
CONSTRAINT PK_3 PRIMARY KEY(B_GROUP)
);
```

```
CREATE TABLE BLOOD(
B_CODE INT(4) NOT NULL AUTO_INCREMENT,
D_ID INT(3),
B_GROUP VARCHAR(4),
PACKETS INT(2),
CONSTRAINT PK_4 PRIMARY KEY(B_CODE),
CONSTRAINT FK_1 FOREIGN KEY(D_ID) REFERENCES DONOR(D_ID) ON DELETE
CASCADE ON UPDATE CASCADE,
CONSTRAINT FK_2 FOREIGN KEY(B_GROUP) REFERENCES
BLOODBANK(B_GROUP) ON DELETE CASCADE ON UPDATE CASCADE
);

CREATE TABLE CONTACT(
CONTACT_ID INT(3) NOT NULL AUTO_INCREMENT,
B_GROUP VARCHAR(4),
C_PACKETS INT(2),
F_NAME VARCHAR(50),
ADRESS VARCHAR(250),
CONSTRAINT PK_5 PRIMARY KEY(CONTACT_ID),
CONSTRAINT FK_3 FOREIGN KEY(B_GROUP) REFERENCES
BLOODBANK(B_GROUP) ON DELETE CASCADE ON UPDATE CASCADE
)ENGINE=InnoDB AUTO_INCREMENT=100 DEFAULT CHARSET=latin1;

CREATE TABLE NOTIFICATIONS(
N_ID INT(3) NOT NULL AUTO_INCREMENT,
NB_GROUP VARCHAR(4),
N_PACKETS INT(2),
NF_NAME VARCHAR(50),
NADRESS VARCHAR(250),
CONSTRAINT PK_6 PRIMARY KEY(N_ID)
)ENGINE=InnoDB AUTO_INCREMENT=100 DEFAULT CHARSET=latin1;
```

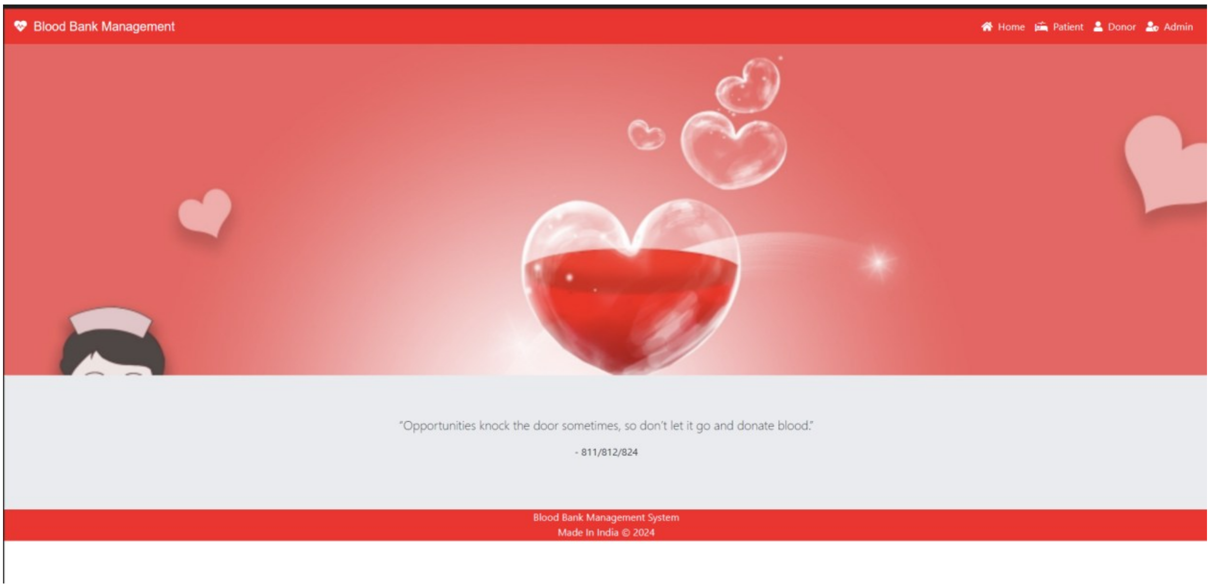
TRIGGERS:

```
mysql> delimiter //
mysql> CREATE TRIGGER agecheck BEFORE INSERT ON DONOR FOR EACH ROW IF
NEW.age < 21 THEN SET NEW.age = 0; END IF;//
mysql> delimiter ;
```

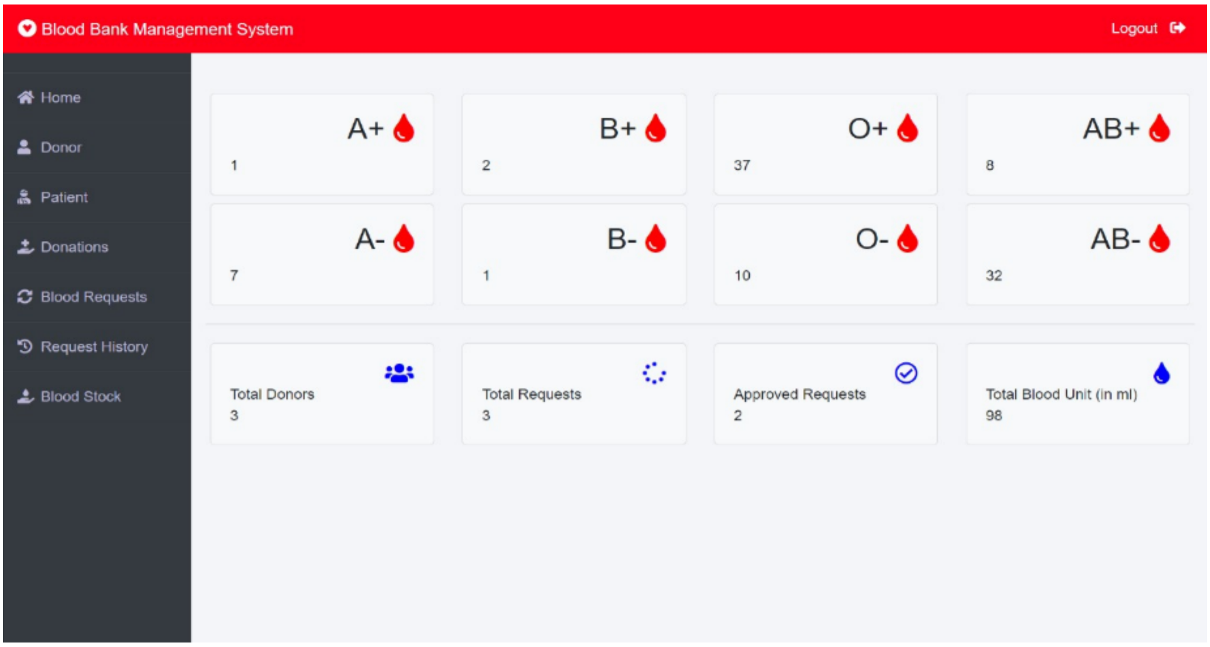
RESULT AND DISCUSSION

7.1 Output Screens

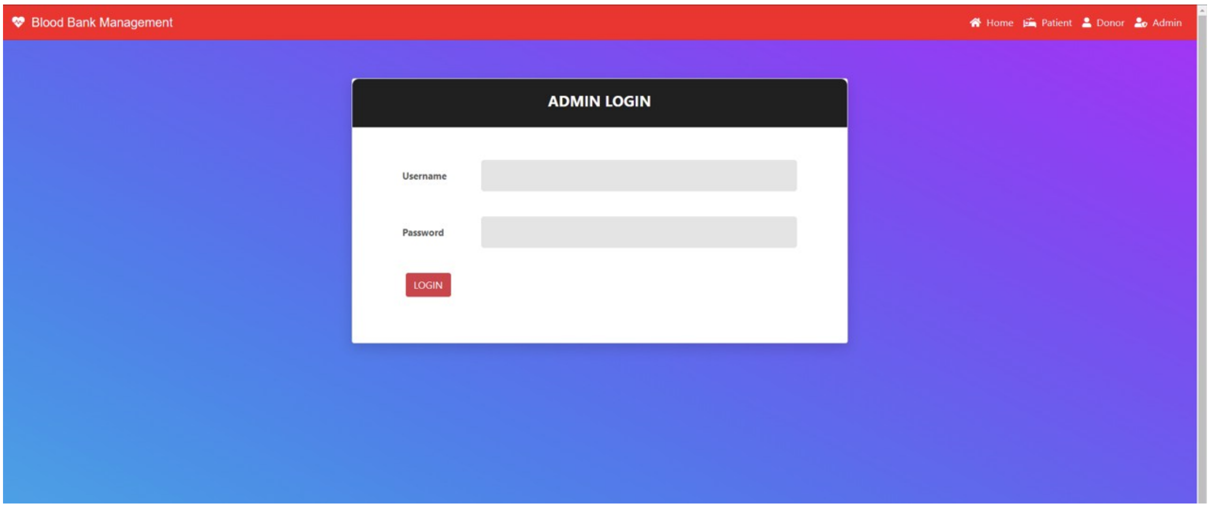
1. Home Page



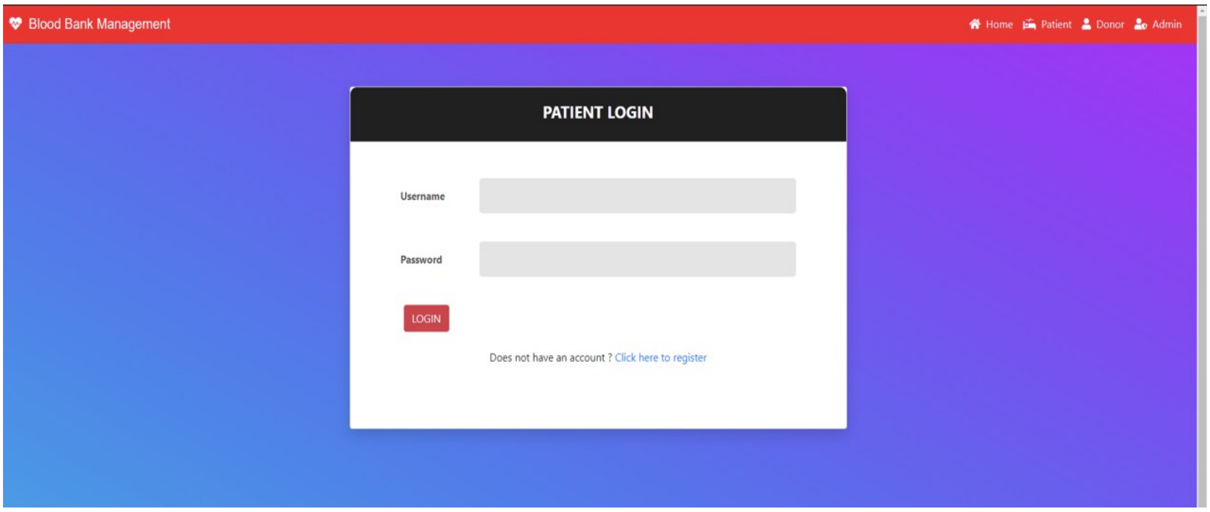
2. Dashboard



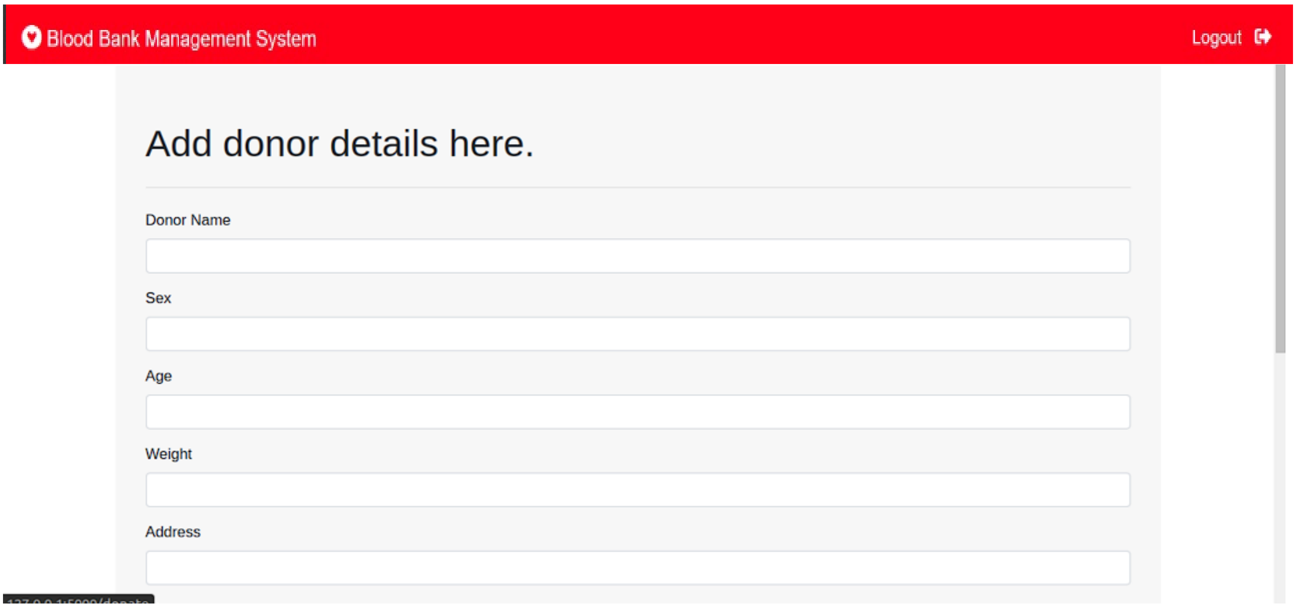
3. Admin login



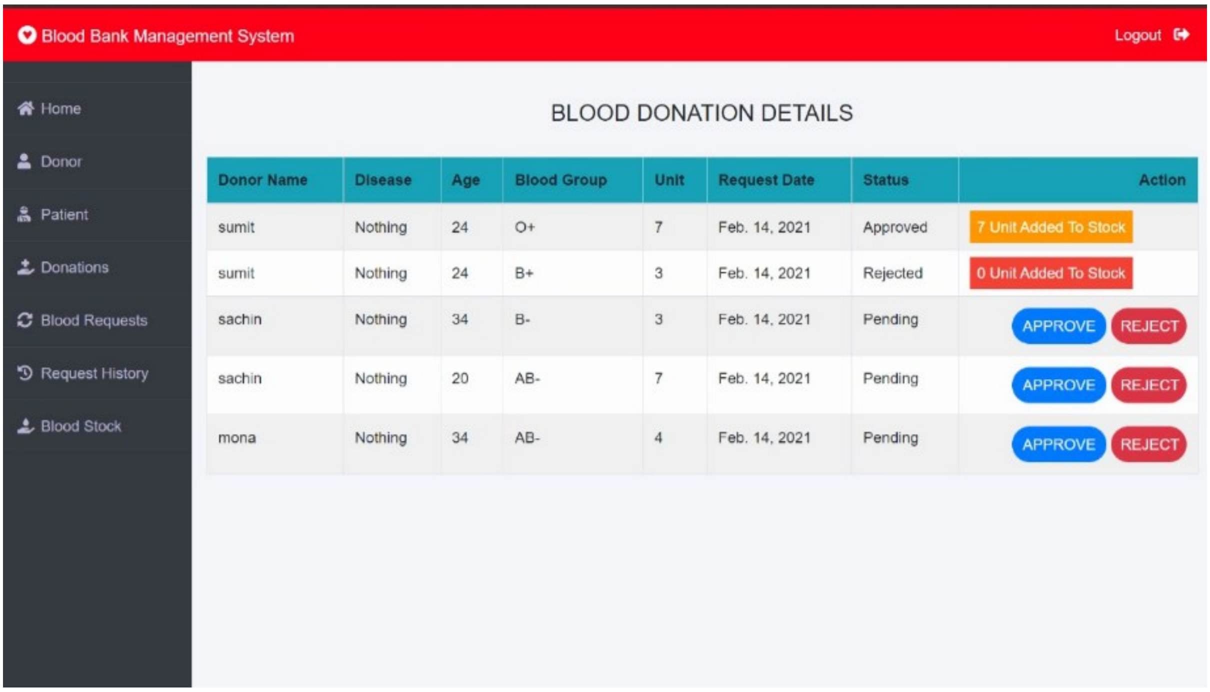
4. Patient Login



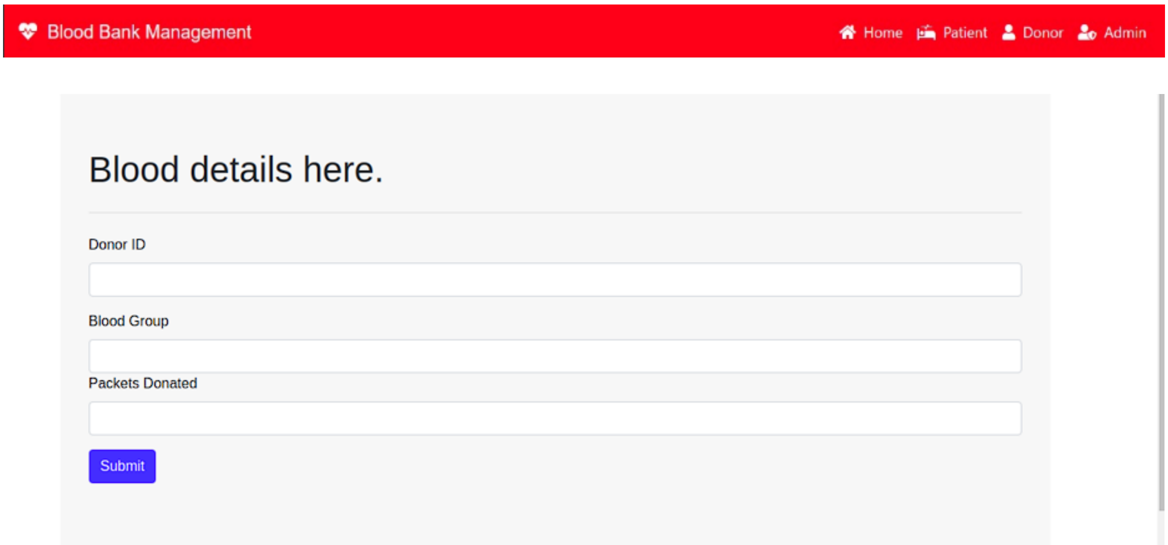
5. Donor Register



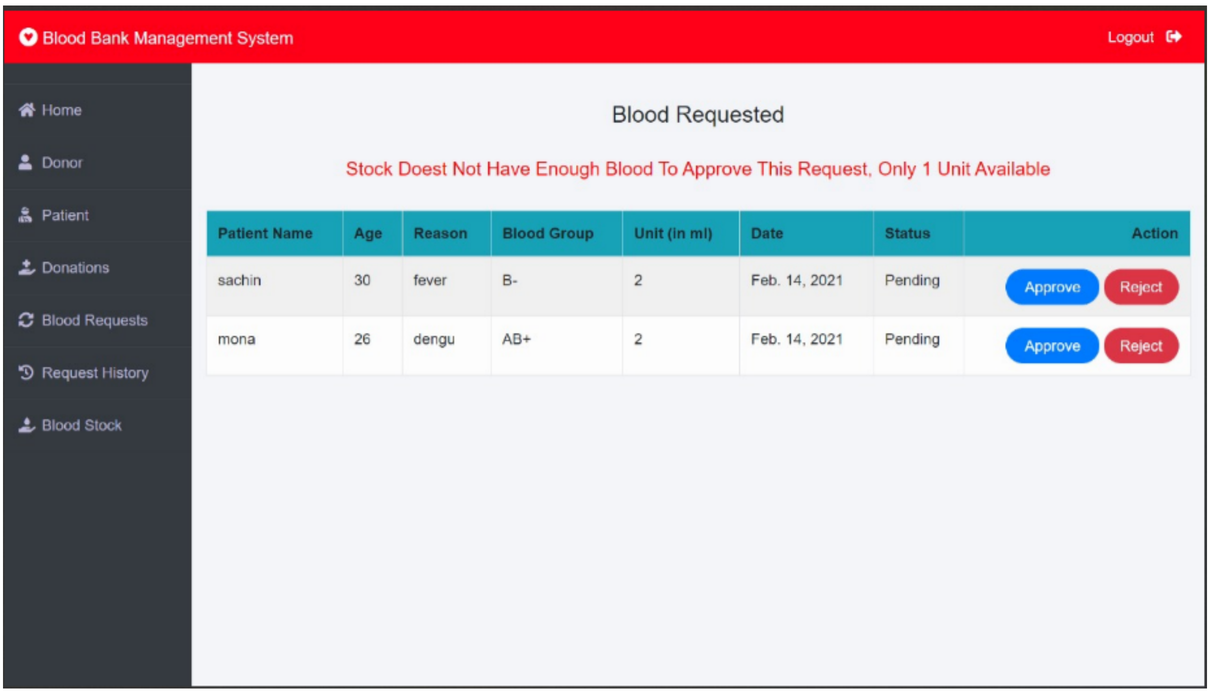
6. Donor Logs



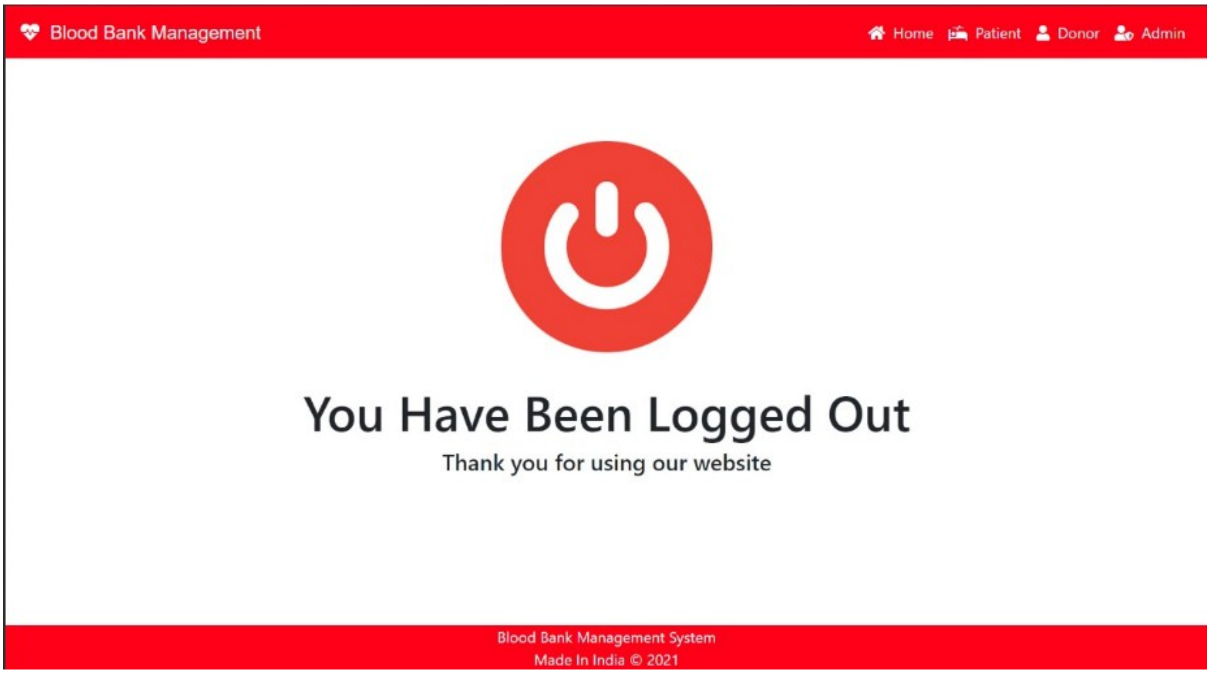
7. Add Donor Blood Details



8. Blood Requests Page



9. Logged Out



Blood Bank Management
Home
Patient
Donor
Admin
You Have Been Logged Out
Thank you for using our website
Blood Bank Management System
Made In India © 2021

7.2 CONCLUSION

In an era marked by continual technological progress, each passing day brings forth new innovations, effectively streamlining and expediting various processes. The proposed system emerges as a solution poised to harness these advancements, particularly in the realm of delivering essential blood supplies swiftly during emergencies. By leveraging modern web-based technologies, this system endeavors to significantly reduce the time required to fulfill urgent blood requests. Through its robust web application, it establishes a vital channel of communication and synchronization between blood donors and the designated blood bank. This digital platform not only facilitates seamless interaction but also empowers users to promptly engage with donors, especially in critical situations where time is of the essence.

Moreover, the system's efficacy hinges significantly on the integrity and reliability of its underlying database infrastructure. Recognizing the pivotal role databases play in ensuring smooth operations, the system mandates regular checks for consistency within the databases housing information on both recipients and the blood bank. These routine inspections are imperative to validate data accuracyand integrity, thereby bolstering the system's overall functionality. Furthermore, continuous monitoring and maintenance of these databasesare essential to preemptively address any potential issues or discrepancies that may arise. By upholding stringent standards of data management and quality assurance, the system endeavors to uphold its promise of efficient and reliable blood supply management, particularly in critical Emergency

7.3 BIBLIOGRAPHY

BOOKS

[1] Python: For Beginners A Crash Course Guide To Learn Python in 1 Week by Timothy C. Needham

[2] MySQL: The Complete Reference by Vikram Vaswani

[3] Flask Framework Cookbook by Shalabh Aggarwal

[4] Essentials of Blood Banking by Mehdi S.R.

REFERENCES

[1] https://docs.python.org/3/tutorial/

[2] https://www.fullstackpython.com/flask.html

[3] http://www.mysqltutorial.org/

[4] https://www.javatpoint.com/mysql-tutorial

[5] https://www.coursera.org/learn/python-databases

[6] https://code.tutsplus.com/tutorials/creating-a-web-app-from-scratch-using-python-flask-and-mysql--cms-22972

[7] https://www.google.com/