

Blood Bank Web Application

A PROJECT REPORT

Submitted By -

Janu Singh (211B152)

Jyoti Singh (211B157)

Sachin Singh (211B261)

Under Guidance Of: Dr. Rahul Pachauri



Jan 2025 - May 2025

Submitted in partial fulfillment for the award of the degree of

Bachelor Of Technology

IN

Computer Science & Engineering

**Department of Computer Science & Engineering
JAYPEE UNIVERSITY OF ENGINEERING & TECHNOLOGY,
AB ROAD, RAGHOGARH, DT. GUNA-473226 MP, INDIA**

Declaration by the Student

We hereby declare that the work reported in the B. Tech. project entitled as “**Blood Bank Web Application**”, in partial fulfillment for the award of degree of Bachelor of Technology submitted at Jaypee University of Engineering and Technology, Guna, as per best of our knowledge and belief there is no infringement of intellectual property right and copyright. In case of any violation I will solely be responsible.

Janu Singh (211B152)

Jyoti Singh (211B157)

Sachin Singh (211B261)

Department of Computer Science and Engineering

Jaypee University of Engineering and Technology

Guna, M.P., India

Date: 14/05/2025



JAYPEE UNIVERSITY OF ENGINEERING & TECHNOLOGY

Grade 'A+' Accredited with by NAAC & Approved U/S 2(f) of the UGC Act, 1956
A.B.Road,Raghogarh,Dist:Guna(M.P.)India,Pin-473226
Phone: 07544 267051, 267310-14, Fax: 07544 267011
Website: www.juet.ac.in

CERTIFICATION

This is to certify that the project titled “**Blood Bank Web Application**” is the bonafide work carried out by **Janu Singh(211b152), Jyoti Singh(211b157) and Sachin Singh(211b261)**. We are students of B.Tech (CSE) at Jaypee University of Engineering and Technology Guna (MP) during the academic year 2024-2025 in partial fulfillment of the requirements for the award of the degree Of Bachelor of Technology (Computer Science and Engineering) and that the project has not formed the basis for the award previously of any other degree, diploma, fellowship or any other similar title.

Dr. Rahul Pachauri

Associate Professor

Signature of the Guide

Department of Computer Science & Engineering
Jaypee University of Engineering and
Technology, Guna, 473226

Date: 14/05/2025

ACKNOWLEDGEMENT

We thank the almighty for giving us the courage & perseverance in completing the project. This project itself is an acknowledgement for all those who have given us their heart-felt cooperation in making it a grand success.

We are also thankful to the project coordinator, **Dr. Rahul Pachauri** for extending their sincere & heartfelt guidance throughout this project work. Without their supervision and guidance, stimulating & constructive criticism, this project would never come out in this form. It is a pleasure to express our deep and sincere gratitude to the Project Guide **Dr. Rahul Pachauri** and are profoundly grateful for the unmatched help rendered by him.

Last but not the least, we would like to express our deep sense and earnest thanksgiving to our dear parents for their moral support and heartfelt cooperation in doing the project. We would also like to thank our friends, whose direct or indirect help has enabled us to complete this work successfully.

Janu Singh (211B152)

Jyoti Singh (211B157)

Sachin Singh (211B261)

Date:14/05/2025

SUMMARY

The project "Blood Bank Web Application" is designed to streamline and enhance the process of blood donation, inventory management, and blood request handling through a centralized, digital platform. This web-based solution serves as a bridge between donors, recipients, and blood banks, ensuring timely availability and efficient distribution of blood units.

The application aims to simplify core operations such as donor registration, blood stock monitoring, and request processing, while also promoting awareness about voluntary blood donation. It supports real-time updates on blood availability and provides automated notifications for donation drives, urgent requirements, and donor eligibility.

Key objectives include reducing the manual workload in blood banks, improving the accuracy of donor and stock records, and ensuring faster response to emergencies. The system covers modules for admin management, hospital access, donor engagement, and reporting tools for analytical insights.

Primary deliverables of the project consist of a user-friendly interface, secure login systems for different roles, comprehensive database integration, and a reliable notification system. While the application offers significant benefits like accessibility, transparency, and operational efficiency, it faces constraints such as data security concerns, internet dependency, and the need for regular system updates.

Assumptions include consistent user engagement, accurate data entry by users, and system scalability for multiple locations. Potential risks involve system downtime, misuse of data, and lack of donor participation, which may affect the efficiency of operations.

Overall, the Blood Bank Web Application presents a practical and socially impactful approach to managing blood donation services, offering a scalable and responsive solution that enhances public health support and emergency responsiveness.

TABLE OF CONTENTS

Title Page.....	i
Declaration by the student.....	ii
Certificate.....	iii
Acknowledgement.....	iv
Summary.....	v
 Chapter 1: Introduction.....	 1-2
1.1 Background.....	1
1.2 Problem Statements.....	1
1.3 Objective of the Project.....	1
1.4 Scope of the Project.....	2
 Chapter 2: System Analysis.....	 3-4
2.1 Existing System and Its Limitations.....	3
2.2 Proposed System Overview.....	3
2.3 Feasibility Study.....	4
2.3.1 Technical Feasibility.....	4
2.3.2 Operational Feasibility.....	4
2.3.3 Economic Feasibility.....	4
2.4 Requirements Gathering.....	4
 Chapter 3: Software Requirements Specification.....	 5-7
3.1 Introduction.....	5
3.2 Functional Requirements.....	5
3.3 Non-Functional Requirements.....	6
3.4 Software and Hardware Requirements.....	7
3.5 Use Case Overview.....	7
 Chapter 4: System and Database Design.....	 8-19
4.1 System Architecture.....	8
4.2 Data Flow Diagram (DFD).....	8
4.3 Entity Relationship Diagram (ERD).....	9
4.4 UML Diagram.....	9-11
4.5 Component Design.....	12
4.6 User Interface Design.....	12
4.7 Database Requirements.....	13
4.8 MongoDB Collections.....	13-17
4.9 Database Relationship.....	18
4.10 Sample Queries.....	18-19
 Chapter 5: Technology Stack and Implementation.....	 20-29
5.1 Frontend-React.js.....	20
5.2 Backend-Node.js and Express.js.....	21
5.3 Database-MongoDB.....	22

5.4 Additional Tools and Libraries.....	22-23
5.5 Development Tools.....	23
5.6 Frontend Implementation (React.js).....	24-26
5.7 Backend Implementation (Node.js and Express.js).....	27
5.8 Database Implementation (MongoDB).....	27
5.9 Integration of Frontend and Backend.....	27-28
5.10 Testing and Debugging.....	28-29
Chapter 6: Testing and Security Features... ..	30-41
6.1 Types of Testing.....	31-32
6.2 Security Testing.....	33-34
6.3 Performance Testing.....	34
6.4 Bug Fixing and Optimization.....	35-41
Chapter 7: Results and Conclusion.....	42-45
Appendix.....	46-51
References.....	52-53

Chapter 1

Introduction

1.1 Background:

Blood is an essential component of the human body, and in medical emergencies, timely access to the right blood type can save lives. However, traditional blood donation systems often rely on outdated, paper-based procedures that can delay the process of finding suitable donors or fulfilling urgent blood requests. A lack of real-time data, poor communication between blood banks, and inefficient inventory management contribute to the shortage and misallocation of blood supplies.

1.2 Problem Statement:

Despite technological advancements, many hospitals and blood banks still lack a centralized, automated system for managing blood donations and requests. Individuals in need often face difficulty locating a suitable donor, especially during emergencies. Manual tracking of donor records, inventory levels, and communication between stakeholders leads to inefficiency and delays.

1.3 Objective of the Project:

The main objective of this project is to design and implement a **web-based blood bank management system** that:

- Provides real-time access to blood availability data.
- Allows users to register as blood donors or recipients.
- Enables admins to manage blood inventory and donor records.
- Offers search functionality for locating donors based on blood type and location.
- Enhances communication between users, hospitals, and blood banks.

1.4 Scope of the Project:

The proposed system is developed using the **MERN stack** (MongoDB, Express.js, React.js, Node.js), ensuring a modern, scalable, and efficient application. The project includes the following features:

- Secure user authentication (donor, recipient, admin).
- Donor registration and profile management.
- Blood request submission and approval system.
- Real-time blood inventory tracking by administrators.
- Search functionality to find blood based on location and type.
- Dashboard and notification features for admins and users.

Chapter 2

System Analysis

Section 2.1: Existing System and Its Limitations:

The traditional blood bank systems used by hospitals and organizations are often manual and lack centralization. In many cases, blood donation data is stored in spreadsheets or paper files, making it difficult to search, retrieve, or update records in real-time. Communication between blood banks and potential donors is slow and inefficient. This outdated process leads to:

- Delayed access to blood during emergencies.
- Difficulty in locating a specific blood type.
- Inaccurate tracking of blood stock levels.
- Higher chances of data loss or human error.
- No real-time updates for users regarding availability.

Section 2.2: Proposed System Overview:

The proposed Blood Bank Web Application overcomes these limitations by providing a centralized, online platform where users, donors, and administrators can interact in real time. Built using the MERN stack (MongoDB, Express.js, React.js, and Node.js), this application ensures fast, secure, and scalable performance. Key benefits of the proposed system include:

- Real-time blood inventory visibility.
- Secure donor and recipient registration.
- Blood search functionality by type and location.
- Admin dashboard for managing donations, requests, and stock.
- Notifications and reminders for eligible donors.

Section 2.3: Feasibility Study:

2.3.1 Technical Feasibility:

The MERN stack offers a robust, scalable, and modular architecture ideal for full-stack web applications. Node.js and Express handle the backend logic efficiently, while React ensures a responsive and user-friendly frontend. MongoDB provides flexible schema-less storage, suitable for rapidly evolving user and donation data.

2.3.2 Operational Feasibility:

The system is designed to be intuitive and easy to use, even for users with minimal technical knowledge. Donors, recipients, and admins can perform their respective functions through a clean web interface, accessible on desktop or mobile browsers. Minimal training is required to operate the system.

2.3.3 Economic Feasibility:

By leveraging open-source technologies (React, Node.js, MongoDB), the development and maintenance costs are significantly reduced. Hosting options such as Vercel, Netlify, and MongoDB Atlas offer free tiers, making the project cost-effective for initial deployment.

Section 2.4 Requirements Gathering:

To ensure the system aligns with real-world needs, requirements were collected through:

- Interviews with healthcare professionals and blood bank managers.
- Online surveys of frequent blood donors.
- Analysis of existing web and mobile-based blood donation platforms.
- Feedback from potential users (e.g., college students, NGOs, volunteers).

Chapter 3

Software Requirements Specification

3.1 Introduction:

This document outlines the functional and non-functional requirements of the Blood Bank Web Application. It serves as a reference for developers, testers, and stakeholders to understand the expected behavior and constraints of the system.

3.2 Functional Requirements:

Functional requirements describe what the system should do and how it should behave in response to specific inputs.

1) User Registration & login

- a. Users can register as donors or recipients.
- b. Secure login/logout with password encryption.

2) Profile Management

- a. Donors can update their personal information and donation history.
- b. Admins can view and manage user profiles.

3) Blood Inventory Management

- a. Admin can add, update, or remove blood units
- b. System automatically updates stock after donations or withdrawals.

4) Blood Search Module

- a. Users can search for blood by type and location.
- b. Results display donor contact info and availability.

5) Blood Request System

- a. Recipients can raise blood requests.
- b. Admin approves or declines based on availability.

6) Notifications

- a. System sends alerts/reminders to eligible donors.
- b. Admin is notified of new blood requests.

7) Admin Dashboard

- a. View statistics on blood availability, donors, and pending requests.
- b. Manage user roles and database entries.

3.3 Non-Functional Requirements:

Non-functional requirements define system attributes such as performance, usability, reliability, and security.

1. Performance

- Application should load within 2 seconds under normal conditions.

2. Scalability

- The system must support increasing users and data without performance degradation.

3. Security

- All passwords must be hashed.
- Role-based access control (RBAC) should restrict unauthorized actions.

4. Availability

- System should have >99% uptime.

5. Usability

- Simple, intuitive UI/UX using React.
- Responsive design for desktops, tablets, and mobile devices.

6. Maintainability

- Code should follow modular, reusable practices for easier debugging and updates.

3.4 Software & Hardware Requirements

Software Requirements

- **Frontend:** React.js
- **Backend:** Node.js with Express
- **Database:** MongoDB
- **Other Tools:** GitHub, Postman, VS Code, MongoDB Atlas

Hardware Requirements

- **Client-Side:** Any device with modern browser (Chrome, Firefox, Edge)
- **Server-Side:** Node.js-compatible server or cloud instance (Heroku, Render, etc.)

3.5 Use Case Overview

A use case diagram identifies the key interactions between users and the system. Major use cases include:

- Register/Login
- Update Profile
- Search for Blood
- Request Blood
- Manage Inventory (Admin)
- Approve/Reject Requests (Admin)

Chapter 4

System and Database Design

4.1 System Architecture

The Blood Bank Web Application follows a three-tier architecture, using the MERN stack:

1. Frontend (Client Layer) – Built with React.js, this layer includes all user interfaces and handles interactions with end users (donors, recipients, admins).
2. Backend (Application Layer) – Managed by Node.js and Express.js, it processes requests, applies business logic, and connects with the database.
3. Database Layer – MongoDB is used for storing all data including user details, blood inventory, donation history, and requests.

4.2 Data Flow Diagram (DFD)

Level 0 DFD (Context Level)

Shows a high-level overview of the system, involving:

- Users (Donors, Recipients, Admins)
- The Web Application
- MongoDB Database

Level 1 DFD

Breaks down the system into sub-processes:

- Registration/Login
- Blood Search
- Inventory Update

- Blood Request Handling
- Notifications

4.3 Entity Relationship Diagram (ERD)

The ER Diagram represents the logical structure of the database.

Entities and Relationships:

- **User:** Stores personal details, role (donor/recipient/admin)
- **Blood Inventory:** Tracks blood type, quantity, date added
- **Request:** Links recipient to blood type requested and status
- **Donation:** Records donation event, donor ID, blood type, date

4.4 UML Diagrams

Class Diagram

Defines the system's object structure:

- Classes: User, Blood Inventory, Request, Donation
- Attributes and Methods for each class
- Relationships between classes (association, inheritance)

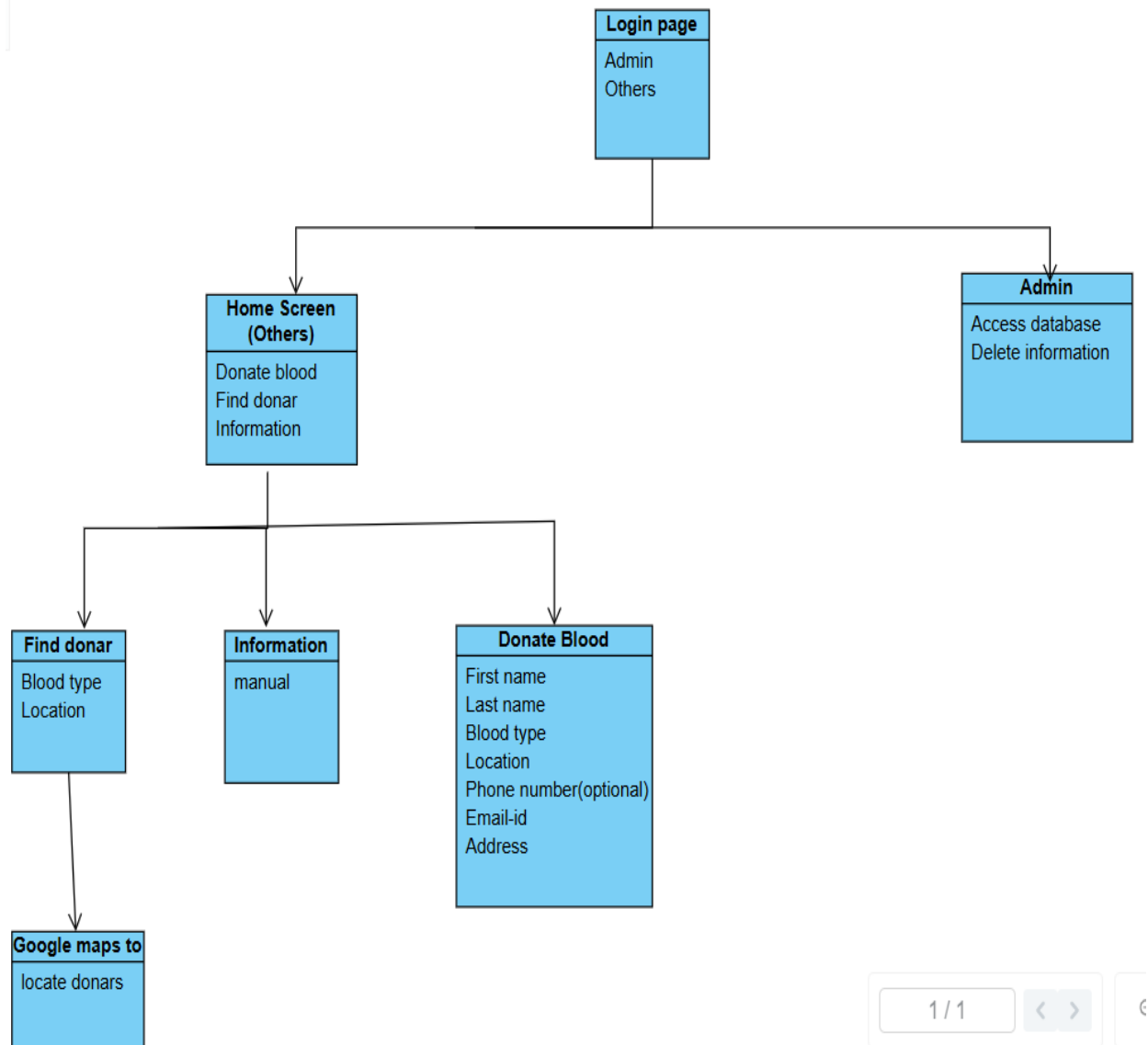


Figure-1: Class Diagram

Use Case Diagram

Describes user interactions:

- Actors: Donor, Recipient, Admin
- Use cases: Register, Login, Request Blood, Update Inventory

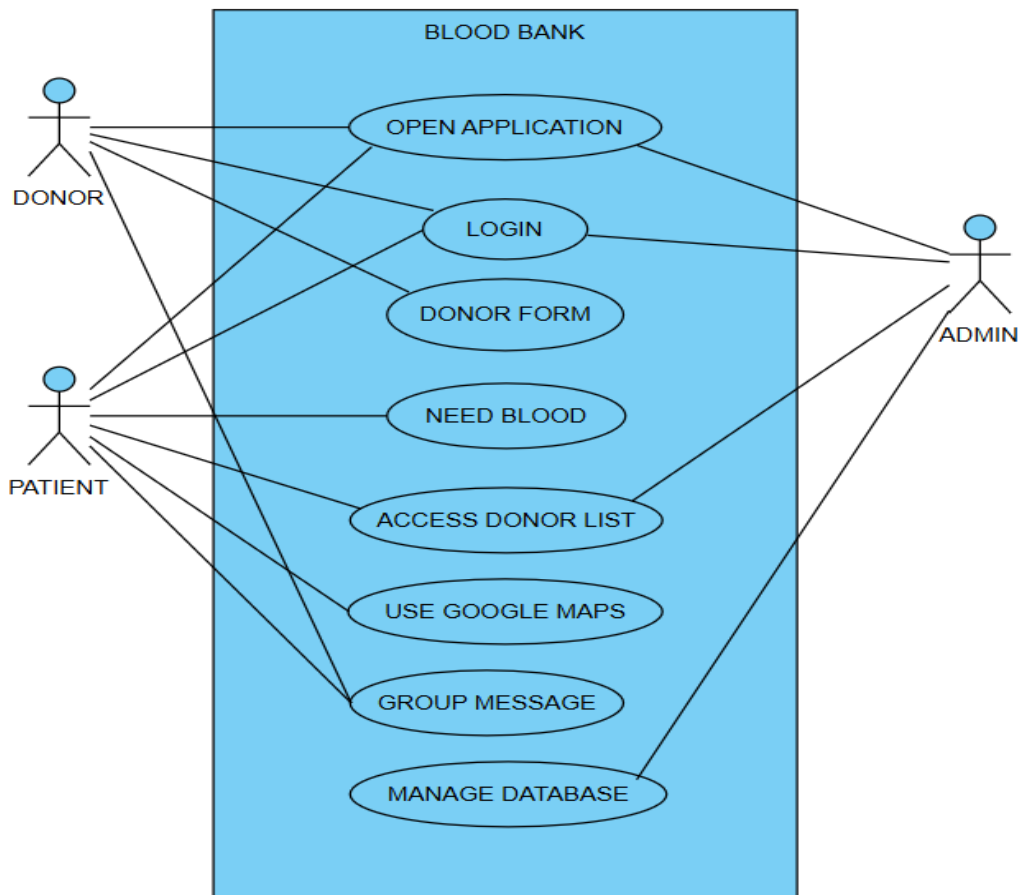


Figure-2: Use Case Diagram

Sequence Diagram

Shows message flow in operations like:

- User registration
- Search for blood
- Request approval by Admin

4.5 Component Design

Each module in the application is broken down into components:

- **Frontend Components (React):**
 - LoginForm, Dashboard, SearchBar, RequestForm, DonorProfile, AdminPanel
- **Backend Modules (Node/Express):**
 - AuthController, UserController, InventoryController, RequestController
- **Database Collections (MongoDB):**
 - Users, BloodInventory, Requests, Donations

4.6 User Interface Design

User interfaces are designed with usability and responsiveness in mind.

- Clean layout using React and Tailwind CSS
- Navigation bar for role-based access
- Dashboard cards and charts for admin insights
- Forms with validation for secure input

4.7 Database Requirements

The application requires a database to store the following types of information:

1. **User Data** – Personal information, login credentials, donation history, and roles (donor/recipient/admin).
2. **Blood Inventory** – Records of available blood types, quantities, and storage details.
3. **Blood Request Data** – Requests made by recipients, including the requested blood type and the status of the request.
4. **Donation Records** – Information on blood donations made by users, including date, donor ID, and blood type donated.

4.8 MongoDB Collections

Users Collection

Stores information about users (donors, recipients, and admins).

Schema:

json

CopyEdit

```
{  
  
  "_id": ObjectId("..."),  
  
  "name": "John Doe",  
  
  "email": "johndoe@example.com",  
  
  "password": "hashed_password",  
  
  "role": "donor", // Can be 'donor', 'recipient', 'admin'  
  
  "contact": "+123456789",
```

```

"donation_history": [

  {

    "blood_type": "A+",

    "date": ISODate("2025-01-01"),

    "status": "Completed"

  }

]

}

```

Fields:

- `_id`: Unique identifier for each user.
- `name`: Full name of the user.
- `email`: Email address used for login.
- `password`: Hashed password for secure authentication.
- `role`: Defines the user's role (either donor, recipient, or admin).
- `contact`: Contact number for notifications.
- `donation_history`: Array to store past donations made by the user.

Blood Inventory Collection

Stores information about available blood types and quantities.

Schema:

json

CopyEdit

```
{
  "_id": ObjectId("..."),
  "blood_type": "A+",
  "quantity": 10,
  "storage_location": "Blood Bank XYZ",
  "expiration_date": ISODate("2025-12-31"),
  "last_updated": ISODate("2025-05-01")
}
```

Fields:

- `_id`: Unique identifier for each inventory record.
- `blood_type`: Type of blood (e.g., A+, O-, AB+).
- `quantity`: Quantity of blood available in the inventory.
- `storage_location`: The name or location of the blood bank storing the blood.
- `expiration_date`: The date when the blood expires.
- `last_updated`: Timestamp of the last update made to the inventory.

Blood Requests Collection

Stores details of blood requests raised by recipients.

Schema:

json

CopyEdit

```
{
```

```

    "_id": ObjectId("..."),

    "recipient_id": ObjectId("..."),

    "blood_type": "O-",

    "quantity_needed": 2,

    "status": "Pending", // Possible values: Pending, Approved, Fulfilled

    "request_date": ISODate("2025-05-01"),

    "approval_date": ISODate(""),

    "fulfilled_by": ObjectId("...") // Admin approval or donor info
}

```

Fields:

- `_id`: Unique identifier for each request.
- `recipient_id`: Reference to the recipient user ID (who made the request).
- `blood_type`: The type of blood requested.
- `quantity_needed`: Quantity of blood requested.
- `status`: The status of the request (Pending, Approved, Fulfilled).
- `request_date`: Date the request was made.
- `approval_date`: Date when the request was approved by the admin.
- `fulfilled_by`: Reference to the donor or admin who approved/fulfilled the request.

Donations Collection

Stores information about the blood donations made by users.

Schema:

json

CopyEdit

```
{  
  
  "_id": ObjectId("..."),  
  
  "donor_id": ObjectId("..."),  
  
  "blood_type": "B+",  
  
  "quantity_donated": 1,  
  
  "donation_date": ISODate("2025-04-15"),  
  
  "status": "Successful", // Values: Successful, Failed  
  
  "donation_location": "Blood Bank ABC"  
  
}
```

Fields:

- `_id`: Unique identifier for each donation.
- `donor_id`: Reference to the donor's user ID.
- `blood_type`: Blood type donated.
- `quantity_donated`: Amount of blood donated (in liters or units).
- `donation_date`: Date the donation was made.
- `status`: Status of the donation (Successful or Failed).
- `donation_location`: Blood bank or hospital where the donation occurred.

4.9 Database Relationships

MongoDB is a NoSQL database, meaning it doesn't use traditional relational tables. However, we do establish relationships between collections using references.

- **User to Donations:** A one-to-many relationship between users and donations. Each user can have multiple donations, but each donation is associated with one user.
- **User to Blood Requests:** A one-to-many relationship. A user can make multiple blood requests, but each request is linked to one recipient.
- **Blood Inventory to Donations:** A many-to-many relationship. A single blood type can be donated by multiple donors, and a blood type can be used in many requests. This relationship is tracked through the inventory and request collections.

4.10 Sample Queries

Find Available Blood Types

js

CopyEdit

```
db.blood_inventory.find({ "quantity": { $gt: 0 } })
```

Get Donor Profile

js

CopyEdit

```
db.users.find({ "_id": ObjectId("donor_id") })
```

Create New Blood Request

js

CopyEdit

```
db.blood_requests.insert({
```

```
"recipient_id": ObjectId("recipient_id"),  
"blood_type": "A+", "quantity_needed": 2, "status": "Pending",  
"request_date": ISODate("2025-05-10")  
});
```

Chapter 5

Technology Stack and Implementation

The Blood Bank Web Application is built using modern technologies to ensure scalability, responsiveness, and ease of maintenance. The application follows the MERN stack architecture, combining the power of MongoDB, Express.js, React.js, and Node.js.

5.1 Frontend - React.js

React.js is a JavaScript library for building user interfaces, especially single-page applications (SPAs), where the user experience is smooth and dynamic without reloading the page.

Why React.js?

- **Component-Based Architecture:** React allows the application to be broken down into small, reusable components that can manage their state independently.
- **Virtual DOM:** React uses a virtual DOM, which optimizes rendering and improves performance by minimizing the number of changes to the real DOM.
- **Rich Ecosystem:** React has a large ecosystem of libraries and tools that help with state management (e.g., Redux) and routing (e.g., React Router).

Features in the Blood Bank Application:

- **User Interface:** A responsive UI for donors, recipients, and admins.
- **Form Handling:** Dynamic forms for user registration, login, and blood requests.
- **Dashboard:** An admin dashboard to manage blood inventory and requests.
- **Search Functionality:** Blood search based on type and location, utilizing dynamic state management.

5.2 Backend - Node.js and Express.js

Node.js is a server-side JavaScript runtime that allows for the creation of scalable and efficient applications. **Express.js** is a lightweight framework built on top of Node.js to simplify routing and middleware management.

Why Node.js and Express.js?

- **Non-blocking I/O:** Node.js is ideal for building high-performance applications that need to handle many simultaneous requests, such as our web app, where real-time blood request tracking is crucial.
- **JavaScript Across the Stack:** Both the frontend (React) and backend (Node.js) use JavaScript, making it easier to develop and maintain the application.
- **Express Middleware:** Express simplifies the creation of APIs with built-in middleware, allowing for easy handling of requests, security features (e.g., authentication), and error handling.

Features in the Blood Bank Application:

- **API for User Management:** Handles authentication, user registration, and profile updates.
- **Blood Request API:** Manages the blood request process, including creation, updates, and status changes.
- **Admin Control:** Provides endpoints for the admin to manage blood inventory, donors, and requests.
- **Real-Time Communication:** Supports asynchronous handling of requests to ensure quick responses and updates.

5.3 Database - MongoDB

MongoDB is a NoSQL database that stores data in flexible, JSON-like documents, allowing for easy storage and retrieval of structured and unstructured data.

Why MongoDB?

- **Scalability:** MongoDB can scale horizontally, making it ideal for handling large volumes of user and donation data, especially as the application grows.
- **Flexibility:** The schema-less nature of MongoDB allows for rapid changes to the data model, which is essential for evolving applications like a blood bank system.
- **Rich Querying:** MongoDB provides powerful querying capabilities, including geospatial searches, which help users find donors by location.

Features in the Blood Bank Application:

- **User Data:** Stores information related to donors, recipients, and admin profiles.
- **Blood Inventory:** Manages blood types, quantities, expiration dates, and locations.
- **Requests & Donations:** Tracks blood requests, statuses, and donations made by users.

5.4 Additional Tools and Libraries

Authentication - JWT (JSON Web Tokens)

JWT is used for securing the API and managing user sessions. It is a compact, URL-safe means of representing claims to be transferred between two parties.

Why JWT?

- **Stateless Authentication:** JWT allows the application to be stateless, where the server does not need to store session information. Instead, the token is stored client-side (e.g., in localStorage or cookies).
- **Secure:** JWTs can be signed with a secret or public/private key pair, ensuring that the token is valid and has not been tampered with.

Frontend Styling - Tailwind CSS

Tailwind CSS is a utility-first CSS framework that makes it easier to design custom user interfaces without writing custom styles.

Why Tailwind CSS?

- **Flexibility:** Tailwind provides utility classes that can be combined to create custom designs.
- **Responsiveness:** Tailwind is built with responsiveness in mind, making it easy to design mobile-friendly UIs.
- **Productivity:** With pre-defined classes for spacing, typography, colors, etc., Tailwind speeds up the development process.

Hosting - Heroku/Netlify

For deployment, the **application** is hosted on cloud platforms like **Heroku** or **Netlify**, which provide easy integration with GitHub for continuous deployment.

Why Heroku/Netlify?

- **Ease of Use:** Both platforms simplify the deployment process, allowing for rapid scaling.
- **Free Tiers:** Both Heroku and Netlify provide generous free-tier hosting, suitable for development and small-scale production.
- **CI/CD Integration:** Seamless integration with GitHub repositories allows automatic deployment upon code commits.

5.5 Development Tools

- **Git & GitHub:** For version control and collaborative development. GitHub also hosts the source code and acts as a repository for the application.
- **Postman:** Used for testing the APIs and ensuring the backend endpoints are functioning as expected.
- **Visual Studio Code (VS Code):** The primary Integrated Development Environment (IDE) used for writing and debugging the code.

5.6 Frontend Implementation (React.js)

The frontend of the Blood Bank Web Application is built using React.js to provide a dynamic, interactive user interface. The application is designed to be responsive, ensuring that it works seamlessly across desktop and mobile devices.

Key Features:

1. User Registration and Login

- Users (donors, recipients) can register with their details, including name, contact information, blood type, etc.
- Secure authentication is implemented using JWT (JSON Web Tokens).
- Login functionality allows users to access their profiles and perform actions based on their roles (donor, recipient, admin).

2. Profile Management

- Donors and recipients can update their personal information.
- Donation history is displayed for donors.
- Admins can view and manage profiles via the admin panel.

3. Blood Search & Request

- A blood search form allows recipients to search for available blood by type and location.
- Recipients can submit blood requests, which can be approved or rejected by the admin.

4. Admin Dashboard

- The dashboard provides an overview of blood availability, donor statistics, and pending requests.

- Admins can update the blood inventory and approve or fulfill blood requests.

5. Responsive Design

- The layout is designed using Tailwind CSS for a clean and modern look, with a focus on responsiveness and usability.

Implementation Steps:

- Set up the React project using create-react-app.
- Implement routing with React Router to handle page navigation (login, dashboard, profile, etc.).
- Create reusable components for forms, buttons, and cards.
- Handle form submissions using React state and form validation.
- Integrate Redux for managing the state of the application (e.g., user authentication status, blood inventory).
- Use Axios to make API calls to the backend for data fetching and submission.

5.7 Backend Implementation (Node.js & Express.js)

The backend is developed using **Node.js** and **Express.js**, which allows us to handle HTTP requests, manage user authentication, and interact with the MongoDB database.

Key Features:

1. User Authentication & Authorization

- **JWT** authentication is used to secure the API endpoints. Upon login, a token is generated and sent to the client, which is then used in subsequent requests to authenticate the user.

- Role-based access control (RBAC) is implemented to ensure that only authorized users (admins) can perform certain actions (e.g., approving blood requests).

2. **CRUD Operations for Blood Inventory**

- Admins can create, read, update, and delete records from the **blood_inventory** collection.
- Blood donations are tracked by associating them with donors.

3. **Blood Request Management**

- Recipients can create blood requests, which are stored in the **blood_requests** collection.
- Admins approve or fulfill these requests, updating their status accordingly.

4. **Error Handling & Validation**

- Implemented proper error handling using **try-catch** blocks and custom error messages for invalid operations (e.g., unauthorized access).
- Input validation ensures that all fields (e.g., email, password) are correctly formatted before data is processed.

Implementation Steps:

- Set up a Node.js server using **Express.js**.
- Create routes for user registration, login, and profile management.
- Develop CRUD APIs for blood inventory and request management.
- Integrate **JWT** for secure authentication and authorization.
- Use **Mongoose** to interact with the **MongoDB** database for storing user and blood information.
- Implement middleware for validation, error handling, and logging.

5.8 Database Implementation (MongoDB)

MongoDB is used as the database for storing all the application data, including user details, blood inventory, donation records, and requests. The data is stored in collections, which are flexible and scalable for handling dynamic data structures.

Key Features:

1. Users Collection:

Stores details of users, including their role (donor, recipient, or admin), personal information, and donation history.

2. Blood Inventory Collection:

Tracks available blood units by type, quantity, location, and expiration date. Admins update this collection after blood donations or withdrawals.

3. Blood Requests Collection:

Stores blood requests made by recipients, including the requested blood type, quantity, and status of the request (pending, approved, fulfilled).

4. Donation Collection:

Records blood donations, including donor information, blood type, donation date, and the status of the donation (successful or failed).

5.9 Integration of Frontend and Backend

The frontend and backend of the Blood Bank Web Application are integrated using **RESTful APIs**. The React frontend makes **HTTP requests** to the Express backend, which processes the data and returns responses.

Key Integration Points:

1. User Authentication

- When a user logs in, the frontend sends the user credentials to the backend.

- The backend authenticates the credentials and sends a **JWT** token to the frontend, which is stored in **localStorage** or **cookies** for subsequent requests.

2. Blood Search & Request

- The frontend makes API calls to search for available blood types and display them to the recipient.
- Recipients can submit blood requests via the frontend, which are stored in the backend.

3. Admin Operations

- Admins can interact with the frontend to manage blood inventory and approve or reject requests, with updates reflected in the database.

Implementation Steps:

- Use **Axios** on the frontend to make requests to backend endpoints.
- Implement asynchronous operations using **async/await** for API calls.
- Handle form submissions and data updates via the API, ensuring smooth user interaction.

5.10 Testing and Debugging

Testing is a crucial part of the implementation phase to ensure the application functions as expected. The application undergoes both **unit testing** and **integration testing** to verify the correctness of the individual components and their interactions.

1. Frontend Testing:

- **Jest** is used for unit testing React components.
- **React Testing Library** is used to test component interactions and behavior in the DOM.

2. **Backend Testing:**

- **Mocha** and **Chai** are used for unit and integration testing of Express.js APIs.
- Mock APIs and database interactions are tested to ensure correct functionality.

3. **Bug Fixes and Optimizations:**

- The application is continuously debugged to fix issues related to UI rendering, API interactions, and security flaws.

Chapter 6

Testing and Security Features

Testing is an essential part of the development process that ensures the functionality, reliability, and security of the Blood Bank Web Application. The application undergoes various types of testing, including unit testing, integration testing, and system testing, to verify that all components work as expected. Additionally, security testing is performed to identify potential vulnerabilities in the application.

6.1 Types of Testing

Unit Testing:

Unit testing involves testing individual components of the application to ensure that each part functions correctly in isolation. This type of testing helps identify bugs early in the development process and makes it easier to isolate problems.

- **Frontend Testing:**

The frontend of the application is built using React.js, and unit tests are written using the Jest framework.

- React Testing Library is used to test components and their behavior, such as form submissions, state changes, and rendering.
- Unit tests verify that the components are rendering correctly based on the input data and ensure that the components interact properly with the React state.

Examples of Unit Tests:

- Testing user input in forms (e.g., blood request form).
- Verifying that components correctly update when data is passed as props.
- Checking whether the right API calls are triggered on form submission.

- **Backend Testing:**

For the backend, Mocha and Chai are used for testing Express.js routes and controllers.

- Unit tests focus on individual API endpoints, ensuring they return the correct response for valid inputs and handle errors properly for invalid inputs.
- Sinon.js is used to mock dependencies and external services, such as the database, to isolate and test individual functionalities.

Examples of Unit Tests:

- Testing the POST /login API to verify correct user authentication.
- Validating the GET /blood-inventory API to ensure that the correct blood inventory data is returned.

Integration Testing:

Integration testing ensures that different parts of the application work together as expected. This type of testing focuses on the interaction between multiple components, such as the communication between the frontend and backend, as well as between the backend and the database.

- **Frontend and Backend Integration:**

Integration tests are written to check if the frontend successfully communicates with the backend API. These tests simulate real-world interactions, such as user registration, login, blood request creation, and donation submission.

Examples of Integration Tests:

- Testing the login process by sending a request from the React frontend to the Express backend and verifying the correct token is returned.
- Verifying that a blood request submitted from the frontend creates a new request in the backend database.

- Checking if updates to the blood inventory are reflected on the frontend.
- **Database Integration:**

The integration between the Express.js backend and MongoDB is tested to ensure data is being correctly stored, updated, and retrieved from the database. This ensures that the CRUD operations on collections (e.g., users, blood inventory, requests) are functioning as expected.

Examples of Database Integration Tests:

- Testing the creation and retrieval of user profiles.
- Verifying that blood donation records are correctly added to the database and associated with the donor.
- Ensuring that blood requests are saved and that the correct request status is updated.

System Testing

System testing verifies that the entire application works as intended when all components are integrated. This includes testing the full flow of the application from frontend to backend to database and vice versa.

- **End to End Testing:**

End-to-end testing simulates real user interactions and tests the entire system as a whole. Tools like Cypress or Selenium can be used for automated end-to-end tests, ensuring that the user experience works seamlessly from the frontend to the backend.

Examples of System Tests:

- Registering a new user, logging in, making a blood donation, and then creating a blood request to ensure all flows are working correctly.
- Admins logging in and managing the blood inventory, ensuring that the updates reflect correctly in the system.

- Verifying that notifications or updates are correctly triggered when a blood request is fulfilled.

6.2 Security Testing

Security testing focuses on identifying vulnerabilities and potential threats in the application. Given the sensitive nature of the data (personal information, blood donation records), it is crucial to ensure that the application is secure.

Authentication & Authorization

- **JWT Authentication:**

The **JWT** authentication mechanism is thoroughly tested to ensure that tokens are securely issued, verified, and expire after the appropriate time. This prevents unauthorized access to user data.

- **Token Expiry Testing:** Ensure that expired tokens cannot be used to access protected resources.
- **Authorization Testing:** Test that users with different roles (donor, recipient, admin) can only access the resources permitted by their role. For example, only admins should have access to blood inventory management.

Data Encryption

- **Password Storage:**

Passwords are hashed using algorithms like **bcrypt** before storing them in the database. Security testing ensures that passwords cannot be retrieved in their original form from the database.

- Test the strength of password hashing and ensure that brute force attacks cannot easily recover the original passwords.

Input Validation

- **SQL/NoSQL Injection:**

Even though MongoDB is a NoSQL database, security testing is conducted to ensure that the application is resistant to NoSQL injection attacks.

- Input fields, such as user registration or donation forms, are tested for malicious input to ensure proper validation and sanitization.

- **Cross-Site Scripting (XSS):**

All user inputs are validated to prevent **XSS** attacks. This ensures that malicious scripts are not executed in the browser, compromising user data.

Cross-Site Request Forgery (CSRF)

- **CSRF Protection:**

Ensure that the application includes anti-CSRF tokens in every form submission, protecting against CSRF attacks, where an attacker can trick a user into making unintended requests on their behalf.

6.3 Performance Testing

Performance testing ensures that the application can handle a large number of users and data without performance degradation.

Load Testing

- **Simulating High Traffic:**

Tools like **Apache JMeter** or **Artillery** are used to simulate multiple users interacting with the application simultaneously. The goal is to ensure that the application can handle a high load, especially during peak times (e.g., when many users are making blood requests).

Stress Testing

- **Breaking The System:**

Stress testing helps identify the system's breaking point by increasing the load beyond its capacity and observing how it handles failure scenarios (e.g., database errors, timeouts).

6.4 Bug Fixing and Optimization

During testing, bugs are identified and fixed to improve the application's stability, performance, and security. The development team addresses these issues through debugging, code refactoring, and optimization.

Common Bug Fixes and Optimizations:

- **UI bugs:** Fixed issues related to form validation, rendering errors, and user interface responsiveness.
- **API issues:** Fixed bugs in the backend API that caused incorrect responses or failed database interactions.
- **Performance improvements:** Optimized database queries and API endpoints to handle larger datasets efficiently.

Authentication and Authorization

User Authentication

User authentication ensures that only authorized individuals can access certain features of the application. The Blood Bank Web Application uses **JWT (JSON Web Token)** for secure authentication.

- **Login Process:**

Users log in by providing their credentials (username/email and password). Upon successful verification, a **JWT** is issued by the backend, which the frontend stores in **localStorage** or **cookies**. This token is included in the headers of subsequent requests to authenticate the user.

- **Token Expiry and Renewal:**

JWT tokens have an expiration time, after which the user must log in again. This reduces the risk of token misuse.

- **Password Hashing:**

Passwords are not stored in plaintext. Instead, they are securely hashed using **bcrypt** before being stored in the database. This ensures that even if the database is compromised, the passwords remain safe.

Role-Based Access Control (RBAC)

The application uses **Role-Based Access Control (RBAC)** to define what actions different users can perform. There are three primary roles in the system: **Admin**, **Donor**, and **Recipient**.

- **Admin Role:**

Admins have full access to the system, including managing user accounts, updating blood inventory, and approving blood requests.

- **Donar Role:**

Donors can view and update their profile and donation history. They can also submit blood donations.

- **Recipient Role:**

Recipients can create blood requests, view available blood types, and monitor the status of their requests.

Each role has a set of permissions to ensure that users can only access resources that are relevant to their role.

Data Protection and Encryption

Data Encryption in Transit

To protect sensitive data during communication between the client (frontend) and server (backend), the application uses **SSL/TLS** encryption to establish a secure **HTTPS** connection. This prevents data from being intercepted or tampered with while it is being transmitted over the internet.

- **HTTPS Protocol:**

All data exchanged between the client and the server is encrypted using **HTTPS** to prevent eavesdropping and man-in-the-middle attacks.

Data Encryption at Rest

Sensitive data such as user credentials and blood donation information are encrypted at rest to ensure that data remains secure even if the database is compromised.

- **Password Hashing:**

As mentioned earlier, **bcrypt** is used to hash passwords before they are stored in the database. This ensures that even if the database is exposed, the passwords cannot be easily recovered.

- **Sensitive Data:**

Any other sensitive information stored in the database, such as personal medical records, is encrypted using strong encryption algorithms (e.g., **AES** - Advanced Encryption Standard) to protect it from unauthorized access.

Input Validation and Sanitization

Prevention of SQL/NoSQL Injection

The Blood Bank Web Application uses **MongoDB**, a NoSQL database. However, **input validation** and **sanitization** are essential to prevent injection attacks, where attackers may manipulate input data to execute malicious queries.

- **Input Validation:**

All user input, including form data (e.g., registration, blood request), is validated to ensure it conforms to the expected format. For example, emails are checked for a valid email format, and numeric fields are validated to contain only numbers.

- **Input Sanitization:**

Inputs are sanitized to remove any malicious code that could potentially be used for injection attacks. Special characters such as <, >, and & are escaped to prevent **cross-site scripting (XSS)** attacks.

Cross-Site Scripting (XSS) Prevention

The application implements **output encoding** and **input validation** to prevent cross-site scripting (XSS) attacks, where attackers inject malicious scripts into web pages that can be executed in the user's browser.

- **Safe Rendering:**

- User-generated content, such as profile information or blood requests, is rendered in a secure manner by encoding any HTML tags or JavaScript before displaying them.

- **Libraries for XSS Prevention:**

The frontend uses libraries like **React.js**, which automatically escapes output to prevent XSS vulnerabilities. This reduces the risk of attackers injecting harmful scripts.

Secure API Endpoints

JWT Authentication for API Security

All sensitive API endpoints (such as login, blood request submission, and donation records) are protected with **JWT authentication**. The token must be passed in the request headers, and the backend validates it before processing the request.

- **Access Tokens:**

The access token contains user-specific information (e.g., user ID, role) and is used to verify that the user making the request has the proper permissions.

- **Authorization Middleware:**

Express middleware checks for the presence of a valid JWT token in incoming API requests. If the token is missing or invalid, the request is rejected with a **401 Unauthorized** status.

Rate Limiting

To protect against **denial-of-service (DoS)** attacks, the application uses **rate limiting** to restrict the number of requests that a user or IP address can make in a given time frame. This helps prevent abuse of the system and protects server resources.

- **Rate Limiting:**

The backend uses middleware to limit requests per minute (e.g., 100 requests per minute) to sensitive endpoints, such as login and blood request submission.

Cross-Site Request Forgery (CSRF) Protection

CSRF attacks trick users into performing unwanted actions on a website without their consent. The application prevents CSRF attacks by implementing the following measures:

- **CSRF Tokens:**

Every form submission (e.g., blood donation form, blood request form) includes a unique CSRF token that is validated by the server before processing the request.

- **SameSite Cookies:**

The application uses **SameSite cookies** to restrict third-party websites from sending unauthorized requests on behalf of authenticated users.

Session Management

Secure Session Handling

The application uses secure session management practices to protect user sessions.

- **Session Expiry:**

User sessions automatically expire after a certain period of inactivity, requiring users to log in again to continue using the application.

- **Session Hijacking Prevention:**

- The application uses **secure cookies** and **HTTPOnly** flags to prevent session hijacking. These cookies are not accessible via JavaScript and are only transmitted over HTTPS.

Monitoring and Logging

Activity Logging

All critical actions performed by users (e.g., login attempts, blood request submissions, donation updates) are logged for audit purposes. This helps in tracking potential security incidents and allows administrators to monitor suspicious activity.

- **Log Storage:**

Logs are stored securely, with access restricted to authorized personnel only.

- **Alerting:**

The system sends alerts to administrators in case of suspicious activities, such as multiple failed login attempts or unusual API request patterns.

Security Best Practices

- **Security Headers:**

The application uses HTTP security headers (e.g., **Content Security Policy (CSP)**, **Strict-Transport-Security (HSTS)**, **X-Content-Type-Options**) to prevent attacks such as clickjacking, XSS, and MIME-sniffing.

- **Regular Security Audits:**

Periodic security audits and vulnerability assessments are conducted to identify and mitigate potential risks in the application.

- **User Education:**

Users are educated on the importance of using strong, unique passwords and are encouraged to enable multi-factor authentication (MFA) if available.

Chapter 7

Results and Conclusion

Outcome:

The outcome of this project was the successful development of a Blood Bank Management System that enables users to efficiently manage blood donations, inventory, and requests. With a user-friendly interface and role-based access, the system provides a one-stop platform for hospitals, donors, and administrators to collaborate and ensure timely blood availability. It streamlines operations, enhances communication, and ensures transparency and accuracy in blood stock management—making life-saving support accessible and efficient.

Result:

Throughout the course of this project, we've gained valuable insights into developing a reliable and responsive healthcare-oriented web platform. The Blood Bank Management System not only fulfilled the essential requirements of managing blood donations and requests but also introduced features that improved the overall experience of both users and administrators.

The system's role-based access helped create a secure environment for distinct users such as donors, hospitals, and administrators, while the real-time tracking of blood inventory minimized shortages and wastage. The integration of notifications and alerts ensured timely donor reminders and urgent request broadcasts, significantly improving response times in critical situations.

Our commitment to user-centric design is evident in the simplicity and clarity of the interface, which enabled easy navigation and quick access to important functionalities. This contributed to a more engaging experience for donors and staff, resulting in a wider adoption rate among users. The inclusion of donor history tracking and eligibility checks enhanced donor management while ensuring compliance with medical donation guidelines.

Moreover, the application helped bridge the gap between hospitals and potential donors by offering location-based search capabilities, making the entire process of requesting and donating blood more responsive and community-driven.

Conclusion:

The Blood Bank Web Application was conceptualized and developed with the objective of addressing the challenges associated with managing blood donation and transfusion services. In this project, we successfully designed, implemented, and tested a full-stack web application that connects donors, recipients, and administrators through a centralized and secure platform.

Summary of Achievements:

Throughout the development cycle, several key milestones were achieved:

- **Comprehensive User Management:** The system enables secure user registration and login, supporting multiple roles such as donors, recipients, and administrators.
- **Effective Blood Inventory Management:** Real-time updates ensure that blood stock levels are accurately tracked and reflected in the system.
- **Streamlined Donation and Request Processes:** Donors can easily register donations, and recipients can submit and track blood requests.
- **Robust Backend Architecture:** Built using **Node.js** and **Express.js**, the backend ensures fast and scalable API handling.
- **Intuitive Frontend Interface:** Developed using **React.js**, the frontend offers a user-friendly interface and responsive design.
- **Secure Data Handling:** Security measures including **JWT authentication**, **password hashing**, and **HTTPS encryption** were implemented to protect user data.
- **Database Optimization:** Using **MongoDB**, the system stores and retrieves data efficiently, supporting dynamic queries and schema flexibility.

Challenges Faced:

Like any software project, the development of this application came with its own set of challenges:

- **Designing a Scalable Architecture:** Ensuring that the application remains performant as the number of users grows required careful architectural planning.
- **Security and Privacy Concerns:** Special attention was given to protecting user data and ensuring that only authorized users could access sensitive features.
- **Testing Edge Cases:** Managing diverse user interactions (e.g., overlapping requests, expired blood units) demanded extensive testing and validation.

Social Impact:

This application has the potential to create a positive societal impact by:

- **Reducing Delays** in finding and accessing blood for patients in need.
- **Encouraging Voluntary Donation** by simplifying the donation process and improving donor engagement.
- **Improving Transparency** in how blood inventory is managed and allocated.
- **Supporting Health Infrastructure** in both urban and rural settings where blood management systems are often underdeveloped

Learning Outcomes:

Working on this project allowed team members to gain valuable experience in:

- **Full-stack Web Development** using modern technologies like React, Node.js, Express, and MongoDB.
- **Project Planning and Team Collaboration**, including effective task delegation and deadline management.
- **Agile Methodology** for iterative design, testing, and deployment.

- **Problem Solving and Debugging**, particularly in real-time data handling and user session management.

Final Thoughts:

The Blood Bank Web Application is a step toward leveraging technology to solve critical health-related issues. While the current version serves as a strong foundation, there is ample scope for enhancement in terms of features, scalability, and integration. With further development, the application can become a powerful tool in saving lives by ensuring timely access to safe and available blood.

The successful completion of this project not only demonstrates the technical competence of the development team but also reflects a commitment to applying software engineering for meaningful and socially impactful causes.

Appendix A

The appendix provides supplementary materials and references that support the main content of this report. It includes screenshots of the application interface, sample code snippets, user interface flow diagrams, and other artifacts created during the project development.

Application Screenshots

1. User Registration Page

A form where users can sign up as a **donor**, **recipient**, or **admin**. It includes validations for input fields and secure password handling.

Register Page

☒ Donar ☐ Admin ☐ Hospital ☐ Organisation

Name

Email

Password

Website

Address

Phone

Already User Please [Login!](#)

Figure-3: Register Page

2. Login Page

Secure login interface with JWT authentication and “Forgot Password” option.

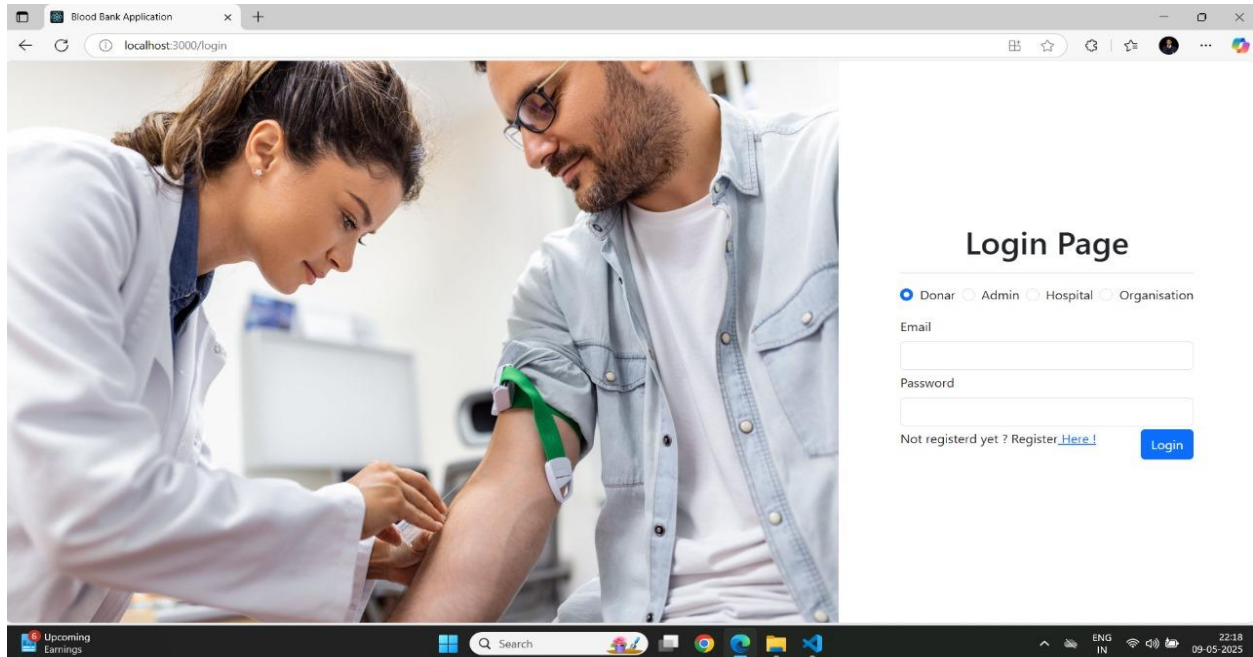


Figure-4: Login Page

3. Admin Dashboard

Displays donor profile, donation history, upcoming blood donation drives, and a form to register new donations.

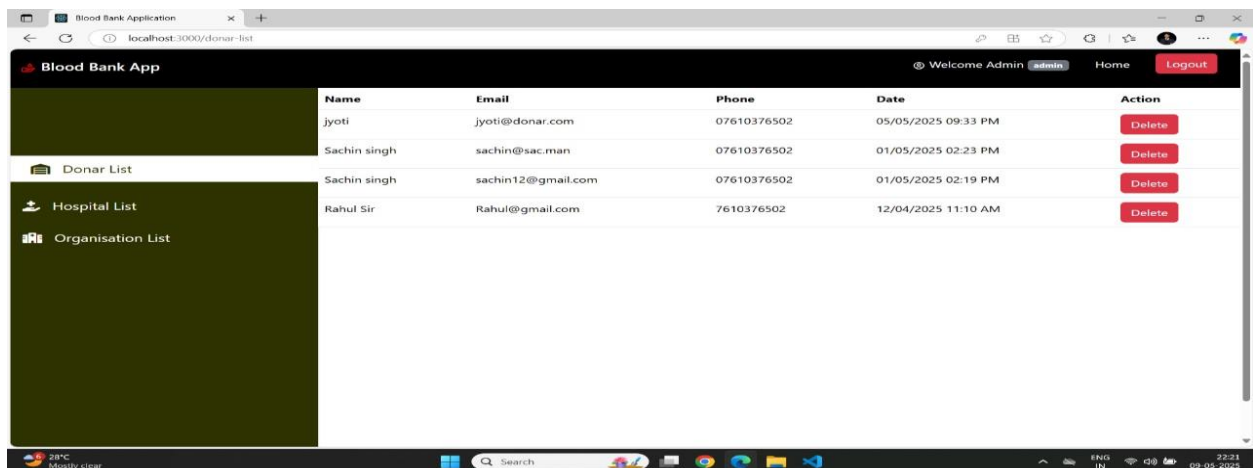


Figure-5: Admin Dashboard

4. Admin Home Dashboard

Comprehensive control panel for managing users, viewing analytics, approving/rejecting requests, and monitoring inventory levels.

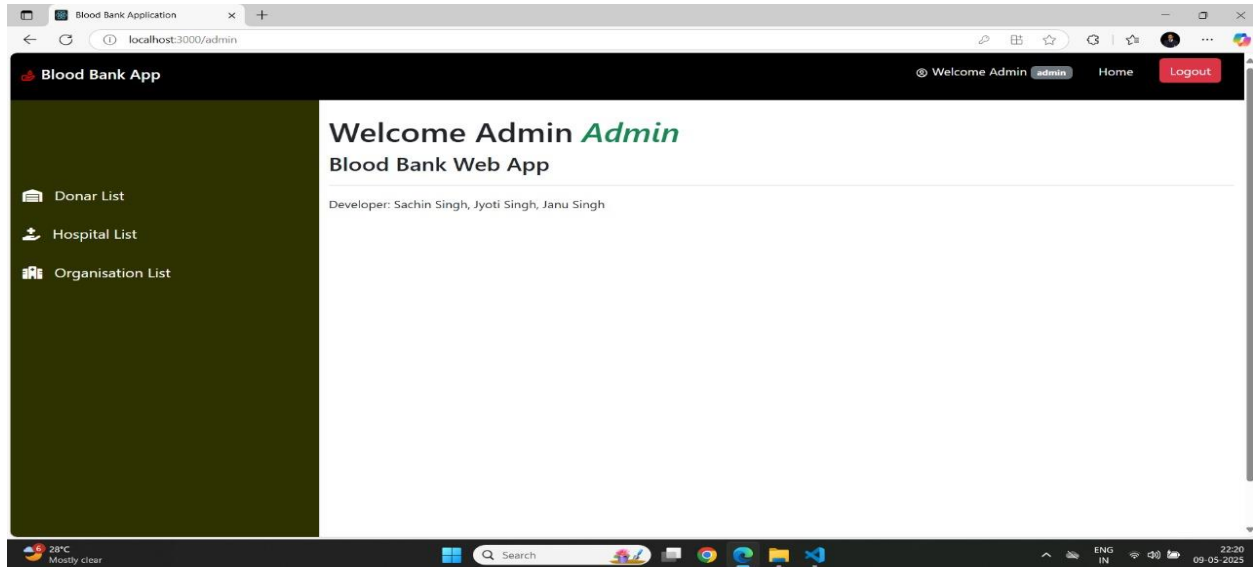


Figure-6: Admin Home Dashboard

5. Blood Inventory Management

Live dashboard for current blood stock by type (A+, A-, B+, etc.), quantity available, and expiration alerts.

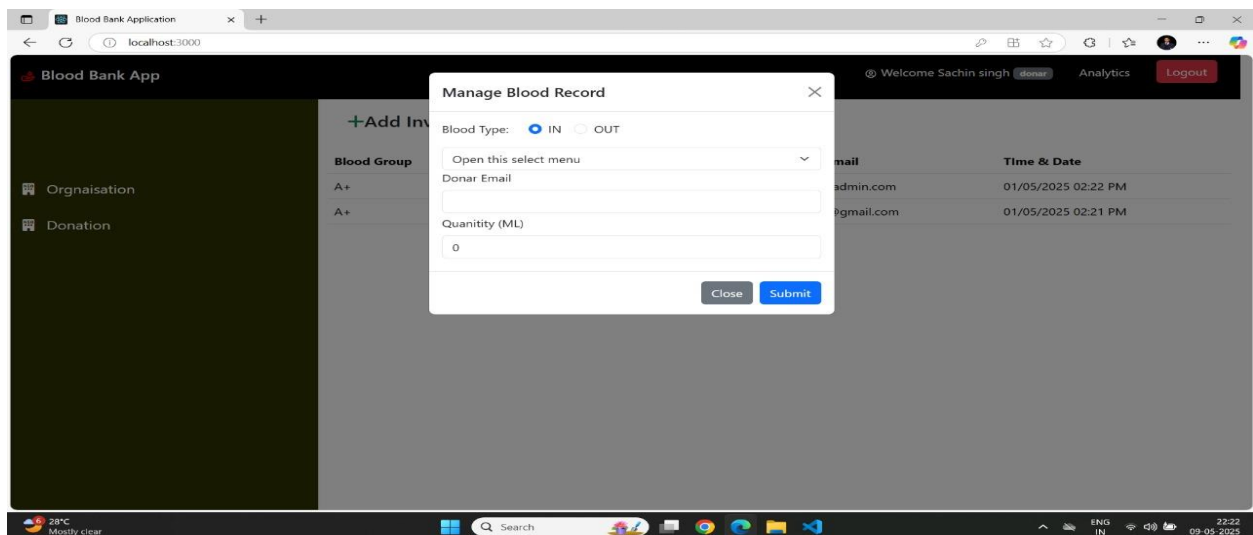


Figure-7: Inventory Form

Sample Code Snippets

React - Login Form Component

```
JS Login.js U X
client > src > pages > auth > JS Login.js > Login
1  import React from "react";
2  import Form from "../../component/shared/Form/Form";
3  import {useSelector} from 'react-redux'
4  import Spinner from "../../component/shared/Spinner";
5
6  const Login = () => {
7    const {loading,error}=useSelector(state=>state.auth)
8
9    return(
10     <>
11       {error && <span>{alert(error)}</span>}
12       {loading ? (<Spinner/>): (
13         <div className="row g-0">
14           <div className="col-md-8 form-banner">
15             
16           </div>
17           <div className="col-md-4 form-container">
18             <Form
19               formTitle={'Login Page'}
20               submitBtn={'Login'}
21               formType={"login"}
22             />
23           </div>
24         </div>
25       </div>
26     </>
27   );
28 }
29
30 );
31 };
32
33
34 export default Login;
35
```

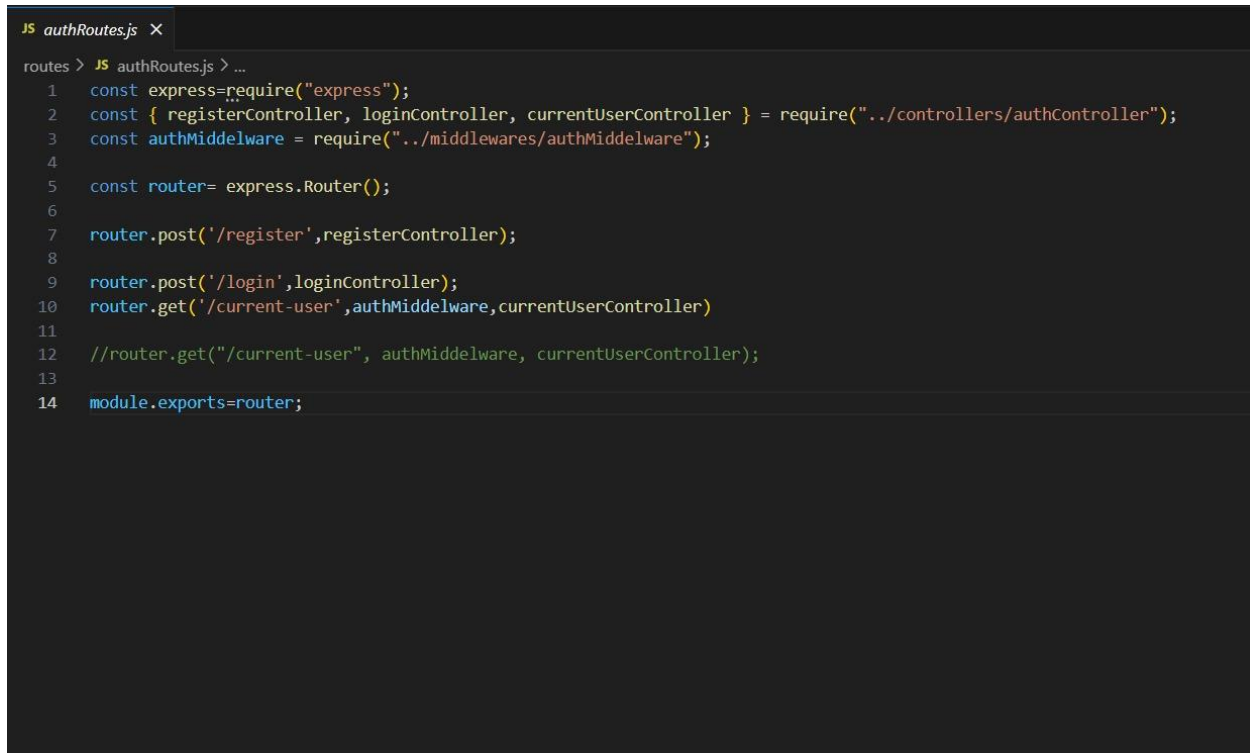
Figure-8: Login Form Component

React - Register Form Component

```
JS Register.js U X
client > src > pages > auth > JS Register.js > [e] Register
1  import React from "react";
2  import Form from "../../component/shared/Form/Form";
3  import { useSelector } from "react-redux";
4  import Spinner from "../../component/shared/Spinner";
5  const Register = () => {
6    const {loading,error}=useSelector(state=>state.auth)
7    return(
8      <>
9        {error && <span>{alert(error)}</span>}
10       {loading ?( <Spinner/>):(
11         <div className="row g-0">
12           <div className="col-md-8 form-banner">
13             
14           </div>
15           <div className="col-md-4 form-container">
16             <Form
17               formTitle={'Register Page'}
18               submitBtn={'Register'}
19               formType={"register"}
20             />
21           </div>
22         </div>
23       </div>
24     )}
25   </>
26 );
27 };
28
29 export default Register;
30
31
32
33
```

Figure-9: Register Form Component

Express.js - Blood Request Endpoint



```
JS authRoutes.js X
routes > JS authRoutes.js > ...
1  const express=require("express");
2  const { registerController, loginController, currentUserController } = require("../controllers/authController");
3  const authMiddelware = require("../middlewares/authMiddelware");
4
5  const router= express.Router();
6
7  router.post('/register',registerController);
8
9  router.post('/login',loginController);
10 router.get('/current-user',authMiddelware,currentUserController)
11
12 //router.get("/current-user", authMiddelware, currentUserController);
13
14 module.exports=router;
```

Figure-10: Blood Request Endpoint

References

The development of this project was guided by a combination of academic resources, online documentation, research articles, and official technology documentation. Below is the list of references used during the course of designing and implementing the Blood Bank Web Application.

Books & Academic Sources

1. **Ian Sommerville**, *Software Engineering*, 10th Edition, Pearson Education, 2015.
2. **Roger S. Pressman**, *Software Engineering: A Practitioner's Approach*, 8th Edition, McGraw-Hill Education, 2014.
3. **Andrew S. Tanenbaum**, *Modern Operating Systems*, 4th Edition, Pearson, 2014.
4. **Elmasri and Navathe**, *Fundamentals of Database Systems*, 7th Edition, Pearson, 2016.

Online Documentation

5. [React Official Documentation](#) – For building the frontend components and UI logic.
6. Node.js Official Docs – For backend JavaScript runtime environment.
7. [Express.js Documentation](#) – For creating the backend REST APIs.
8. [MongoDB Documentation](#) – For database schema, CRUD operations, and performance tuning.
9. [Mongoose.js Documentation](#) – For MongoDB object modeling in Node.js.
10. [JWT.io](#) – For understanding and implementing JSON Web Token-based authentication.

API & Tools References

11. Postman Learning Center – Used for testing and verifying REST API endpoints.
12. [GitHub Docs](#) – For version control, branching, and collaborative development.
13. [Tailwind CSS Documentation](#) – For responsive UI and styling.

Research Articles & Blogs

14. "The Importance of Blood Donation and its Role in Healthcare Systems" – *Journal of Public Health*, 2020.
15. Blog: "Secure Web Application Architecture with Node.js & MongoDB" – Medium.com, 2022.
16. Article: "Scalable Web Application Design: Best Practices for Node.js and React" – Dev.to, 2021.

General Websites and Tutorials

17. [W3Schools](#) – For general HTML, CSS, and JavaScript references.
18. [GeeksforGeeks](#) – For coding examples, algorithm support, and backend logic.
19. [Stack Overflow](#) – For issue resolution and developer community support.
20. [MDN Web Docs](#) – For frontend web standards and browser compatibility.

STUDENTS PROFILE

Name: Janu Singh
Enrolment No.: 211B152
Email: 211b152@juetguna.in
Address: Satna, Madhya Pradesh
University: Jaypee University of engineering and Technology, Guna
Contact: 9343438546



Name: Jyoti Singh
Enrolment No.: 211B157
Email: 211b157@juetguna.in
Address: Rewa, Madhya Pradesh
University: Jaypee University of engineering and Technology, Guna
Contact: 7049022984



Name: Sachin Singh
Enrolment No.: 211B261
Email: 211b261@juetguna.in
Address: Rewa, Madhya Pradesh
University: Jaypee University of engineering and Technology, Guna
Contact: 7610376502



