Project Title:- Blood Bank Management System using MERN Stack

A Project Report for Industrial Internship
In the partial fulfilment for the award of the degree of

Submitted by
Anirudha Gorai
Ayandeep Roy
Prithwish Narayan Majumder
Ankit Saha
Abhishek Chakroborty
Sourabh Mandal

B.TECH

In the

Department OF **Computer Science and Engineering**

SILIGURI INSTITUTE OF TECHNOLOGY

at

Ardent Computech Pvt. Ltd.

Ardent® Computech Pvt. Ltd.
Module – 132, Ground Floor, SDF Building,
Sector V, Salt Lake, Kolkata – 700091.
☎+91 33 40073507
✉ training@ardentcollaborations.com

CERTIFICATE FROM SUPERVISOR

This is to certify that**,** Student have successfully completed the project titled "Blood Bank Management System" under my supervision during the period from “29th February” to “11st March” which is in partial fulfilment of requirements for the award of the **B.Tech** degree and submitted to the Department of Computer Science and Engineering.

____
Signature of the Supervisor
Date: 11/03/2024
Name of the Project Supervisor: **Sourav Goswami**

ACKNOWLEDGEMENT

The achievement that is associated with the successful completion of any task would be incomplete without mentioning the names of those people whose endless cooperation made it possible. Their constant guidance and encouragement made all our efforts successful.

We take this opportunity to express our deep gratitude towards our project mentor, *SOURAV GOSWAMI* for giving such valuable suggestions, guidance and encouragement during the Development of this project work.

Last but not the least we are grateful to all the faculty members of Ardent Computech Pvt. Ltd. for their support.

BBMS

TABLE OF CONTENTS

S.NO.	Name of the Topic	Page No.
1.	**Company Profile**	**10**
2	**Introduction**	**12-14**
2.a	Objective	13
2.b	Scope	14
3	**System Analysis**	**15-28**
3.a	Identification of Need	16-17
3.b	Feasibility Study	18
3.c	Work flow	19-22
3.d	Study of the System	23
3.e	Input & Output	24
3.f	Software Requirements Specification(SRS)	25-26
3.g	Software Engineering Paradigm Applied	27-28
4	**System Design**	**29-54**
4.a	Data Flow Diagram(DFD)	30-34
4.b	Sequence Diagram	35-36
4.c	Entity-Relationship Diagram	37-41
4.d	Use-case Diagram	42-45
4.e	Modularization Details	46-47
4.f	Database Design	48-54
5	**Output Screen**	**55-80**
5.a	User Interface Design	56
5.b	Snapshots	57-80
6	**Implementation & Testing**	**81-**
6.a	Introduction	81
6.b	Objectives of Testing	82-83

6.c	Test Cases	84-89
6.d	White Box Testing	90
6.e	Black Box Testing	91
6.f	Output Testing	92
6.g	Goal of Testing	93
6.h	Integration Test Reports	94-95
7	**Gantt Chart**	**……**
8	**System Security Measures**	**96-99**
8.a	Database security measures	97
8.b	System security measures	98
8.c	Limitations	99
9	**Conclusion**	**100**
10	**Future Scope & Further Enhancements**	**101**
11	**Bibliography/References**	**102**

1. ARDENT COMPUTECH PVT.LTD.

Ardent Computech Private Limited is an ISO 9001-2008 certified Software Development Company in India. It has been operating independently since 2003. It was recently merged with ARDENT TECHNOLOGIES.

Ardent Technologies

ARDENT TECHNOLOGIES is a Company successfully providing its services currently in UK, USA, Canada and India. The core line of activity at ARDENT TECHNOLOGIES is to develop customized application software covering the entire responsibility of performing the initial system study, design, development, implementation and training. It also deals with consultancy services and Electronic Security systems. Its primary clientele includes educational institutes, entertainment industries, resorts, theme parks, service industry, telecom operators, media and other business houses working in various capacities.

Ardent Collaborations

ARDENT COLLABORATIONS, the Research Training and Development Department of ARDENT COMPUTECH PVT LTD is a professional training Company offering IT enabled services & industrial trainings for B-Tech, MCA, BCA, MSc and MBA fresher’s and experienced developers/programmers in various platforms. Summer Training / Winter Training / Industrial training will be provided for the students of B.TECH, M.TECH, MBA and MCA only. Deserving candidates may be awarded stipends, scholarships and other benefits, depending on their performance and recommendations of the mentors.

Associations

Ardent is an ISO 9001:2008 company.

It is affiliated to National Council of Vocational Training (NCVT), Directorate General of Employment & Training (DGET), Ministry of Labor & Employment, and Government of India.

BBMS

(BLOOD BANK MANAGEMENT SYSTEM)

2. INTRODUCTION

Greetings and welcome to our Blood Bank App! Our mission is simple: connect hearts through life-saving blood donations. Greetings, and a warm welcome to the compassionate community of our Blood Bank App. Here, we're all about transforming the act of blood donation into a seamless, human-centric experience, connecting hearts and saving lives in the process.

Picture this: a user-friendly platform that not only simplifies the blood donation process but also creates a sense of purpose and belonging. Whether you're eager to be a hero donor, donating that life-giving elixir, or you find yourself in the position of needing urgent assistance, our app is designed with your needs in mind.

In the fast-paced world of modern technology, we're bringing a touch of humanity back to the forefront. Real-time updates ensure that you are always in the loop, while our intuitive interfaces make navigation as simple as a friendly conversation. We believe in making a positive impact through accessible and easy-to-use tools.

2a. OBJECTIVE

The primary objective of a blood bank management project is to improve the efficiency and effectiveness of blood banking operations. This translates to several key goals:
Safe & Available Blood: Ensure sufficient and timely supply of blood components.
Streamlined Donor Management: Simplify registration, profile management, and communication with donors.
Efficient Blood Processes: Optimize blood collection, scheduling, screening, and donation recording.
Improved Blood Allocation: Match blood types and patient needs with available inventory faster.
Enhanced Communication: Increase transparency for donors and hospitals regarding blood availability.

2b. TECHNOLOGY USES

MERN stands for a group of four technologies used to build modern web applications:

MongoDB: This is a NoSQL document database that offers flexibility and scalability for storing data. Unlike relational databases, MERN uses a document-oriented structure where data is stored in JSON-like documents. This makes it a good fit for storing complex and varied data types often encountered in web applications.

Express.js: This is a popular web framework built on top of Node.js. It provides a streamlined way to create web servers, handle HTTP requests and responses, and define application routing. Express.js simplifies back-end development by offering a structured approach and a rich set of features.

React.js: This is a JavaScript library for building user interfaces (UI). React uses a component-based approach where complex UIs are broken down into smaller, reusable components. This promotes code reusability, maintainability, and easier collaboration among developers.

Node.js: This is a JavaScript runtime environment that allows you to run JavaScript code outside of the browser. Node.js plays a crucial role in MERN by enabling the creation of servers written in JavaScript. This simplifies full-stack development using a single language (JavaScript) for both front-end and back-end logic.

Here's a simplified explanation of how these technologies work together in a MERN stack application:

Front-end (React): The user interacts with the visual elements of the web application built using React components.
Front-end to Back-end communication (API calls): When a user interacts with the front-end, React components might make API calls to the back-end server.
Back-end (Express.js and Node.js): The Express.js server receives these API requests, processes them using Node.js, and interacts with the MongoDB database as needed.
Data retrieval or manipulation (MongoDB): The back-end server retrieves or manipulates

data from the MongoDB database based on the API request.
Back-end to Front-end communication (Response): The back-end server sends a response (data or confirmation) back to the front-end.
Front-end updates: The React components in the front-end receive the response and update the UI accordingly.
Benefits of using MERN Stack:

Full-stack JavaScript: Using JavaScript for both front-end and back-end simplifies development and reduces the need for context switching between different languages.
Large Community and Resources: The MERN stack has a vast developer community and abundant learning resources, making it easier to find help and support.
Scalability and Performance: Both MongoDB and Node.js are known for their scalability, making MERN suitable for web applications that need to handle growing user bases.
Flexibility: The document-oriented nature of MongoDB and the component-based structure of React offer flexibility in data storage and UI development.

2c. SCOPE

The scope of the project includes the development of a web-based platform for managing and tracking blood donations, connecting donors with recipients, and providing real-time information on blood shortages and needs. The platform will include both a user-facing interface and an blood bank’s interface for managing the data.

2d. DETAILS OF SOLUTION ASPECTS

A blood bank management system (BBMS) deals with various functionalities, and the solution aspect can be approached from different angles. Here's a breakdown of some key areas where the system can offer solutions:

Inventory Management:

Blood Unit Tracking: The system can track blood units throughout their lifecycle, from donation to transfusion. This includes blood type, Rh factor, collection date, expiry date, storage location, and remaining shelf life.
Real-time Inventory Visibility: Blood banks can gain real-time insights into their blood stock levels, identify critical shortages, and optimize blood product allocation across different departments or hospitals.
Automated Alerts: Low stock alerts can trigger notifications to initiate blood donation drives or request replenishment from other blood banks.
Donor Management:

Donor Registration and Screening: The system can streamline donor registration, manage donor

information, and automate eligibility screening processes based on pre-defined criteria.
Donor Tracking and Communication: Track donor history, record donation frequency, and facilitate communication for recall donations or provide feedback.
Appointment Scheduling and Management: Manage donor appointments, send reminders, and automate follow-up communication.
Blood Safety and Quality Control:

Testing and Results Management: Track blood testing procedures, record test results, and ensure compliance with quality control standards.
Issue Tracking and Reporting: Identify and report any issues related to blood quality, storage, or transfusion reactions.
Donor Deferral Management: Manage donor deferrals based on test results or eligibility criteria.
Blood Transfusion Management:
equest Management: Manage blood requisition requests from hospitals or departments, ensuring compatibility and timely delivery.
Cross-Matching: The system can facilitate efficient blood cross-matching to identify compatible blood for transfusion based on patient blood type and other factors.
Transfusion Tracking: Track transfused blood units, record patient details, and maintain a log of transfusion events for future reference.
Additional Solutions:

Reporting and Analytics: Generate reports on blood stock levels, donor trends, blood usage patterns, and identify areas for improvement.
Security and Access Control: Implement role-based access control to ensure only authorized personnel can access sensitive data.
Integration with Hospital Systems: Integrate with existing hospital management systems to streamline data exchange and improve overall workflow efficiency.

By implementing these solutions, a blood bank management system can significantly improve blood bank operations, enhance blood safety, and ensure efficient blood product management for patient care.

SYSTEM ANALYSIS

3a. IDENTIFICATION OF NEED

System analysis is a process of gathering and interpreting facts, diagnosing problems and the information to recommend improvements on the system. It is a problem solving activity that requires intensive communication between the system users and system developers. System analysis or study is an important phase of any system development process .The system studies the minutest detail and gets analyzed. The system analysist plays the role of the interrogator and dwells deep into the working of the present system. The System is viewed as a whole and the input to the system are identified. The outputs from the organization are traced to the various processes. System analysis is concerned with becoming aware of the problem ,identifying the relevant and Decisional variables ,analysis and synthesizing the various factors and determining an optimal or at least a satisfactory solution or program of action.

A detailed study of the process must be made by various techniques like interviews, questionnaires etc. The data collected by these sources must be 9scrutinized to arrive to a conclusion. The conclusion is an understanding of how the system functions. This system is called the existing system .Now the existing system is subjected to close study and problem area are identified .The designer now function as a problem solver and tries to sort out the difficulties that the enterprise faces. The solution are given as proposals .The proposal is then weighed with the existing system analytically and the best one is selected .The proposal is presented to the user for an endorsement by the user .The proposal is reviewed on user request and suitable changes are made. This is loop that ends as soon as the user is satisfied with proposal.

3b. FEASIBILITY STUDY

Feasibility study is made to see if the project on completion will serve the purpose the organization for the amount of work.

Effort and the time that spend on it. Feasibility study lets the developer foresee the future of the project and the usefulness. A Feasibility study of a system proposal is according to its workability, which is the impact on the organization, ability to meet their user needs and effective use of resources .Thus when a new application is proposed it normally goes through a feasibility study before it is approved for development.

The document provide the feasibility of the project that is being designed and lists various area that were considered very carefully during the feasibility study of this project such as Technical , Economic and operational feasibilities.

3c. WORK FLOW

A blood bank management system (BBMS) streamlines the entire process of blood collection, testing, storage, and distribution. Here's a typical workflow:

Donor Management:

1. **Donor Registration:** New donors can register online or at the blood bank itself. The system captures their details, medical history, and contact information.
2. **Donor Screening:** During donation, staff verify donor eligibility through a questionnaire and basic health checks.
3. **Blood Collection:** Blood collection is performed by trained staff following safety protocols. The system tracks the collected blood volume and donor ID.

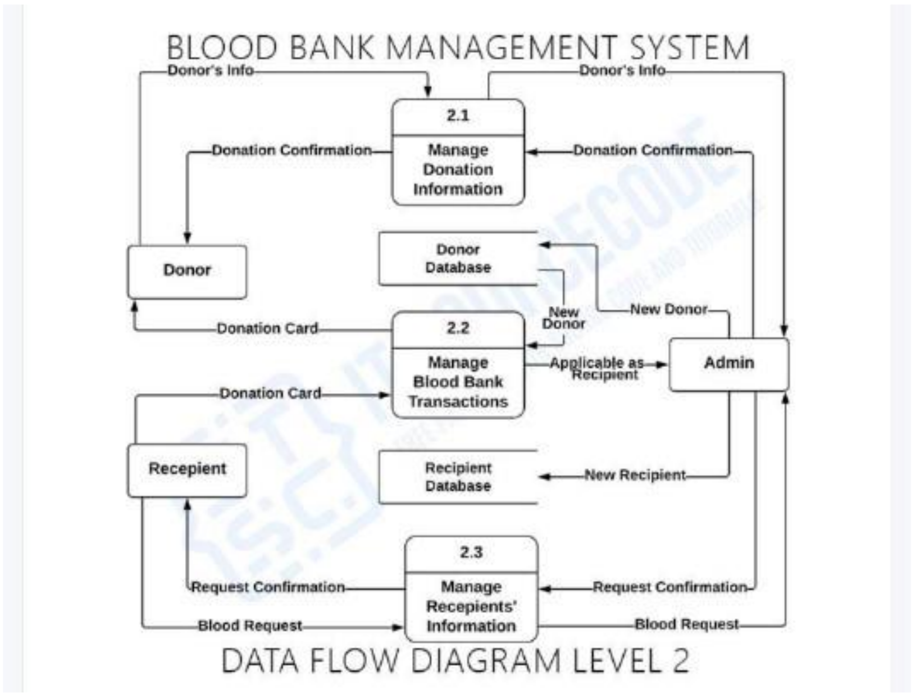
Blood Processing & Testing:

1. **Component Separation:** Collected blood is centrifuged to separate red blood cells (RBCs), plasma, and platelets.
2. **Blood Testing:** Each blood component undergoes rigorous testing for infectious diseases and blood type compatibility.
3. **Labeling & Storage:** The system assigns unique identification labels to each blood component based on test results and expiry dates. Blood is stored under controlled temperature conditions.

Inventory Management:

1. **Blood Stock Tracking:** The BBMS tracks blood inventory levels for each blood type and component.
2. **Expiry Monitoring:** The system generates alerts for blood nearing its expiry date,

allowing for proper utilization or disposal.



Request & Distribution:

1. **Hospital Requests:** Hospitals submit blood requisition requests electronically or through the BBMS interface.
2. **Blood Compatibility Matching:** The system matches patient blood type with compatible blood units based on urgency and inventory.
3. **Blood Issuance:** Blood is prepared for transfusion and released to the requesting hospital with proper documentation.

Additional Features:

- Donor Relationship Management (DRM): BBMS can manage donor communication, send appointment reminders, and express gratitude for donations.
- Reporting & Analytics: The system generates reports on blood collection, inventory levels, usage patterns, and expiry dates.

- Integration: BBMS can integrate with hospital information systems for seamless blood request and tracking.

By automating these workflows, a BBMS ensures efficient blood bank operations, minimizes errors, and ultimately helps deliver life-saving blood transfusions to patients in need.

3d. STUDY OF THE SYSTEM

A study of a blood bank management system (BBMS) goes beyond the technical implementation using a specific framework like MERN stack. Here's a broader approach to studying a BBMS:

Understanding the System Landscape:

- **Existing Systems:** Research existing BBMS solutions in hospitals, blood banks, or national registries. Analyze their functionalities, user interface design, and reported strengths and weaknesses.
- **Regulatory Requirements:** Familiarize yourself with regulations governing blood collection, testing, storage, and distribution set by national or regional health authorities.

Functional Requirements Analysis:

- **User Needs:** Identify the needs of different user groups - donors, blood bank staff, hospital staff, and potentially blood recipients (if given controlled access). Conduct user interviews or surveys to understand their workflows and pain points.
- **Workflow Mapping:** Map out the existing blood donation, processing, storage, and distribution workflows. Identify bottlenecks, inefficiencies, and opportunities for improvement.

System Design and Features:

- **Core Functionalities:** Define the core functionalities required for efficient blood bank management. This includes donor management, blood processing & testing, inventory control, request processing & distribution, and reporting & analytics.
- **Advanced Features:** Consider incorporating advanced features like donor relationship management (DRM) for communication and appointment reminders, integration with

hospital information systems for seamless blood request exchange, and real-time blood availability dashboards.

Technical Considerations:

- **Technology Stack Evaluation:** While MERN stack is a popular choice, explore other options like LAMP (Linux, Apache, MySQL, PHP) or MEAN (MongoDB, Express.js, Angular, Node.js) based on specific project requirements and team expertise.
- **Data Security & Privacy:** Prioritize data security measures like user authentication, authorization, data encryption, and secure data storage practices to protect sensitive donor and blood information.

Implementation and Testing:

- **Agile Development:** Consider adopting an agile development methodology for iterative development, allowing for continuous feedback and improvement throughout the development process.
- **Testing Strategies:** Implement thorough testing strategies including unit testing, integration testing, and user acceptance testing to ensure system functionality, performance, and usability.

Evaluation and Impact Assessment:

- **Performance Monitoring:** Monitor system performance metrics like response times, resource utilization, and uptime to identify areas for optimization.
- **User Feedback:** Continuously gather feedback from users to identify areas for improvement and ensure the BBMS effectively addresses their needs.
- **Impact Analysis:** Evaluate the impact of the BBMS on blood bank operations such as increased blood collection efficiency, reduced wastage, and improved response times to blood requests.

By following a comprehensive study approach, you can gain a deeper understanding of BBMS functionalities, user needs, technical considerations, and best practices. This knowledge can be applied to develop or improve a BBMS that effectively manages the critical aspects of blood donation and delivery, ultimately saving lives.

3e. INPUT AND OUTPUT

A Blood Bank Management System (BBMS) deals with data related to donors, blood units, hospitals, and the overall blood donation process. Here's a breakdown of the typical inputs and outputs of a BBMS:

Inputs:

- **Donor Information:** During registration, the system captures donor details like name, contact information, medical history, blood type, and donation frequency.
- **Blood Collection Data:** Staff input details like blood volume collected, date of donation, and any relevant observations during the donation process.
- **Blood Testing Results:** Laboratory technicians input results from blood tests for infectious diseases, blood type confirmation, and Rh factor.
- **Hospital Requests:** Hospitals submit requests for blood specifying blood type, quantity, and urgency level.
- **Inventory Management:** Data on blood unit storage location, expiry dates, and current inventory levels is continuously updated.

Outputs:

- **Donor Cards/Reports:** Donors receive donor cards or reports with their blood type, donation history, and next eligible donation date.
- **Blood Labels:** The system generates unique identification labels for each blood unit containing details like blood type, component (RBCs, plasma, platelets), collection date, and expiry date.
- **Inventory Reports:** The BBMS provides reports on current blood stock levels for different blood types and components, including those nearing expiry.

- **Blood Issue Reports:** The system generates reports for issued blood units, including recipient details, hospital information, and blood type provided.
- **Alerts & Notifications:** The BBMS can trigger alerts for low blood stock levels of specific types, expiring blood units, and appointment reminders for upcoming blood drives or eligible donors.
- **Analytics & Dashboards:** The system can provide dashboards and reports for visualizing blood donation trends, usage patterns, and potential blood shortages.

These inputs and outputs are essential for managing blood bank operations efficiently. The BBMS ensures accurate data capture, tracking, and reporting, ultimately facilitating a safe and timely blood supply for patients in need.

3f. SOFTWARE REQUIREMENT SPECIFICATIONS

Software Requirements Specification provides an overview of the entire project. It is a description of a software system to be developed, laying out functional and nonfunctional requirements. The software requirements specification document enlists enough and necessary requirements that are required for the project development. To derive the requirements we need to have clear and thorough understanding of the project to be developed. This is prepared after the detailed communication with project team and the customer.

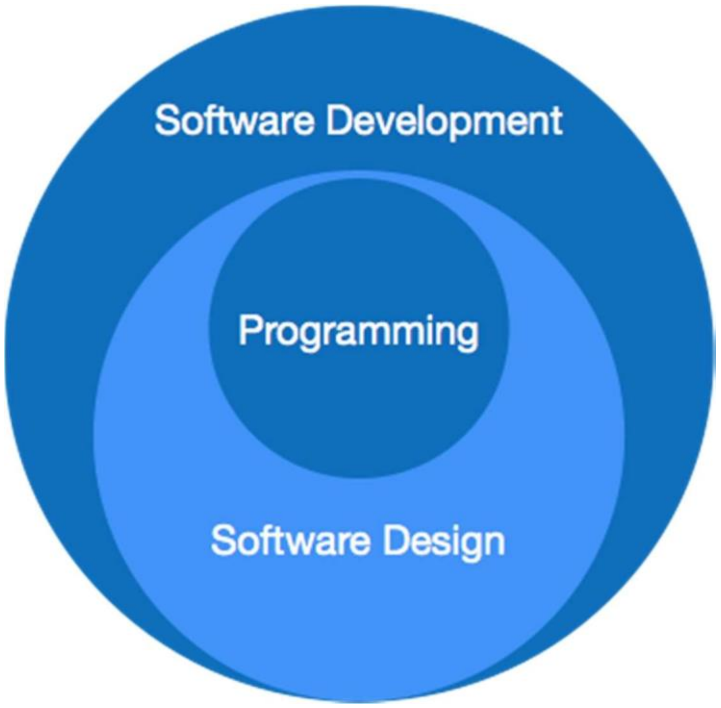
The developer is responsible for:-

- ✓ Developing the system, which meets the SRS and solving all the requirements of the system?
- ✓ Demonstrating the system and installing the system at client's location after acceptance testing is successful.
- ✓ Submitting the required user manual describing the system interfaces to work on it and also the documents of the system.
- ✓ Conducting any user training that might be needed for using the system.
- ✓ Maintain the system for a period of one year after installation.

- **Operating System:**
    - Supported OS: Windows 10 (64-bit), macOS (latest stable version), Linux (major distributions like Ubuntu, Debian, Fedora)
    - Minimum requirement: Ability to run command-line tools and install software packages.
- **Hardware:**
    - Minimum: Processor - Intel Core i3 or equivalent, RAM - 8 GB, Storage - 50 GB free space
    - Recommended: Consider upgrading hardware specifications (especially RAM) for larger projects and improved performance.
- **Software:**
    - **Essential:**
        - Node.js (latest Long-Term Support (LTS) version recommended) with npm (or yarn) package manager included.
        - MongoDB (latest stable version recommended).
        - Code editor or IDE (e.g., Visual Studio Code, Atom, WebStorm, Sublime Text).
    - **Recommended:**
        - Version control system (Git with a Git client like GitHub Desktop).
        - Web browser (Chrome, Firefox, Safari, Edge).
        - Familiarity with the command line (terminal/command prompt).

3g. SOFTWARE ENGINEERING PARADIGM APPLIED

Software paradigms refer to the methods and steps, which are taken while designing the software. There are many methods proposed and are in work today, but we need to see where in the software engineering these paradigms stand. These can be combined into various categories, though each of them is contained in one another.



Programming paradigm is a subset of Software design paradigm which is further a subset of Software development paradigm.

There are two levels of reliability. The first is meeting the right requirement. A carefully and through systems study is needed to satisfy this aspect of reliability. The second level of systems reliability involves the actual working delivered to the user. At this level, the systems reliability is interwoven with software engineering and development. There are three approaches to reliability.

1. Error avoidance: Prevents errors from occurring in software.

2. Error detection and correction: In this approach errors are recognized whenever they are encountered and correcting the error by effect of error of the system does not fail.

3. Error tolerance: In this approach errors are recognized whenever they occur, but enables the system to keep running through degraded perform or Appling values that instruct the system to continue process.

Maintenance:

The key to reducing need for maintenance, while working, if possible to do essential tasks.

1. More accurately defining user requirement during system development.

2. Assembling better systems documents.

3. Using some effective methods for designing, processing, and login and communicating information with project team members.

4. Making better use of existing tools and techniques.

SYSTEM DESIGN

4a. DATA FLOW DIAGRAM

A **data flow diagram** (**DFD**) is a graphical representation of the "flow" of data through an information system, modelling its process aspects. A **DFD** is often used as a preliminary step to create an overview of the system, which can later be elaborated.

DFDs can also be used for the visualization of data processing (structured design).

A DFD shows what kind of information will be input to and output from the system, where the data will come from and go to, and where the data will be stored. It does not show information about the timing of process or information about whether processes will operate in sequence or in parallel (which is shown on a flowchart).

This context-level DFD is next "exploded", to produce a Level 1 DFD that shows some of the detail of the system being modeled. The Level 1 DFD shows how the system is divided into sub-systems (processes), each of whi1ch deals with one or more of the data flows to or from an external agent, and which together provide all of the functionality of the system as a whole. It also identifies internal data stores that must be present in order for the system to do its job, and shows the flow of data between the various parts of the system.

Data flow diagrams are one of the three essential perspectives of the structured-systems analysis and design method SSADM. The sponsor of a project and the end users will need to be briefed and consulted throughout all stages of a system's evolution. With a data flow diagram, users are able to visualize how the system will operate, what the system will accomplish, and how the system will be implemented. The old system's dataflow diagrams can be drawn up and compared with the new system's data flow diagrams to draw comparisons to implement a more efficient system. Data flow diagrams can be used to provide the end user with a physical idea of where the data they input ultimately has an effect upon the structure of the whole system from order to dispatch to report.

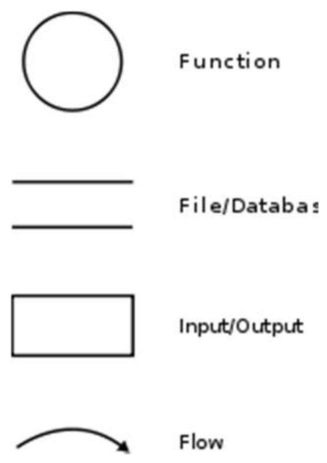
How any system is developed can be determined through a data flow diagram model.

In the course of developing a set of *leveled* data flow diagrams the analyst/designer is forced to address how the system may be decomposed into component sub-systems, and to identify the transaction data in the data model.

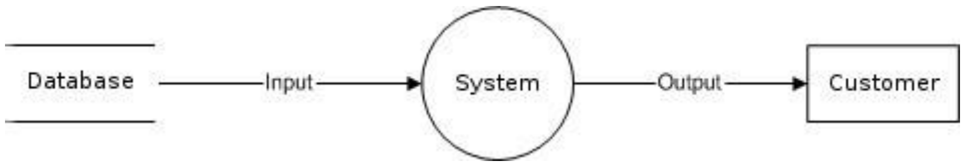
Data flow diagrams can be used in both Analysis and Design phase of the SDLC.

There are different notations to draw data flow diagrams. defining different visual representations for processes, data stores, data flow, and external entities.[6]

DFD NOTATION



DFD EXAMPLE



Steps to Construct Data Flow Diagram:-

Four Steps are generally used to construct a DFD.

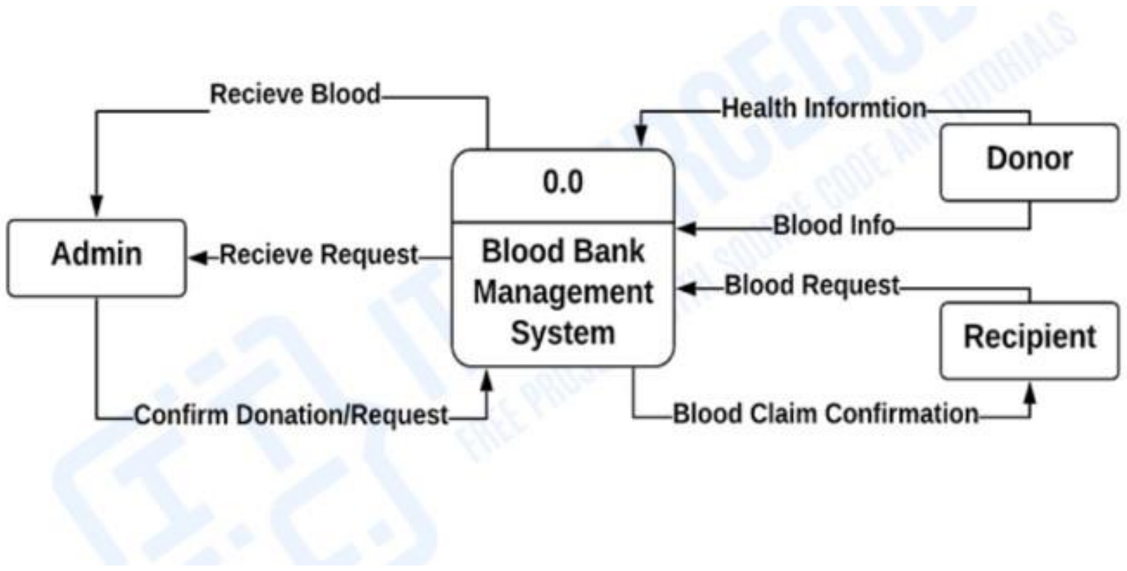
- Process should be named and referred for easy reference. Each name should be representative of the reference.
- The destination of flow is from top to bottom and from left to right.
- When a process is distributed into lower level details they are numbered.
- The names of data stores, sources and destinations are written in capital letters.

Rules for constructing a Data Flow Diagram:-

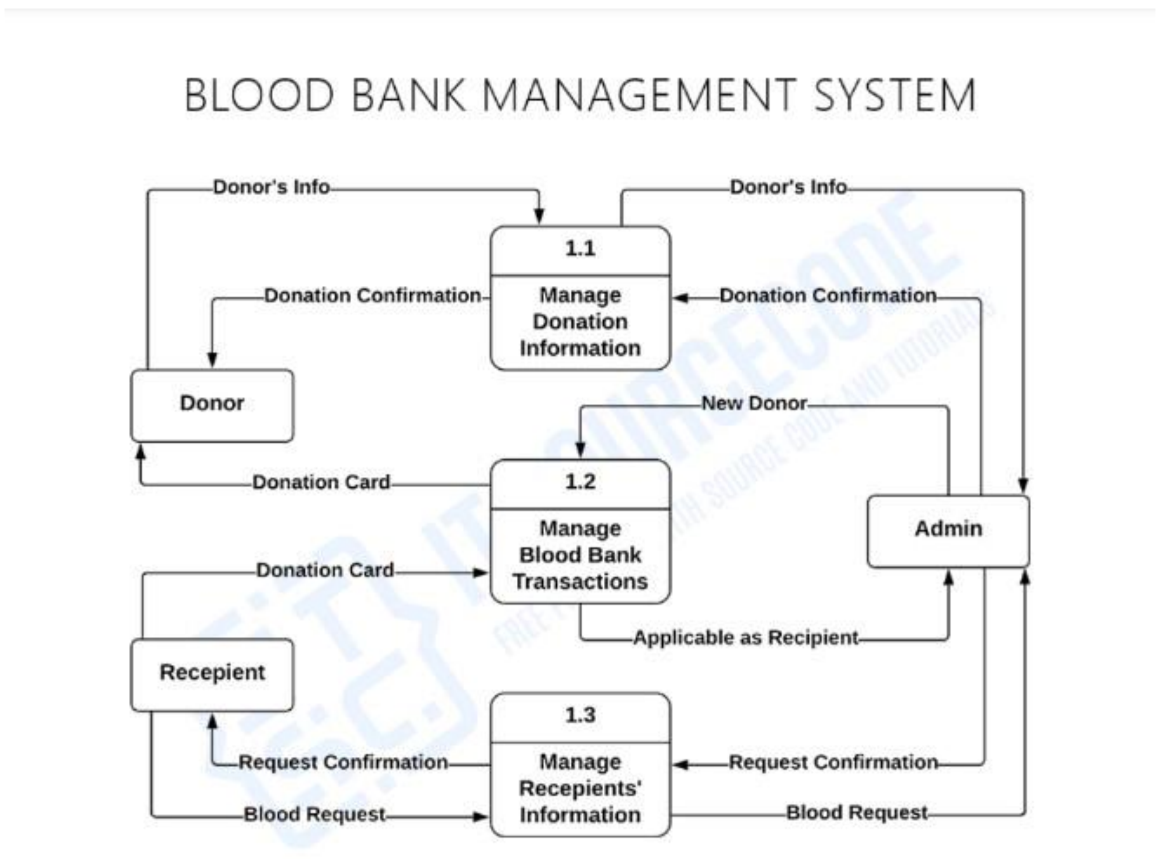
- Arrows should not cross each other.
- Squares, Circles, Files must bear a name.
- Decomposed data flow squares and circles can have same names.
- Draw all data flow around the outside of the diagram.

DATA FLOW DIAGRAM

LEVEL-0 DFD DIAGRAM



LEVEL-1 DFD DIAGRAM

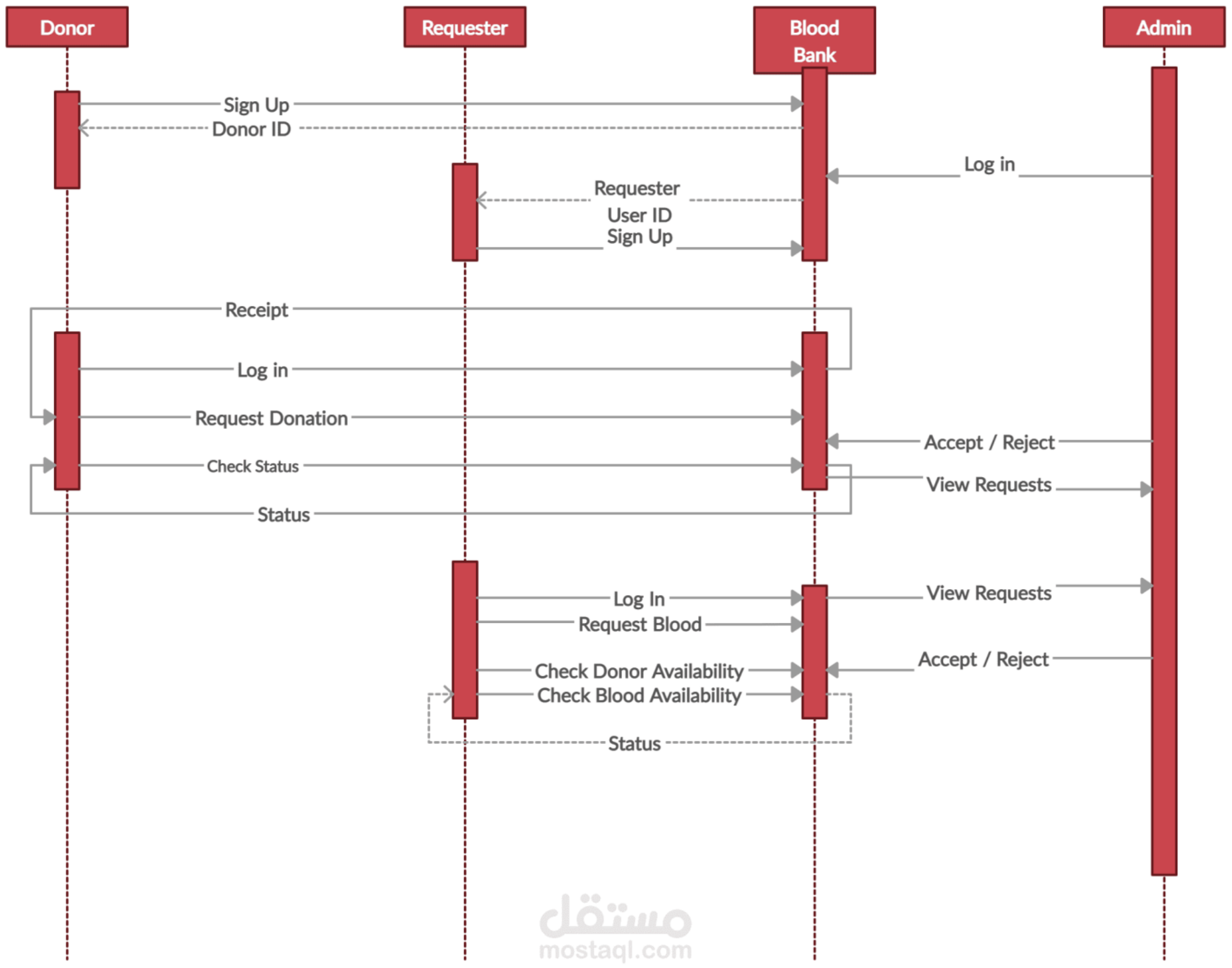


4b. SEQUENCE DIAGRAM

A **Sequence diagram** is an interaction diagram that shows how processes operate with one another and what is their order. It is a construct of a Message Sequence Chart. A sequence diagram shows object interactions arranged in time sequence. It depicts the objects and classes involved in the scenario and the sequence of messages exchanged between the objects needed to carry out the functionality of the scenario. Sequence diagrams are typically associated with use case realizations in the Logical View of the system under development. Sequence diagrams are sometimes called **event diagrams** or **event scenarios**.

A sequence diagram shows, as parallel vertical lines (*lifelines*), different processes or objects that live simultaneously, and, as horizontal arrows, the messages exchanged between them, in the order in which they occur. This allows the specification of simple runtime scenarios in a graphical manner.

Sequence diagram is the most common kind of **interaction diagram**, which focuses on the **message** interchange between a number of **lifelines**.
Sequence diagram describes an interaction by focusing on the sequence of messages that are exchanged, along with their corresponding occurrence specifications on the lifelines.
The following nodes and edges are typically drawn in a **UML sequence diagram**: **lifeline**, **execution-specification**, **message**, fragment, interaction, **state invariant**, **continuation**, **destruction occurrence**.


Donor
Requester
Blood Bank
Admin
Sign Up
Donor ID
Log in
Requester User ID Sign Up
Receipt
Log in
Request Donation
Accept / Reject
Check Status
View Requests
Status
View Requests
Log In
Request Blood
Accept / Reject
Check Donor Availability
Check Blood Availability
Status

4c. ENTITY RELATIONSHIP DIAGRAM

In software engineering, an **entity–relationship model**(**ER model**) is a data model for describing the data or information aspects of a business domain or its process requirements, in an abstract way that lends itself to ultimately being implemented in a database such as a relational. The main components of ER models are entities (things) and the relationships that can exist among them.

An entity–relationship model is the result of using a systematic process to describe and define a subject area of business data. It does not define business process; only visualize business data. The data is represented as components (entities) that are linked with each other by relationships that express the dependencies and requirements between them, such as: one building may be divided into zero or more apartments, but one apartment can only be located in one building. Entities may have various properties (attributes) that characterize them. Diagrams created to represent these entities, attributes, and relationships graphically are called entity–relationship diagrams.

An ER model is typically implemented as a database. In the case of a relational database, which stores data in tables, every row of each table represents one instance of an entity. Some data fields in these tables point to indexes in other tables; such pointers are the physical implementation of the relationships.

The three schema approach to software engineering uses three levels of ER models that may be developed.

Conceptual data model

The conceptual ER model normally defines master reference data entities that are commonly used by the organization. Developing an enterprise-wide conceptual ER model is useful to support documenting the data architecture for an organization.A conceptual ER model may be used as the foundation for one or more logical data models . The purpose of the conceptual ER model is then to establish structural metadata commonality for the master data entities between the set of logical ER models. The conceptual data model may be used to form commonality relationships between ER models as a basis for data model integration.

Logical data model

The logical ER model contains more detail than the conceptual ER model. In addition to master data entities, operational and transactional data entities are now defined. The details of each data entity are developed and the relationships between these data entities are established. The logical ER model is however developed independent of technology into which it can be implemented.

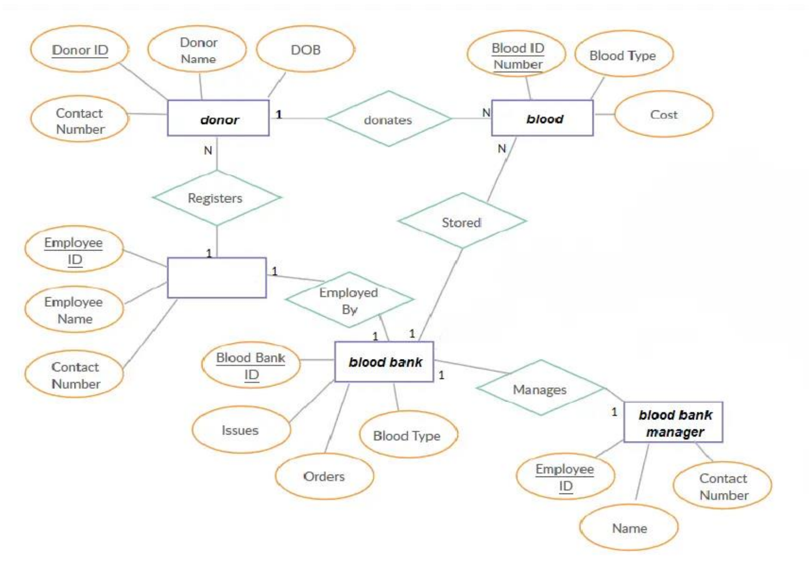
Physical data model

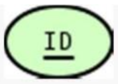
One or more physical ER models may be developed from each logical ER model. The physical ER model is normally developed to be instantiated as a database. Therefore, each physical ER model must contain enough detail to produce a database and each physical ER model is technology dependent since each database management system is somewhat different.

The physical model is normally instantiated in the structural metadata of a database management system as relational database objects such as database tables, database indexes such as unique keyindexes, and database constraints such as a foreign key constraint or a commonality constraint. The ER model is also normally used to design modifications to the relational database objects and to maintain the structural metadata of the database.

The first stage of information system design uses these models during the requirements analysis to describe information needs or the type of information that is to be stored in a database. The data modelingtechnique can be used to describe any ontology (i.e. an overview and classifications of used terms and their relationships) for a certain area of interest. In the case of the design of an information system that is based on a database, the conceptual data model is, at a later stage (usually called logical design), mapped to a logical data model, such as the relational model; this in turn is mapped to a physical model during physical design. Note that sometimes, both of these phases are referred to as "physical design". It is also used in database management system.

Entity–relationship modeling





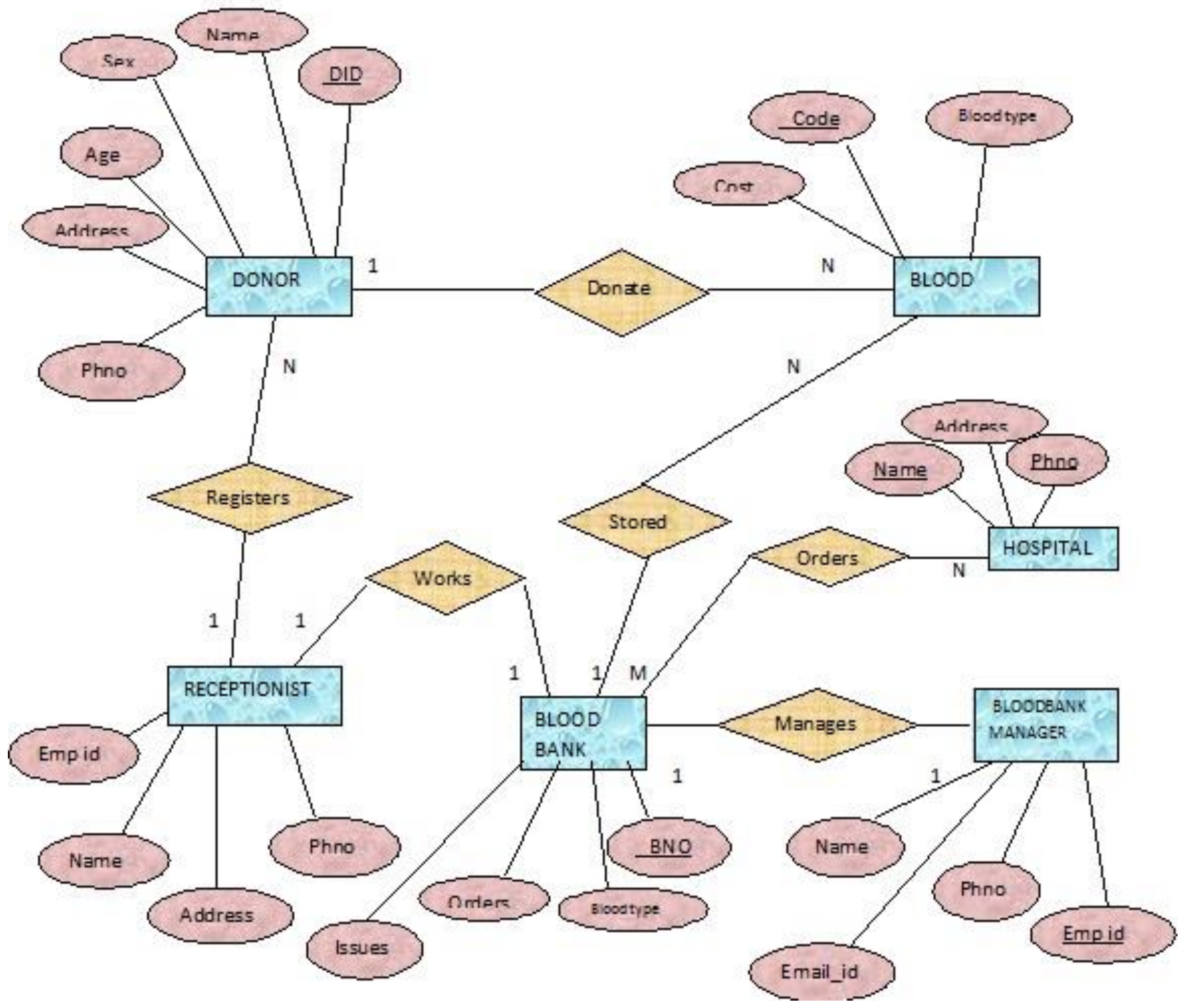
Primary key

Cardinality constraints are expressed as follows:

- A double line indicates a *participation constraint*, totality or subjectivity : all entities in the entity set must participate in *at least one* relationship in the relationship set;
- an arrow from entity set to relationship set indicates a key constraint, i.e. injectivity: each entity of the entity set can participate in *at most one* relationship in the relationship set;
- A thick line indicates both, i.e. bijectivity: each entity in the entity set is involved in *exactly one* relationship.
- An underlined name of an attribute indicates that it is a key: two different entities or relationships with this attribute always have different values for this attribute.

ER-DIAGRAM

THE COMPLETE ER- DIAGRAM



4d. USE CASE DIAGRAM

A **use case diagram** at its simplest is a representation of a user's interaction with the system that shows the relationship between the user and the different use cases in which the user is involved. A use case diagram can identify the different types of users of a system and the different use cases and will often be accompanied by other types of diagrams as well.

So only static behavior is not sufficient to model a system rather dynamic behavior is more important than static behavior. In UML there are five diagrams available to model dynamic nature and use case diagram is one of them. Now as we have to discuss that the use case diagram is dynamic in nature there should be some internal or external factors for making the interaction.

These internal and external agents are known as actors. So use case diagrams are consists of actors, use cases and their relationships. The diagram is used to model the system/subsystem of an application. A single use case diagram captures a particular functionality of a system.

So to model the entire system numbers of use case diagrams are used.

The purpose of use case diagram is to capture the dynamic aspect of a system. But this definition is too generic to describe the purpose.

Because other four diagrams (activity, sequence, collaboration and State chart) are also having the same purpose. So we will look into some specific purpose which will distinguish it from other four diagrams.

Use case diagrams are used to gather the requirements of a system including internal and external influences. These requirements are mostly design requirements. So when a system is analyzed to gather its functionalities use cases are prepared and actors are identified.

Now when the initial task is complete use case diagrams are modelled to present the outside view.

So in brief, the purposes of use case diagrams can be as follows:

- Used to gather requirements of a system.
- Used to get an outside view of a system.
- Identify external and internal factors influencing the system.
- Show the interacting among the requirements are actors.

How to draw Use Case Diagram?

Use case diagrams are considered for high level requirement analysis of a system. So when the requirements of a system are analyzed the functionalities are captured in use cases.

So we can say that uses cases are nothing but the system functionalities written in an organized manner. Now the second things which are relevant to the use cases are the actors. Actors can be defined as something that interacts with the system.

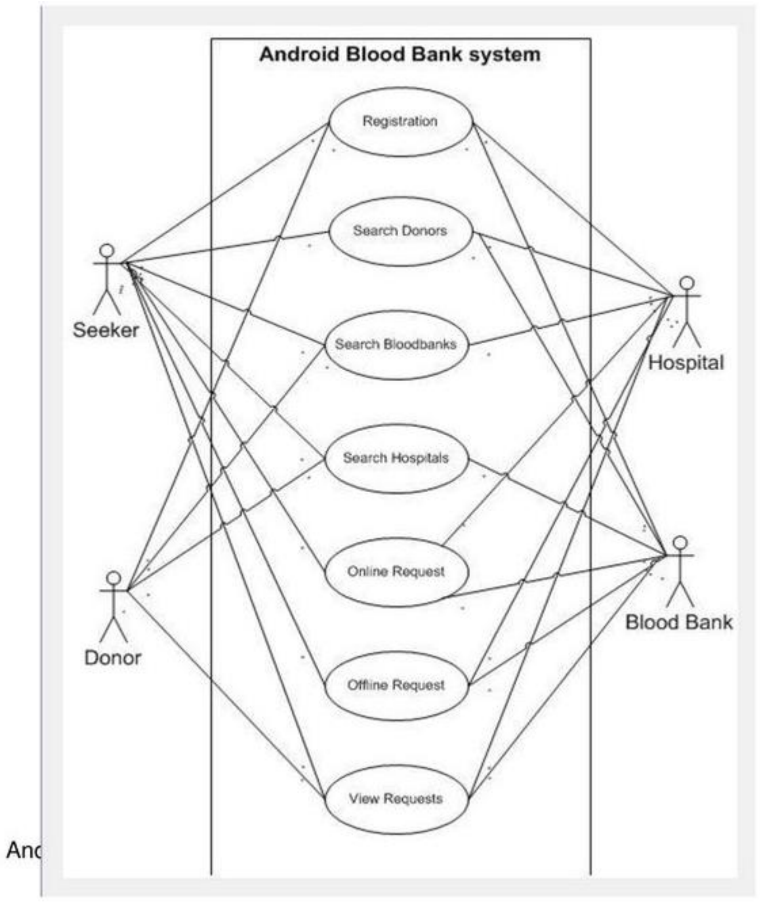
The actors can be human user, some internal applications or may be some external applications. So in a brief when we are planning to draw a use case diagram we should have the following items identified.

- Functionalities to be represented as an use case
- Actors
- Relationships among the use cases and actors.

Use case diagrams are drawn to capture the functional requirements of a system. So after identifying the above items we have to follow the following guidelines to draw an efficient use case diagram.

- The name of a use case is very important. So the name should be chosen in such a way so that it can identify the functionalities performed.
- Give a suitable name for actors.
- Show relationships and dependencies clearly in the diagram.
- Do not try to include all types of relationships. Because the main purpose of the diagram is to identify requirements.
- Use note whenever required to clarify some important points.

USE CASE DIAGRAM



4.e MODULARIZATION DETAILS

As Modularization has gained increasing focus from companies outside its traditional industries of aircraft and automotive, more and more companies turn to it as strategy and product development tool. I intend to explain the importance aspects of modularization and how it should be initiated within a company. After determining the theoretical steps of modularization success described in literature, I intend to conduct a multiple case study of companies who have implemented modularization in order to find how real world modularization was initiated and used to improve the company's competitiveness. By combining theory and practical approach to modularization I will derive at convergence and divergence between theoretical implementation to modularization and real world implementation to modularization. This gives a valuable input for both implantations in companies as well as new aspects to be further.

DATA INTEGRITY AND CONSTRAINTS

Data integrity is normally enforced in a database system by a series of integrity constraints or rules. Three types of integrity constraints are an inherent part of the relational data model: entity integrity, referential integrity and domain integrity:

- *Entity integrity* concerns the concept of a primary key. Entity integrity is an integrity rule which states that every table must have a primary key and that the column or columns chosen to be the primary key should be unique and not null.

- Concerns the concept of a foreign key. The referential integrity rule states that any foreign-key value can only be in one of two states. The usual state of affairs is that the foreign-key value refers to a primary key value of some table in the database.

Occasionally, and this will depend on the rules of the data owner, a foreign-key value can be null. In this case we are explicitly saying that either there is no relationship between the objects represented in the database or that this relationship is unknown.

- *Domain integrity* specifies that all columns in a relational database must be declared upon a defined domain. The primary unit of data in the relational data model is the data item. Such data items are said to be non-decomposable or atomic. A domain is a set of values of the same type.

4f. DATABASE DESIGN

A database is an organized mechanism that has capability of storing information through which a user can retrieve stored information in an effective and efficient manner. The data is the purpose of any database and must be protected.

The database design is two level processes. In the first step, user requirements are gathered together and a database is designed which will meet these requirements as clearly as possible. This step is called information Level design and it is taken independent of any individual DBMS.

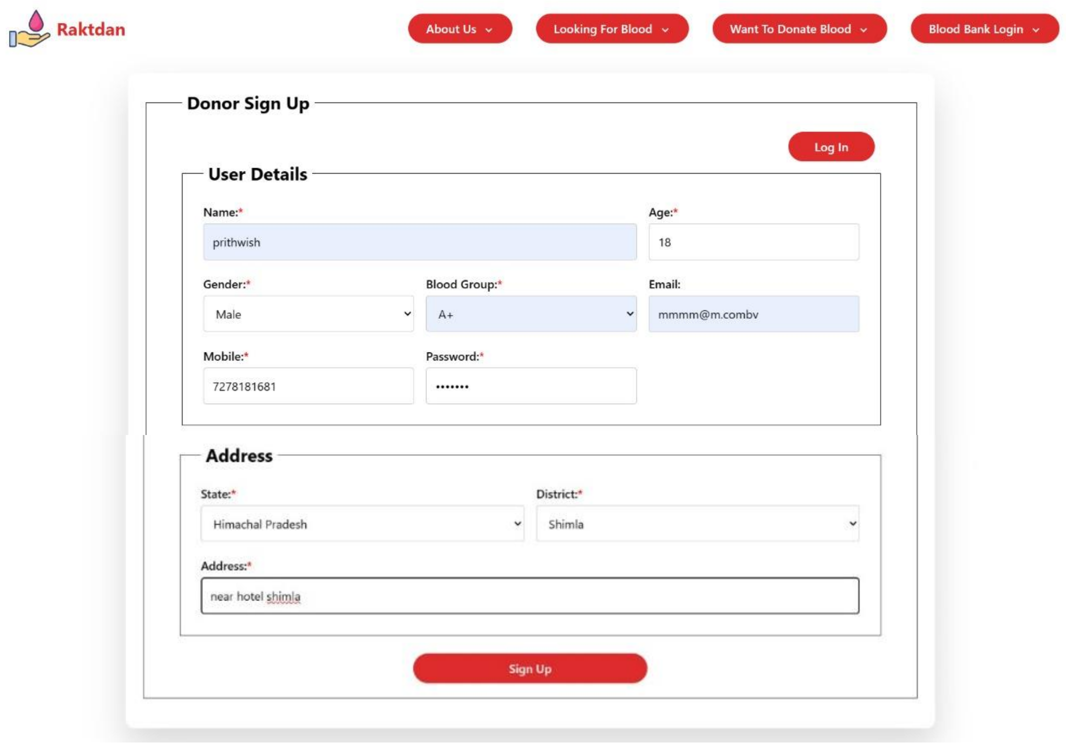
In the following snapshots we display the way we have used SQL Server as the back-end RDBMS for our project and the various entities that have been used along with their table definition and table data.

DATA DICTIONARY

config,Db.js :

```
const mongoose = require('mongoose');

const connectDB = async () => {

   mongoose.connect(process.env.MONGO_URL, { useNewUrlParser: true, useUnifiedTopology: true,
useCreateIndex: true }, (e) => {
       console.log(e ? e : "Connected successfully to database");
   });

};

module.exports = connectDB;
```

Donor Data Input:



Donor Sign Up

User Details

Name:* prithwish

Age:* 18

Gender:* Male

Blood Group:* A+

Email: mmmm@m.combv

Mobile:* 7278181681

Password:* •••••••

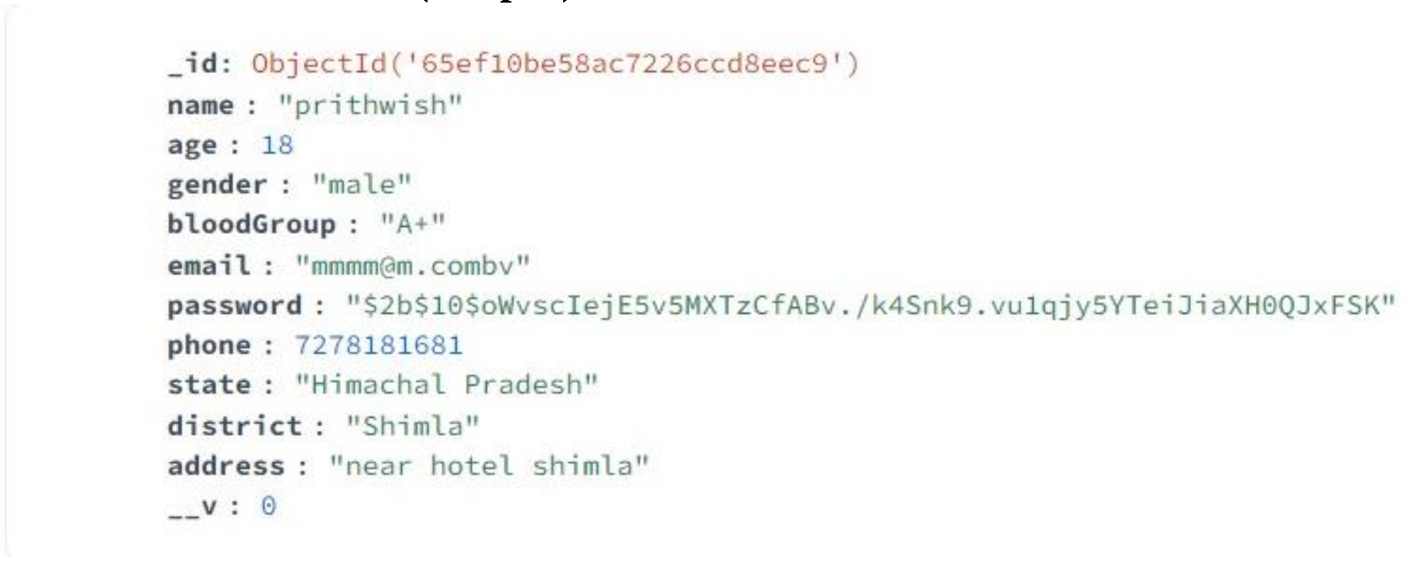
Address

State:* Himachal Pradesh

District:* Shimla

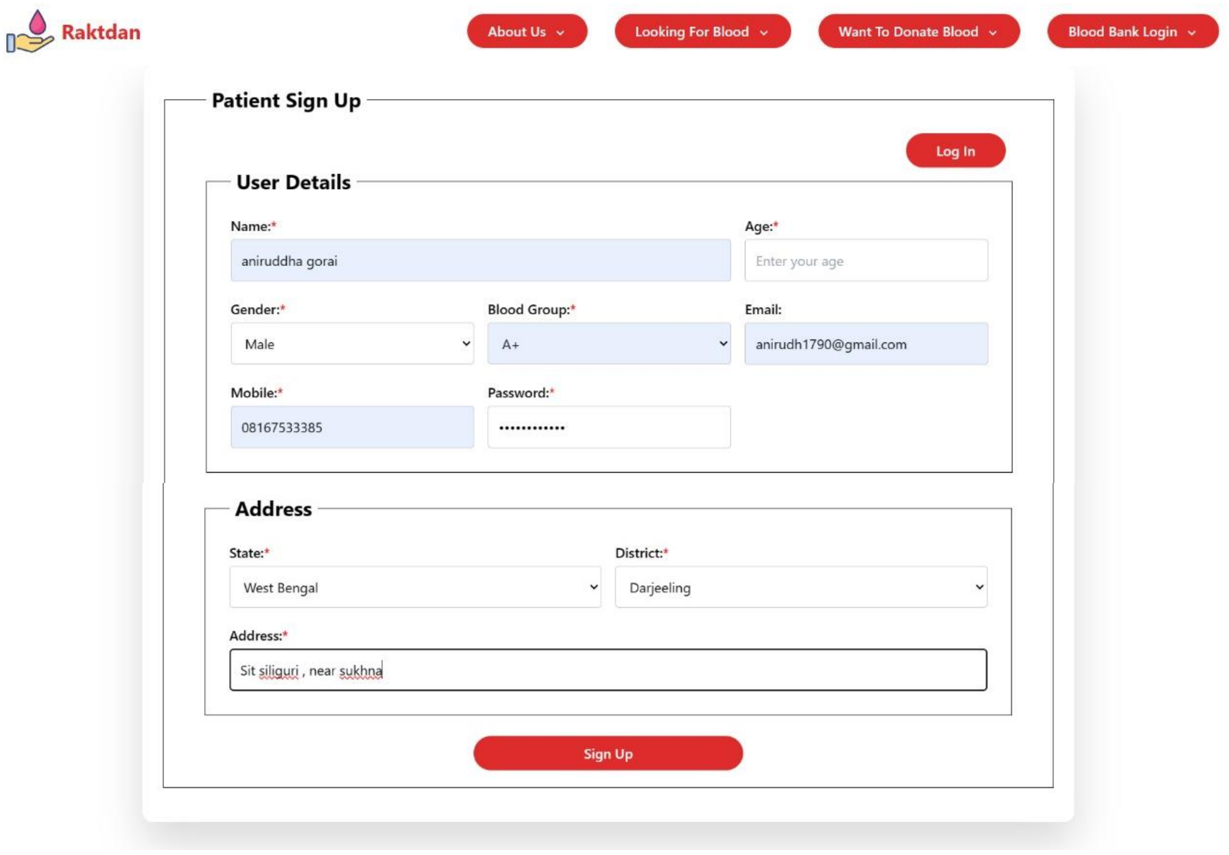
Address:* near hotel shimla

Database(Output):



```
_id: ObjectId('65ef10be58ac7226ccd8eec9')
name : "prithwish"
age : 18
gender : "male"
bloodGroup : "A+"
email : "mmmm@m.combv"
password : "$2b$10$oWvscIejE5v5MXTzCfABv./k4Snk9.vu1qjy5YTeiJiaXH0QJxFSK"
phone : 7278181681
state : "Himachal Pradesh"
district : "Shimla"
address : "near hotel shimla"
__v : 0
```

New Patient Registration Data:



Patient Sign Up

User Details

Name:* aniruddha gorai

Age:* Enter your age

Gender:* Male

Blood Group:* A+

Email: anirudh1790@gmail.com

Mobile:* 08167533385

Password:* ••••••••••••

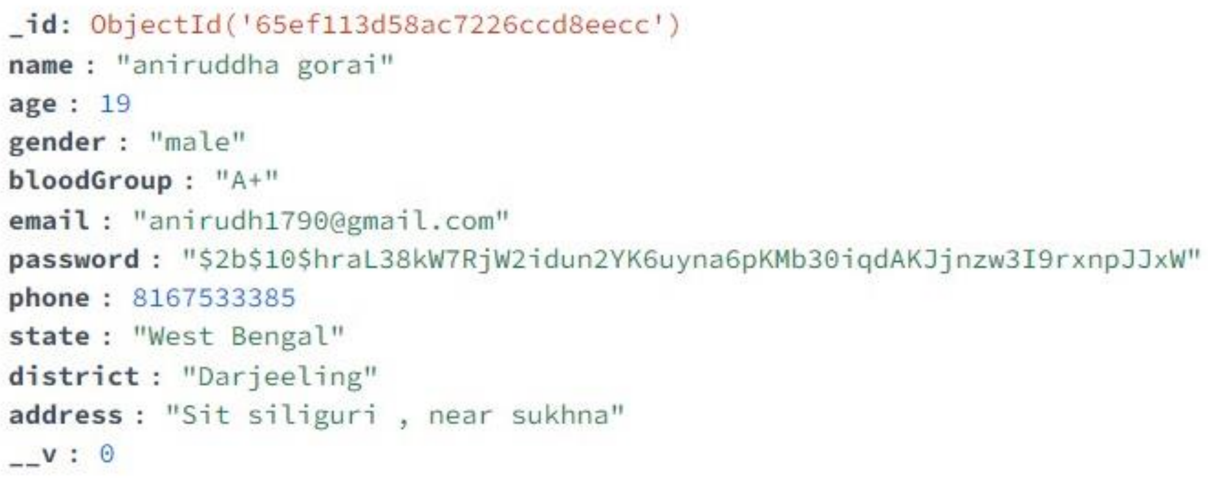
Address

State:* West Bengal

District:* Darjeeling

Address:* Sit siliguri , near sukhna

Database(Output):



```
_id: ObjectId('65ef113d58ac7226ccd8eecc')
name : "aniruddha gorai"
age : 19
gender : "male"
bloodGroup : "A+"
email : "anirudh1790@gmail.com"
password : "$2b$10$hraL38kW7RjW2idun2YK6uyna6pKMb30iqdAKJjnzw3I9rxnpJJxW"
phone : 8167533385
state : "West Bengal"
district : "Darjeeling"
address : "Sit siliguri , near sukhna"
__v : 0
```

OUTPUT SCREEN

5a. USER INTERFACE DESIGN

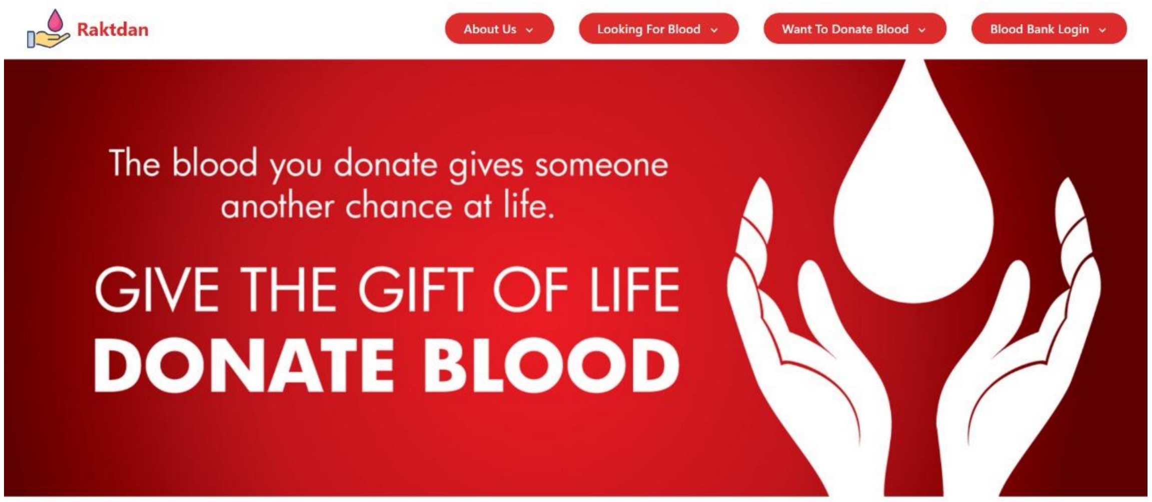
User interface design (UID) or **user interface engineering** is the design of user interfaces for machines and software, such as computers, home appliances, mobile devices, and other electronic devices, with the focus on maximizing the user experience. The goal of user interface design is to make the user's interaction as simple and efficient as possible, in terms of accomplishing user goals (user-centered design).

Good user interface design facilitates finishing the task at hand without drawing unnecessary attention to it. Graphic design and typography are utilized to support its usability, influencing how the user performs certain interactions and improving the aesthetic appeal of the design; design aesthetics may enhance or detract from the ability of users to use the functions of the interface. The design process must balance technical functionality and visual elements (e.g., mental model) to create a system that is not only operational but also usable and adaptable to changing user needs.

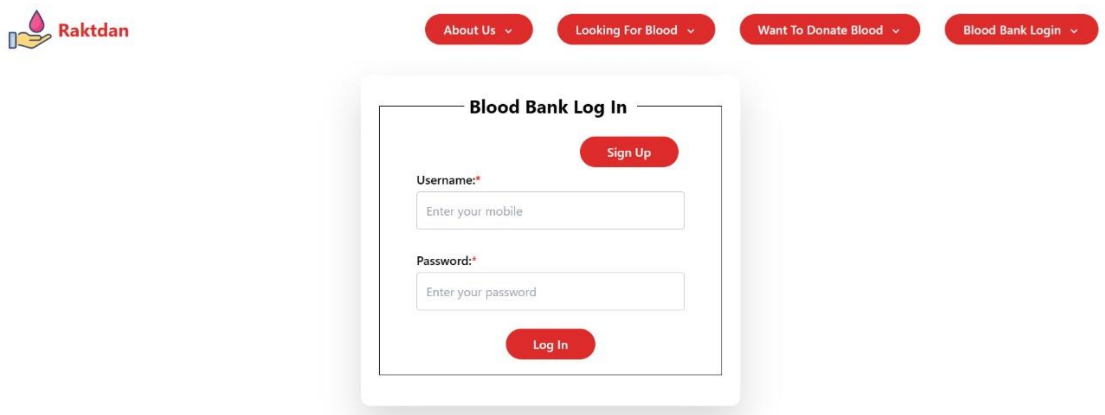
Interface design is involved in a wide range of projects from computer systems, to cars, to commercial planes; all of these projects involve much of the same basic human interactions yet also require some unique skills and knowledge. As a result, designers tend to specialize in certain types of projects and have skills centered on their expertise, whether that be software design, user research, web design, or industrial design.

SNAPSHOTS

Home page



Blood Bank login page



Blood Bank login code:

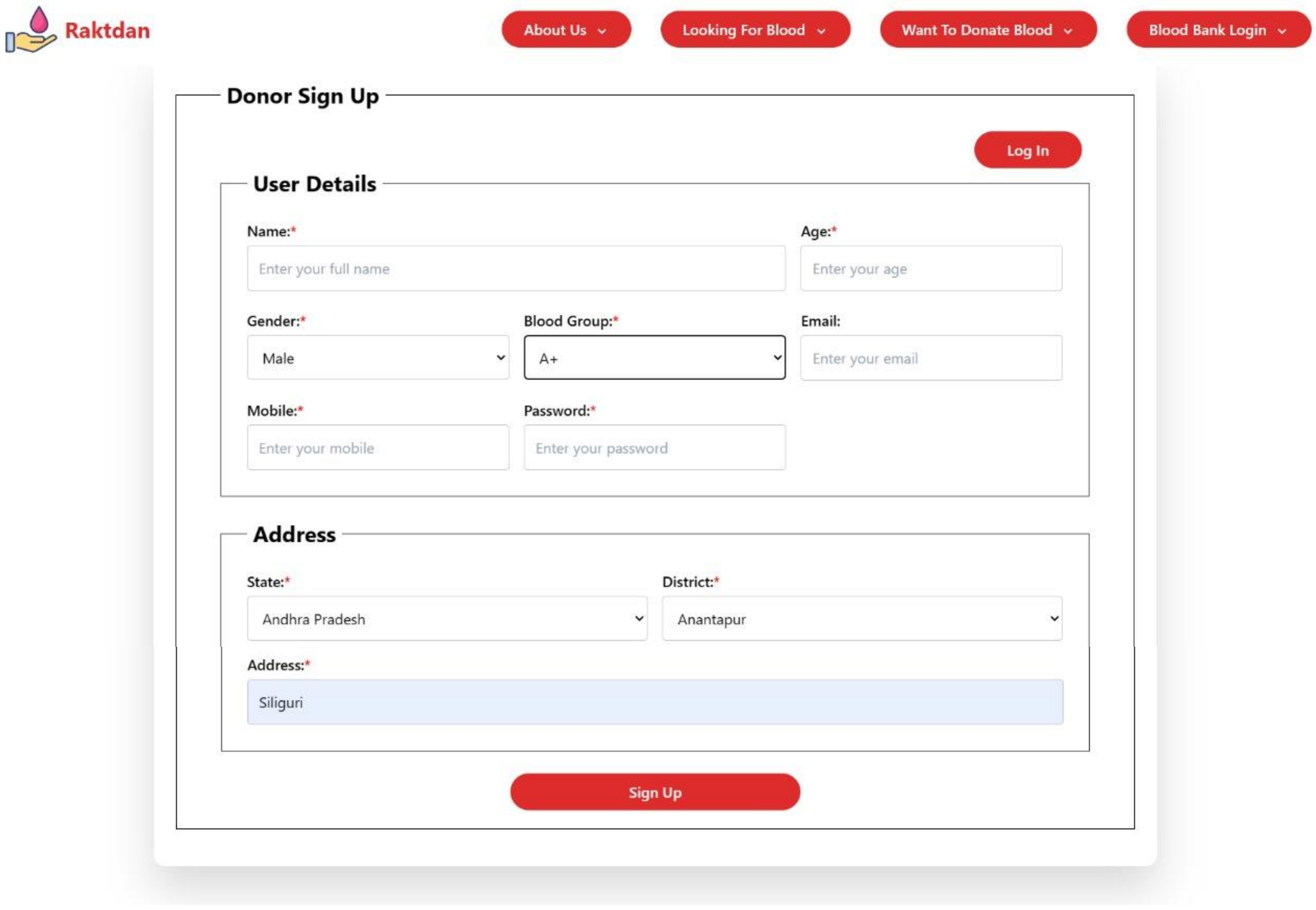
```
      import React,{useState} from 'react'
import axios from 'axios';
import {useHistory} from "react-router-dom"
const Login = ({setLoginUser}) => {
const history = useHistory()
   const [user,setUser] = useState({
      name:"",
      password: ""
   })
   const handleChange = e =>{
   const {name,value} = e.target
   setUser({
   ...user,//spread operator
   [name]:value
   })
   }

   const login =()=>{
      axios.post("http://localhost:6969/Login",user)
      .then(res=>{alert(res.data.message)
      setLoginUser(res.data.user)
   history.push("/")})
   }
   return (
      <>
<div class="flex flex-col w-full max-w-md px-4 py-8 bg-white rounded-lg shadow dark:bg-gray-800 sm:px-6 md:px-8 lg:px-10">
   <div class="self-center mb-6 text-xl font-light text-gray-600 sm:text-2xl
```

```
dark:text-white">
        Login To Your Account
    </div>
    <div class="mt-8">
        <form action="#" autoComplete="off">
            <div class="flex flex-col mb-2">
                <div class="flex relative ">
                    <span class="rounded-l-md inline-flex  items-center px-3 border-t bg-
white border-l border-b  border-gray-300 text-gray-500 shadow-sm text-sm">
                        <svg width="15" height="15" fill="currentColor" viewBox="0 0
1792 1792" xmlns="http://www.w3.org/2000/svg">
                            <path d="M1792 710v794q0 66-47 113t-113 47h-1472q-66 0-
113-47t-47-113v-794q44 49 101 87 362 246 497 345 57 42 92.5 65.5t94.5 48
110 24.5h2q51 0 110-24.5t94.5-48 92.5-65.5q170-123 498-345 57-39 100-
87zm0-294q0 79-49 151t-122 123q-376 261-468 325-10 7-42.5 30.5t-54 38-52
32.5-57.5 27-50 9h-2q-23 0-50-9t-57.5-27-52-32.5-54-38-42.5-30.5q-91-64-262-
182.5t-205-142.5q-62-42-117-115.5t-55-136.5q0-78 41.5-130t118.5-
52h1472q65 0 112.5 47t47.5 113z">
                            </path>
                        </svg>
                    </span>
                    <input type="text" id="sign-in-email" class=" rounded-r-lg flex-1
appearance-none border border-gray-300 w-full py-2 px-4 bg-white text-gray-
700 placeholder-gray-400 shadow-sm text-base focus:outline-none focus:ring-2
focus:ring-purple-600 focus:border-transparent" name="email"
value={user.email}  onChange={handleChange} placeholder="Your email"/>
                    </div>
                </div>
                <div class="flex flex-col mb-6">
```

```
          <div class="flex relative ">
            <span class="rounded-l-md inline-flex  items-center px-3 border-t
bg-white border-l border-b  border-gray-300 text-gray-500 shadow-sm text-
sm">
              <svg width="15" height="15" fill="currentColor" viewBox="0 0
1792 1792" xmlns="http://www.w3.org/2000/svg">
                <path d="M1376 768q40 0 68 28t28 68v576q0 40-28 68t-68
28h-960q-40 0-68-28t-28-68v-576q0-40 28-68t68-28h32v-320q0-185 131.5-
316.5t316.5-131.5 316.5 131.5 131.5 316.5q0 26-19 45t-45 19h-64q-26 0-45-
19t-19-45q0-106-75-181t-181-75-181 75-75 181v320h736z">
                </path>
              </svg>
            </span>
            <input type="password" id="sign-in-email" class=" rounded-r-lg
flex-1 appearance-none border border-gray-300 w-full py-2 px-4 bg-white text-
gray-700 placeholder-gray-400 shadow-sm text-base focus:outline-none
focus:ring-2 focus:ring-purple-600 focus:border-transparent" name="password"
value={user.password}  onChange={handleChange} placeholder="Your
password"/>
            </div>
          </div>
          <div class="flex items-center mb-6 -mt-4">
            <div class="flex ml-auto">
              <a href="#" class="inline-flex text-xs font-thin text-gray-500
sm:text-sm dark:text-gray-100 hover:text-gray-700 dark:hover:text-white">
                Forgot Your Password?
              </a>
            </div>
          </div>
```

```
              <div class="flex w-full">
                <button type="submit" class="py-2 px-4  bg-purple-600 hover:bg-
purple-700 focus:ring-purple-500 focus:ring-offset-purple-200 text-white w-full
transition ease-in duration-200 text-center text-base font-semibold shadow-md
focus:outline-none focus:ring-2 focus:ring-offset-2  rounded-lg  "
onClick={login}>
                  Login
                </button>
              </div>
            </form>
          </div>
          <div class="flex items-center justify-center mt-6">
            <a href="#" target="_blank" class="inline-flex items-center text-xs font-
thin text-center text-gray-500 hover:text-gray-700 dark:text-gray-100
dark:hover:text-white"  onClick={history.push("/Register")}>
              <span class="ml-2">
                You don&#x27;t have an account?
              </span>
            </a>
          </div>
        </div>

        </>
    )
}
export default Login
```

Donor Registration page:



Donor Sign Up

User Details

Name:* Enter your full name

Age:* Enter your age

Gender:* Male

Blood Group:* A+

Email: Enter your email

Mobile:* Enter your mobile

Password:* Enter your password

Address

State:* Andhra Pradesh

District:* Anantapur

Address:* Siliguri

Donor Registration page code:

```
        import React , {useState} from 'react'
import axios from "axios";
const Register = () => {
   const [user,setUser] = useState({
      name:"",
      email:"",
      password: ""
   })
   const handleChange = e =>{
   const {name,value} = e.target
   setUser({
   ...user,//spread operator
   [name]:value
   })
   }
//register function
  const egister = ()=>{
  const {name,email,password} = user
  if (name && email && password){
   axios.post("http://localhost:6969/Register",user )
   .then(res=>console.log(res))
  }
  else{
     alert("invalid input")
  };
   return (
      <>
<div class="flex flex-col max-w-md px-4 py-8 bg-white rounded-lg shadow dark:bg-gray-800 sm:px-6
md:px-8 lg:px-10">
   <div class="self-center mb-2 text-xl font-light text-gray-800 sm:text-2xl dark:text-white">
      Create a new account
   </div>
   <span class="justify-center text-sm text-center text-gray-500 flex-items-center dark:text-gray-400">
      Already have an account ?
      <a href="#" target="_blank" class="text-sm text-blue-500 underline hover:text-blue-700">
         Sign in
        </a>
    </span>
    <div class="p-6 mt-8">
        <form action="#">
            <div class="flex flex-col mb-2">
                <div class=" relative ">
                    <input type="text" id="create-account-pseudo" class=" rounded-lg border-transparent flex-
1 appearance-none border border-gray-300 w-full py-2 px-4 bg-white text-gray-700 placeholder-gray-
400 shadow-sm text-base focus:outline-none focus:ring-2 focus:ring-purple-600 focus:border-
transparent" name="name" value={user.name} onChange={handleChange} placeholder="FullName"/>
                    </div>
                </div>
                <div class="flex gap-4 mb-2">
                    <div class=" relative ">
                        <input type="text" id="create-account-first-name" class=" rounded-lg border-transparent
flex-1 appearance-none border border-gray-300 w-full py-2 px-4 bg-white text-gray-700 placeholder-
gray-400 shadow-sm text-base focus:outline-none focus:ring-2 focus:ring-purple-600 focus:border-
transparent" name="email" value={user.email} onChange={handleChange} placeholder="Email"/>
                        </div>

                        </div>
                        <div class="flex flex-col mb-2">
                            <div class=" relative ">
                                <input type="password" id="create-account-email" class=" rounded-lg border-
transparent flex-1 appearance-none border border-gray-300 w-full py-2 px-4 bg-white text-gray-700
placeholder-gray-400 shadow-sm text-base focus:outline-none focus:ring-2 focus:ring-purple-600
focus:border-transparent" name="password" value={user.password} onChange={handleChange}
placeholder="password"/>
                                </div>
                            </div>
                            <div class="flex w-full my-4">
                                <button type="submit" class="py-2 px-4  bg-purple-600 hover:bg-purple-700
focus:ring-purple-500 focus:ring-offset-purple-200 text-white w-full transition ease-in duration-200
text-center text-base font-semibold shadow-md focus:outline-none focus:ring-2 focus:ring-offset-2
rounded-lg " onClick={egister} >
                                    Register
                                </button>                       </div>
                        </form>
```

```
                                                       </div>
                                                   </div>

        </>
    )
}
}
export default Register
```

……………………………………

Validation code

App.js :

```
import { BrowserRouter, Routes, Route } from "react-router-dom";
import React, { useContext } from 'react';
import AuthContext from "./Components/context/AuthContext";
import Home from "./Components/Main/Home";
import Navbar from "./Components/Main/Navbar";
import About from "./Components/Main/About";
import Camps from "./Components/Main/Camps";
import Contact from "./Components/Main/Contact";
import Banks from "./Components/Main/Banks"
import AboutDonation from "./Components/Main/AboutDonation";
import Auth from "./Components/Auth/Auth";
import User from "./Components/User/User";
import Bank from "./Components/Bank/Bank";
import "./App.css";

export default function App() {
	const { loggedIn, user } = useContext(AuthContext);
	return (
		<>
			<BrowserRouter>
				<Routes>
					<Route path="/" element={<Navbar logIn={loggedIn} user={loggedIn &&
user.latitude ? "bank" : "user"} />}>
						<Route index element={<Home />} />
						{!loggedIn && <>
							<Route path="/:type/:handle" element={<Auth logIn={loggedIn} />}
/>
						</>}
						{loggedIn && (user.hospital ?
							<Route path="/bank/:handle?" element={<div><Bank /></div>} /> :
							<Route path="/user/:handle?" element={<div><User /></div>} />)
						}
						<Route path="about" element={<About />} />
						<Route path="aboutBloodDonation" element={<AboutDonation />} />
						<Route path="bloodDirect" element={<Banks />} />
						<Route path="bloodCamps" element={<Camps />} />
						<Route path="contactUs" element={<Contact />} />
```

```
                                    <Route path="*" element={<div>404</div>} />
                              </Route>
                        </Routes>
                  </BrowserRouter>
            </>
      );
}
```

Backend Using Node.js and React.js

Source Code:

authrouter.js :

```
const router = require("express").Router();
const { User, BloodBank } = require("../models/models");
const bcrypt = require("bcrypt");
const jwt = require("jsonwebtoken");

// register

router.post("/:handle", async (req, res) => {
  try {
    // validation
    const handle = req.params.handle;
    const existingUser = handle == "bank" ?
      await BloodBank.findOne({ phone: req.body.phone }) :
      await User.findOne({ phone: req.body.phone });
    if (existingUser)
      return res.status(400).json({
        errorMessage: "An account with this email already exists.",
      });

    // hash the password

    const salt = await bcrypt.genSalt();
    const passwordHash = await bcrypt.hash(req.body.password, salt);
    req.body.password = passwordHash;
    // save a new user account to the db
    const newUser = handle == "bank" ? new BloodBank(req.body) : new User(req.body);
    const savedUser = await newUser.save();

    // sign the token
    const token = jwt.sign({ user: savedUser._id, type: handle }, process.env.JWT_SECRET);

    // send the token in a HTTP-only cookie

    res.cookie("token", token, {
      httpOnly: true,
      secure: true,
      sameSite: "none",
    }).send();
  } catch (err) {
    console.error(err);
    res.status(500).send();
  }
});

// log in

router.post("/login/:handle", async (req, res) => {
  try {
    const { phone, password } = req.body;
    const handle = req.params.handle;
    const existingUser = await (handle == "bank" ? BloodBank.findOne({ phone: phone }) :
User.findOne({ phone: phone }));
    if (!existingUser)
      return res.status(401).json({ errorMessage: "Wrong username or password." });
    const passwordCorrect = await bcrypt.compare(
      password,
      existingUser.password
    );
    if (!passwordCorrect)
      return res.status(401).json({ errorMessage: "Wrong username or password." });

    // sign the token
```

```
      const token = jwt.sign(
        {
          user: existingUser._id,
          type: handle
        },
        process.env.JWT_SECRET
      );

      // send the token in a HTTP-only cookie

      res
        .cookie("token", token, {
          httpOnly: true,
          secure: true,
          sameSite: "none",
        })
        .send();
    } catch (err) {
      console.error(err);
      res.status(500).send();
    }
});

router.get("/logout", (req, res) => {
    res
      .cookie("token", "", {
        httpOnly: true,
        secure: true,
        sameSite: "none",
      })
      .send();
    console.log("Logged Out")
});

router.get("/loggedIn", async (req, res) => {
    try {
      const token = req.cookies.token;
      if (!token) return res.json({ auth: false });
```

```
        const verified = jwt.verify(token, process.env.JWT_SECRET);
        const user = await (verified.type == "bank" ? BloodBank : User).findOne({ _id: verified.user }, {
password: 0, donations: 0, requests: 0, stock: 0, __v: 0 });
        console.log("logged in")
        res.send({ auth: true, user: user });
    } catch (err) {
        console.log(err);
        res.json({ auth: false });
    }
});

module.exports = router;
```

bankRouter.js :

```
const router = require("express").Router();
const auth = require("../middleware/auth");
const { User, BloodBank, Donations, Requests, Camp } = require("../models/models");

router.post("/:handle", auth, async (req, res) => {
    try {
        const filter = req.params.handle === "bank" ? {} : { password: 0, requests: 0, donations: 0, stock: 0,
__v: 0 };
        const banks = await BloodBank.find(req.body, filter);
        res.json(banks);
    } catch (err) {
        console.error(err);
        res.status(500).send();
    }
});

router.get("/allBanks/:state/:district", async (req, res) => {
    try {
        const banks = await BloodBank.find({ state: req.params.state, district: req.params.district }, {
password: 0, _id: 0, donations: 0, requests: 0, stock: 0 });
        res.json(banks);
    } catch (err) {
        console.error(err);
        res.status(500).send();
```

```
  }
});

router.put("/updateStock", auth, async (req, res) => {
  try {
    const prevStock = await BloodBank.findOne({ _id: req.user }, { stock: 1 });
    await BloodBank.updateOne(
      { _id: req.user },
      { $set: { ["stock." + req.body.bloodGroup]: prevStock.stock[req.body.bloodGroup] +
req.body.units } }
    )
    res.status(200).send();
  } catch (err) {
    console.error(err);
    res.status(500).send();
  }
});

router.put("/deleteStock", auth, async (req, res) => {
  try {
    const prevStock = await BloodBank.findOne({ _id: req.user }, { stock: 1 });
    if (prevStock.stock[req.body.bloodGroup] < req.body.units) {
      res.status(404).send("Not enough blood");
    } else {
      await BloodBank.updateOne(
        { _id: req.user },
        { $set: { ["stock." + req.body.bloodGroup]: prevStock.stock[req.body.bloodGroup] -
req.body.units } }
      )
      res.status(200).send();
    }
  } catch (err) {
    console.error(err);
    res.status(500).send();
  }
});

router.get("/getStock", auth, async (req, res) => {
  try {
    const data = await BloodBank.findOne(
      { _id: req.user },
      { _id: 0, stock: 1 }
    )
    res.status(200).send(data);
  } catch (err) {
    console.error(err);
    res.status(500).send();
  }
});

router.put("/donations", auth, async (req, res) => {
  try {
    Donations.updateOne({ _id: req.body.id }, { status: req.body.status }, (err, user) => {
      if (err) {
        res.status(404).send("Donation not found");
      } else {
        res.status(200).send("Status updated");
      }
    });
  } catch (err) {
    console.error(err);
    res.status(500).send();
  }
});

router.put("/requests", auth, async (req, res) => {
  try {
    Requests.updateOne({ _id: req.body.id }, { status: req.body.status }, (err, user) => {
      if (err) {
        res.status(404).send("Request not found");
      } else {
        res.status(200).send("Status updated");
      }
    });
  } catch (err) {
    console.error(err);
```

```
        res.status(500).send();
    }
});

router.get("/donations", auth, async (req, res) => {
    try {
        const data = await Donations.find({ bankId: req.user }).populate('userId', '-__v -password -requests
-donations -stock');
        res.json(data);
    } catch (err) {
        console.error(err);
        res.status(500).send();
    }
});

router.get("/requests", auth, async (req, res) => {
    try {
        const data = await Requests.find({ bankId: req.user }).populate('userId', '-__v -password -requests -
donations -stock');
        res.json(data);
    } catch (err) {
        console.error(err);
        res.status(500).send();
    }
});

router.put("/", auth, async (req, res) => {
    try {
        console.log(req.user);
        BloodBank.updateOne({ _id: req.user }, req.body, (err, user) => {
            if (err) {
                res.status(404).send("BloodBank not found");
            } else {
                res.status(200).send("BloodBank updated");
            }
        });
    } catch (err) {
        console.error(err);
        res.status(500).send();
    }
});

module.exports = router;
```

campRouter.js :

```
const router = require("express").Router();
const auth = require("../middleware/auth");
const { Camp } = require("../models/models");

router.post("/", auth, async (req, res) => {
    try {
        req.body.bankId = req.user;
        req.body.donors = [];
        const newCamp = new Camp(req.body);
        await newCamp.save();
        res.status(200).send();
    } catch (err) {
        console.error(err);
        res.status(500).send();
    }
});

router.get("/:state?/:district?", auth, async (req, res) => {
    try {
        let query = {};
        if (req.params.state) {
            query.state = req.params.state;
            query.district = req.params.district;
        } else {
            query.bankId = req.user;
        }
        const data = await Camp.find(query).populate('bankId', '-_id -__v -password -requests -donations -
stock').populate({
            path: "donors._id",
            select: '-__v -password'
        });
    res.json(data);
  } catch (err) {
    console.error(err);
    res.status(500).send();
  }
});

router.get("/allCamps/:state/:district/:date", async (req, res) => {
  try {
    if (req.params.date) {
      const data = await Camp.find({
        state: req.params.state,
        district: req.params.district,
        date: new Date(req.params.date)
      }, { donors: 0, _id: 0 }).populate("bankId","-_id -password -donations -requests -stock +name");
      res.json(data);
    } else{
      res.json({});
    }
  } catch (err) {
    console.error(err);
    res.status(500).send();
  }
});

router.put("/:id/:userId?", auth, async (req, res) => {
  try {
    if (req.params.userId) {
      await Camp.update(
        {
          _id: req.params.id,
          donors: { $elemMatch: { _id: req.params.userId, status: 0 } }
        },
        { $set: { "donors.$.units": req.body.units, "donors.$.status": 1 } }
      )
    } else {
      if (await Camp.find({
        _id: req.params.id,
        donors: { $elemMatch: { _id: req.user } }
      }) != []) {
        await Camp.updateOne(
          { _id: req.params.id },
          { $push: { donors: { _id: req.user } } }
        );
      }
    }
    res.status(200).send();
  } catch (err) {
    console.error(err);
    res.status(500).send();
  }
})

module.exports = router;
```

App.js

```
const express = require('express');
const cors = require("cors");
const dotenv = require("dotenv");
const cookieParser = require("cookie-parser");
const connectDB = require('./config/db');
import ('body-parser');

const app = express();
const port = 3177;

dotenv.config();


connectDB();




app.use(cookieParser());
app.use(express.json());
app.use(
        cors({
                origin: [
                        "http://localhost:3000",
                ],
                credentials: true,
        })
);

app.use("/auth", require("./routers/authRouter"));
app.use("/user", require("./routers/userRouter"));
app.use("/bank", require("./routers/bankRouter"));
app.use("/camps", require("./routers/campRouter"));

app.listen(port, () =>
        console.log(`Server running at http://localhost:${port}`)
);
```

userRouter.js :

```
const router = require("express").Router();
const auth = require("../middleware/auth");
const { User, Donations, Requests, BloodBank } = require("../models/models");

router.get("/", auth, async (req, res) => {
   try {
      console.log("hum yha hain")
      const user = await User.find({ _id: req.user });
      console.log(user);
      res.json(user);
   } catch (err) {
      console.error(err);
      res.status(500).send();
   }
});

router.post("/donate", auth, async (req, res) => {
   try {
      req.body.userId = req.user;
```

```
        const date = new Date();
        req.body.date = date.toLocaleTimeString() + " " + date.toLocaleDateString();
        const newDonation = new Donations(req.body);
        const saved = await newDonation.save();
        await BloodBank.update(
            { _id: req.body.bankId },
            { $push: { donations: { _id: saved._id } } }
        );
        res.send("done")
    } catch (err) {
        console.error(err);
        res.status(500).send();
    }
});

router.post("/request", auth, async (req, res) => {
    try {
        req.body.userId = req.user;
        const date = new Date();
        req.body.date = date.toLocaleTimeString() + " " + date.toLocaleDateString();
        const newRequest = new Requests(req.body);
        const saved = await newRequest.save();
        await BloodBank.update(
            { _id: req.body.bankId },
            { $push: { requests: { _id: saved._id } } }
        );
        res.send("done")
    } catch (err) {
        console.error(err);
        res.status(500).send();
    }
})

router.get("/donations", auth, async (req, res) => {
    try {
        const data = await Donations.find({ userId: req.user }).populate('bankId', '-_id -__v -password -
requests -donations -stock');
        res.json(data);
```

```
  } catch (err) {
    console.error(err);
    res.status(500).send();
  }
});

router.get("/requests", auth, async (req, res) => {
  try {
    const data = await Requests.find({ userId: req.user }).populate('bankId', '-_id -__v -password -
requests -donations -stock');
    res.json(data);
  } catch (err) {
    console.error(err);
    res.status(500).send();
  }
});

router.put("/", auth, async (req, res) => {
  try {
    console.log(req.user);
    User.updateOne({ _id: req.user }, req.body, (err, user) => {
      if (err) {
        res.send(404, "User not found");
      } else {
        res.send(200, "User updated");
      }
    });
  } catch (err) {
    console.error(err);
    res.status(500).send();
  }
});

module.exports = router;
```

IMPLEMENTATION AND TESTING

A software system test plan is a document that describes the objectives, scope, approach and focus of software testing effort. The process of preparing a test plan is a usual way to think the efforts needed to validate the acceptability of a software product. The complete document will help people outside the test group understand the "WHY" and "HOW" product validation. It should be through enough to be useful but not so through that no one outside the test group will read it.

6a. INTRODUCTION

Testing is the process of running a system with the intention of finding errors. Testing enhances the integrity of a system by detecting deviations in design and errors in the system. Testing aims at detecting error-prone areas. This helps in the prevention of errors in a system. Testing also adds value to the product by conforming to the user requirements.

The main purpose of testing is to detect errors and error-prone areas in a system. Testing must be thorough and well-planned. A partially tested system is as bad as an untested system. And the price of an untested and under-tested system is high.

The implementation is the final and important phase. It involves user-training, system testing in order to ensure successful running of the proposed system. The user tests the system and changes are made according to their needs. The testing involves the testing of the developed system using various kinds of data. While testing, errors are noted and correctness is the mode.

6b. OBJECTIVES OF TESTING:

The objective our test plan is to find and report as many bugs as possible to improve the integrity of our program. Although exhaustive testing is not possible, we will exercise a broad range of tests to achieve our goal. Our user interface to utilize these functions is designed to be user-friendly and provide easy manipulation of the tree. The application will only be used as a demonstration tool, but we would like to ensure that it could be run from a variety of platforms with little impact on performance or usability.

Process Overview

The following represents the overall flow of the testing process:

1. Identify the requirements to be tested. All test cases shall be derived using the current Program Specification.

2. Identify which particular test(s) will be used to test each module.

3. Review the test data and test cases to ensure that the unit has been thoroughly verified and that the test data and test cases are adequate to verify proper operation of the unit.

4. Identify the expected results for each test.

5. Document the test case configuration, test data, and expected results.

6. Perform the test(s).

7. Document the test data, test cases, and test configuration used during the testing process. This information shall be submitted via the Unit/System Test Report (STR).

8. Successful unit testing is required before the unit is eligible for component integration/system testing.

9. Unsuccessful testing requires a Bug Report Form to be generated. This document shall describe the test case, the problem encountered, its possible cause, and the sequence of events that led to the problem. It shall be used as a basis for later technical analysis.

10. Test documents and reports shall be submitted. Any specifications to be reviewed, revised, or updated shall be handled immediately.

6c. TEST CASES

A test case is a document that describe an input, action, or event and expected response, to determine if a feature of an application is working correctly. A test case should contain particular such as test case identifier, test condition, input data
Requirement expected results. The process of developing test cases can help find problems in the requirement or design of an application, since it requires completely thinking through the operation of the application.

TESTING STEPS

Unit Testing:

Unit testing focuses efforts on the smallest unit of software design. This is known as module testing. The modules are tested separately. The test is carried out during programming stage itself. In this step, each module is found to be working satisfactory as regards to the expected output from the module.

Integration Testing:

Data can be lost across an interface. One module can have an adverse effect on another, sub functions, when combined, may not be linked in desired manner in major functions. Integration testing is a systematic approach for constructing the program structure, while at the same time conducting test to uncover errors associated within the interface. The objective is to take unit tested modules and builds program structure. All the modules are combined and tested as a whole.

Validation:

At the culmination of the integration testing, Software is completely assembled as a package. Interfacing errors have been uncovered and corrected and a final series of software test begin in validation testing. Validation testing can be defined in many ways, but a simple definition is that the validation succeeds when the software functions in a manner that is expected by the customer. After validation test has been conducted, one of the three possible conditions exists.

a) The function or performance characteristics confirm to specification and are accepted.

b) A deviation from specification is uncovered and a deficiency lists is created.

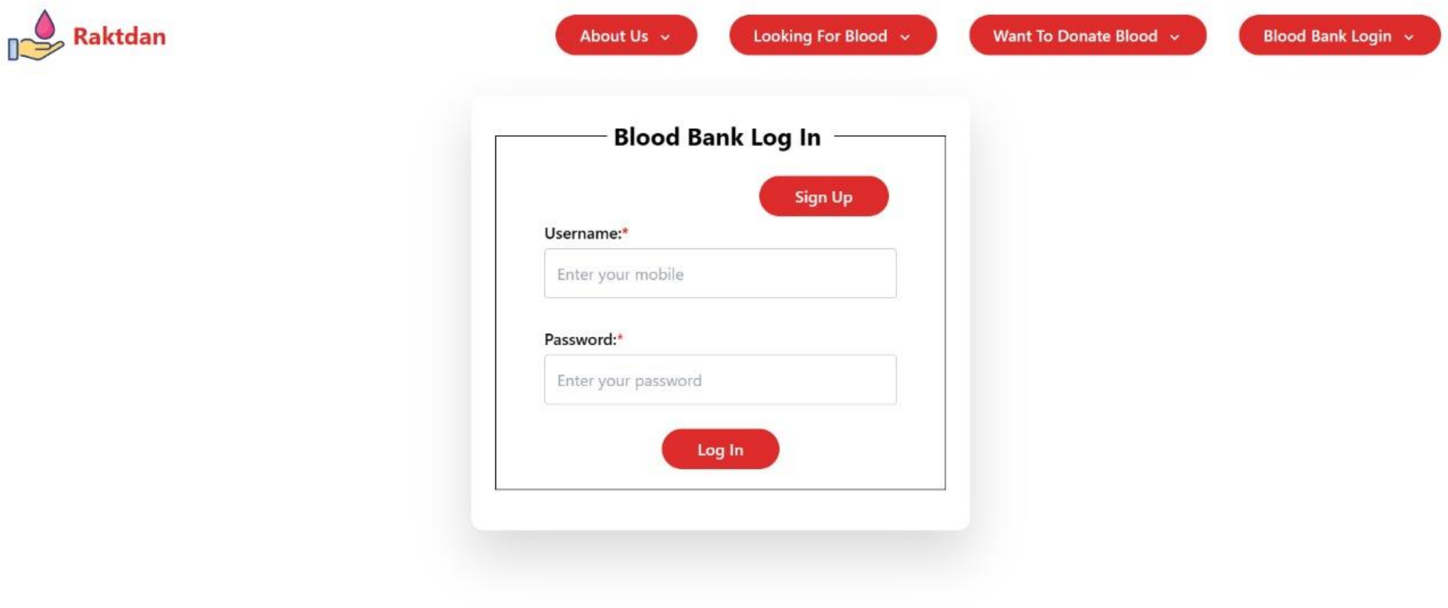
c) Proposed system under consideration has been tested by using validation test and found to be working satisfactory.

Blood Donor Eligibility Requirements

This table outlines the basic requirements for blood donation:

Requirement	Description
Age	Minimum age to donate varies depending on location. Generally between 18-75 years old.
Weight	Minimum weight requirement to ensure sufficient blood volume donation (varies by location, typically around 50 kg or 110 lbs).
Identification	Valid government-issued photo ID is required to verify identity.
General Health	Feel well overall, free from any symptoms of illness like fever, cold, or flu.
Blood History	No recent history of blood transfusions (waiting period applies).
Medications	Certain medications might temporarily disqualify you. Discuss with staff if you take any medications.
Medical Conditions	Some pre-existing medical conditions might be disqualifying. Be honest about your health history during screening.
Lifestyle	Low-risk lifestyle regarding activities that could increase infection risk.
Blood Type and Rh Factor	All blood types are needed, but some are in higher demand depending on local needs.

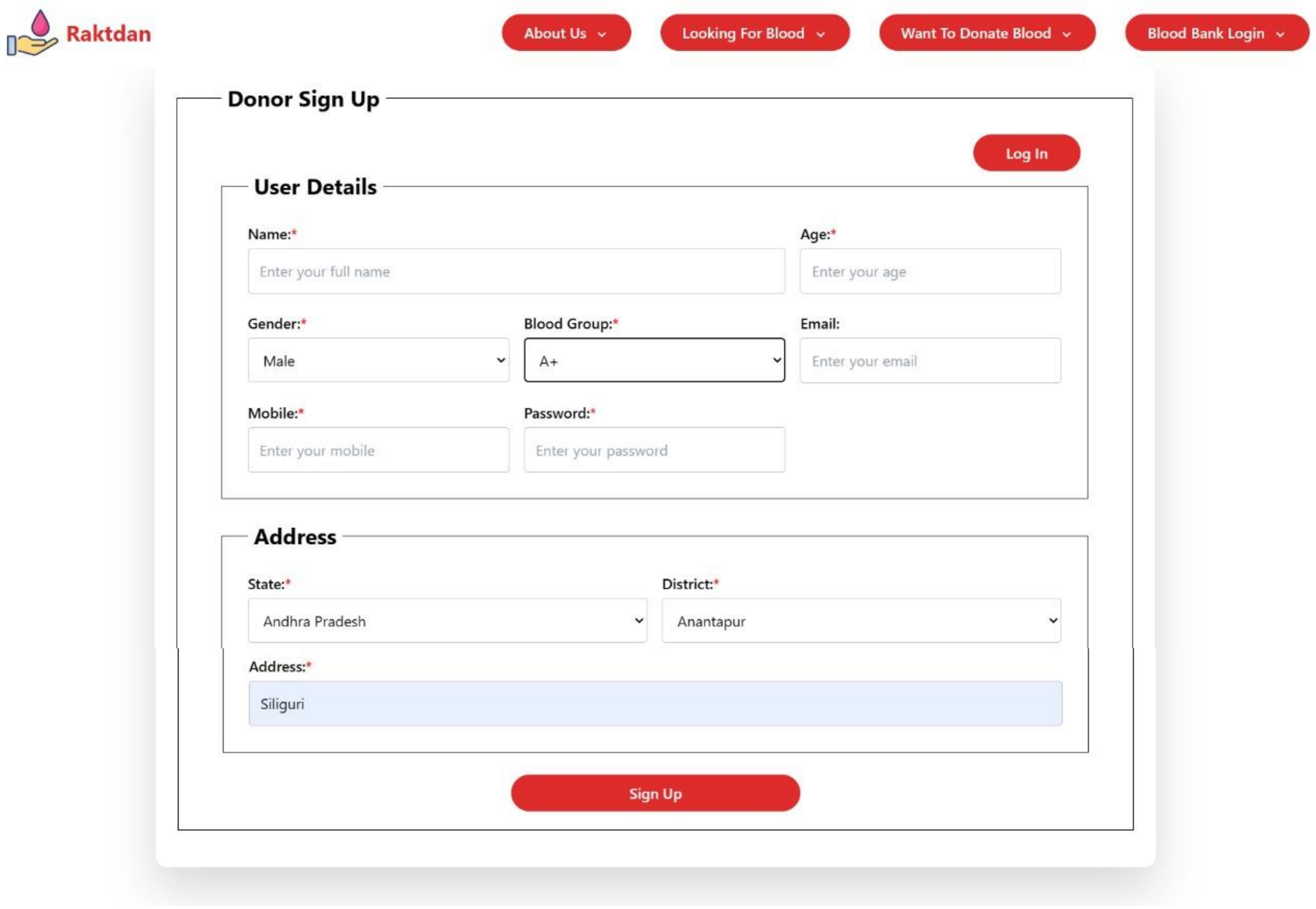
SNAPSHOT OF DONOR LOGIN



Patient Information for Treatment Decisions

Field Name	Description	Mandatory
Patient Demographics		
Patient ID	Unique identifier assigned to the patient	Yes
Last Name	Patient's last name	Yes
First Name	Patient's first name	Yes
Date of Birth	Patient's date of birth (DD/MM/YYYY)	Yes
Medical Information		
Blood Type	ABO and Rh factor	Yes
Current Medical Condition	Primary reason for seeking treatment	Yes
Medical History (Summary)	Brief overview of past medical conditions and surgeries	Yes
Allergies	List of any allergies the patient has	Yes
Medications (Active)	List of medications the patient is currently taking	Yes
Additional Information Relevant to Treatment		
Recent Test Results	Lab test results, X-rays, or other relevant diagnostic reports	Yes (if applicable)
Symptoms	Description of the patient's current symptoms	Yes
Family Medical History (if relevant)	History of relevant medical conditions in the patient's family	Yes (if applicable)
Treatment Considerations		
Previous Treatments (if applicable)	Any prior treatments for the current or similar conditions	Yes (if applicable)
Lifestyle Factors (if applicable)	Smoking, alcohol use, dietary habits, etc. that might impact treatment	Yes (if applicable)

SNAPSHOTS OF DONOR REGISTRATION



Donor Sign Up

User Details

Name:* Enter your full name		Age:* Enter your age
Gender:* Male	Blood Group:* A+	Email: Enter your email
Mobile:* Enter your mobile	Password:* Enter your password	

Address

State:* Andhra Pradesh

District:* Anantapur

Address:* Siliguri

6 d. WHITE BOX TESTING

In white box testing, the UI is bypassed. Inputs and outputs are tested directly at the code level and the results are compared against specifications. This form of testing ignores the function of the program under test and will focus only on its code and the structure of that code. Test case designers shall generate cases that not only cause each condition to take on all possible values at least once, but that cause each such condition to be executed at least once. To ensure this happens, we will be applying Branch Testing. Because the functionality of the program is relatively simple, this method will be feasible to apply.

Each function of the binary tree repository is executed independently; therefore, a program flow for each function has been derived from the code.

6e. BLACK BOX TESTING

Black box testing typically involves running through every possible input to verify that it results in the right outputs using the software as an end-user would. We have decided to perform Equivalence Partitioning and Boundary Value Analysis testing on our application.

System Testing

The goals of system testing are to detect faults that can only be exposed by testing the entire integrated system or some major part of it. Generally, system testing is mainly concerned with areas such as performance, security, validation, load/stress, and configuration sensitivity. But in our case well focus only on function validation and performance. And in both cases we will use the black-box method of testing.

6f. OUTPUT TESTING

The output of testing a blood bank management system built with MERN stack will take various forms depending on the specific test type. Here's a breakdown of what you might expect:

Functional Testing:

- **Pass/Fail Reports:** Automated tests should generate reports indicating which functionalities passed and which failed. These reports detail error messages or unexpected behavior for failed tests.
- **Manual Test Cases:** Documented test cases with manual execution will have success or failure notations along with comments or screenshots capturing the observed behavior.

Performance and Scalability Testing:

- **Performance Metrics:** Reports showing response times, throughput (requests processed per unit time), and resource usage (CPU, memory) under varying loads. These metrics help identify performance bottlenecks.
- **Scalability Analysis:** Documentation outlining how the system handles increased data volume and user base. This might include charts or graphs depicting performance changes with increasing load.

Security Testing:

- **Vulnerability Reports:** Security scanners will produce reports listing identified vulnerabilities along with severity ratings and recommendations for remediation.

- **Penetration Testing Reports:** Penetration testers will provide detailed reports outlining the attempted attacks, their success or failure, and recommendations for fixing any exploited weaknesses.

Usability and User Experience:

- **User Feedback:** Gathered through surveys, interviews, or usability testing sessions. This feedback can be qualitative (descriptive comments) or quantitative (ratings on ease of use, satisfaction).
- **Accessibility Reports:** Automated accessibility testing tools will generate reports highlighting areas where the application might not meet WCAG standards, along with suggestions for improvement.

6g. GOAL OF TESTING

The testing of a blood bank management system built with the MERN stack targets several goals, all aiming to ensure a smooth, reliable, and potentially life-saving application. Here's a breakdown of key testing objectives:

Functionality:

- **Core Features:** Verify core functionalities like donor registration, blood type matching, appointment scheduling, inventory management, and blood request processing work flawlessly.
- **User Roles:** Ensure different user roles (donors, administrators, medical staff) have the appropriate access and can perform intended actions within the system.
- **Data Integrity:** Test data accuracy and completeness throughout the system, including blood unit information, donor details, and medical records (with appropriate anonymization).

Performance and Scalability:

- **Load Testing:** Simulate real-world usage scenarios with high numbers of concurrent users to assess system responsiveness and stability under load.

- **Performance Optimization:** Identify bottlenecks and fine-tune the MERN stack components (MongoDB, Express.js, React.js, Node.js) for optimal performance.
- **Scalability Testing:** Evaluate how the system handles increased data volume and user base, ensuring it can grow efficiently to meet future demands.

Security:

- **Authentication and Authorization:** Test login processes, role-based access controls, and data encryption to ensure unauthorized access is prevented and sensitive information is protected.
- **Vulnerability Scanning:** Identify potential security weaknesses in the MERN stack or custom code to mitigate vulnerabilities before deployment.
- **Penetration Testing:** Simulate real-world attacks (ethical hacking) to assess the system's resilience against unauthorized intrusion attempts.

Usability and User Experience:

- **User Interface (UI) Testing:** Evaluate the user interface for clarity, ease of navigation, and intuitive workflows for all user roles.
- **User Acceptance Testing (UAT):** Involve potential users (donors, medical staff) to provide feedback on the system's usability and identify areas for improvement.

- **Accessibility Testing:** Ensure the system is accessible to users with disabilities, adhering to WCAG (Web Content Accessibility Guidelines) standards.

6h. INTEGRATION TEST REPORTS

Software testing is always used in association with verification and validation. In the testing phase of this project our aim is to find the answer to following two questions.

- Whether the software matches with the specification (i.e. process base) to verify the product.

- Whether this software in one client what wants (i.e. product base) to validate the product.

  Unit testing and integration testing has been carried out to find the answer to above questions. In unit testing each individual module was test to find any unexpected behaviour if exists. Later all the module was integrated and flat file was generated.

FUNCTIONAL TESTING

These are the points concerned during the stress test:

- Nominal input: character is in putted in the place of digits and the system has to flash the message "Data error"
- Boundary value analysis: exhaustive test cases have designed to create an output report that produces the maximum (and minimum) allowable number of table entries.

Testing Method Used

We have adopted a testing method which is a mix of both (structural) and black box (functional) testing. For modules we have adopted white box testing. Then we integrated the module into sub - systems and further into the system. These we adopted black box testing for checking the correctness of the system.

Requirements Validated and Verified:

- The data is getting entered properly into database.
- The Screens are being loaded correctly
- The Various functions specified are being performed completely.

SYSTEM SECURITY MEASURES

8.a DATABASE SECURITY

MongoDB offers a variety of features and best practices to secure your data. Here's a rundown of some key aspects:

Authentication and Authorization:

- **Separate Credentials:** Create unique logins for each user or process requiring MongoDB access.
- **Role-Based Access Control (RBAC):** Assign permissions based on roles (e.g., admin, developer) rather than individual users.

Data Encryption:

- **Encryption at Rest:** Encrypt data stored on disk using the WiredTiger storage engine or file-system level encryption.
- **Encryption in Transit:** Enforce TLS/SSL encryption for all communication between applications and MongoDB, including internally between mongod and mongos instances.
- **Client-Side Field Level Encryption:** Encrypt sensitive data fields within your application before sending them to MongoDB.

Auditing and Logging:

- Enable auditing to track user activity, including CRUD operations, authentication attempts, and role changes.
- Regularly review logs to identify suspicious activity.

Network Security:

- **Restrict Access:** Use IP whitelisting to limit connections to MongoDB from authorized IP addresses only.
- **Keep Software Updated:** Apply security patches promptly to address vulnerabilities in MongoDB and underlying systems.

8.b SYSTEM SECURITY

If we talk about the system security in our proposed system we have implemented with the help of maintain the session throughout the system's use. Once a user has logged out than he/she will not be able to perform any task before signing back again.

A high level of authentic login is given to the system so this is a very tedious task to enter without authorization and authentication.

Mitigating vulnerabilities across the stack:

- **Stay Updated:** Regularly update all libraries, frameworks, and Node.js itself to address known vulnerabilities. Outdated software is a prime target for attackers.
- **Input Validation and Sanitization:** Validate and sanitize all user inputs on the server-side to prevent attacks like SQL injection and Cross-Site Scripting (XSS). Don't trust user input!
- **Secure User Authentication:** Implement a robust user authentication system using secure password hashing (e.g., bcrypt) and avoid storing passwords in plain text. Consider using JSON Web Tokens (JWT) for session management.
- **Authorization:** Enforce proper authorization to ensure users can only access resources they are permitted to. Don't assume a user has access because they are logged in.

MongoDB Security:

- **Authentication and Authorization:** Use MongoDB's authentication framework with strong credentials and role-based access control (RBAC).

- **Data Encryption:** Encrypt data at rest (using MongoDB's encryption features) and in transit (using TLS/SSL). Sensitive data deserves an extra layer of protection.
- **Network Security:** Restrict access to the MongoDB database by limiting connections to authorized IP addresses. Don't leave your database wide open.

Node.js and Express.js Security:

- **Helmet:** Use the Helmet middleware to set security headers for your Express.js application, helping mitigate common web vulnerabilities.
- **Secure Coding Practices:** Follow secure coding practices to avoid common vulnerabilities in your Node.js code.

8c. LIMITATIONS:

- ✓ Since it is an online project, customers need internet connection to use it.
- ✓ People who are not familiar with computers can’t use this software.
- ✓ Customer must have debit card or credit card to book tickets.

9. CONCLUSION

This project has been appreciated by all the users in the organization. It is easy to use, since it uses the GUI provided in the user dialog. User friendly screens are provided. The usage of software increases the efficiency, decreases the effort. It has been efficiently employed as a Site management mechanism. It has been thoroughly tested and implemented.

10. FUTURE SCOPE AND FURTHER ENHANCEMENTS

The future of blood bank management systems (BBMS) is bright, with advancements in technology poised to further improve efficiency, safety, and accessibility in blood banking. Here are some exciting prospects to look forward to:

Advanced Blood Inventory Management:

Predictive Analytics: Machine learning algorithms can analyze blood usage patterns and predict potential shortages, allowing for proactive blood collection drives and targeted resource allocation.
Blockchain Technology: Blockchain can ensure secure and transparent blood tracking throughout the supply chain, from donation to transfusion.
Enhanced Donor Management and Recruitment:

Mobile Apps: Donor recruitment and communication can be streamlined through mobile apps with features like appointment scheduling, donation reminders, and loyalty programs.
Social Media Integration: Utilizing social media platforms for targeted donor outreach campaigns can expand the donor pool and raise awareness about blood donation.
Precision Blood Matching and Personalized Medicine:

Advanced Blood Typing: Emerging technologies might enable more precise blood typing, identifying additional compatibility factors beyond ABO and Rh systems for personalized blood transfusions.
Blood Component Management: Systems might manage blood components (red blood

cells, plasma, platelets) more precisely, allowing for targeted use based on specific patient needs.
Improved System Integration and Automation:

Interoperability Standards: Adoption of standardized data formats and APIs will facilitate seamless integration between BBMS and hospital information systems, improving workflow efficiency.
Robotic Process Automation: Repetitive tasks like blood labeling, sorting, and inventory management can be automated using robotics, freeing up staff time for higher-value activities.
Focus on Data-Driven Decision Making:

Real-time Blood Availability Data: Hospitals can access real-time data on blood availability across different blood banks, optimizing blood requisition processes and reducing wastage.
Data Analytics for Blood Utilization Trends: Analyzing blood usage data can help identify areas for improvement in blood product management and optimize resource allocation.
Telemedicine Integration:

11. BIBLIOGRAPHY

- https://www.w3schools.com
- https://www.freecodecamp.org/news/how-to-secure-your-mern-stack-application/
- https://www.npmjs.com/package/@react-google-maps/api
- https://www.tutorialspoint.com
- https://www.youtube.com

THANK
YOU