Blood Donation Management

1. Abstract

This project is aimed at developing an online Blood Donation Management System. Blood Donation Management System is a browser-based system that is designed to store, process, retrieve and manage information on blood donors as well as Blood Banks. It maintains all the information pertaining to blood donors, different blood groups available, and blood banks and helps them manage in a better way. The aim is to provide transparency in this field and make the process of donating blood to a blood bank hassle free which makes the system effective while also increasing the availability of donors for those in need.

Introduction

A blood donation is a process whereby a person voluntarily has blood drawn to be used for future transfusions when in need at hospitals for treatment procedures that require them. Donation may be of whole blood (blood drawn directly from the body) or of specific components of the blood; such as red blood cells, white blood cells, plasma, and platelets. Blood banks often participate in the process of collecting blood and other procedures such as conducting donation drives, approving blood requests and updating donation information and requirements.

The inspiration of this project is to help blood banks and to develop a blood bank information system which focuses on making an online system that is accessible for both donors and blood banks.

The purpose of the Blood Donation Management System project is to develop and implement a computerized system to manage the information of blood donors and to connect the blood banks to these donors. This system acts as a bridge between Donors and Blood Banks for easy, fast and efficient donation of blood. Donors can directly register themselves for blood donations, by providing the necessary information. They can also update the personal information through the system, without having to contact the blood bank registry or administrator. The administrator is also responsible for maintaining the records of registered blood banks and their blood requests.

The entire project has been developed keeping in view of the distributed client server computing technology, in mind. The Blood Donation Admin is to create an e-Information about the donor and organization or blood banks that are related to donating the blood. Through this application any person who is interested in donating the blood can register himself. But if any organization wants to register itself with this site it has to be verified and approved by Admin. Moreover, if any registered blood bank wants to make request blood online, it can also take the help of this site. Admin is the main authority who can do addition, deletion, and modification if required.

The project has been planned to be having the view of distributed architecture, with centralized storage of the database. The application for the storage of the data has been planned using the constructs of MY-SQL through XAMPP Server and all the user interfaces have been designed using the Web technologies. User The database connectivity is planned using the “Apache Web Server” methodology. The standards of security and data protective mechanism have been given a big choice for proper usage. The application takes care of different modules and their associated reports, which are produced as per the applicable strategies and standards.

The entire project has been developed keeping in view of the distributed client server computing technology, in mind. The user interfaces are browser independent to provide uniform access for the overall system. The internal database has been selected as MY-SQL database. The basic constructs of table spaces, clusters and indexes have been exploited to provide higher consistency and reliability for the data storage. The MySQL was a choice because it provides the constructs of high-level reliability and security. The total front end was implemented completely using Handlebars, CSS and JavaScript while the backend is managed by NodeJs. We have extensively used ExpressJs as Web Framework for NodeJs. At all proper levels high care was taken to check that the system managed the data consistency with proper rules or validations. The database connectivity was planned using the latest “Apache Web Server” technology provided by XAMPP. The authentication and authorization was crosschecked at all the relevant stages.

2. Project Objectives

* To have latest updates on blood donation drive and donors available.
* Use of Web Services and Remoting.
* To have a centralized system to manage blood donation.
* To effectively monitor or manage the related data.
* To provide security with different level of authentication.
* To have proper co-ordination between Blood Banks, Donors, Users and Applications.
* To improve donor recruitment and retention by providing a user-friendly online donation process, tracking donor information and allowing them to view donation drives.
* To increase data security and compliance with regulations by providing a secure and auditable system for storing and managing sensitive donor and patient information.
* To spread awareness on Blood donation and educate the people

Functional Requirements

The functional requirements of a Blood Donation Management System (BDMS) will vary depending on the specific needs and goals of the blood banks that is using it. However, some common functional requirements that might be included are:

Donor registration and management: The BDMS should enable the registration of new donors and the management of existing donor information, including personal details, contact information, etc.

Appointment scheduling: The system should allow donors to view and contact to blood donation drives by blood banks.

Blood Bank Information management: The system should enable the registration and management of blood banks and all the related information including contact information and their requirements and blood drives.

User management: The BDMS should have a user management system that controls access to the system and different functionalities based on the user's role.

Data security: The system should have robust security features to protect donor information and comply with data privacy regulations.

Compliance: The system should comply with all relevant regulations and guidelines for blood donation and management.

Mobile access: The system should have a mobile version to enable access from multiple devices and locations

Integration: The system should be able to integrate with other systems, such as electronic medical records, to improve information sharing and coordination.

Feedback and suggestion management: The system should have a mechanism for receiving and managing feedback, messages and suggestions from users, to improve the system and user experience.

Dashboard and alerts: The system should have a dashboard to provide real-time data on key performance indicators, such as donors count, donor activity, and drive information. The system should also have tabs to enable users to view drives from various blood banks, or any other important information.

Non Functional Requirements

Non-functional requirements are the requirements that specify the system's overall characteristics such as performance, security, usability, and maintainability. Some of non-functional requirements for our Blood Donation Management System (BDMS) include:

Performance: The system is able to handle a large number of concurrent users, process data quickly and efficiently, and generate reports and statistics in a timely manner.

Security: The system is designed to protect donor and patient information from unauthorized access and ensure compliance with data privacy regulations.

Usability: The system is user-friendly, easy to navigate and understand, and accessible for all users, including those with disabilities.

Scalability: The system is able to handle an increase in the number of donors, blood banks, and blood requests over time.

Maintainability: The system is easy to maintain and update, with minimal downtime and disruptions to service.

Accessibility: The system is accessible remotely via web and mobile devices to allow easy access by users and blood banks.

Integration: The system is able to integrate with other relevant systems such as electronic medical records and inventory management systems.

Compliance: The system should be compliant with all relevant regulations, standards, and guidelines for blood donation and management.

Implementation

The idea of these whole project is to overcome all the above difficulties while providing a centralized database to manage all the data related to blood donation and also to provide various functionalities to this data at the same time.

This approach includes usage of Web services to provide a platform to connect both blood banks and blood donors. It also includes a section to educate the people about the importance of blood donation and also to spread awareness in the society and answer all frequently asked questions.

The entire project has been developed keeping in view of the distributed client server computing technology, in mind. The user interfaces are browser independent to provide uniform access for the overall system. The internal database has been selected as MY-SQL database. The whole front end was implemented completely using Handlebars, CSS and JavaScript. The backend was supported by NodeJs. We have extensively used ExpressJs as Web Framework for NodeJs. This provides high efficiency and fast response times. The database connectivity was planned using the latest “Apache Web Server” technology provided by XAMPP. The authentication and authorization was crosschecked at all the relevant stages.

Technologies used in this project development are: - Handlebars, CSS, JavaScript, MY-SQL, Apache Web Server, XAMPP, NodeJs, ExpressJs, Body-Parser, Nodemon, Git, and GitHub.

Handlebars

Handlebars is a popular, open-source JavaScript library that is used for building semantic templates. It provides a simple way to create and manage templates for dynamic web pages. It allows developers to separate the structure of a web page (i.e. the HTML) from the data that is displayed on the page, reducing the amount of code needed to display data on a web page, which makes it easier to maintain and update the code.

Handlebars can be used with a variety of JavaScript frameworks, such as Ember.js and Backbone.js, as well as with server-side frameworks like Express.js. It is compatible with both client-side and server-side rendering.

CSS

CSS, or Cascading Style Sheets, is a style sheet language used for describing the presentation of a document written in a markup language. It is used to control the layout and formatting of web pages, including colors, fonts, spacing, and other visual elements.

With CSS, developers can separate the presentation of a web page from its structure and content, making it easier to maintain and update the design of a website. This is done by linking a CSS file to an HTML document, and then applying styles to specific elements on the page using selectors and declarations.

JavaScript

JavaScript is a versatile and powerful language that can be used for both front-end and back-end development. On the front-end, it can be used to create interactive user interfaces and handle user interactions. On the back-end, it can be used to create servers, web applications and work with databases.

JavaScript is a programming language that is primarily used to create interactive and dynamic user interfaces for websites. It can be used to add a wide range of functionality to web pages, such as form validation, image sliders, and interactive maps.

MY-SQL

MySQL is a free and open-source relational database management system (RDBMS). It is widely used in web applications and data-driven websites to store and manage data. MySQL is known for its reliability, ease of use, and performance.

MySQL stores data in tables, which are similar to spreadsheets, with rows and columns. Each table is made up of fields (or columns) and records (or rows). Each field contains a specific type of data, such as text, numbers, or dates. MySQL uses a variant of the SQL (Structured Query Language) to interact with the database, which allows developers to insert, update, retrieve, and delete data in the database.

XAMPP

XAMPP is an open-source software that provides an easy way to install and run Apache web server, PHP, and MySQL on a local machine. It stands for Cross-Platform (X), Apache (A), Maria DB (M) (a fork of MySQL), PHP (P) and Perl (P). This package allows developers to create and test web applications on their local computers, without the need for a live web server.

Apache Web Server

Apache is an open-source web server software that is widely used on the Internet. It is known for its stability, security, and flexibility, and it is supported by a large community of developers.

Apache is responsible for handling HTTP requests and returning the appropriate response to the client. When a client (such as a web browser) requests a web page, the request is sent to the Apache web server, which then processes the request and returns the requested page to the client.

NodeJs

Node.js is an open-source, cross-platform, JavaScript runtime environment that executes JavaScript code outside of a web browser. It allows developers to use JavaScript on the server-side to create server-side applications, networking tools, and other types of back-end tools.

Node.js uses an event-driven, non-blocking I/O model, which makes it efficient for building scalable and high-performance network applications. It also provides a large library of modules, known as the Node Package Manager (NPM), which can be easily installed and used in Node.js applications.

Node.js uses the JavaScript language, which makes it a popular choice among web developers who are already familiar with JavaScript and want to use it for both the front-end and back-end of their web applications. Node.js can also be integrated with other technologies such as MongoDB, ExpressJs, and AngularJS to build full-stack web applications.

ExpressJs

Express.js is a web application framework for Node.js, it is a minimal and flexible framework that provides a robust set of features for web and mobile applications. It is designed to simplify the process of building web applications with Node.js.

Express.js provides a simple and easy-to-use API for handling HTTP requests and responses, routing, middleware, and other web application-related tasks. It also allows developers to easily connect with databases and other web services.

Code Snippets

These are few codes written in NodeJs and ExpressJs that includes a database access to fetch, update or delete the data stored and display these data in the website where required. We have also ensured to make sure the security and authentication for each of the SQL query operation is maintained using username and userPassword in each levels.

const mysql = require('mysql')

const connection = mysql.createConnection({

host : 'localhost',

user : 'admin',

password : 'admin',

database: 'blood\_donation',

port :3306,

})

connection.connect(function(err) {

if(err) {

console.log("Not connected")

throw err

}

console.log("Connected")

})

module.exports = {connection, mysql}

Figure 4.1 Code Snippet of Connection Module

Figure 4.1 Code Snippet of Connection Module which is used for connecting purpose. This is used to connect back end and front end. If the connection fails then a message occurs saying “Not Connected”.

router.get('', (req, res) => {

connection.query(`select Name, Bgroup from users LIMIT 7`, async (err, result, field) => {

connection.query(`select \* from faq`, async (err, results, field) => {

res.render('index.hbs', {

'result': result,

'faq': results

})

})

})

})

Figure 4.1 Code Snippet of render the landing page(website) from backend

Figure 4.1 Code Snippet of render the landing page from backend. It also includes fetching the top 7 donors as well as FAQ’s from the database and sending it to the page.

exports.userSignUp = async (req, res) => {

const userName = req.body.name

const userEmail = req.body.email

const userPassword = req.body.password

const userGroup = req.body.group

const userPhNo = req.body.PhNo

const userAge = req.body.age

const userGender = req.body.gender

const userLocation = req.body.location

const encryptedPassword = userPassword

connection.query(`insert into users(Name, Email, Bgroup, PhNo, Age, Gender, Location, Password) Values ("${userName}", "${userEmail}", "${userGroup}", "${userPhNo}", "${userAge}", "${userGender}", "${userLocation}", "${encryptedPassword}")`, (err, results, field) => {

if (err) {

console.log(err)

if (err.errno === 1062) {

return res.status(406).send({

"message": "The entered email is already registered"

})

}

res.status(500).send({

"message": "Server error"

})

} else {

const finish = () => {

res.redirect('/')

}

setTimeout(finish, 1500)

}

})

};

|  |
| --- |
|  |
|  |  |

Figure 4.1 Code Snippet of signing up/registering a donor

Figure 4.1 shows the code snippet of signing up/registering a donor via inserting the donor’s record along with other attributes into the databse table name “users”

exports.userLogin = (req, res) => {

const userName = req.body.name

const userEmail = req.body.email

const userPassword = req.body.password

const newPassword = req.body.newPasswd

let age = req.body.age

let gender = req.body.gender

let location = req.body.location

const updateFlag = req.body.update

if(updateFlag == undefined) { //Logging In

connection.query(`select \* from users where Email = "${userEmail}"`, async (err,results,field) => {

if (err) {

res.status(500).send({

"message": "Server error"

})

console.log(err)

} else {

if (results.length == 0) {

return res.status(401).send({

"message": "The entered credentials do not match"

})

}

match = (userPassword == results[0].Password) && (userName == results[0].Name) && (userEmail == results[0].Email)

if (match) {

res.render('userdash.hbs', {

"user": results[0]

})

} else {

res.status(401).send({

"message": "The entered credentials do not match"

})

}

}

})

}

else if(updateFlag == 0) { // redirecting to userdashboard after authenticating

connection.query(`Select \* from users where Email = "${userEmail}"`, (err, results, fields) => {

if (err) {

res.status(500).send({

"message": "Server error"

})

console.log(err)

} else {

if (results.length == 0) {

return res.status(401).send({

"message": "The entered credentials do not match"

})

}

match = (userPassword == results[0].Password)

if (match) {

connection.query('UPDATE users SET Password = ? WHERE Email = ?',[newPassword,userEmail], (error, userresults, fields) => {

if(error) {

console.log(error)

res.status(500).send({

"message": "Server error"

})

}

else {

connection.query(`select \* from users where Email = "${userEmail}"`, async (err, results, field) => {

if (err) {

res.status(500).send({

"message": "Server error"

})

console.log(err)

} else {

if (results.length == 0) {

return res.status(401).send({

"message": "The entered credentials do not match"

})

}

res.render('userdash.hbs', {

"user": results[0]

})

}

})

}

})

}

else{

return res.status(401).send({

"message": "The entered credentials do not match"

})

}

}

})

}

else if(updateFlag == 1){ //updating user’s data by the user and redirecting back to userdashboard

connection.query(`Select \* from users where Email = '${userEmail}'`, async (err, results, fields) => {

if (err) {

res.status(500).send({

"message": "Server error"

})

console.log(err)

} else {

if(age == '')

age = results[0].Age

if(gender == '')

gender = results[0].Gender

if(location == '')

location = results[0].Location

connection.query('UPDATE users SET Age = ?, Gender = ?, Location = ? WHERE Email = ?', [age, gender, location, userEmail], (err, results, field) => {

if (err) {

console.log(err)

res.status(500).send({

"message": "Server error"

})

}

else {

connection.query(`select \* from users where Email = "${userEmail}"`, async (err, results, field) => {

if (err) {

res.status(500).send({

"message": "Server error"

})

console.log(err)

} else {

if (results.length == 0) {

return res.status(401).send({

"message": "The entered credentials do not match"

})

}

res.render('userdash.hbs', {

"user": results[0]

})

}

})

}

})

}

})

}

else {

connection.query(`select \* from users where Email = "${userEmail}"`, async (err, results, field) => {

if (err) {

res.status(500).send({

"message": "Server error"

})

console.log(err)

} else {

console.log(results)

if (results.length == 0) {

return res.status(401).send({

"message": "The entered credentials do not match"

})

}

// const match = await bcrypt.compare(userPassword, results[0].password)

match = (userPassword == results[0].Password) && (userEmail == results[0].Email)

if (match) {

// req.session.userName = results[0].Name

// res.status(200).send()

res.render('userdash.hbs', {

"user": results[0]

})

} else {

res.status(401).send({

"message": "The entered credentials do not match"

})

}

}

})

}

};

exports.userReqPage = (req, res) => {

const userName = req.body.name

const userEmail = req.body.email

const userPassword = req.body.password

connection.query(`select \* from Request\_Blood`, async (err, results, field) => {

if (err) {

res.status(500).send({

"message": "Server error"

})

console.log(err)

} else {

res.render('userReqPage', {

'name': userName,

'email': userEmail,

'password': userPassword,

'result': results

})

}

})

}

exports.unregister = async (req, res) => {

connection.query(`delete from users where Email="${req.body.email}"`, (err, results, field) => {

if (err) {

console.log(err)

res.status(500).send({

"message": "Server error"

})

} else {

// res.status(200).send()

res.redirect("/")

}

})

};

exports.bloodBankLogin = (req, res) => {

const userName = req.body.name

const userPassword = req.body.password

const userReq = req.body.requirements

const userPhNo = req.body.phno

const updateFlag = req.body.flag

const userDesc = req.body.desc

console.log(req.body)

if(updateFlag == 1) { //Requesting blood

connection.query(`insert into Request\_Blood(Name, PhNo, Requirements, Description) Values ("${userName}", "${userPhNo}", "${userReq}", "${userDesc}")`, (err, results, field) => {

if (err) {

console.log(err)

if (err.errno === 1062) {

return res.status(406).send({

"message": "The entered email is already registered"

})

}

res.status(500).send({

"message": "Server error"

})

}

})

}

else if(updateFlag == 2) { //updating the amount of blood requested

connection.query(`update Request\_Blood Set Requirements = "${userReq}", Description = "${userDesc}" where Name = "${userName}"`, (error, userresults, fields) => {

if(error) {

console.log(error)

res.status(500).send({

"message": "Server error"

})

}

})

}

//redirecting back to blood bank dashboard after authenticating

connection.query(`select \* from BloodBank where Name = "${userName}"`, async (err, results, field) => {

if (err) {

res.status(500).send({

"message": "Server error"

})

console.log(err)

} else {

if (results.length == 0) {

return res.status(401).send({

"message": "The entered credentials do not match"

})

}

console.log(results)

match = (userPassword == results[0].Password) &&(userName == results[0].Name)

if (match) {

// req.session.userName = results[0].Name

// res.status(200).send()

connection.query(`Select \* from users`, async (error, userresults, fields) => {

res.render('bloodbankdash', {

result: userresults,

user: userName,

'phno': results[0].PhNo,

'requirements': userReq,

password: userPassword

})

})

} else {

res.status(401).send({

"message": "The entered credentials do not match"

})

}

}

})

};

exports.adminPage = (req, res) => { //admin Login

const adminName = req.body.name

const adminPassword = req.body.password

console.log(req.body)

connection.query(`select \* from Admin where Name = "${adminName}"`, async (err, results, field) => {

if (err) {

res.status(500).send({

"message": "Server error"

})

console.log(err)

} else {

if (results.length == 0) {

return res.status(401).send({

"message": "The entered credentials do not match"

})

}

// const match = await bcrypt.compare(userPassword, results[0].password)

console.log(results)

match = (adminPassword == results[0].Password) &&(adminName == results[0].Name)

if (match) {

res.render('adminPage.hbs', {

adminResult: results[0]

})

} else {

res.status(401).send({

"message": "The entered credentials do not match"

})

}

}

})

}

exports.reqPage = (req, res) => { //user’s notifications page

const user = req.body.user

const password = req.body.password

const phno = req.body.phno

const userReq = req.body.requirements

const userDesc = req.body.desc

connection.query(`select \* from Request\_Blood where Name = "${user}"`, async (err, results, field) => {

if (err) {

res.status(500).send({

"message": "Server error"

})

console.log(err)

} else {

if(results.length == 0) { //if no notifications, displaying empty

res.render('reqPage', {

'user': user,

'password': password,

'phno': phno,

'requirements': userReq,

'desc': userDesc,

})

}

else { //else displaying request blood table

res.render('reqPage', {

'user': results[0].Name,

'password': password,

'phno': results[0].PhNo,

'requirements': results[0].Requirements,

'desc': results[0].Description,

})

}

}

})

}

Testing

Software testing is the process of used to identify the correctness, security, completeness and

quality of developed computer software. This includes the process of executing the program or

applications with the intent of finding errors. An individual unit, functions or procedures of

developed project is verified and validated and these units are fit for use.

5.1Testing process

Best testing process is to test each subsystem separately, as we have done in project. Best done

during implementation. Best done after small sub-steps of the implementation rather than large

chunks. Once each lowest level unit has been tested, units are combined with related units and

retested in combination. This proceeds hierarchically bottom-up until the entire system is tested

as a whole. Typical levels of testing:

1.Module- package, abstract data type, class

2.Sub-system- collection of related modules, cluster of classes, method-message paths

3.Acceptance testing- whole system with real data (involve customer, user, etc) Alpha testing is

acceptance testing with a single client (common for bespoke systems).

Beta testing involves distributing system to potential customers to use and provide feedback. In

this project, beta testing has been followed. This exposes system to situations and errors that

might not be anticipated by us.

5.1.1Unit testing

Unit testing is the process of testing individual software components unit or modules. Since it

needs the detailed knowledge of the internal program design and code this task is done by the

programmer and not by testers.

5.1.2Integration Testing

Integration testing is another aspect of testing that is generally done in order to uncover errors

associated with the flow of data across interfaces. The unit-tested modules are grouped together

and tested in small segment, which makes it easier to isolate and correct errors. This approach

is continued until we have integrated all modules to form the system.

As completion of each module, it has been combined with the remaining module to ensure that

the project is working properly as expected.

5.1.3 System Testing

System testing tests a completely integrated system to verify that it meets its requirements. After

the completion of the entire module, they are combined together to test whether the entire project

is working properly.

5.2 Test Cases

A Test Case is a software testing document, which consists of events, action, input, output,

expected result and actual result. Technically a test case includes test description, procedure,

expected result and remarks. Test cases should be based primarily on the software requirements

and developed to verify correct functionality and to establish conditions that reveal potential

errors.

.

.

.

CONCLUSION

Our proposed project Blood Donation Management helps the donors to join the community in simplest

manner which saves time and hassle. Blood Donation Management System is online

registration and connecting technique. There is a database which maintains all the names of

donors and blood banks with complete information. Further , users can view various blood requests from blood banks and also contact them if interested. This is not only limited to donors but blood banks also find this system very useful in order to contact donors and conduct blood donation drives. Finally admin is the ultimate authority in this system who control, manage all the related data in the system. From this, he can further analyse the data and derive some of the reports and statistics for future enhancements. By this project we, oursolves had a lot to learn and practise. We have gained a knowledge on how the client server architecture actually works and how such systems are developed.

This project provided practical knowledge of not only programming and developing interface to users through web application but also in handling data, users, procedures related to Blood Donation and managing database. It also provides knowledge about the latest technology used in developing web enabled application and database technology that will be great demand in future. This will provide better opportunities and guidance in future for developing projects independently.

FUTURE ENHANCEMENT

In the future enhancement of this project, we would like to include some additional requirements

that can be implemented and integrated into the application code. It includes having portals for screening and monitoring donors as well as integrating with other sub-systems in the hospital, making it much more reliable and flexible, and also making much more user interactive.

3. Existing System

Most of the times, traditional approach to Blood Donation management is being handled manually through number of clerks, employees, volunteers, etc. It also faces many difficulties such as:

* Cannot Upload and Download the latest updates.
* No use of Web Services and Remoting.
* Risk of mismanagement and of data when the project is under development.
* Less Security.
* No proper coordination between different Applications and Users.
* Fewer Users – Friendly

These are most common problems in traditional approach which leads to data redundancy, risk of mismanagement, too much messy data.

4. Proposed System (Engineering Approach)

The idea of these whole project is to overcome all the above difficulties while providing a centralized database to manage all the data related to blood donation and also to provide various functionalities to this data at the same time.

This approach includes usage of Web services to provide a platform to connect both blood banks and blood donors. It also includes a section to educate the people about the importance of blood donation and also to spread awareness in the society.

Technologies to be used in this project development are: - HTML, CSS, JavaScript, MongoDB, Firebase Git, and GitHub.

5. Project Benefits and Deliverables

The project is identified by the merits of the system offered to the user. The merits of this project are as follows: -

* It’s a web-enabled project.
* This project offers user to enter the data through simple and interactive forms. This is very helpful for the client to enter the desired information through so much simplicity.
* The user is mainly more concerned about the validity of the data, whatever he is entering. There are checks on every stages of any new creation, data entry or updation so that the user cannot enter the invalid data, which can create problems at later date.
* Sometimes the user finds in the later stages of using project that he needs to update some of the information that he entered earlier. There are options for him by which he can update the records. Moreover, there is restriction for his that he cannot change the primary data field. This keeps the validity of the data to longer extent.
* User is provided the option of monitoring the records he entered earlier. He can see the desired records with the variety of options provided by him.
* From every part of the project the user is provided with the links through framing so that he can go from one option of the project to other as per the requirement. This is bound to be simple and very friendly as per the user is concerned. That is, we can say that the project is user friendly which is one of the primary concerns of any good project.
* Data storage and retrieval will become faster and easier to maintain because data is stored in a systematic manner and in a single central database.
* Decision making process would be greatly enhanced because of faster processing of information since data collection from information available on computer takes much less time than manual system.
* Allocating of sample results becomes much faster because at a time the user can see the records of last years.
* Easier and faster data transfer through latest technology associated with the computer and communication.
* Through these features it will increase the efficiency, accuracy and transparency.

6. Project Limitations

* The size of the database increases day-by-day, increasing the load on the database back up and data maintenance activity.
* Training for simple computer operations is necessary for the users working on the system.

7. Conclusion

It will be a great pleasure for us to work on this exciting and challenging project. This project will provide practical knowledge of not only programming and developing interface to users through web application but also in handling data, users, procedures related to Blood Donation and managing database. It also provides knowledge about the latest technology used in developing web enabled application and database technology that will be great demand in future. This will provide better opportunities and guidance in future in developing projects independently.