

# **CHURCH MANAGEMENT SYSTEM**

## **Midterm Report**

**COURSE NAME:** APPLIED RESEARCH PROJECT

**COURSE ID:** CSIS 4495\_003

**GROUP MEMBERS:** BASIL RUGOYI

**STUDENT ID:** 300371550

**VIDEO DEMO LINK:** <https://youtu.be/NaonABnkMMs>

# Table of Contents

<u>Introduction</u> .....	4
○ <u>Domain Overview and Background</u> .....	4
○ <u>Problem Framing</u> .....	4
○ <u>Literature Review and Knowledge Gaps</u> .....	5
○ <u>Hypotheses, Assumptions and Benefits</u> .....	5
<u>Summary of the Initially Proposed Research Project</u> .....	5
<u>Changes to the Proposal</u> .....	6
○ <u>Feature Scope Adjustments</u> .....	6
○ <u>Technical Refinements</u> .....	6
○ <u>Timeline Adjustments</u> .....	7
<u>Project Planning and Timeline</u> .....	7
○ <u>Project Phases and Milestones</u> .....	7
○ <u>Team Responsibilities</u> .....	8
○ <u>Project Management Approach</u> .....	8
○ <u>Gantt Chart</u> .....	9
<u>Implemented Features</u> .....	10
○ <u>Project Code Structure and Component Classification</u> .....	10
○ <u>Controllers</u> .....	10
○ <u>Middleware</u> .....	16
○ <u>Models</u> .....	19
○ <u>Routes</u> .....	21
○ <u>Partials</u> .....	24
○ <u>Welcome Screen</u> .....	26
○ <u>Login Screen</u> .....	28
○ <u>Authentication and Authorization</u> .....	30
○ <u>User Management</u> .....	31
○ <u>Member Management</u> .....	39
○ <u>Dashboard</u> .....	49

○ <a href="#"><u>Chatbot Module</u></a> .....	50
○ <a href="#"><u>Server Configuration Module</u></a> .....	52
<a href="#"><u>AI Use Section</u></a> .....	53
<a href="#"><u>Work Date / Hours Logs</u></a> .....	54
<a href="#"><u>Closing and References</u></a> .....	55
<a href="#"><u>Appendix</u></a> .....	55
<a href="#"><u>Conclusion</u></a> .....	55

## **1. Introduction**

### **1.1 Domain Overview and Background**

Churches and faith-based organizations increasingly rely on digital systems to manage administrative operations such as member records, staff access, reporting, and internal communication. Traditionally, many churches continue to rely on manual or semi-digital methods such as spreadsheets, paper records, or fragmented tools, which introduces inefficiencies, data inconsistencies, and security risks. As congregations grow, these limitations become more pronounced, creating the need for a centralized, secure, and role-based administrative system.

The Church Administration System addresses this domain by providing a web-based platform that supports user authentication, member management, and role-controlled access to administrative features. The system is designed with modern web technologies and follows industry best practices in security and modular architecture.

### **1.2 Problem Framing**

The primary problem addressed in this research project is the lack of an integrated, secure, and scalable administrative platform tailored to church operations. Furthermore, manual systems make it difficult for church leadership to analyse trends in attendance and membership growth. Key questions guiding this research include:

- How can sensitive church data be protected while remaining accessible to authorized users?
- In what ways can attendance and membership data be accurately captured and analysed using modern web technologies?
- How can a centralized, web-based modular database system improve the efficiency, maintainability and future scalability of church administration?

These questions are important because church data often contains sensitive personal information, and unauthorized access can lead to privacy breaches and operational disruption.

### **1.3 Literature Review and Knowledge Gaps**

Previous studies on information systems in non-profit and religious organizations indicate that digitization improves operational efficiency, data accuracy, and transparency. Research on church management systems has largely focused on basic membership databases or financial accounting tools. However, many existing solutions lack integration, real-time reporting, and flexibility for customization.

There is a notable gap in research and practical implementation of lightweight, customizable, full-stack JavaScript-based systems tailored specifically for church administration. This project aims to address this gap by developing an integrated system that combines membership management, attendance tracking, and reporting within a single platform.

### **1.4 Hypotheses, Assumptions, and Benefits**

This project assumes that church administrators are willing to adopt digital tools if they are user-friendly and cost-effective. The primary hypothesis is that a centralized web-based system will significantly improve data accuracy, accessibility, and administrative efficiency.

The expected benefits of this research include improved operational workflows, enhanced security, reduced administrative workload, better decision-making through data analysis, and a reusable framework that can be adapted by other churches or non-profit organizations. There is also the benefit of a foundation for future features such as analytics and automation.

## **2. Summary of the Initially Proposed Research Project**

The initially proposed project focused on building a web-based church administration system that enables secure login, member, finance, attendance management, and role-based user control.

The proposal outlined the use of the below technologies: -

- **Operating System / Platform:** Windows 10 / Web-based platform
- **Programming Languages:** JavaScript, HTML, CSS
- **Backend Framework:** Node.js with Express.js
- **Database:** MongoDB
- **Frontend:** HTML5, CSS3, Vanilla JavaScript
- **Development Tools:** Visual Studio Code, Postman, GitHub

The core features included authentication, member records, and administrative reporting. The system is expected to reduce record duplication, improve data accuracy, and enable faster retrieval of information. The reporting features are to support data-driven decision-making, while the modular design will allow future enhancements.

### **3. Changes to the Proposal**

Several refinements were made to the initial proposal as development progressed:

#### **3.1 Feature Scope Adjustments**

Additional modules such as user reports, edit-user functionality, and a chatbot interface were introduced. These changes were justified based on emerging requirements discovered during implementation and testing.

#### **3.2 Technical Refinements**

The authentication strategy was refined to fully integrate middleware-based authorization checks. Partial templates were introduced to reduce duplication across pages. These changes improved maintainability and security.

### **3.3 Timeline Adjustments**

Development began earlier than originally planned to validate architectural decisions, and to have part of the system in-place and working by the end of February 2026. This is to reduce technical risk later in the project lifecycle.

## **4. Project Planning and Timeline**

### **4.1 Project Phases and Milestones**

#### **Project Timeline (February 10 – April 7, 2026)**

<b>Phase</b>	<b>Task Description</b>	<b>Start Date</b>	<b>End Date</b>	<b>Key Deliverables</b>
Phase 3	Backend Development & Authentication. Core Module Development (Members, Users, Users Reports)  Initial Chatbot Integration.	Feb 10	Feb 23	REST APIs, JWT authentication, role-based access. Functional backend and frontend modules.  Church information chatbot.
Phase 4	Backend Development. Core Module Development (Finance, Attendance, Finance Reports)	Feb 24	Mar 2	Functional backend and frontend modules. Data persistence.
Phase 5	Core Module Development. (Refinement of modules, Edit & Delete functions for Users & Members)	Mar 3	Mar 9	Fully functional CRUD operations, refined business logic, stable core modules
Phase 6	User Interface Enhancements. Navigation improvements.	Mar 10	Mar 16	Improved user experience, polished UI, responsive layouts.

Phase	Task Description	Start Date	End Date	Key Deliverables
Phase 7	Advanced Features & Integration. Chatbot enhancement and contextual responses.	Mar 17	Mar 23	Enhanced chatbot functionality, integration of system modules.
Phase 8	Testing, Debugging & Validation	Mar 24	Mar 30	Test cases, bug fixes, validation results
Phase 9	Documentation & Final Report Preparation	Mar 31	Apr 6	Final report, technical documentation
Phase 10	Final Review & Submission	Apr 7	Apr 7	Final submission

## 4.2 Team Responsibilities

- **Team Lead (Basil Rugoyi): (Working Alone)**
  - Overall project coordination
  - Backend and database development
  - Final integration and documentation

## 4.3 Project Management Approach

### Project Management Tool: Gantt Chart

A **Gantt Chart-based planning approach** is being used to track milestones and deadlines. Tasks are divided into weekly deliverables to ensure steady progress and accountability. The Gantt Chart provides a structured visual representation of project phases, task dependencies, milestones, and deliverables. This approach enables effective progress tracking, accountability, and timely completion of the research project.

# Gantt Chart (Weeks, Tasks, Milestones)

Task / Phase	Week 1 (Jan 10-19)	Week 2 (Jan 20-26)	Week 3 (Jan 27-Feb 2)	Week 4 (Feb 3-9)	Week 5 (Feb 10-16)	Week 6 (Feb 17-23)	Week 7 (Feb 24-Mar 2)	Week 8 (Mar 3-9)	Week 9 (Mar 10-16)	Week 10 (Mar 17-23)	Week 11 (Mar 24-30)	Week 12 (Mar 31-Apr 7)
Project Planning & Proposal Drafting												
Literature Review & Background Research												
Requirements Analysis & System Design												
Database Design (MongoDB)												
Backend Development (Node.js / Express)												
Frontend Development (HTML, CSS, JS)												
Chatbot Integration & UI Enhancements												
System Testing & Bug Fixes												
Documentation & Final Report Writing												
Final Review & Submission												

## **5. Implemented Features**

### **Project Code Structure and Component Classification**

#### **5.1. Controllers (Backend)**

##### **Purpose:**

Controllers contain the business logic of the application. They receive requests from routes, interact with models, and return responses to the client.

Listed below are the Controllers and the code implemented in the system.

##### **memberController.js**

###### **Responsibilities:**

- Create a new church member
- Retrieve all members
- Retrieve a single member by ID
- Update member details
- Delete a member

###### **Key Functions:**

- createMember()
- getAllMembers()
- getMemberById()
- updateMember()
- deleteMember()

###### **Used By:**

- Members management module
- Members report module
- Edit-member functionality

Below is the implemented memberController.js code (screenshot).



The screenshot shows a code editor window with the file 'memberController.js' open. The code is written in JavaScript and defines several methods for managing members in a database. The methods include creating a new member, getting all members, getting a single member by ID, and updating a member. Error handling is implemented using try-catch blocks and returning JSON responses with validation errors.

```
JS memberController.js U X
backend > controllers > JS memberController.js > deleteMember > deleteMember
1 const Member = require("../models/Member");
2
3 // Create a new member
4 exports.createMember = async (req, res) => {
5   try {
6     const member = new Member(req.body);
7     await member.save();
8     res.status(201).json(member);
9   } catch (error) {
10    res.status(400).json({
11      error: "Validation failed",
12      details: error.message
13    });
14  }
15};
16
17 // Get all members
18 exports.getAllMembers = async (req, res) => {
19   try {
20     const members = await Member.find();
21     res.status(200).json(members);
22   } catch (error) {
23     res.status(500).json({
24       error: "Validation failed",
25       details: error.message
26     });
27   }
28 };
29
30 // Get a single member by ID
31 exports.getMemberById = async (req, res) => {
32   try {
33     const member = await Member.findById(req.params.id);
34     if (!member) {
35       return res.status(404).json({ message: "Member not found" });
36     }
37     res.status(200).json(member);
38   } catch (error) {
39     res.status(500).json({
40       error: "Validation failed",
41       details: error.message
42     });
43   }
44 };
45
46 // Update a member
47 exports.updateMember = async (req, res) => {
48   try {
49     console.log("UPDATE PAYLOAD:", req.body);
50     const member = await Member.findByIdAndUpdate(
51       req.params.id,
52       { $set: req.body },
53       [ { new: true },
54     ]);
55     res.status(200).json(member);
56   } catch (error) {
57     res.status(500).json({
58       error: "Validation failed",
59       details: error.message
60     });
61   }
62 };
63
```

```
54     |     |     |     runValidators: true
55     |     |     |
56     |     |     );
57     |     |     |
58     |     |     if (!member) {
59     |     |     |     return res.status(404).json({ message: "Member not found" });
60     |     |     }
61     |     |     res.status(200).json(member);
62     |     |   } catch (error) {
63     |     |     |     res.status(400).json({
64     |     |     |       error: "Validation failed",
65     |     |     |       details: error.message
66     |     |     });
67     |     |   }
68     |     | };
69     |     |
70     |     // Delete a member
71     |     exports.deleteMember = async (req, res) => {
72     |     |     try {
73     |     |     |     await Member.findByIdAndDelete(req.params.id);
74     |     |     |     res.status(200).json({ message: "Member deleted successfully" });
75     |     |     } catch (error) {
76     |     |     |     res.status(500).json({
77     |     |     |       error: "Validation failed",
78     |     |     |       details: error.message
79     |     |     });
80     |     |   }
81     |     | };

```

## **userController.js**

## Responsibilities:

- Create system users
  - Retrieve all users (Admin-only)
  - Delete users
  - Update user details (name, email, role)

## Key Functions:

- createUser()
  - getAllUsers()
  - getAllUsers()
  - updateUser()
  - deleteUser()

## Used By:

- Create-user module
  - Users report module
  - Edit-user functionality

Below is the implemented userController.js code (screenshot).

The screenshot shows a code editor window with the file 'userController.js' open. The code is written in JavaScript and defines several functions for managing users:

```
JS userController.js U X
backend > controllers > JS userController.js > ...
1 const User = require("../models/User");
2 const bcrypt = require("bcryptjs");
3
4 exports.createUser = async (req, res) => {
5   try {
6     console.log("CREATE USER BODY:", req.body);
7     const { name, email, password, role } = req.body;
8
9     const user = new User({
10       name,
11       email,
12       password,
13       role
14     );
15
16     await user.save();
17
18     res.status(201).json({ message: "User created successfully" });
19   } catch (err) {
20     console.error("CREATE USER ERROR:", err);
21
22     if (err.code === 11000) {
23       return res.status(400).json({ message: "User already exists" });
24     }
25
26     res.status(500).json({ message: "Failed to create user" });
27   }
28 };
29
● 30 exports.getAllUsers = async (req, res) => {
31   try {
32     const users = await User.find().select("-password"); // hide passwords
33     res.json(users);
34   } catch (err) {
35     res.status(500).json({ message: "Failed to load users" });
36   }
37 };
38
39 // GET single user (ADMIN only)
40 exports.getUserById = async (req, res) => {
41   try {
42     const user = await User.findById(req.params.id).select("-password");
43
44     if (!user) {
45       return res.status(404).json({ message: "User not found" });
46     }
47
48     res.json(user);
49   } catch (err) {
50     res.status(500).json({ message: "Failed to load user" });
51   }
52 };
53
54 exports.updateUser = async (req, res) => {
55   try {
56     const { name, email, role, password } = req.body;
57
58     const updateData = { name, email, role };
59
60     const user = await User.findById(req.params.id);
61
62     if (!user) {
63       return res.status(404).json({ message: "User not found" });
64     }
65
66     user.name = updateData.name;
67     user.email = updateData.email;
68     user.role = updateData.role;
69
70     if (password) {
71       user.password = bcrypt.hashSync(password, 10);
72     }
73
74     await user.save();
75
76     res.json({ message: "User updated successfully" });
77   } catch (err) {
78     res.status(500).json({ message: "Failed to update user" });
79   }
80 };
81
82 exports.deleteUser = async (req, res) => {
83   try {
84     const user = await User.findByIdAndDelete(req.params.id);
85
86     if (!user) {
87       return res.status(404).json({ message: "User not found" });
88     }
89
90     res.json({ message: "User deleted successfully" });
91   } catch (err) {
92     res.status(500).json({ message: "Failed to delete user" });
93   }
94 };
95
96 exports.login = async (req, res) => {
97   try {
98     const user = await User.findOne({ email: req.body.email });
99
100    if (!user) {
101      return res.status(404).json({ message: "User not found" });
102    }
103
104    const isMatch = bcrypt.compareSync(req.body.password, user.password);
105
106    if (!isMatch) {
107      return res.status(401).json({ message: "Incorrect password" });
108    }
109
110    const token = jwt.sign({ id: user._id }, process.env.JWT_SECRET);
111
112    res.json({ token });
113  } catch (err) {
114    res.status(500).json({ message: "Failed to login" });
115  }
116};
117
```

```

59   // Only hash password if it was changed
60   if (password && password.trim() !== "") {
61     updateData.password = await bcrypt.hash(password, 10);
62   }
63
64   const updatedUser = await User.findByIdAndUpdate(
65     req.params.id,
66     updateData,
67     { new: true }
68   ).select("-password");
69
70   if (!updatedUser) {
71     return res.status(404).json({ message: "User not found" });
72   }
73
74   res.json(updatedUser);
75 } catch (err) {
76   console.error("UPDATE USER ERROR:", err);
77   res.status(500).json({ message: "Update failed" });
78 }
79 };
80
81 // DELETE user
82 exports.deleteUser = async (req, res) => {
83   try {
84     await User.findByIdAndDelete(req.params.id);
85     res.status(200).json({ message: "User deleted" });
86   } catch (err) {
87     res.status(500).json({ message: "Delete failed" });
88   }
89 };

```

## authController.js

Responsibilities:

- User authentication
- Credential verification
- JWT token generation

Used By:

- Login page
- authGuard middleware

Below is the implemented authController.js code (screenshot).

```

JS authController.js U X
backend > controllers > JS authController.js > ...
1 const User = require("../models/User");
2 const bcrypt = require("bcryptjs");
3 const jwt = require("jsonwebtoken");
4
5 exports.login = async (req, res) => {
6   console.log("LOGIN BODY:", req.body);
7   const { email, password } = req.body;
8
9   const user = await User.findOne({ email });
10  console.log("USER FOUND:", user);
11  if (!user) {
12    return res.status(401).json({ message: "Invalid credentials" });
13  }
14
15  const isMatch = await bcrypt.compare(password, user.password);
16  console.log("PASSWORD MATCH:", isMatch);
17  if (!isMatch) {
18    return res.status(401).json({ message: "Invalid credentials" });
19  }
20
21  const token = jwt.sign(
22    { id: user._id, role: user.role },
23    process.env.JWT_SECRET,
24    { expiresIn: "8h" }
25  );
26
27  res.json({
28    token,
29    role: user.role
30    // name: user.name
31  });
32};

```

## chatbotController.js

Responsibilities:

- Process chatbot questions
- Route user queries to predefined responses
- Return formatted chatbot responses

Key Functions (typical):

- handleChatRequest()
- getChurchInfo()
- processMessage()

Used In:

- Chatbot module
- Church information assistant

Below is the implemented chatbotController.js code (screenshot).

The screenshot shows a code editor window with the file 'chatbotController.js' open. The code is written in JavaScript and defines a function 'chatbotReply' that takes 'req' and 'res' as parameters. It first checks if the message is in lowercase. If it includes 'service', it responds with service times. If it includes 'location', it responds with location details. If it includes 'meeting', it responds with meeting times. If it includes 'group', it lists church groups. Finally, it sends a JSON response with the reply.

```
JS chatbotController.js X
backend > controllers > JS chatbotController.js > chatbotReply > chatbotReply
1 exports.chatbotReply = (req, res) => {
2   const message = req.body.message.toLowerCase();
3
4   let reply = "Sorry, I didn't understand that. Please ask about service times, meetings, or groups.";
5
6   if (message.includes("service")) {
7     reply = `
8     Sunday Service: 10:00 AM - 11:30 AM
9     Wednesday Prayer: 6:00 PM - 7:30 PM
10    `;
11  }
12
13  else if (message.includes("location")) {
14    reply = `
15    Address: 6062 132 St, Surrey, BC V3X 1M9
16    Contact Number: (604) 591-3599
17    Email: info@berea.ca
18    `;
19  }
20
21  else if (message.includes("meeting")) {
22    reply = `
23    Leadership Meeting: First Saturday - 10:00 AM
24    Youth Meeting: Friday - 5:00 PM
25    Women's Fellowship: Thursday - 4:00 PM
26    `;
27  }
28
29  else if (message.includes("group")) {
30    reply = `
31    Church Groups:
32    - Youth Ministry
33    - Women's Fellowship
34    - Men's Fellowship
35    - Choir
36    - Ushering Team
37    `;
38  }
39
40  res.json({ reply });
41 };
42
```

## 5.2. Middleware (Backend)

### Purpose:

Middleware sits between the request and controller to handle authentication, authorization, and security checks.

Listed below is the Middleware and the code implemented in the system.

## auth.js

Purpose:

- Verifies JWT token
- Prevents unauthorized access to protected routes

Used In:

- Users routes
- Members routes
- Reports routes

Below is the implemented auth.js code (screenshot).



The screenshot shows a code editor window with the file 'auth.js' open. The code is written in JavaScript and uses the jsonwebtoken library to verify JWT tokens. It checks if a token is provided in the authorization header, verifies it against a secret, and sets the user object on the request if successful. If the token is invalid or missing, it returns a 401 status with an error message.

```
JS auth.js U X
backend > middleware > JS auth.js > ...
1 const jwt = require("jsonwebtoken");
2
3 module.exports = (req, res, next) => {
4   const token = req.headers.authorization?.split(" ")[1];
5
6   if (!token) {
7     return res.status(401).json({ message: "No token provided" });
8   }
9
10  try {
11    req.user = jwt.verify(token, process.env.JWT_SECRET);
12    next();
13  } catch {
14    res.status(401).json({ message: "Invalid token" });
15  }
16};
17
```

## authorize.js

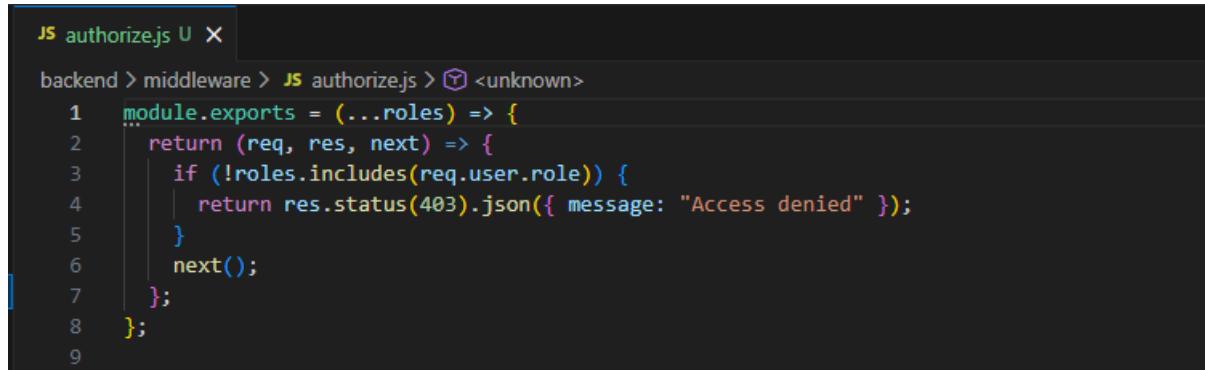
Purpose:

- Role-based access control
- Restricts access based on user roles (ADMIN, STAFF, etc.)

Used In:

- Users report
- Create-user
- Edit-user
- Admin-only routes

Below is the implemented authorize.js code (screenshot).



The screenshot shows a code editor window with the title "JS authorize.js". The file path is "backend > middleware > JS authorize.js > <unknown>". The code is as follows:

```
1 module.exports = (...roles) => {
2   ...
3   return (req, res, next) => {
4     if (!roles.includes(req.user.role)) {
5       return res.status(403).json({ message: "Access denied" });
6     }
7     next();
8   };
9 };
```

## authGuard.js (Frontend)

Purpose:

- Client-side route protection
- Redirects users if:
  - Token is missing
  - Role is unauthorized
- Handles logout functionality

Used In:

- Dashboard
- Members
- Users report
- Edit-member
- Edit-user

Below is the implemented authGuard.js code (screenshot).

```

JS authGuard.js ×
frontend > JS authGuard.js > ...
1  console.log("authGuard loaded");
2
3  window.authToken = localStorage.getItem("token");
4  window.userRole = localStorage.getItem("role");
5
6  if (!window.authToken) {
7    |  window.location.href = "login.html";
8  }
9
10 function logout() {
11   |  localStorage.clear();
12   |  window.location.href = "welcome.html";
13 }
14
15 function requireRole(...allowedRoles) {
16   |  if (!allowedRoles.includes(window.userRole)) {
17   |    |  alert("Access denied");
18   |    |  window.location.href = "dashboard.html";
19   |  }
20 }
21 /* MAKE FUNCTIONS GLOBAL */
22 window.logout = logout;
23 window.requireRole = requireRole;

```

### 5.3. Models (Backend)

#### Purpose:

Models define the database structure and schema using Mongoose and represent collections in MongoDB.

Listed below are the Models and the code implemented in the system.

#### User.js

##### Fields:

- name
- email
- password
- role

##### Used By:

- Authentication
- User management
- Authorization middleware

Below is the implemented User.js code (screenshot).

The screenshot shows a code editor window with the tab bar at the top labeled "JS User.js". Below the tab bar, the file path "backend > models > JS User.js > ..." is displayed. The main area contains the code for the User.js file, which defines a Mongoose schema for a User document. The schema includes fields for name, email, password, and role, with validation rules like required and unique. It also includes a pre-save middleware function to hash the password before saving. The code ends with an export statement.

```
JS User.js  U X
backend > models > JS User.js > ...
1  const mongoose = require("mongoose");
2  const bcrypt = require("bcryptjs");
3
4  const userSchema = new mongoose.Schema({
5    name: {
6      type: String,
7      required: true
8    },
9    email: {
10      type: String,
11      required: true,
12      unique: true
13    },
14    password: {
15      type: String,
16      required: true
17    },
18    role: {
19      type: String,
20      enum: ["ADMIN", "PASTOR", "TREASURER", "STAFF"],
21      default: "STAFF"
22    }
23  }, { timestamps: true });
24
25  // Hash password before save
26  userSchema.pre("save", async function (next) {
27    if (!this.isModified("password")) return next();
28    this.password = await bcrypt.hash(this.password, 10);
29    next();
30  });
31
32  module.exports = mongoose.model("User", userSchema);
--
```

## Member.js

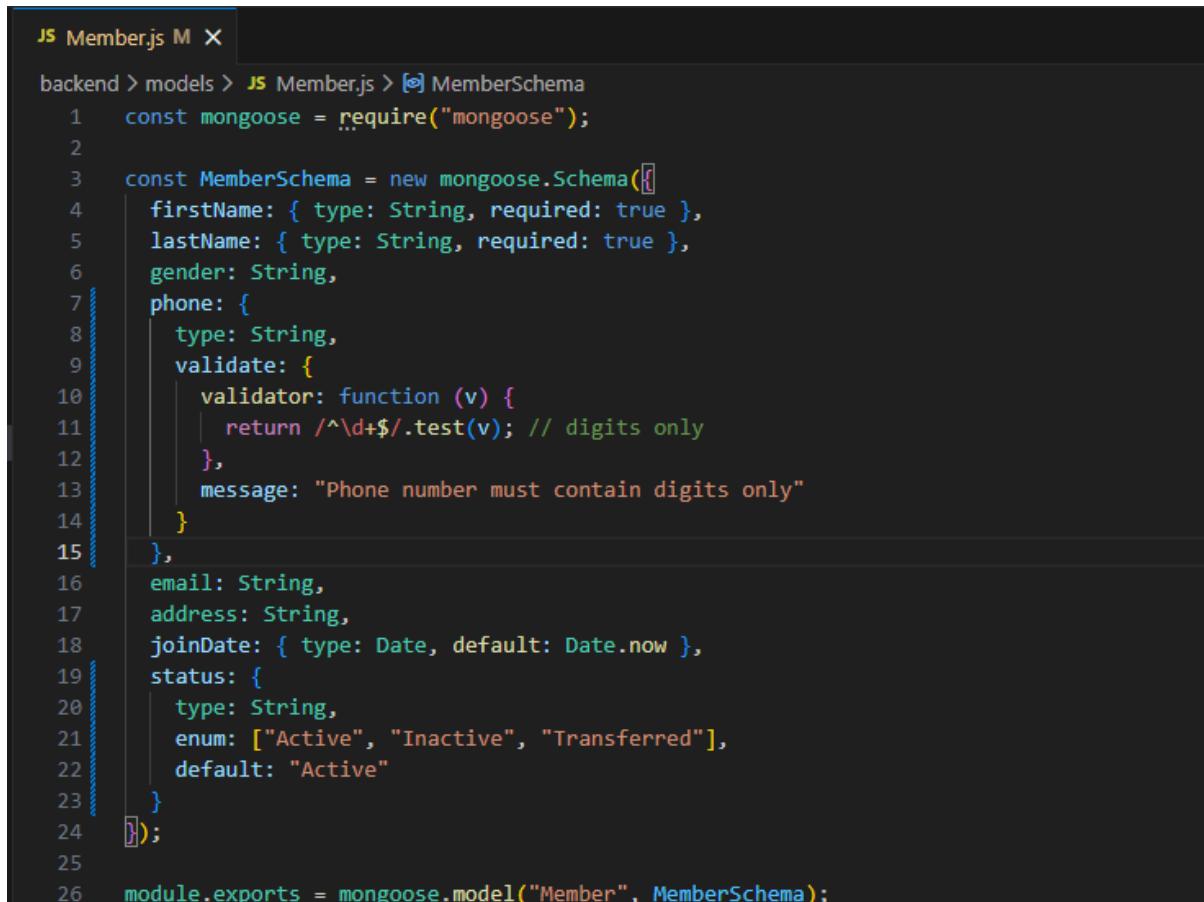
Fields:

- firstName
- lastName
- gender
- phone
- email
- address
- joinDate
- status

## Used By:

- Members module
- Members report
- Edit-member functionality

Below is the implemented Member.js code (screenshot).



```
JS Member.js M X
backend > models > JS Member.js > [o] MemberSchema
1  const mongoose = require("mongoose");
2
3  const MemberSchema = new mongoose.Schema({
4      firstName: { type: String, required: true },
5      lastName: { type: String, required: true },
6      gender: String,
7      phone: {
8          type: String,
9          validate: {
10              validator: function (v) {
11                  return /^\d+$/.test(v); // digits only
12              },
13              message: "Phone number must contain digits only"
14          }
15      },
16      email: String,
17      address: String,
18      joinDate: { type: Date, default: Date.now },
19      status: {
20          type: String,
21          enum: ["Active", "Inactive", "Transferred"],
22          default: "Active"
23      }
24  });
25
26  module.exports = mongoose.model("Member", MemberSchema);
```

## 5.4. Routes (Backend)

### Purpose:

Routes define API endpoints and connect requests to the appropriate controller functions.

Listed below are the Routes and the code implemented in the system.

### routes/members.js

Endpoints:

- POST /api/members – Create member
- GET /api/members – Get all members
- GET /api/members/:id – Get single member
- PUT /api/members/:id – Update member
- DELETE /api/members/:id – Delete member

Middleware Applied:

- auth

Below is the implemented routes/members.js code (screenshot).

```
js members.js M X
backend > routes > js members.js > ...
1  const express = require("express");
2  const router = express.Router();
3  const memberController = require("../controllers/memberController");
4  const auth = require("../middleware/auth");
5  const authorize = require("../middleware/authorize");
6
7  // Create member
8  router.post("/", auth, memberController.createMember);
9
10 // Get all members
11 router.get("/", auth, memberController.getAllMembers);
12
13 // Get single member
14 router.get("/:id", auth, memberController.getMemberById);
15
16 // Update member
17 router.put("/:id", auth, memberController.updateMember);
18
19 // Delete member
20 router.delete("/:id", auth, authorize("ADMIN"), memberController.deleteMember);
21
22 module.exports = router;
```

## routes/users.js

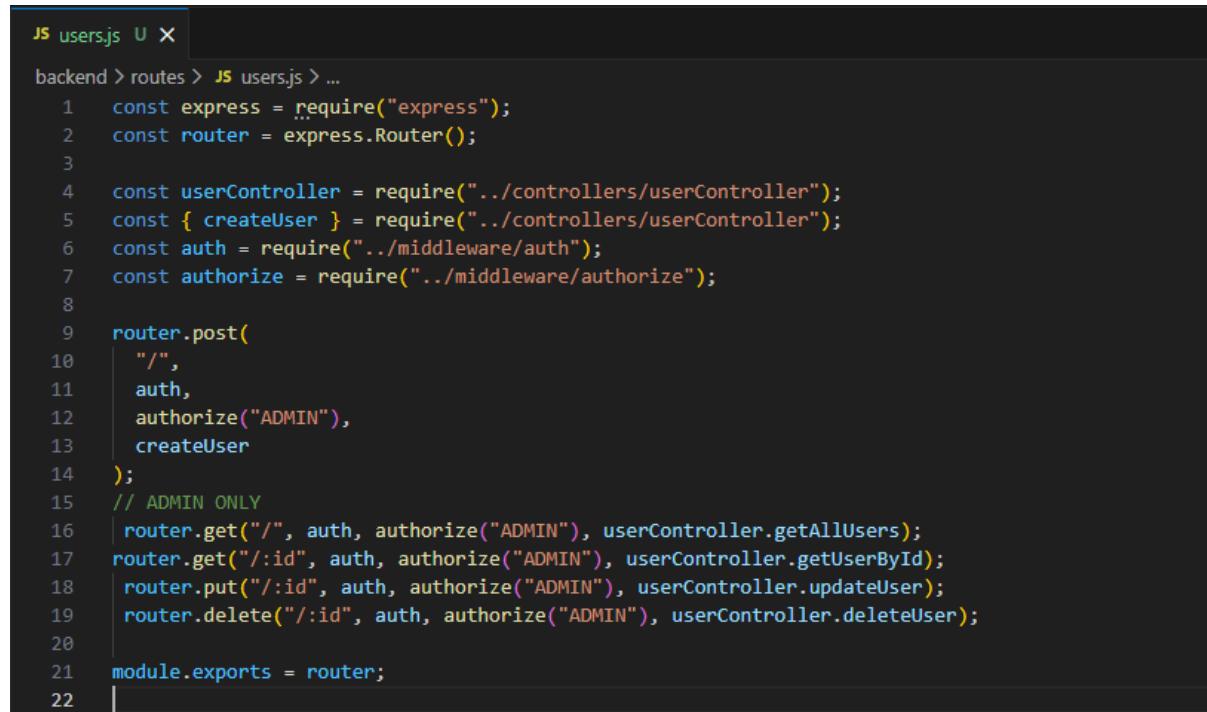
Endpoints:

- POST /api/users – Create user
- GET /api/users – Get all users (ADMIN only)
- GET /api/users/:id – Get single user
- PUT /api/users/:id – Update user
- DELETE /api/users/:id – Delete user

## Middleware Applied:

- auth
- authorize("ADMIN")

Below is the implemented routes/users.js code (screenshot).



The screenshot shows a code editor window with the file 'users.js' open. The code defines an Express router for user operations, including POST, GET, PUT, and DELETE methods. It uses middleware 'auth' and 'authorize("ADMIN")' for specific routes. The code is as follows:

```
JS users.js U X
backend > routes > JS users.js > ...
1 const express = require("express");
2 const router = express.Router();
3
4 const userController = require("../controllers/userController");
5 const { createUser } = require("../controllers/userController");
6 const auth = require("../middleware/auth");
7 const authorize = require("../middleware/authorize");
8
9 router.post(
10   "/",
11   auth,
12   authorize("ADMIN"),
13   createUser
14 );
15 // ADMIN ONLY
16 router.get("/", auth, authorize("ADMIN"), userController.getAllUsers);
17 router.get("/:id", auth, authorize("ADMIN"), userController.getUserById);
18 router.put("/:id", auth, authorize("ADMIN"), userController.updateUser);
19 router.delete("/:id", auth, authorize("ADMIN"), userController.deleteUser);
20
21 module.exports = router;
22 |
```

## routes/auth.js

Endpoints:

- POST /api/auth/login – Login user and issue JWT

Below is the implemented routes/auth.js code (screenshot).



The screenshot shows a code editor window with the file 'auth.js' open. The code defines an Express router with a single POST endpoint for logging in users. The code is as follows:

```
JS auth.js U X
backend > routes > JS auth.js > ...
1 const express = require("express");
2 const router = express.Router();
3 const { login } = require("../controllers/authController");
4
5 router.post("/login", login);
6
7 module.exports = router;
8 |
```

## **routes/chatbot.js**

Endpoints:

- POST /api/chatbot/message
- GET /api/chatbot/info

Middleware Applied:

- Auth

Mapped Controller:

- chatbotController.js

Below is the implemented routes/chatbot.js code (screenshot).



The screenshot shows a code editor window with a dark theme. The file is named 'chatbot.js' and is located in the 'backend > routes' directory. The code is as follows:

```
JS chatbot.js U X
backend > routes > JS chatbot.js > ...
1 const express = require("express");
2 const router = express.Router();
3 const { chatbotReply } = require("../controllers/chatbotController");
4
5 router.post("/", chatbotReply);
6
7 module.exports = router;
8 |
```

## **5.5. Partials (Frontend)**

**Purpose:**

Partials are reusable UI components shared across multiple pages to ensure consistency and reduce duplication.

Listed below are the Partials used in the system.

### **chatbot.html**

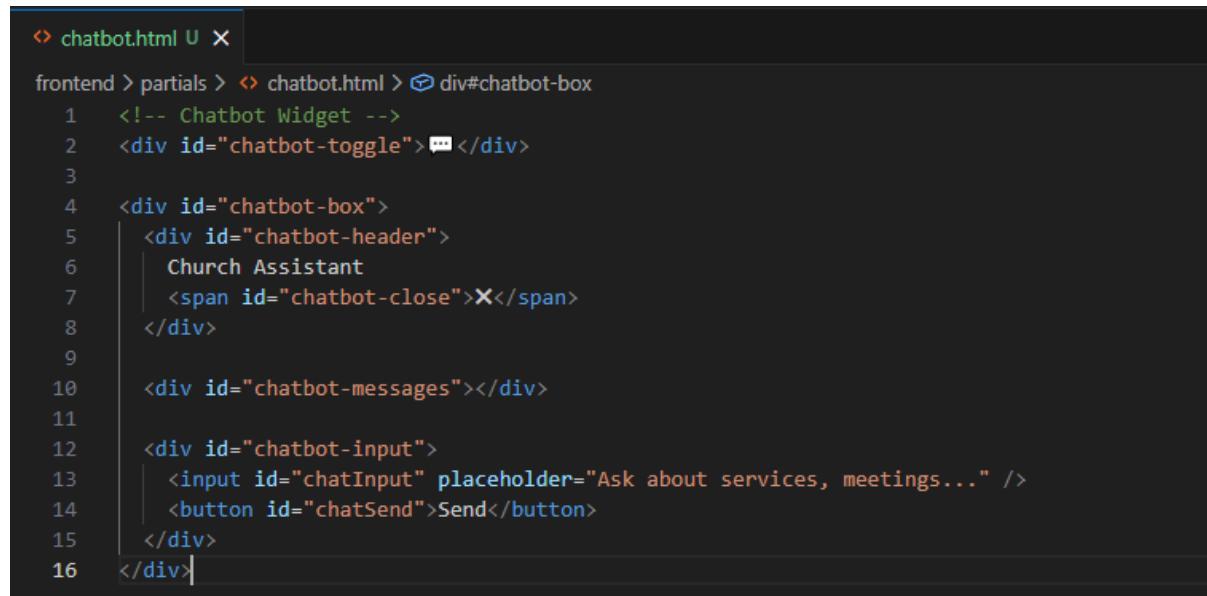
**Purpose:**

- User interface for the church information chatbot
- Allows users to:
  - Ask questions
  - Receive automated responses

Used In:

- Dashboard
- Standalone chatbot page

Below is the implemented chatbot.html code (screenshot).



The screenshot shows a code editor window with the file 'chatbot.html' open. The code is a template for a chatbot interface, structured as follows:

```
chatbot.html U X
frontend > partials > chatbot.html > div#chatbot-box
1  <!-- Chatbot Widget -->
2  <div id="chatbot-toggle">...</div>
3
4  <div id="chatbot-box">
5    <div id="chatbot-header">
6      Church Assistant
7      <span id="chatbot-close">X</span>
8    </div>
9
10   <div id="chatbot-messages"></div>
11
12   <div id="chatbot-input">
13     <input id="chatInput" placeholder="Ask about services, meetings..." />
14     <button id="chatSend">Send</button>
15   </div>
16 </div>
```

## Navigation Bar (navbar)

Features:

- Role-based menu hiding
- Logout option
- Shared across:
  - Dashboard
  - Members
  - Users Report
  - Edit pages

## Tables (Reports)

Used In:

- Members report
- Users report

## Features:

- Dynamic data rendering
- Edit/Delete icons
- Role-based visibility

## Forms

Reusable patterns used in:

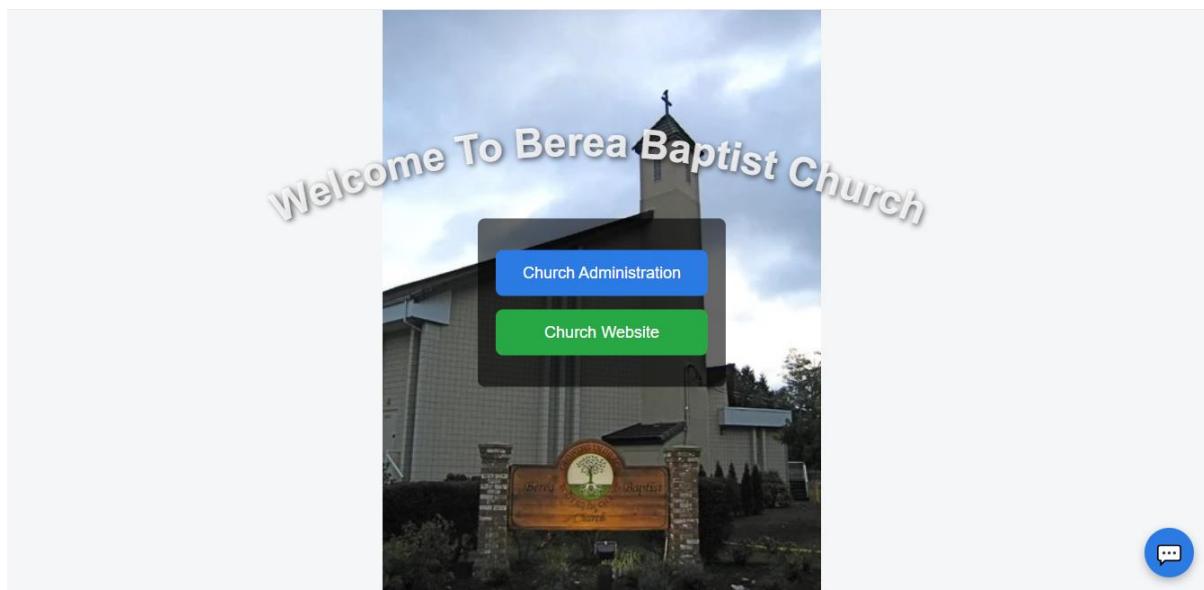
- Add-member
- Edit-member
- Create-user
- Edit-user

## Features:

- Input validation
- Controlled fields
- API integration

## 5.6 Welcome Screen (Frontend)

Implemented a welcome screen with two buttons, to allow the user to choose whether they want to do administration or to go to the church website. Included on the welcome screen is a chatbot for assistance with church information.



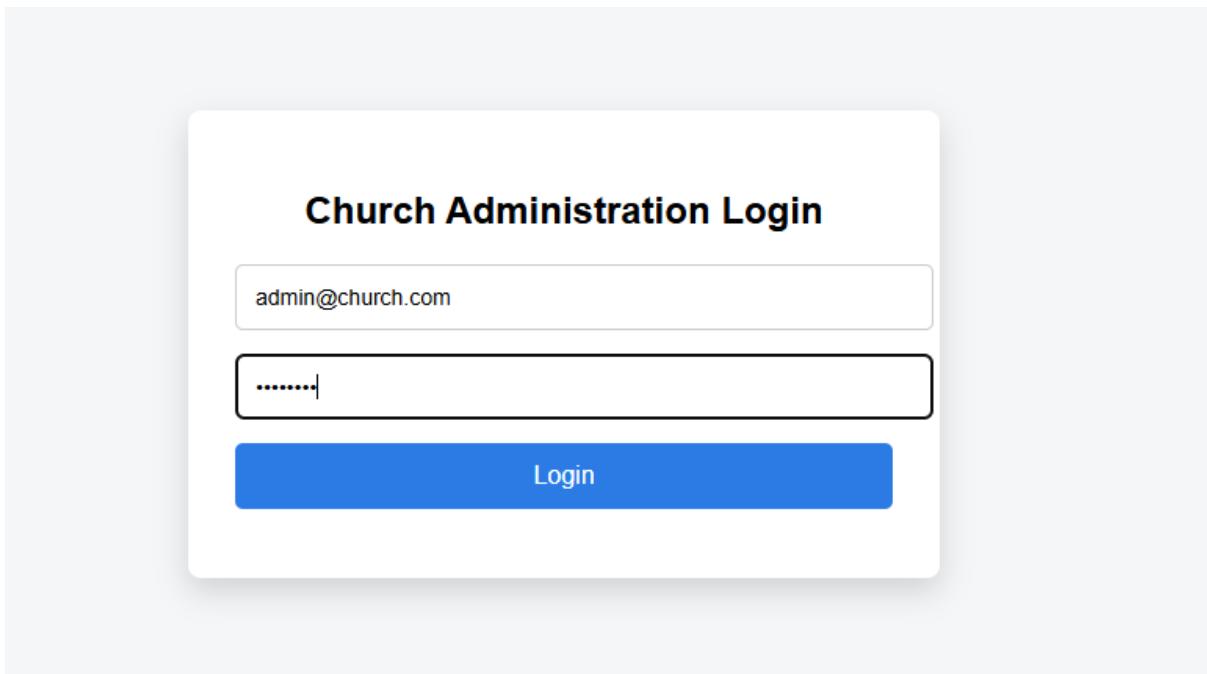
Below is the implemented welcome.html code for the welcome screen.

```
↳ welcome.html M X
frontend > ↳ welcome.html > ⏺ html > ⏺ head
1   <!DOCTYPE html>
2   <html lang="en">
3
4   <head>
5     <meta charset="UTF-8">
6     <title>Welcome | Church System</title>
7     <link rel="stylesheet" href="css/welcome.css">
8     <link rel="stylesheet" href="css/chatbot.css">
9   </head>
10
11  <div id="chatbot-container"></div>
12
13  <script>
14    fetch("partials/chatbot.html")
15      .then(res => res.text())
16      .then(html => {
17        document.getElementById("chatbot-container").innerHTML = html;
18
19        // Load chatbot.js AFTER HTML exists
20        const script = document.createElement("script");
21        script.src = "chatbot.js";
22        script.onload = () => {
23          initChatbot();
24        };
25        document.body.appendChild(script);
26      });
27  </script>
28
29  <body class="welcome-body">
30
31    <div class="welcome-container">
32
33      <div class="image-wrapper">
34        
35        <!-- Arc Text -->
36        <svg class="arc-text" viewBox="0 0 900 320" preserveAspectRatio="xMidYMid meet">
37          <defs>
38            <path id="arcPath" d="M 100 220 Q 450 40 900 220" />
39          </defs>
40
41          <text fill="rgba(255,255,255,0.85)" font-size="48" font-weight="bold" text-anchor="middle"
42            style="text-shadow: 2px 2px 6px rgba(0,0,0,0.7);">
43            <textPath href="#arcPath" startOffset="50%">
44              Welcome To Berea Baptist Church
45            </textPath>
46          </text>
47        </svg>
```

```
48     <div class="welcome-buttons">
49         <a href="login.html" class="welcome-btn">Church Administration</a>
50         <a href="https://www.berea.ca" target="_blank" class="welcome-btn secondary">
51             Church Website
52         </a>
53     </div>
54 </div>
55 </div>
56 </body>
57 </html>
```

## 5.7 Login Screen (Frontend)

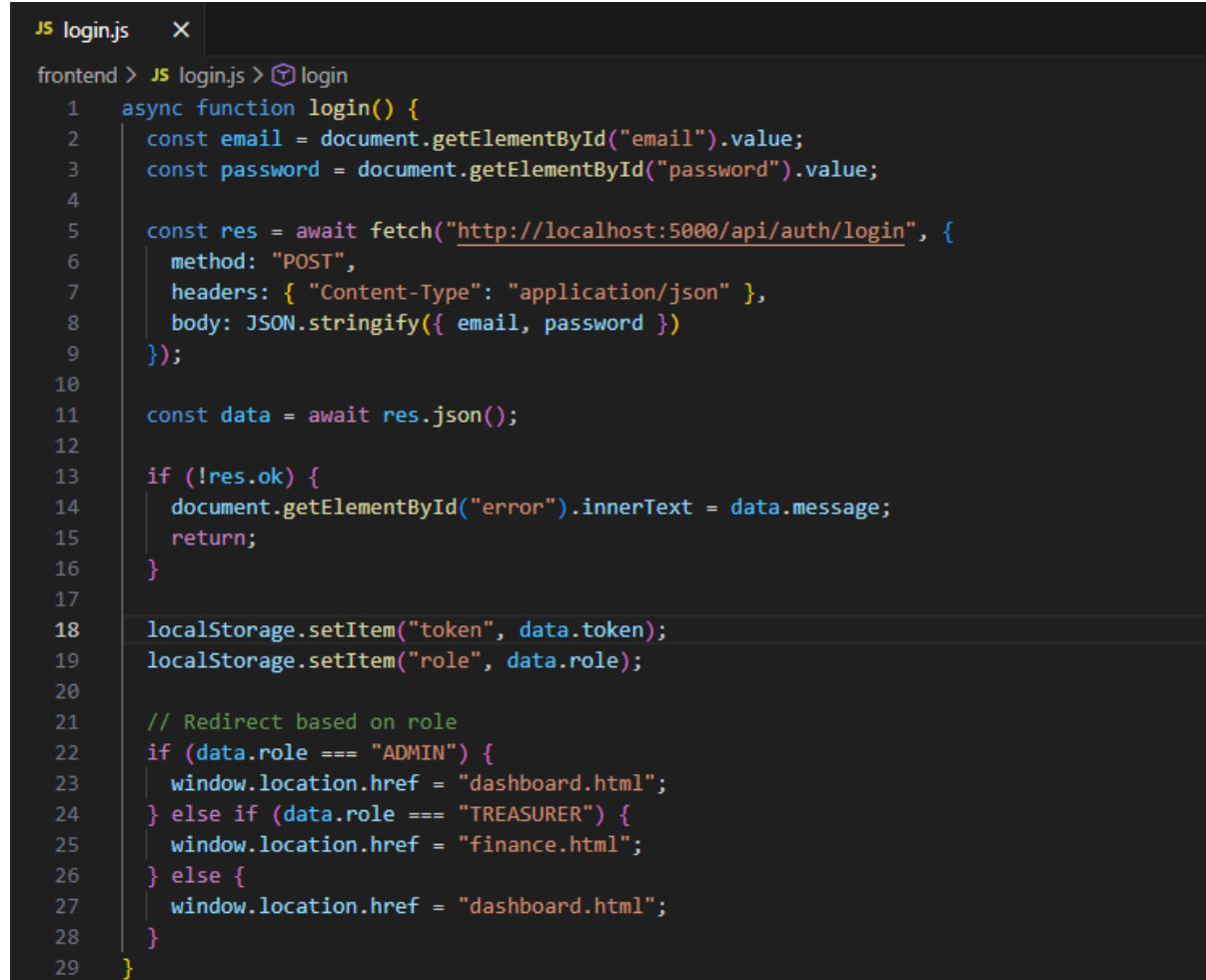
A login screen was implemented to allow for users to be authenticated before gaining access to the system. This will allow the user to view menu items of the dashboard linked to the user's role.



Below is the implemented login.html code to display the login screen.

```
login.html M X
frontend > login.html > html > body > script
1  <!DOCTYPE html>
2  <html>
3
4  <head>
5    <title>Login | Church Admin</title>
6    <link rel="stylesheet" href="css/login.css">
7  </head>
8
9  <body>
10
11  <div class="login-box">
12    <h2>Church Administration Login</h2>
13
14    <input id="email" placeholder="Email">
15    <input id="password" type="password" placeholder="Password">
16    <button onclick="login()">Login</button>
17
18    <p id="error"></p>
19  </div>
20
21  <script>
22    async function login() {
23      const errorEl = document.getElementById("error");
24      errorEl.innerText = ""; // clear previous error
25
26      const res = await fetch("http://localhost:5000/api/auth/login", {
27        method: "POST",
28        headers: { "Content-Type": "application/json" },
29        body: JSON.stringify({
30          email: email.value,
31          password: password.value
32        })
33      );
34
35      const data = await res.json();
36
37      if (res.ok && data.token) {
38        localStorage.setItem("token", data.token);
39        localStorage.setItem("role", data.role);
40        window.location.href = "dashboard.html";
41      } else {
42        errorEl.innerText = data.message || "Invalid login credentials";
43      }
44    }
45  </script>
46 </body>
47
48 </html>
```

Below is the implemented login.js code to authenticate user and check user role.



The screenshot shows a code editor window with the title "JS login.js". The file path is "frontend > JS login.js > login". The code is written in JavaScript and performs the following steps:

- Imports the fetch function.
- Gets the email and password from the document.
- Makes a POST request to "http://localhost:5000/api/auth/login" with the email and password in the body.
- Checks if the response is successful. If not, it sets the error message to the "error" element and returns.
- Stores the token and role in local storage.
- Redirects the user based on their role: ADMIN to "dashboard.html", TREASURER to "finance.html", and otherwise to "dashboard.html".

```
1  async function login() {
2    const email = document.getElementById("email").value;
3    const password = document.getElementById("password").value;
4
5    const res = await fetch("http://localhost:5000/api/auth/login", {
6      method: "POST",
7      headers: { "Content-Type": "application/json" },
8      body: JSON.stringify({ email, password })
9    );
10
11   const data = await res.json();
12
13   if (!res.ok) {
14     document.getElementById("error").innerText = data.message;
15     return;
16   }
17
18   localStorage.setItem("token", data.token);
19   localStorage.setItem("role", data.role);
20
21   // Redirect based on role
22   if (data.role === "ADMIN") {
23     window.location.href = "dashboard.html";
24   } else if (data.role === "TREASURER") {
25     window.location.href = "finance.html";
26   } else {
27     window.location.href = "dashboard.html";
28   }
29 }
```

## 5.8 Authentication and Authorization (Frontend)

Implemented JWT-based authentication with middleware (authGuard) enforcing role restrictions across protected routes.

Below is the implemented authGuard.js code to grant user access to the system after authentication based on user role.

```
JS authGuard.js X
frontend > JS authGuard.js > ...
1  console.log("authGuard loaded");
2
3  window.authToken = localStorage.getItem("token");
4  window.userRole = localStorage.getItem("role");
5
6  if (!window.authToken) {
7    window.location.href = "login.html";
8  }
9
10 function logout() {
11   localStorage.clear();
12   window.location.href = "welcome.html";
13 }
14
15 function requireRole(...allowedRoles) {
16   if (!allowedRoles.includes(window.userRole)) {
17     alert("Access denied");
18     window.location.href = "dashboard.html";
19   }
20 }
21 /* MAKE FUNCTIONS GLOBAL */
22 window.logout = logout;
23 window.requireRole = requireRole;
```

## 5.9 User Management (Frontend)

Modules implemented include:

- create-user
- users-report
- edit-user

These modules allow administrators to manage system users securely.

## 5.9(i) create-user

Below is the create-user screen, visible to the admin user.

The screenshot shows a 'Create New User' form within a 'Church Admin' application. The top navigation bar includes links for 'Dashboard', 'Create User', and 'Users Report'. The form itself has a title 'Create New User' and contains four input fields: 'Full Name', 'Email', 'Password', and a dropdown menu set to 'Staff'. A blue 'Create User' button is at the bottom.

Below is the implemented create-user.html code to display the above screen.

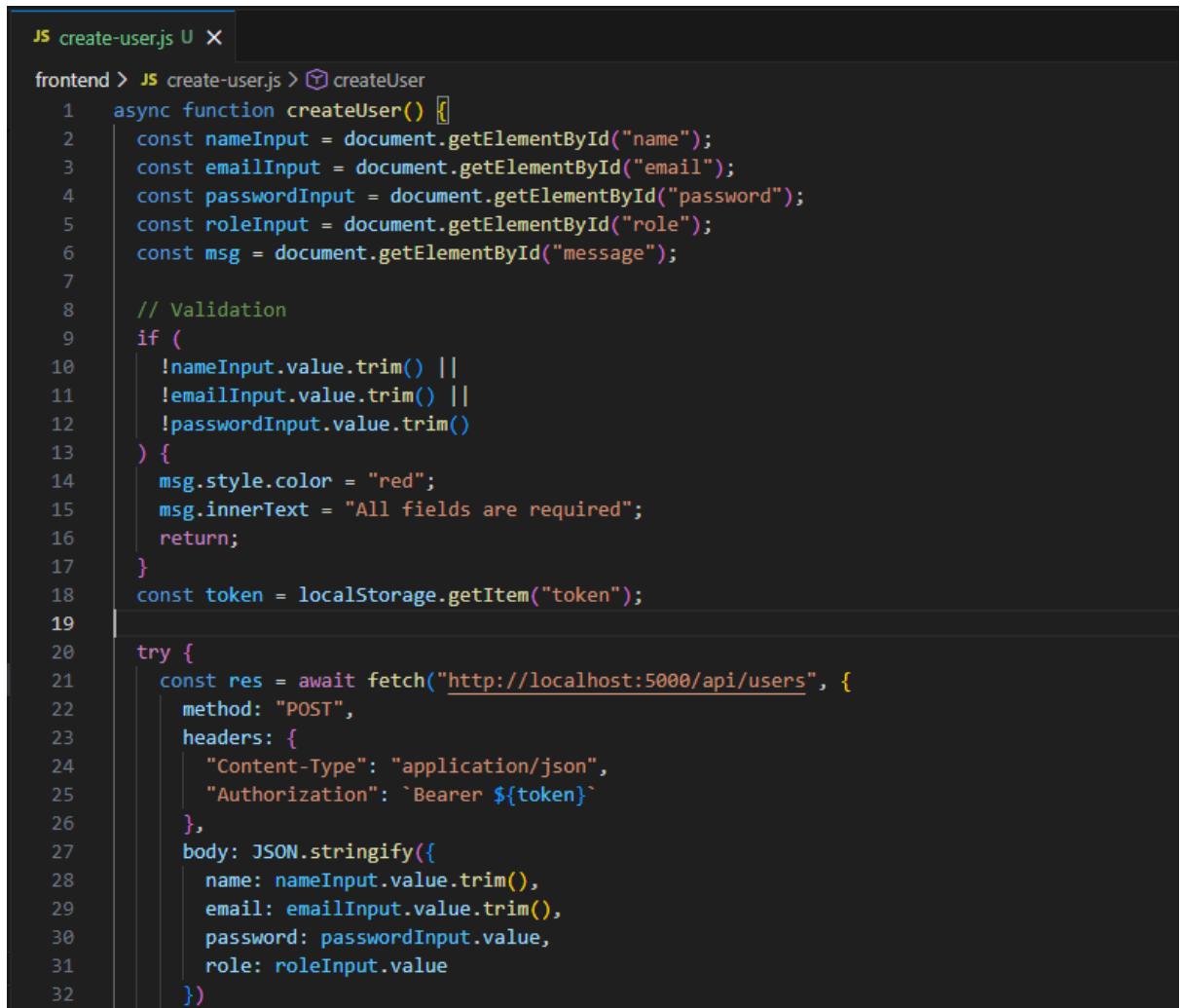
```
↳ create-user.html U X
frontend > ↳ create-user.html > ⏺ html > ⏺ body > ⏺ script
● 1  <!DOCTYPE html>
2  <html lang="en">
3
4  <head>
5    <meta charset="UTF-8">
6    <title>Create User | Admin</title>
7    <link rel="stylesheet" href="css/styles.css">
8    <link rel="stylesheet" href="css/create-user.css">
9  </head>
10
11 <body>
12
13  <nav class="navbar">
14    <h2>Church Admin</h2>
15    <ul>
16      <li><a href="dashboard.html">Dashboard</a></li>
17      <li><a href="create-user.html">Create User</a></li>
18      <li><a href="users-report.html">Users Report</a></li>
19    </ul>
20  </nav>
21
22  <div class="form-container">
23    <h2>Create New User</h2>
24
25    <input id="name" placeholder="Full Name">
26    <input id="email" placeholder="Email">
27    <input id="password" type="password" placeholder="Password">
```

```

29   <select id="role">
30     <option value="STAFF">Staff</option>
31     <option value="PASTOR">Pastor</option>
32     <option value="TREASURER">Treasurer</option>
33     <option value="ADMIN">Admin</option>
34   </select>
35
36   <button type="button" onclick="createUser()">Create User</button>
37   <p id="message"></p>
38 </div>
39
40   <script src="authGuard.js"></script>
41   <script src="create-user.js"></script>
42
43 </body>
44
45 </html>

```

Below is the implemented create-user.js code to validate the input user data and post the data to the mongoDB database.



```

JS create-user.js U X
frontend > JS create-user.js > ⚡ createUser
1  async function createUser() {
2    const nameInput = document.getElementById("name");
3    const emailInput = document.getElementById("email");
4    const passwordInput = document.getElementById("password");
5    const roleInput = document.getElementById("role");
6    const msg = document.getElementById("message");
7
8    // Validation
9    if (
10      !nameInput.value.trim() ||
11      !emailInput.value.trim() ||
12      !passwordInput.value.trim()
13    ) {
14      msg.style.color = "red";
15      msg.innerText = "All fields are required";
16      return;
17    }
18    const token = localStorage.getItem("token");
19
20    try {
21      const res = await fetch("http://localhost:5000/api/users", {
22        method: "POST",
23        headers: {
24          "Content-Type": "application/json",
25          "Authorization": `Bearer ${token}`
26        },
27        body: JSON.stringify([
28          name: nameInput.value.trim(),
29          email: emailInput.value.trim(),
30          password: passwordInput.value,
31          role: roleInput.value
32        ])
33    }

```

```

33 });
34
35 const data = await res.json();
36
37 if (res.ok) {
38     msg.style.color = "green";
39     msg.innerText = data.message || "User created successfully";
40     nameInput.value = "";
41     emailInput.value = "";
42     passwordInput.value = "";
43     roleInput.value = "STAFF";
44 } else {
45     msg.style.color = "red";
46     msg.innerText = data.message || "Failed to create user";
47 }
48 } catch (err) {
49     msg.style.color = "red";
50     msg.innerText = "Server error";
51     console.error(err);
52 }
53 }

```

## 5.9(ii) users-report

Below is the users report. It is visible to the admin user, and displays all users of the system. The admin user is able to delete or edit a user by clicking the icons in the actions column.

Name	Email	Role	Created	Actions
SysAdmin	admin@church.com	ADMIN	2026-01-19	
Testing Test	testing@church.com	TREASURER	2026-01-20	
Someone Test	someone@church.com	STAFF	2026-01-20	
Peter Tosh	peter@church.com	TREASURER	2026-01-28	

On clicking the button [Export CSV], a csv file of the report is exported and downloaded to the downloads folder, where it can be opened in Microsoft excel and viewed.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
1	Name	Email	Role	Created														
2	SysAdmin	admin@church.com	ADMIN	2026/01/19														
3	Testing Test	testing@church.com	TREASURER	2026/01/20														
4	Someone Test	someone@church.com	STAFF	2026/01/20														
5	Peter Tosh	peter@church.com	TREASURER	2026/01/28														
6																		

Below is the implemented users-report.html code to display the users report, and export the report to a csv file after clicking the button [Export CSV].

```
④ users-report.html ✘
frontend > ④ users-report.html > ⏺ html > ⏺ body > ⏺ div.table-container > ⏺ h2
1   <!DOCTYPE html>
2   <html lang="en">
3
4   <head>
5     <meta charset="UTF-8">
6     <title>Users Report</title>
7     <script src="authGuard.js"></script>
8     <link rel="stylesheet" href="css/attReport.css">
9     <!-- Font Awesome -->
10    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.5.0/css/all.min.css">
11  </head>
12
13  <body>
14
15    <nav class="navbar">
16      <h2>Church Admin</h2>
17      <ul>
18        <li><a href="dashboard.html">Dashboard</a></li>
19        <li><a href="users-report.html">Users Report</a></li>
20      </ul>
21    </nav>
22
23    <div class="table-container">
24      <h2>System Users</h2>
25      <div class="filters">
26        <button onclick="exportCSV()">Export CSV</button>
27      </div>
28
29      <table>
30        <thead>
31          <tr>
32            <th>Name</th>
33            <th>Email</th>
34            <th>Role</th>
35            <th>Created</th>
36            <th>Actions</th>
37          </tr>
38        </thead>
39        <tbody id="usersTable"></tbody>
40      </table>
41    </div>
42
43    <script>
44      requireRole("ADMIN"); // ADMIN ONLY
45
46      const API_URL = "http://localhost:5000/api/users";
47      const token = localStorage.getItem("token");
48      let userData = [] // GLOBAL storage for CSV
49
50      async function loadUsers() {
51        try {
52          const res = await fetch(API_URL, {
53            headers: {
54              Authorization: `Bearer ${token}`
55            }
56          });
57          console.log("STATUS:", res.status);
58        } catch (error) {
59          console.error(error);
60        }
61      }
62
63      loadUsers();
64
65      const exportCSV = () => {
66        const csvContent = [
67          "Name,Email,Role,Created,Actions",
68          ...userData.map(user => `${user.name},${user.email},${user.role},${user.created},#`),
69        ].join("\n");
70
71        const blob = new Blob([csvContent], { type: "text/csv" });
72        const url = URL.createObjectURL(blob);
73
74        const link = document.createElement("a");
75        link.href = url;
76        link.download = "users.csv";
77        link.click();
78      };
79
80      document.querySelector("button").addEventListener("click", exportCSV);
81
82    </script>
83  
```

```

59      if (!res.ok) {
60        const errorText = await res.text();
61        console.error("API ERROR:", errorText);
62        alert("Failed to load users");
63        return;
64      }
65
66
67      const users = await res.json();
68      usersData = users; //store for CSV
69
70      const table = document.getElementById("usersTable");
71      table.innerHTML = "";
72
73      users.forEach(user => {
74        const row = document.createElement("tr");
75        row.innerHTML = `
76          <td>${user.name}</td>
77          <td>${user.email}</td>
78          <td>${user.role}</td>
79          <td>${user.createdAt.split("T")[0]}</td>
80          <td class="actions">
81            <i class="fa-solid fa-pen" onclick="editUser('${user._id}')"></i>
82            <i class="fa-solid fa-trash" onclick="deleteUser('${user._id}')"></i>
83          </td>
84        `;
85        table.appendChild(row);
86      });
87    } catch (err) {
88      console.error("FETCH ERROR:", err);
89      alert("Unexpected error loading users");
90    }
91  }
92
93  function editUser(id) {
94    window.location.href = `edit-user.html?id=${id}`;
95  }
96
97  async function deleteUser(id) {
98    if (!confirm("Delete this user?")) return;
99
100   const res = await fetch(`[${API_URL}]/${id}`, {
101     method: "DELETE",
102     headers: {
103       Authorization: `Bearer ${token}`
104     }
105   });
106
107   if (res.ok) {
108     loadUsers();
109   } else {
110     alert("Delete failed");
111   }
112 }
113
114 function editUser(id) {

```

```

● 115     window.location.href = `edit-user.html?id=${id}`;
116   }
117
118   ▼ async function deleteUser(id) {
119     if (!confirm("Delete this user?")) return;
120
121   ▼ const res = await fetch(`.${API_URL}/${id}`, {
122     method: "DELETE",
123     ▼ headers: {
124       Authorization: `Bearer ${token}`
125     }
126   });
127
128   ▼ if (res.ok) {
129     loadUsers();
130   } else {
131     alert("Delete failed");
132   }
133 }
134
135   ▼ function exportCSV() {
136     if (usersData.length === 0) {
137       alert("No data to export");
138       return;
139     }
140
141     let csv = "Name,Email,Role,Created\n";
142   ▼ usersData.forEach(u => {
143
144     csv += `${u.name},${u.email},${u.role},${u.createdAt.split("T")[0]}\n`;
145   });
146
147   const blob = new Blob([csv], { type: "text/csv" });
148   const a = document.createElement("a");
149   a.href = URL.createObjectURL(blob);
150   a.download = "users_report.csv";
151   a.click();
152
153   loadUsers();
154 </script>
155 </body>
156 </html>

```

### 5.9(iii) edit-user

Below is the edit-user screen, visible to the admin user. This screen is visible from the users-report by clicking the edit icon on the report.

The screenshot shows a modal dialog titled "Edit User". Inside the dialog, there are three input fields: one for the name containing "Peter Tosh", one for the email containing "peter@church.com", and a dropdown menu for the role showing "TREASURER". Below the inputs is a blue button labeled "Update User". The background of the page is a light grey, and at the top, there is a blue header bar with the text "Church Admin" on the left and "Dashboard Users Report" on the right.

Below is the implemented edit-user.html code to display the above screen, and update user details.

```
↳ edit-user.html U X
frontend > ↳ edit-user.html > ⏮ html
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <title>Edit User</title>
6      <script src="authGuard.js"></script>
7      <link rel="stylesheet" href="css/styles.css">
8  </head>
9
10 <body>
11     <nav class="navbar">
12         <h2>Church Admin</h2>
13         <ul>
14             <li><a href="dashboard.html">Dashboard</a></li>
15             <li><a href="users-report.html">Users Report</a></li>
16             <li><a href="#" onclick="logout()">Logout</a></li>
17         </ul>
18     </nav>
19
20     <div class="form-container">
21         <h2>Edit User</h2>
22
23         <input id="name" placeholder="Name">
24         <input id="email" placeholder="Email">
25
26         <select id="role">
27             <option value="ADMIN">ADMIN</option>
28             <option value="PASTOR">PASTOR</option>
29             <option value="TREASURER">TREASURER</option>
30             <option value="STAFF">STAFF</option>
31         </select>
32
33         <button onclick="updateUser()">Update User</button>
34
35         <p id="message"></p>
36     </div>
37
38     <script>
39         requireRole("ADMIN");
40
41         const API_URL = "http://localhost:5000/api/users";
42         const token = localStorage.getItem("token");
43         const params = new URLSearchParams(window.location.search);
44         const userId = params.get("id");
45
46         if (!userId) {
47             alert("No user ID provided");
48             window.location.href = "users-report.html";
49         }
50
51         async function loadUser() {
52             const res = await fetch(`${API_URL}/${userId}`, {
```

```

53     headers: { Authorization: `Bearer ${token}` }
54   });
55
56   if (!res.ok) {
57     alert("Failed to load user");
58     return;
59   }
60
61   const user = await res.json();
62   document.getElementById("name").value = user.name;
63   document.getElementById("email").value = user.email;
64   document.getElementById("role").value = user.role;
65 }
66
67 async function updateUser() {
68   const res = await fetch(`${API_URL}/${userId}`, {
69     method: "PUT",
70     headers: {
71       "Content-Type": "application/json",
72       Authorization: `Bearer ${token}`
73     },
74     body: JSON.stringify({
75       name: document.getElementById("name").value,
76       email: document.getElementById("email").value,
77       role: document.getElementById("role").value
78     })
79   });
80
81   if (res.ok) {
82     alert("User updated successfully");
83     window.location.href = "users-report.html";
84   } else {
85     alert("Update failed");
86   }
87 }
88 loadUser();
89 </script>
90 </body>
91
92 </html>

```

## 5.10 Member Management

Implemented modules include:

- add-member
- members-report
- edit-member

These features enable CRUD operations on church member records.

## 5.10(i) add-member

Below is the add-member screen, visible to the admin user.

The screenshot shows a 'Church Admin' application interface. At the top, there is a blue header bar with the text 'Church Admin' on the left and 'Dashboard' 'Add Member' 'Members Report' on the right. Below the header is a white content area titled 'Add Member'. The form contains the following fields:

- First Name (text input)
- Last Name (text input)
- Select Gender (dropdown menu)
- Phone Number (text input)
- Email Address (text input)
- Address (text input)
- Date of Birth (text input with placeholder 'yyyy / mm / dd' and a calendar icon)
- Active (dropdown menu)

At the bottom of the form is a blue 'Save' button.

Below is the implemented add-member.html code to display the above screen, and post the entered details to the mongoDB database.

```
add-member.html U X
frontend > add-member.html > html
1  <!DOCTYPE html>
2  <html lang="en">
3
4  <head>
5    <meta charset="UTF-8">
6    <title>Add Member</title>
7    <link rel="stylesheet" href="css/styles.css">
8  </head>
9
10 <body>
11
12  <nav class="navbar">
13    <h2>Church Admin</h2>
14    <ul>
15      <li><a href="dashboard.html">Dashboard</a></li>
16      <li><a href="add-member.html">Add Member</a></li>
17      <li><a href="members.html">Members Report</a></li>
18    </ul>
19  </nav>
20
21  <div class="form-container">
22
23    <h2>Add Member</h2>
24
25    <input id="firstName" placeholder="First Name">
26    <input id="lastName" placeholder="Last Name">
```

```

27      <!-- Gender Dropdown -->
28      <select id="gender">
29          <option value="">Select Gender</option>
30          <option value="Male">Male</option>
31          <option value="Female">Female</option>
32          <option value="Other">Other</option>
33      </select>
34
35
36      <input
37          id="phone"
38          placeholder="Phone Number"
39          inputmode="numeric"
40          pattern="[0-9]*"
41          oninput="this.value = this.value.replace(/[^0-9]/g, '')"
42      />
43
44      <input id="email" placeholder="Email Address">
45      <input id="address" placeholder="Address">
46
47      <!-- Join Date Calendar -->
48      <input type="date" id="joinDate">
49
50      <!-- Status Dropdown -->
51      <select id="status">
52          <option value="Active" selected>Active</option>
53          <option value="Inactive">Inactive</option>
54          <option value="Transferred">Transferred</option>
55      </select>
56
57      <button onclick="addMember()">Save</button>
58
59      <p id="message"></p>
60
61  </div>
62
63  <script>
64      const token = localStorage.getItem("token");
65
66      async function addMember() {
67          const memberData = {
68              firstName: document.getElementById("firstName").value,
69              lastName: document.getElementById("lastName").value,
70              gender: document.getElementById("gender").value,
71              phone: document.getElementById("phone").value,
72              email: document.getElementById("email").value,
73              address: document.getElementById("address").value,
74              joinDate: document.getElementById("joinDate").value,
75              status: document.getElementById("status").value || "Active"
76          };
77
78          try {
79              const response = await fetch("http://localhost:5000/api/members", {
80                  method: "POST",
81                  headers: { "Content-Type": "application/json",
82                      "Authorization": `Bearer ${token}` },
83                  body: JSON.stringify(memberData)

```

```

84  });
85  const data = await response.json();
86
87  if (!response.ok) {
88    throw new Error(data.message || "Failed to save member");
89  }
90
91  // alert("Member saved successfully!");
92  document.getElementById("message").innerText = "Member added successfully!";
93
94  document.getElementById("firstName").value = "";
95  document.getElementById("lastName").value = "";
96  document.getElementById("gender").value = "";
97  document.getElementById("phone").value = "";
98  document.getElementById("email").value = "";
99  document.getElementById("address").value = "";
100 document.getElementById("joinDate").value = "";
101 document.getElementById("status").value || "Active";
102 } catch (error) {
103   document.getElementById("message").innerText =
104     "Error: " + error.message;
105 }
106 }
107 </script>
108 </body>
109 </html>

```

## 5.10(ii) members-report

Below is the members report. It is visible to the admin user, and displays all members of the church. The admin user is able to delete or edit any member details by clicking the icons in the actions column.

First Name	Last Name	Gender	Phone	Email	Address	Join Date	Status	Actions
John	Doris	Male	1234567890	john.doe@example.com	24 140 Street Surrey	2026-01-08	Active	
Jack	Last	Male	123456789	jlast@gmail.com	23 160 street surrey	2026-01-08	Active	

On clicking the button [Export CSV], a csv file of the report is exported and downloaded to the downloads folder, where it can be opened in Microsoft excel and viewed.

A screenshot of Microsoft Excel showing a table of member data. The table has columns for First Name, Last Name, Gender, Phone, Email, Address, Join Date, and Status. The data shows two rows: John Doe (Male) and Jack Last (Male). The Excel ribbon is visible at the top.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
1	First Name	Last Name	Gender	Phone	Email	Address	Join Date	Status										
2	John	Doris	Male	1234567890	john.doe@example.com	24 140 Street Surrey	2026/01/08	Active										
3	Jack	Last	Male	1234567899	jlast@gmail.com	23 160 street surrey	2026/01/08	Active										
4																		
5																		
6																		

Below is the implemented members.html code to display the members report, and export the report to a csv file after clicking the button [Export CSV].

```
<> members.html M X
frontend > <> members.html > html
1   <!DOCTYPE html>
2   <html lang="en">
3
4   <head>
5     <meta charset="UTF-8">
6     <title>Members</title>
7     <script src="authGuard.js"></script>
8     <link rel="stylesheet" href="css/attReport.css">
9
10    <!-- Font Awesome Icons -->
11    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.5.0/css/all.min.css" />
12
13  </head>
14
15  <body>
16    <!-- Navigation -->
17    <nav class="navbar">
18      <h2>Church Admin</h2>
19      <ul>
20        <li><a href="dashboard.html">Dashboard</a></li>
21        <li><a href="members.html">Members-Report</a></li>
22      </ul>
23    </nav>
24
25    <div class="table-container">
26      <h2>Church Members</h2>
27      <div class="filters">
28        <button onclick="exportCSV()">Export CSV</button>
29      </div>
30
31      <table>
32        <thead>
```

```

33    <tr>
34      <th>First Name</th>
35      <th>Last Name</th>
36      <th>Gender</th>
37      <th>Phone</th>
38      <th>Email</th>
39      <th>Address</th>
40      <th>Join Date</th>
41      <th>Status</th>
42      <th>Actions</th>
43    </tr>
44  </thead>
45  <tbody id="membersTable">
46    <!-- Rows inserted here -->
47  </tbody>
48 </table>
49 </div>
50
51 <script>
52   const API_URL = "http://localhost:5000/api/members";
53   const token = localStorage.getItem("token");
54   let membersData = [];// GLOBAL storage for CSV
55
56 // Load members
57 async function loadMembers() {
58   const token = localStorage.getItem("token");
59   const response = await fetch(API_URL, {
60     headers: {
61       "Authorization": `Bearer ${token}`
62     }
63   });
64   const members = await response.json();
65   membersData = members;// store for CSV
66
67   const table = document.getElementById("membersTable");
68   table.innerHTML = "";
69
70   members.forEach(member => {
71     const row = document.createElement("tr");
72
73     row.innerHTML = `
74       <td>${member.firstName}</td>
75       <td>${member.lastName}</td>
76       <td>${member.gender || ""}</td>
77       <td>${member.phone || ""}</td>
78       <td>${member.email || ""}</td>
79       <td>${member.address || ""}</td>
80       <td>${member.joinDate ? member.joinDate.split("T")[0] : ""}</td>
81       <td>${member.status}</td>
82       <td class="actions">
83         <i class="fa-solid fa-pen" onclick="editMember('${member._id}')"></i>
84         <i class="fa-solid fa-trash" onclick="deleteMember('${member._id}')"></i>
85       </td>
86     `;
87
88     table.appendChild(row);
89   });
90 }
91
92 // Edit member (placeholder)
93 function editMember(id) {
94   alert("Edit member: " + membersData.find(m => m._id === id)?.firstName
95   + " " + membersData.find(m => m._id === id)?.lastName);
96   window.location.href = `edit-member.html?id=${id}`;
97 }

```

```

98  // Delete member
99
100 async function deleteMember(id) {
101   if (!confirm("Are you sure you want to delete this member?")) return;
102
103   const token = localStorage.getItem("token");
104   const response = await fetch(` ${API_URL}/ ${id}` , {
105     method: "DELETE",
106     headers: {
107       "Authorization": `Bearer ${token}`
108     }
109   });
110
111   if (!response.ok) {
112     const error = await response.text();
113     console.error("Delete Failed:", error);
114     alert("Failed to Delete Member");
115     return;
116   }
117
118   alert("Member Deleted Successfully");
119   loadMembers();
120 }
121
122 function exportCSV() {
123   if (membersData.length === 0) {
124     alert("No data to export");
125     return;
126   }
127
128   let csv = "First Name,Last Name,Gender,Phone,Email,Address,Join Date,Status\n";
129   membersData.forEach(m => {
130     csv += `${m.firstName},${m.lastName},${m.gender || ""},${m.phone || ""},${m.email || ""}, ${m.address || ""}, ${m.joinDate ? m.joinDate.split("T")[0] : ""}, ${m.status}\n`;
131   });
132
133   const blob = new Blob([csv], { type: "text/csv" });
134   const a = document.createElement("a");
135   a.href = URL.createObjectURL(blob);
136   a.download = "members_report.csv";
137   a.click();
138 }
139
140 loadMembers();
141 </script>
142
143 </body>
144
145 </html>

```

### 5.10(iii) edit-member

Below is the edit-member screen, visible to the admin user. This screen is visible from the members-report by clicking the edit icon on the report.

### Edit Member

John

Doris

Male

1234567890

john.doe@example.com

24 140 Street Surrey

2026/01/08

Active

[Update Member](#)

Below is the implemented edit-member.html code to display the above screen, and update member details.

```
edit-member.html U X
frontend > edit-member.html > html > body > script
1  <!DOCTYPE html>
2  <html lang="en">
3
4  <head>
5    <meta charset="UTF-8">
6    <title>Edit Member</title>
7    <link rel="stylesheet" href="css/styles.css">
8    <script src="authGuard.js"></script>
9  </head>
10
11 <body>
12
13  <nav class="navbar">
14    <h2>Church Admin</h2>
15    <ul>
16      <li><a href="dashboard.html">Dashboard</a></li>
17      <li><a href="members.html">Members-Report</a></li>
18    </ul>
19  </nav>
20
21  <div class="form-container">
22    <h2>Edit Member</h2>
23
24    <form id="editMemberForm">
25
26      <input id="firstName" placeholder="First Name" required>
27      <input id="lastName" placeholder="Last Name" required>
28
29      <!-- Gender Dropdown -->
30      <select id="gender">
31        <option value="">Select Gender</option>
32        <option value="Male">Male</option>
```

```

33 |     <option value="Female">Female</option>
34 |     <option value="Other">Other</option>
35 |   </select>
36 |
37 |   <!-- <input id="phone" placeholder="Phone Number"> -->
38 |   <input id="phone" placeholder="Phone Number" inputmode="numeric" pattern="[0-9]*"
39 |     oninput="this.value = this.value.replace(/[^0-9]/g, '')" />
40 |   <input id="email" placeholder="Email Address">
41 |   <input id="address" placeholder="Address">
42 |
43 |   <!-- Join Date Calendar -->
44 |   <input id="joinDate" type="date">
45 |
46 |   <!-- Status Dropdown -->
47 |   <select id="status">
48 |     <option value="Active" selected>Active</option>
49 |     <option value="Inactive">Inactive</option>
50 |     <option value="Transferred">Transferred</option>
51 |   </select>
52 |
53 |   <button type="submit">Update Member</button>
54 |
55 | </form>
56 |
57 | <p id="message"></p>
58 | </div>
59 |
60 | <script>
61 |   const API_URL = "http://localhost:5000/api/members";
62 |   const headers = { Authorization: `Bearer ${window.authToken}` };
63 |   const params = new URLSearchParams(window.location.search);
64 |   const memberId = params.get("id");
65 |
66 |   if (!memberId) {
67 |     alert("No member ID provided");
68 |     window.location.href = "members.html";
69 |   }
70 | // Load member data
71 | async function loadMember() {
72 |   try {
73 |     console.log("TOKEN:", window.authToken);
74 |     const res = await fetch(`${API_URL}/${memberId}`, {
75 |       headers: {
76 |         Authorization: `Bearer ${window.authToken}`
77 |       }
78 |     });
79 |
80 |     console.log("STATUS:", res.status);
81 |
82 |     if (!res.ok) {
83 |       const errText = await res.text();
84 |       console.error("API ERROR:", errText);
85 |       throw new Error(`Failed to load member (${res.status})`);
86 |     }
87 |
88 |     const member = await res.json();

```

```

89    |     console.log("MEMBER DATA:", member);
90
91    |     document.getElementById("firstName").value = member.firstName || "";
92    |     document.getElementById("lastName").value = member.lastName || "";
93    |     document.getElementById("gender").value = member.gender || "";
94    |     document.getElementById("phone").value = member.phone || "";
95    |     document.getElementById("email").value = member.email || "";
96    |     document.getElementById("address").value = member.address || "";
97    |     document.getElementById("joinDate").value =
98    |       member.joinDate ? member.joinDate.split("T")[0] : "";
99    |     document.getElementById("status").value = member.status;
100   | } catch (err) {
101   |   alert(err.message);
102   | }
103   |
104
105   // Update member
106   document.getElementById("editMemberForm").addEventListener("submit", async e => {
107     e.preventDefault();
108     console.log("UPDATE SUBMITTED");
109
110   const updatedMember = [
111     |   firstName: document.getElementById("firstName").value,
112     |   lastName: document.getElementById("lastName").value,
113     |   gender: document.getElementById("gender").value,
114     |   phone: document.getElementById("phone").value,
115     |   email: document.getElementById("email").value,
116     |   address: document.getElementById("address").value,
117     |   joinDate: document.getElementById("joinDate").value,
118     |   status: document.getElementById("status").value
119   ];
120
121   console.log("Payload:", updatedMember);
122
123   const res = await fetch(` ${API_URL} / ${memberId} `, {
124     method: "PUT",
125     headers: [
126       |   "Content-Type": "application/json",
127       |   "Authorization": `Bearer ${window.authToken}`
128     ],
129     body: JSON.stringify(updatedMember)
130   });
131
132   if (res.ok) {
133     alert("Member updated successfully");
134     window.location.href = "members.html";
135   } else {
136     alert("Update failed");
137   }
138   });
139   loadMember();
140
141 </script>
142 </body>
143 </html>
```

## 5.11 Dashboard (Frontend)

A centralized dashboard was created using partial templates for navigation and layout consistency.

Below is the dashboard.html code that displays the dashboard of the system.

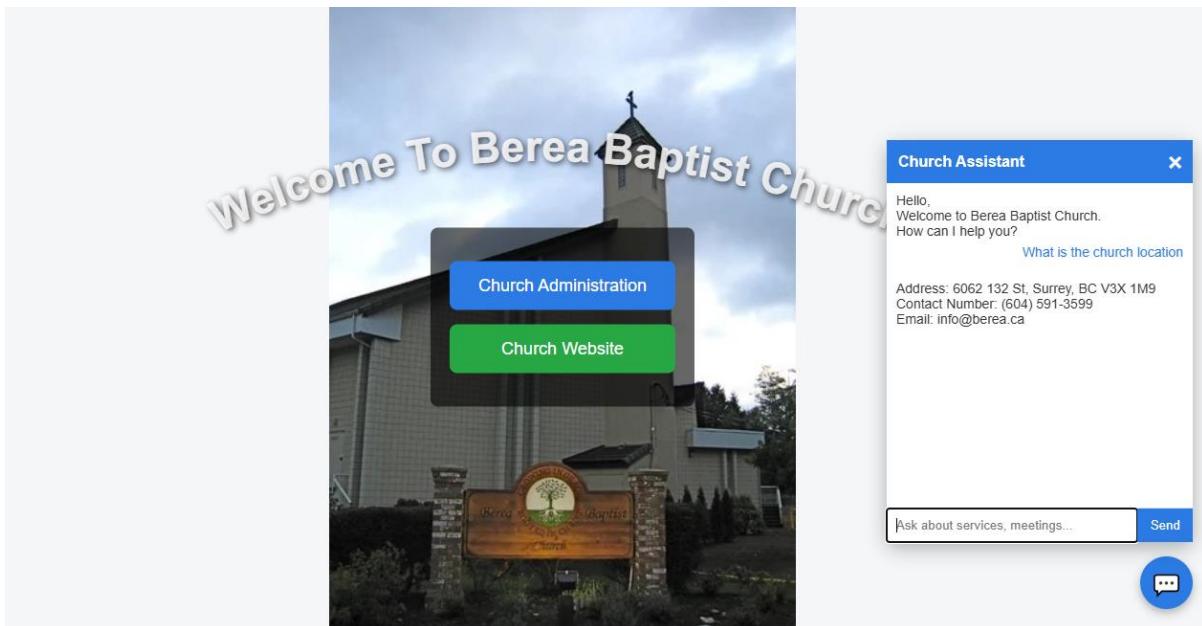
```
↳ dashboard.html M X
frontend > ↳ dashboard.html > ⏺ html
1   <!DOCTYPE html>
2   <html lang="en">
3
4   <head>
5     <meta charset="UTF-8">
6     <title>Church Admin Dashboard</title>
7     <link rel="stylesheet" href="css/styles.css">
8     <link rel="stylesheet" href="css/chatbot.css">
9   </head>
10
11  <div id="chatbot-container"></div>
12
13  <script>
14    fetch("partials/chatbot.html")
15      .then(res => res.text())
16      .then(html => {
17        document.getElementById("chatbot-container").innerHTML = html;
18
19        // Load chatbot.js AFTER HTML exists
20        const script = document.createElement("script");
21        script.src = "chatbot.js";
22        script.onload = () => {
23          | initChatbot();
24        };
25        document.body.appendChild(script);
26      });
27  </script>
28
29  <body>
30    <!-- Navigation Bar -->
31    <nav class="navbar">
32      <h2>Church Admin</h2>
33
34      <ul>
35        <li id="menu-dashboard"><a href="dashboard.html">Dashboard</a></li>
36        <li id="menu-users"><a href="create-user.html">User-Management</a></li>
37        <li id="menu-members"><a href="add-member.html">Member-Management</a></li>
38
39        <!-- Logout always visible -->
40        <li id="menu-logout">
41          <a href="#" onclick="logout()">Logout</a>
42        </li>
43      </ul>
44    </nav>
45
46    <!-- Hero Section -->
47    <section class="hero">
48      
49      <div class="hero-text">
50        <h1>Church Administration System</h1>
51        <p>Manage members, attendance, and finances efficiently</p>
52      </div>
53    </section>
```

```
53  <script src="authGuard.js"></script>
54  <script src="roles.js"></script>
55  <script src="roleMenu.js"></script>
56  <script>
57  |   applyRoleMenu();
58  </script>
59  </body>
60
61  </html>
```

## 5.12 Chatbot Module (Frontend)

A chatbot interface was implemented as a proof-of-concept to support future AI-assisted features.

Below is a screenshot of the welcome screen with the chatbot being displayed on the bottom right corner of the screen.



Below is the chatbot.html code that displays the chatbot on the screen.

```
chatbot.html

frontend > partials > chatbot.html > div#chatbot-box
  1  <!-- Chatbot Widget -->
  2  <div id="chatbot-toggle">✖</div>
  3
  4  <div id="chatbot-box">
  5    <div id="chatbot-header">
  6      Church Assistant
  7      <span id="chatbot-close">✖</span>
  8    </div>
  9
 10   <div id="chatbot-messages"></div>
 11
 12   <div id="chatbot-input">
 13     <input id="chatInput" placeholder="Ask about services, meetings..." />
 14     <button id="chatSend">Send</button>
 15   </div>
 16 </div>
```

Below is the chatbotController.js code that contains replies from the chatbot when interacting with a user.

```
JS chatbotController.js U X
backend > controllers > JS chatbotController.js > chatbotReply > chatbotReply
1 exports.chatbotReply = (req, res) => {
2   const message = req.body.message.toLowerCase();
3
4   let reply = "Sorry, I didn't understand that. Please ask about service times, meetings, or groups.";
5
6   if (message.includes("service")) {
7     reply = `

8 Sunday Service: 10:00 AM - 11:30 AM
9 Wednesday Prayer: 6:00 PM - 7:30 PM
10 `;
11 }
12
13 else if (message.includes("location")) {
14   reply = `

15 Address: 6062 132 St, Surrey, BC V3X 1M9
16 Contact Number: (604) 591-3599
17 Email: info@berea.ca
18 `;
19 }
20
21 else if (message.includes("meeting")) {
22   reply = `

23 Leadership Meeting: First Saturday - 10:00 AM
24 Youth Meeting: Friday - 5:00 PM
25 Women's Fellowship: Thursday - 4:00 PM
26 `;
27 }
28
```

```

29  |   else if (message.includes("group")) {
30  |     reply = ` 
31  |       Church Groups:
32  |       - Youth Ministry
33  |       - Women's Fellowship
34  |       - Men's Fellowship
35  |       - Choir
36  |       - Ushering Team
37  |     `;
38  |   }
39
40   res.json({ reply });
41 }
42

```

## 5.13 Application Core / Server Configuration Module

### server.js

The server.js file serves as the main entry point of the backend application. It is responsible for initializing the Express server, configuring middleware, establishing the database connection, and registering all application routes. This file coordinates the interaction between different system modules such as authentication, member management, attendance tracking, financial records, and the chatbot feature.

The file ensures that the application is properly configured before handling client requests and starts the HTTP server to listen for incoming connections.

#### **Responsibilities Include :-**

- Initializes the Express application
- Configures middleware (JSON parsing, CORS, authentication middleware)
- Connects to the MongoDB database
- Registers route modules (users, members, attendance, finance, chatbot)
- Handles global error configuration
- Starts the backend server on a specified port

Below is the server.js code that is the core backend file of the system.

```

JS server.js M X
backend > JS server.js > ...
1  require("dotenv").config();
2  const express = require("express");
3  const path = require("path");
4  const cors = require("cors");
5  const connectDB = require("./config/db");
6  |
7  const app = express();
8
9  // Connect to MongoDB
10 connectDB();
11
12 // Middleware
13 app.use(cors());
14 app.use(express.json());
15
16 // Routes
17 app.use("/api/members", require("./routes/members"));
18 app.use("/api/chatbot", require("./routes/chatbot"));
19 app.use("/api/auth", require("./routes/auth"));
20 app.use("/api/users", require("./routes/users"));
21
22 // Default route → welcome.html
23 app.get("/", (req, res) => {
24   | res.sendFile(path.join(__dirname, "../frontend/welcome.html"));
25 });
26
27 // Serve frontend static files
28 app.use(express.static(path.join(__dirname, "../frontend")));
29
30 const open = require("open").default;
31
32 const PORT = process.env.PORT || 5000;
33 app.listen(PORT, () => {
34   | console.log(`Server running on port ${PORT}`);
35   | open(`http://localhost:${PORT}`);
36 });

```

## 6. AI Use Section

AI Tool Name	Version / Account Type	Specific Use	Value Added
ChatGPT	GPT-5.2 – Free	Code clarification, architecture guidance.	Manual integration, debugging, and optimization.

## Appendix: Prompt History

(All AI prompts and responses used during development have been documented and stored in the appendix.)

## 7. Work Date / Hours Logs

### Work Log Table

Date	Task Description	Hours
Feb 11	Implemented controllers for users and members.	1.5
Feb 12	Created models and schemas for MongoDB.	2
Feb 13	Developed authentication middleware.	1.5
Feb 14	Implemented add-member and edit-member modules.	2
Feb 15	Developed dashboard and partial templates.	1.5
Feb 16	Implemented users-report and members-report modules.	2
Feb 17	Integrated authGuard across protected pages.	1.5
Feb 18	Developed edit-user and create-user modules.	2
Feb 19	Implemented chatbot proof-of-concept.	1
Feb 20	Bug fixes and role-based access testing.	1.5
Feb 21	UI refinements and layout consistency.	1
Feb 22	Code refactoring and documentation.	1.5
Feb 23	Repository cleanup and final commits.	1

*Note: Logs will be updated daily to reflect actual work completed.*

## **8. Closing and References**

### **Acknowledgements**

Acknowledgement is given to course materials and online documentation that supported this research and development.

### **References**

- OWASP Foundation. (2023). *Web Application Security Guidelines*.
- MongoDB Documentation. (2024)
- Express.js Documentation. (2024).
- JWT.io Documentation

## **Appendix**

### **AI Prompt History**

- "Explain JWT authentication with middleware in Express"
- "How to structure MVC architecture in Node.js"

## **9. Conclusion**

This proposal outlines a structured approach to developing a full-stack JavaScript-based church administration system that modernizes record-keeping, improves efficiency, and supports data-driven leadership through scalable web technologies, while providing a foundation for future system enhancements.