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

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)

Phase	Task Description	Start Date	End Date	Key Deliverables
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

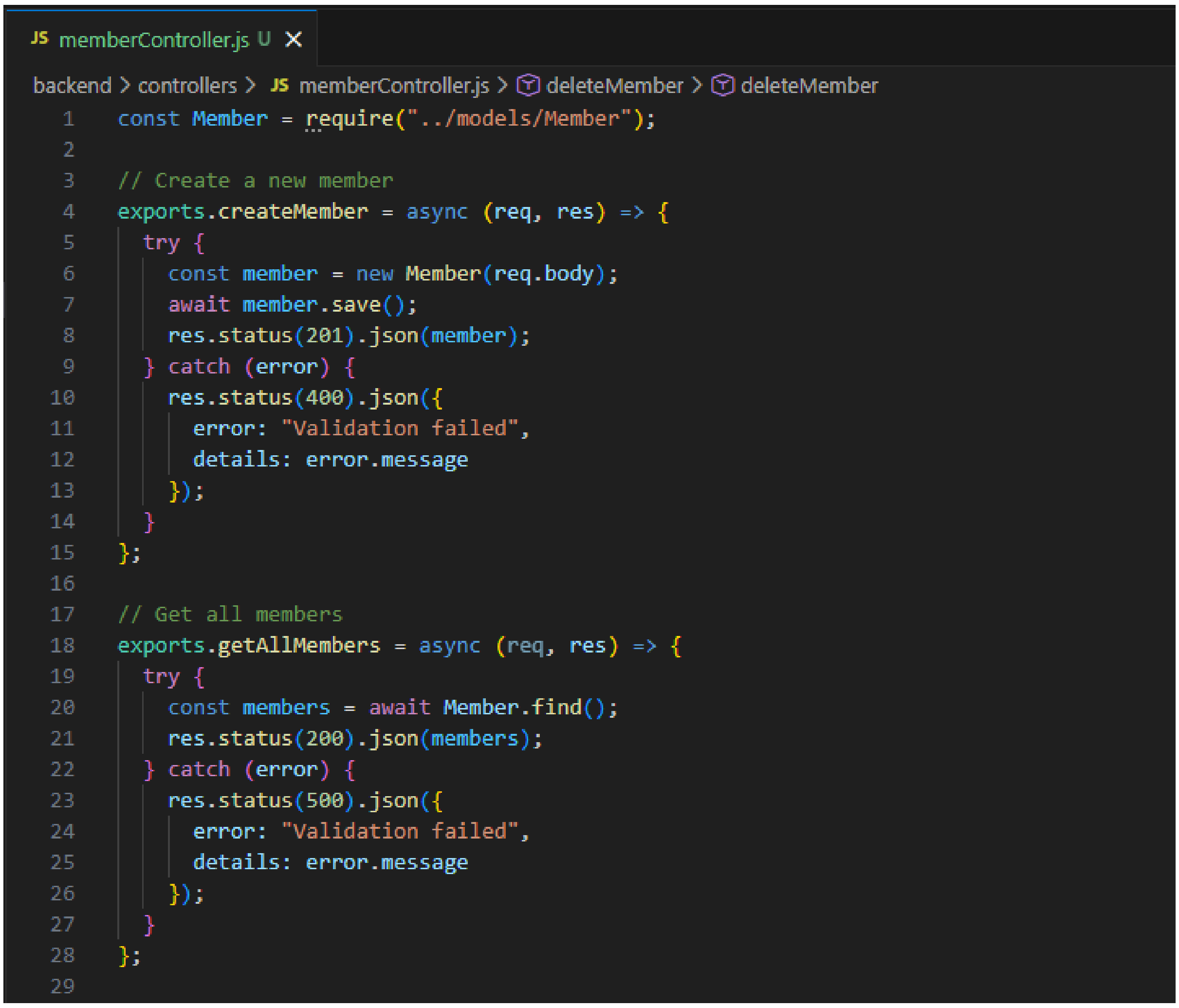
Key Functions:

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

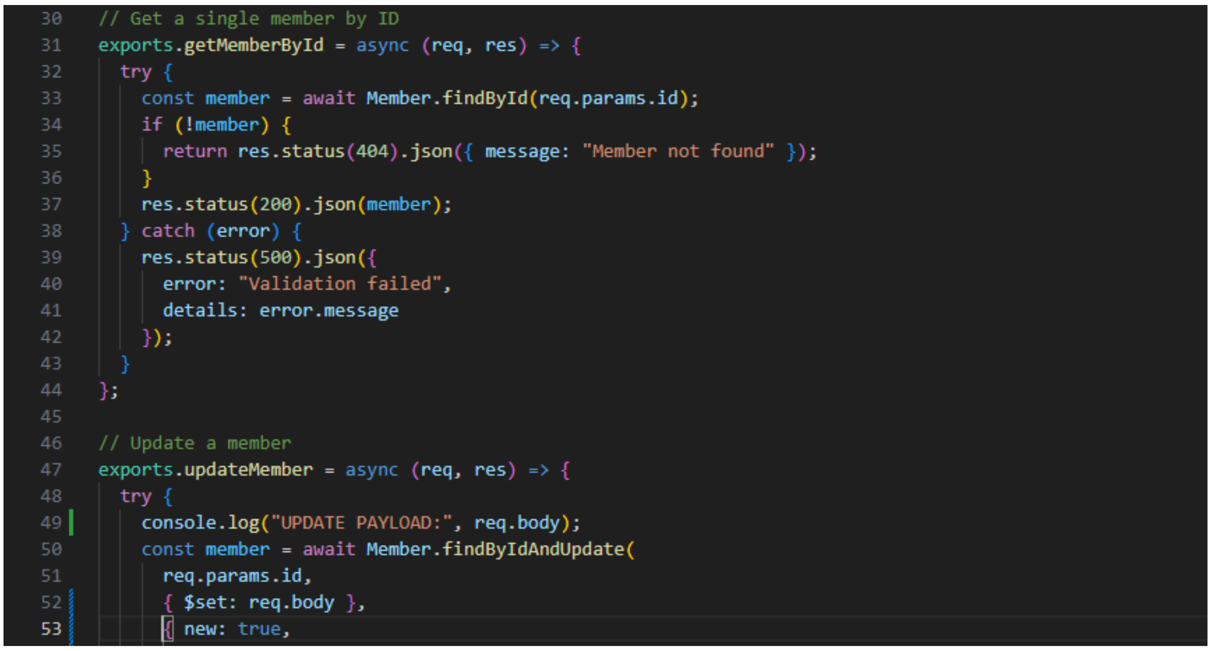
Used By:

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

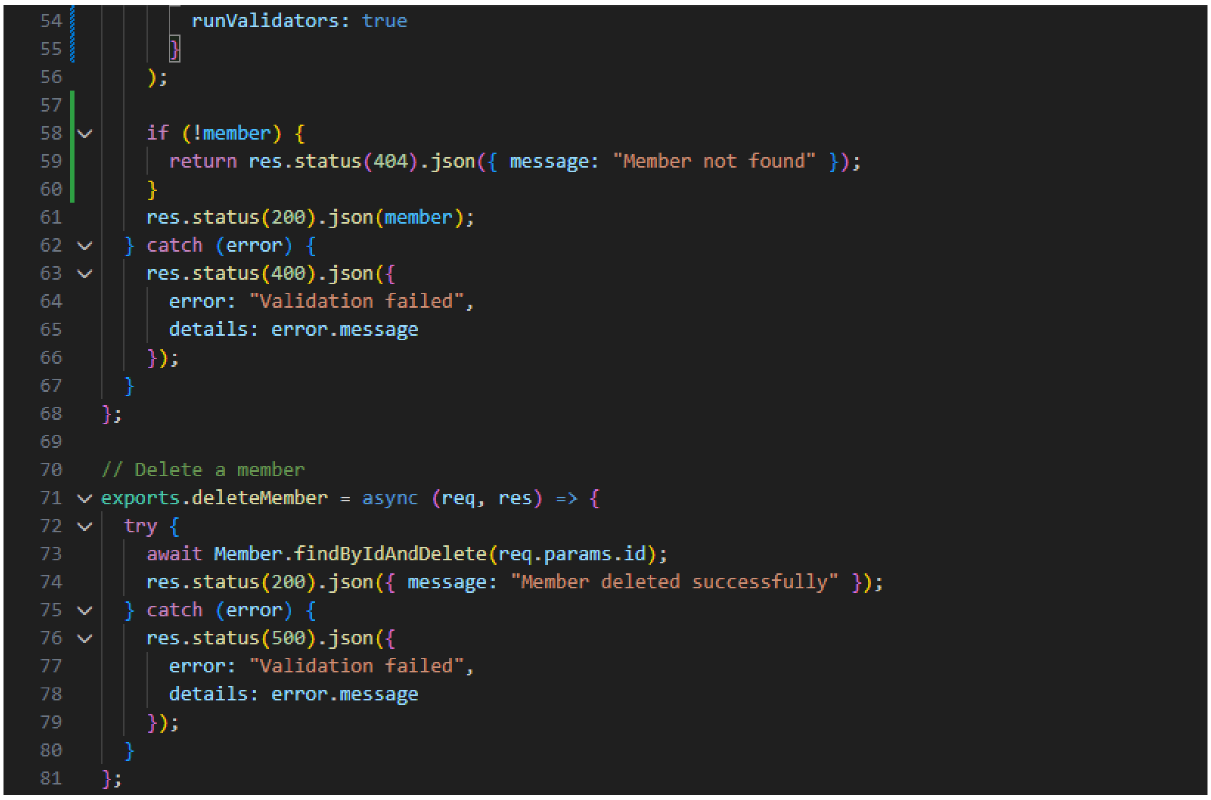
Below is the implemented memberController.js code (screenshot).


JS memberController.js U ✕
backend > controllers > JS memberController.js > deleteMember > deleteMember

```javascript
const Member = require("../models/Member");

// Create a new member
exports.createMember = async (req, res) => {
  try {
    const member = new Member(req.body);
    await member.save();
    res.status(201).json(member);
  } catch (error) {
    res.status(400).json({
      error: "Validation failed",
      details: error.message
    });
  }
};

// Get all members
exports.getAllMembers = async (req, res) => {
  try {
    const members = await Member.find();
    res.status(200).json(members);
  } catch (error) {
    res.status(500).json({
      error: "Validation failed",
      details: error.message
    });
  }
};

```



```javascript
// Get a single member by ID
exports.getMemberById = async (req, res) => {
  try {
    const member = await Member.findById(req.params.id);
    if (!member) {
      return res.status(404).json({ message: "Member not found" });
    }
    res.status(200).json(member);
  } catch (error) {
    res.status(500).json({
      error: "Validation failed",
      details: error.message
    });
  }
};

// Update a member
exports.updateMember = async (req, res) => {
  try {
    console.log("UPDATE PAYLOAD:", req.body);
    const member = await Member.findByIdAndUpdate(
      req.params.id,
      { $set: req.body },
      { new: true,
```


```
        runValidators: true
      }
    );

    if (!member) {
      return res.status(404).json({ message: "Member not found" });
    }
    res.status(200).json(member);
  } catch (error) {
    res.status(400).json({
      error: "Validation failed",
      details: error.message
    });
  }
};

// Delete a member
exports.deleteMember = async (req, res) => {
  try {
    await Member.findByIdAndDelete(req.params.id);
    res.status(200).json({ message: "Member deleted successfully" });
  } catch (error) {
    res.status(500).json({
      error: "Validation failed",
      details: error.message
    });
  }
};
```

userController.js

Responsibilities:

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

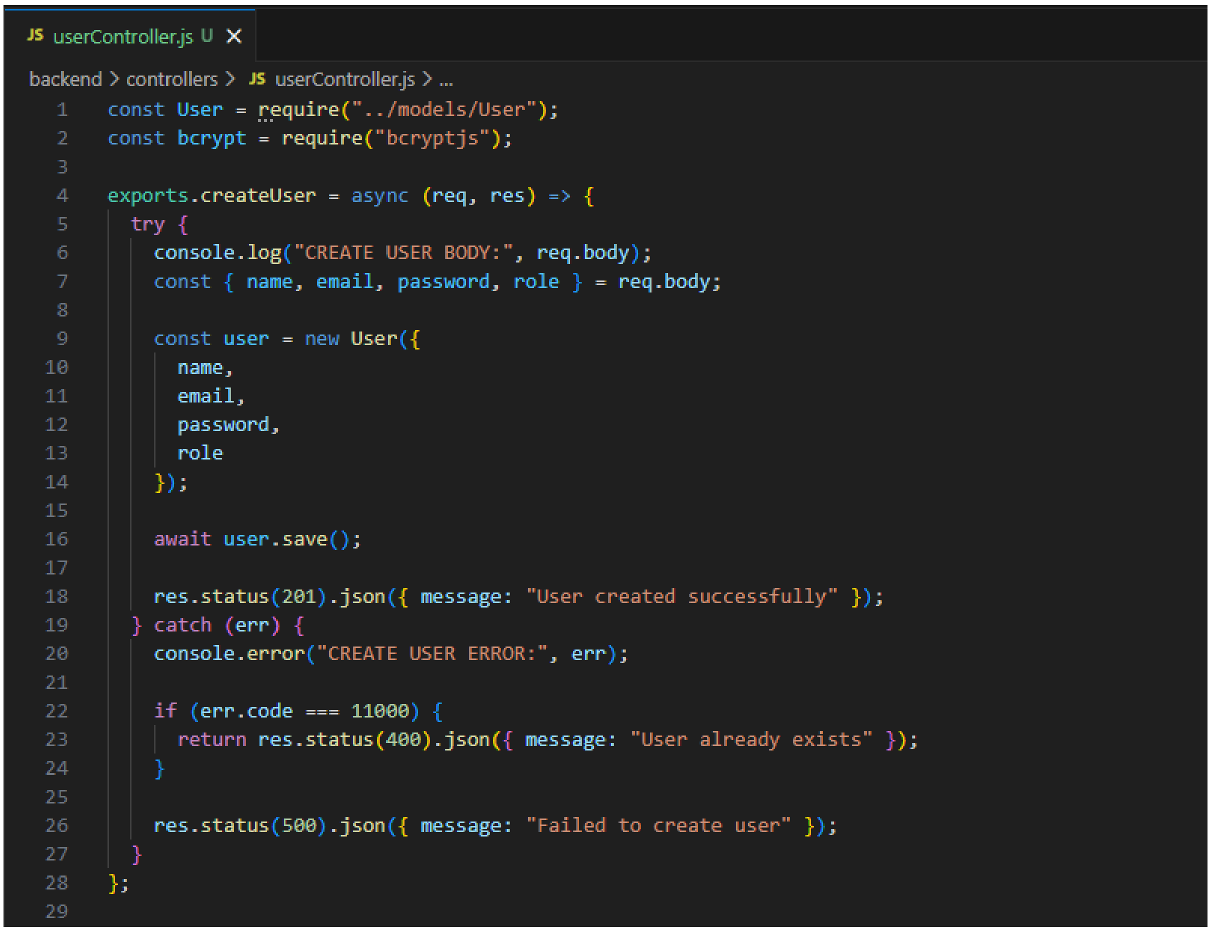
Key Functions:

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

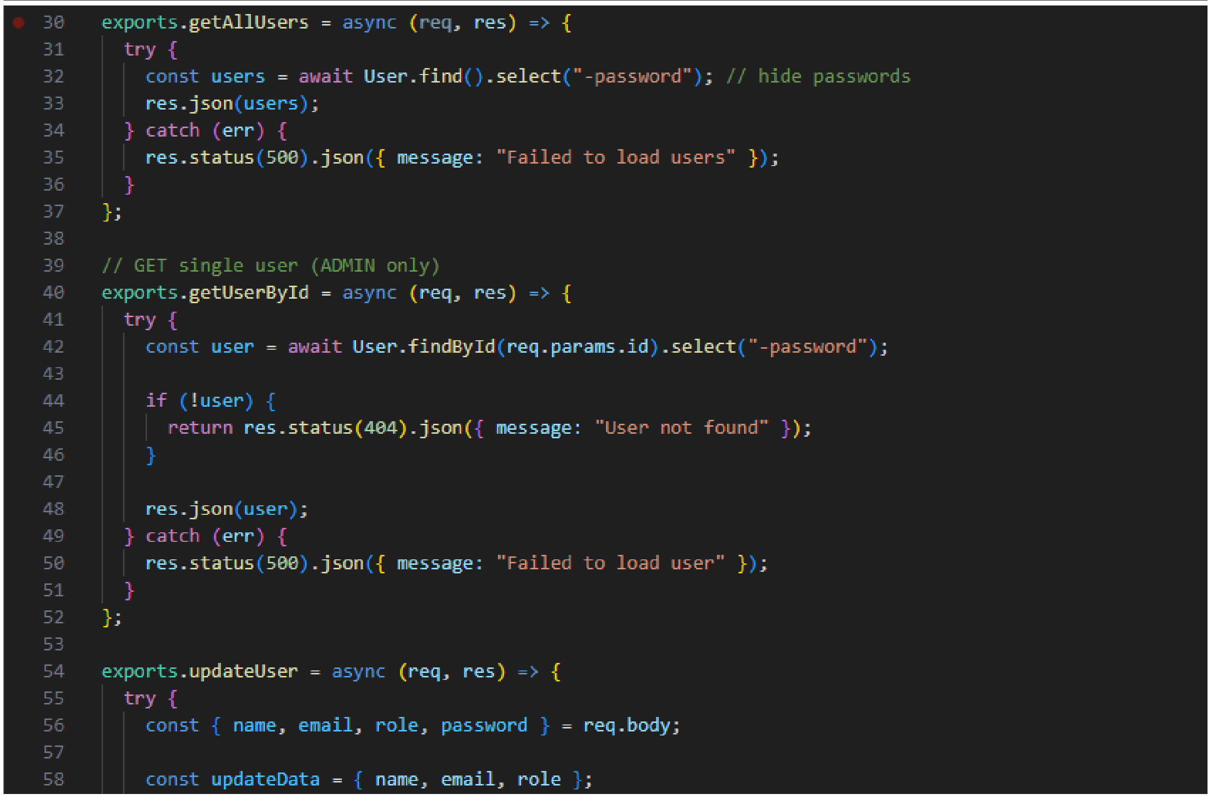
Used By:

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

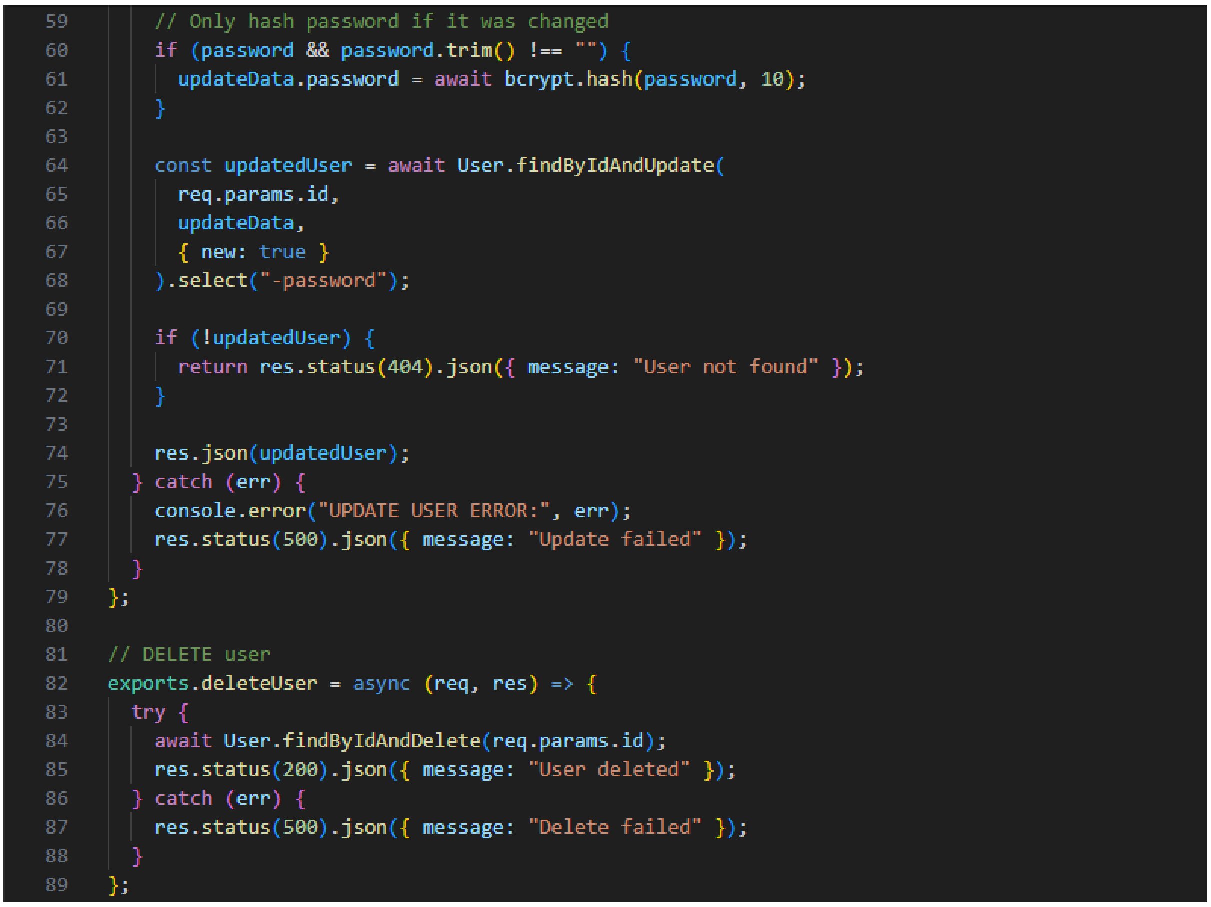
Below is the implemented userController.js code (screenshot).



```javascript
const User = require("../models/User");
const bcrypt = require("bcryptjs");

exports.createUser = async (req, res) => {
  try {
    console.log("CREATE USER BODY:", req.body);
    const { name, email, password, role } = req.body;

    const user = new User({
      name,
      email,
      password,
      role
    });

    await user.save();

    res.status(201).json({ message: "User created successfully" });
  } catch (err) {
    console.error("CREATE USER ERROR:", err);

    if (err.code === 11000) {
      return res.status(400).json({ message: "User already exists" });
    }

    res.status(500).json({ message: "Failed to create user" });
  }
};

```



```javascript
exports.getAllUsers = async (req, res) => {
  try {
    const users = await User.find().select("-password"); // hide passwords
    res.json(users);
  } catch (err) {
    res.status(500).json({ message: "Failed to load users" });
  }
};

// GET single user (ADMIN only)
exports.getUserById = async (req, res) => {
  try {
    const user = await User.findById(req.params.id).select("-password");

    if (!user) {
      return res.status(404).json({ message: "User not found" });
    }

    res.json(user);
  } catch (err) {
    res.status(500).json({ message: "Failed to load user" });
  }
};

exports.updateUser = async (req, res) => {
  try {
    const { name, email, role, password } = req.body;

    const updateData = { name, email, role };
```



```
    // Only hash password if it was changed
    if (password && password.trim() !== "") {
      updateData.password = await bcrypt.hash(password, 10);
    }

    const updatedUser = await User.findByIdAndUpdate(
      req.params.id,
      updateData,
      { new: true }
    ).select("-password");

    if (!updatedUser) {
      return res.status(404).json({ message: "User not found" });
    }

    res.json(updatedUser);
  } catch (err) {
    console.error("UPDATE USER ERROR:", err);
    res.status(500).json({ message: "Update failed" });
  }
};

// DELETE user
exports.deleteUser = async (req, res) => {
  try {
    await User.findByIdAndDelete(req.params.id);
    res.status(200).json({ message: "User deleted" });
  } catch (err) {
    res.status(500).json({ message: "Delete failed" });
  }
};
```

authController.js

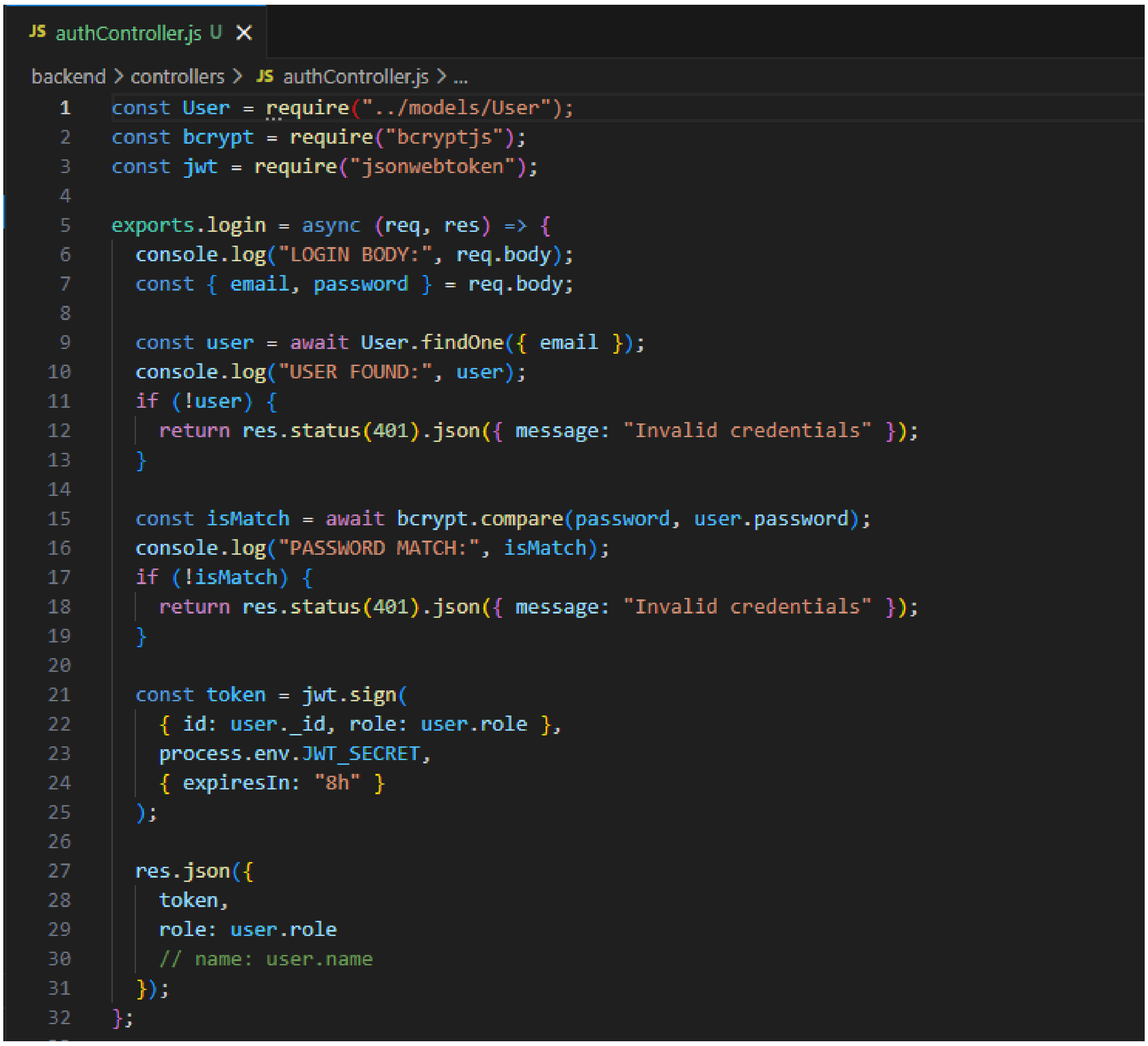
Responsibilities:

- 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 ×

backend > controllers > JS authController.js > ...

```javascript
const User = require("../models/User");
const bcrypt = require("bcryptjs");
const jwt = require("jsonwebtoken");

exports.login = async (req, res) => {
  console.log("LOGIN BODY:", req.body);
  const { email, password } = req.body;

  const user = await User.findOne({ email });
  console.log("USER FOUND:", user);
  if (!user) {
    return res.status(401).json({ message: "Invalid credentials" });
  }

  const isMatch = await bcrypt.compare(password, user.password);
  console.log("PASSWORD MATCH:", isMatch);
  if (!isMatch) {
    return res.status(401).json({ message: "Invalid credentials" });
  }

  const token = jwt.sign(
    { id: user._id, role: user.role },
    process.env.JWT_SECRET,
    { expiresIn: "8h" }
  );

  res.json({
    token,
    role: user.role
    // name: user.name
  });
};
```

chatbotController.js

Responsibilities:

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

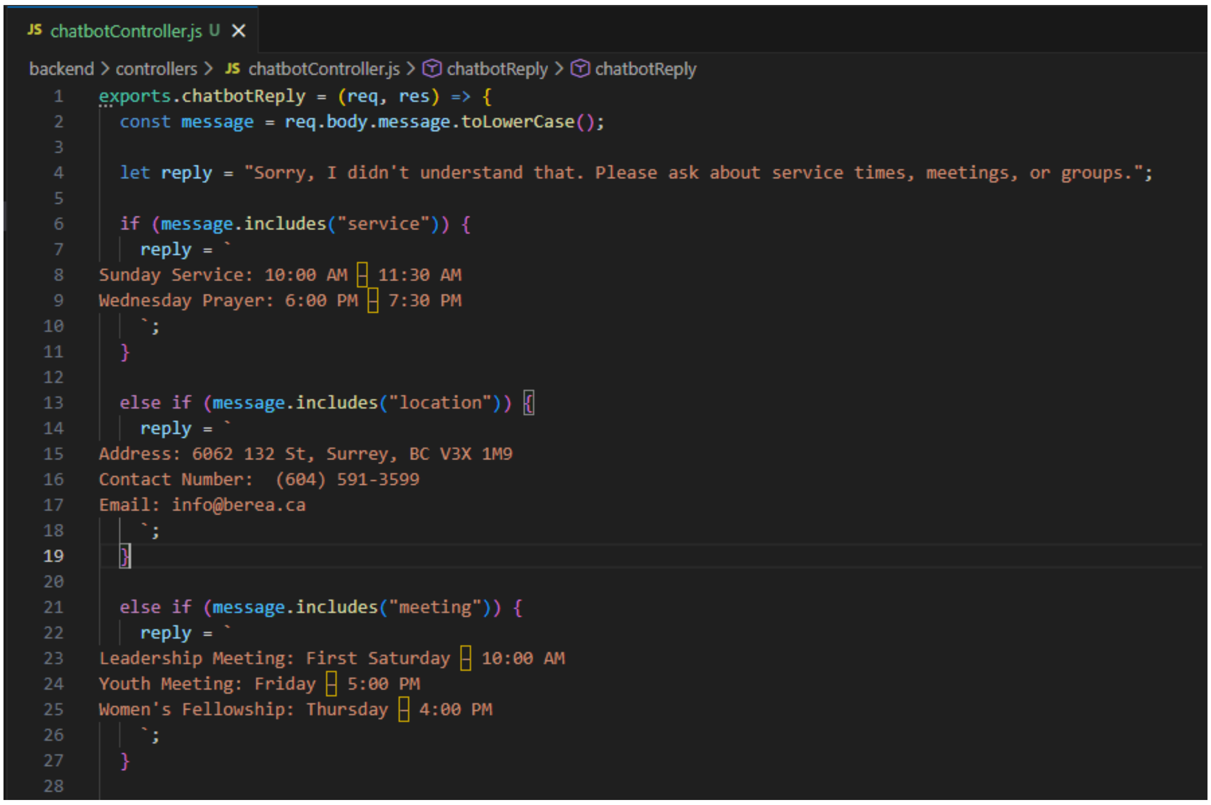
Key Functions (typical):

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

Used In:

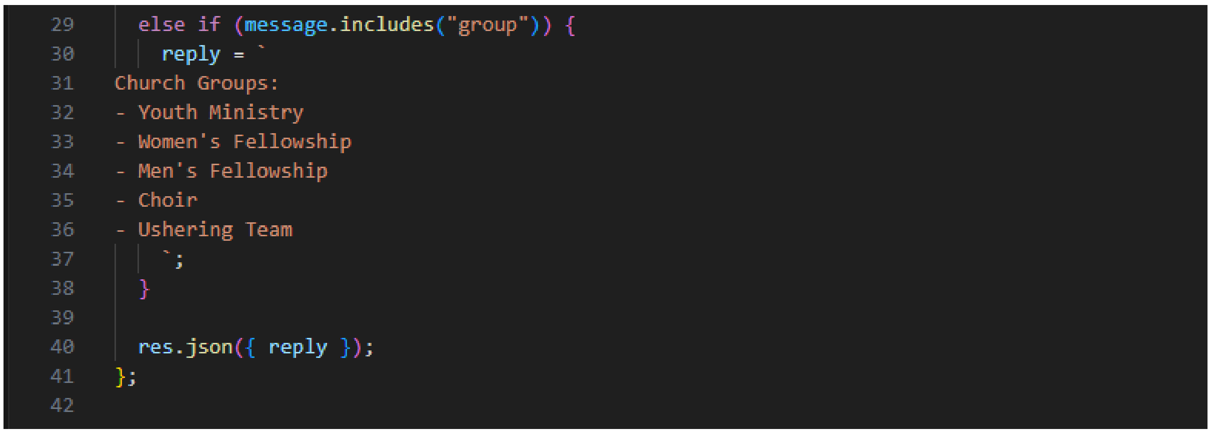
- Chatbot module
- Church information assistant

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



JS chatbotController.js U
backend > controllers > JS chatbotController.js > chatbotReply > chatbotReply

```
exports.chatbotReply = (req, res) => {
  const message = req.body.message.toLowerCase();

  let reply = "Sorry, I didn't understand that. Please ask about service times, meetings, or groups.";

  if (message.includes("service")) {
    reply = `
Sunday Service: 10:00 AM □ 11:30 AM
Wednesday Prayer: 6:00 PM □ 7:30 PM
    `;
  }

  else if (message.includes("location")) {
    reply = `
Address: 6062 132 St, Surrey, BC V3X 1M9
Contact Number:  (604) 591-3599
Email: info@berea.ca
    `;
  }

  else if (message.includes("meeting")) {
    reply = `
Leadership Meeting: First Saturday □ 10:00 AM
Youth Meeting: Friday □ 5:00 PM
Women's Fellowship: Thursday □ 4:00 PM
    `;
  }
```



```
  else if (message.includes("group")) {
    reply = `
Church Groups:
- Youth Ministry
- Women's Fellowship
- Men's Fellowship
- Choir
- Ushering Team
    `;
  }

  res.json({ reply });
};
```

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

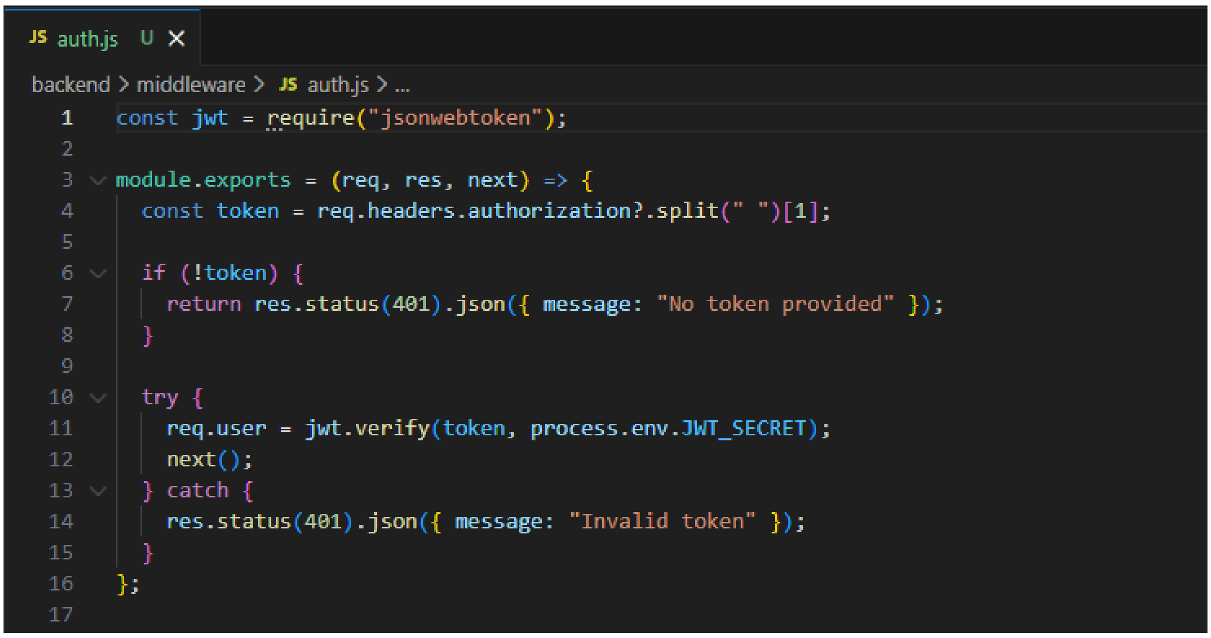
Purpose:

- 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).



```javascript
const jwt = require("jsonwebtoken");

module.exports = (req, res, next) => {
  const token = req.headers.authorization?.split(" ")[1];

  if (!token) {
    return res.status(401).json({ message: "No token provided" });
  }

  try {
    req.user = jwt.verify(token, process.env.JWT_SECRET);
    next();
  } catch {
    res.status(401).json({ message: "Invalid token" });
  }
};
```

authorize.js

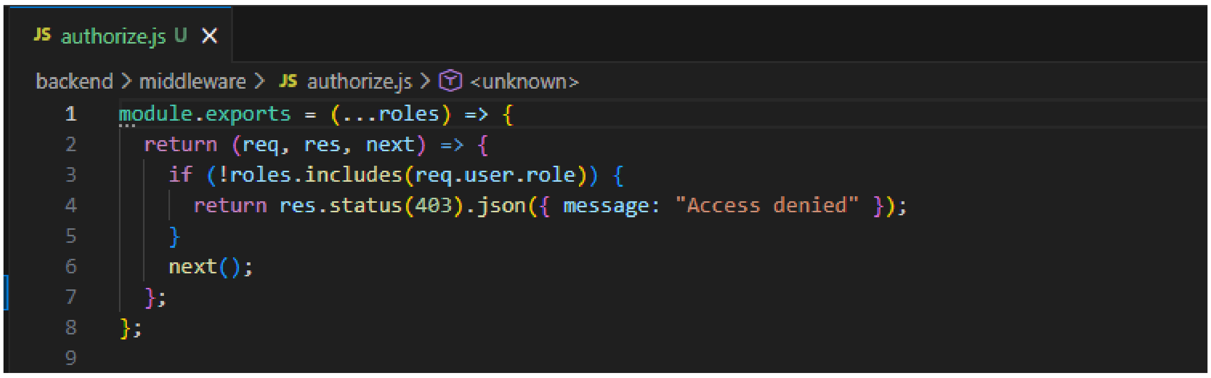
Purpose:

- 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).



authorize.js U ×

backend > middleware > authorize.js > <unknown>

```javascript
module.exports = (...roles) => {
  return (req, res, next) => {
    if (!roles.includes(req.user.role)) {
      return res.status(403).json({ message: "Access denied" });
    }
    next();
  };
};

```

authGuard.js (Frontend)

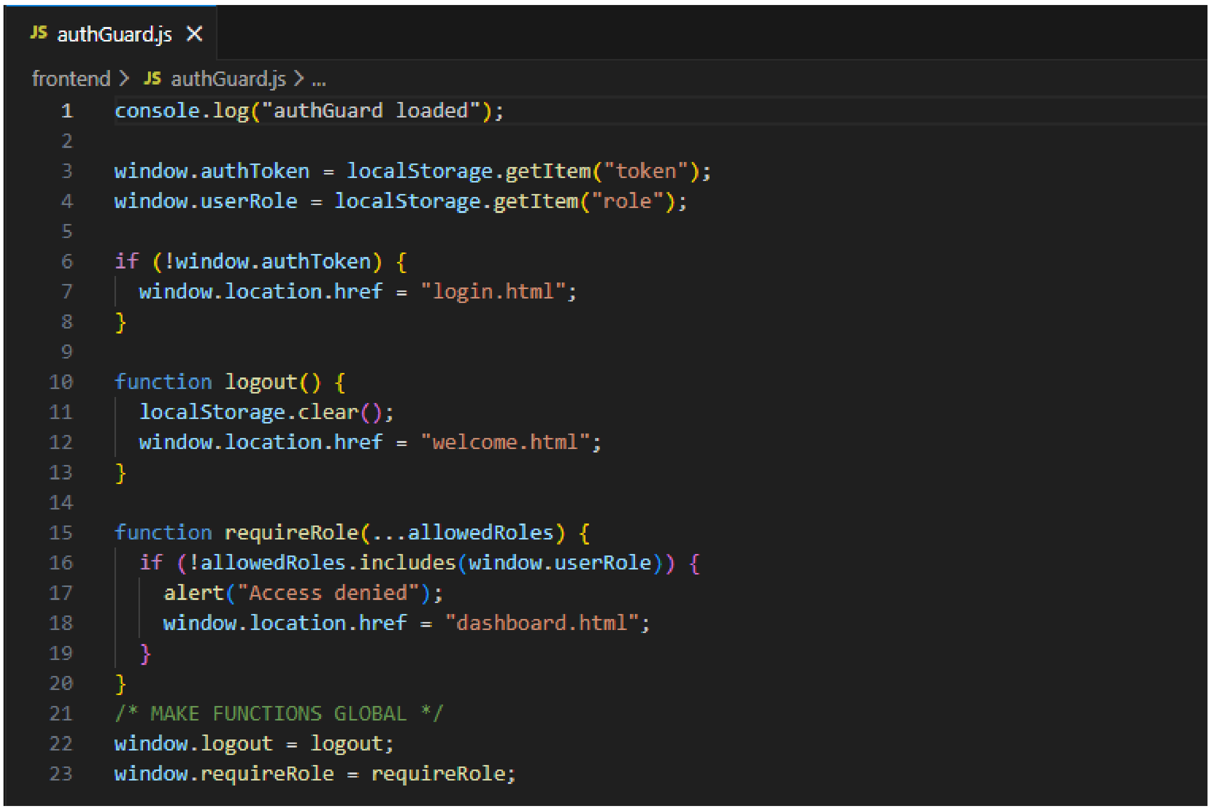
Purpose:

- 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).



```javascript
console.log("authGuard loaded");

window.authToken = localStorage.getItem("token");
window.userRole = localStorage.getItem("role");

if (!window.authToken) {
  window.location.href = "login.html";
}

function logout() {
  localStorage.clear();
  window.location.href = "welcome.html";
}

function requireRole(...allowedRoles) {
  if (!allowedRoles.includes(window.userRole)) {
    alert("Access denied");
    window.location.href = "dashboard.html";
  }
}
/* MAKE FUNCTIONS GLOBAL */
window.logout = logout;
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

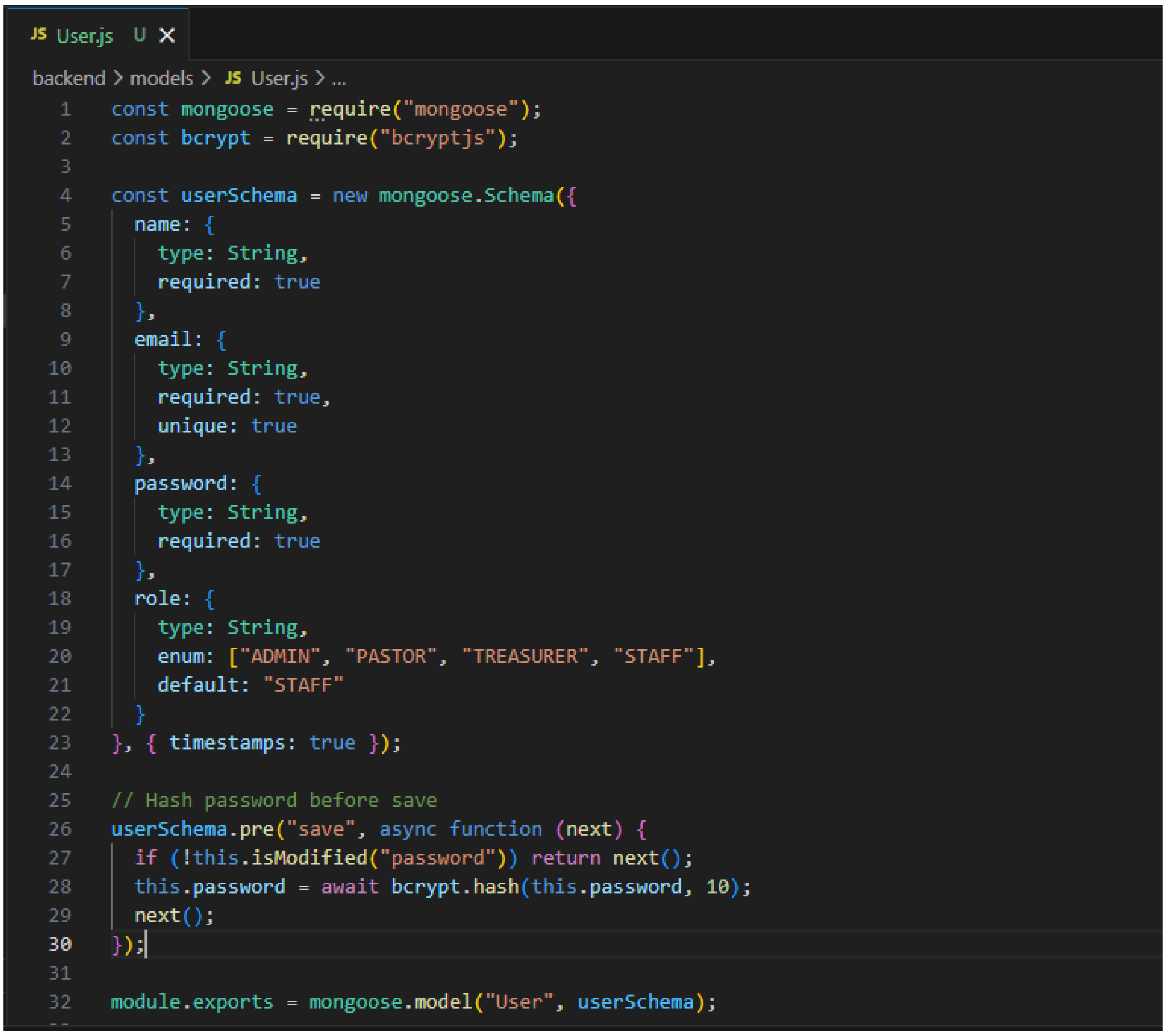
Fields:

- name
- email
- password
- role

Used By:

- Authentication
- User management
- Authorization middleware

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



```
const mongoose = require("mongoose");
const bcrypt = require("bcryptjs");

const userSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true
  },
  email: {
    type: String,
    required: true,
    unique: true
  },
  password: {
    type: String,
    required: true
  },
  role: {
    type: String,
    enum: ["ADMIN", "PASTOR", "TREASURER", "STAFF"],
    default: "STAFF"
  }
}, { timestamps: true });

// Hash password before save
userSchema.pre("save", async function (next) {
  if (!this.isModified("password")) return next();
  this.password = await bcrypt.hash(this.password, 10);
  next();
});

module.exports = mongoose.model("User", userSchema);
```

Member.js

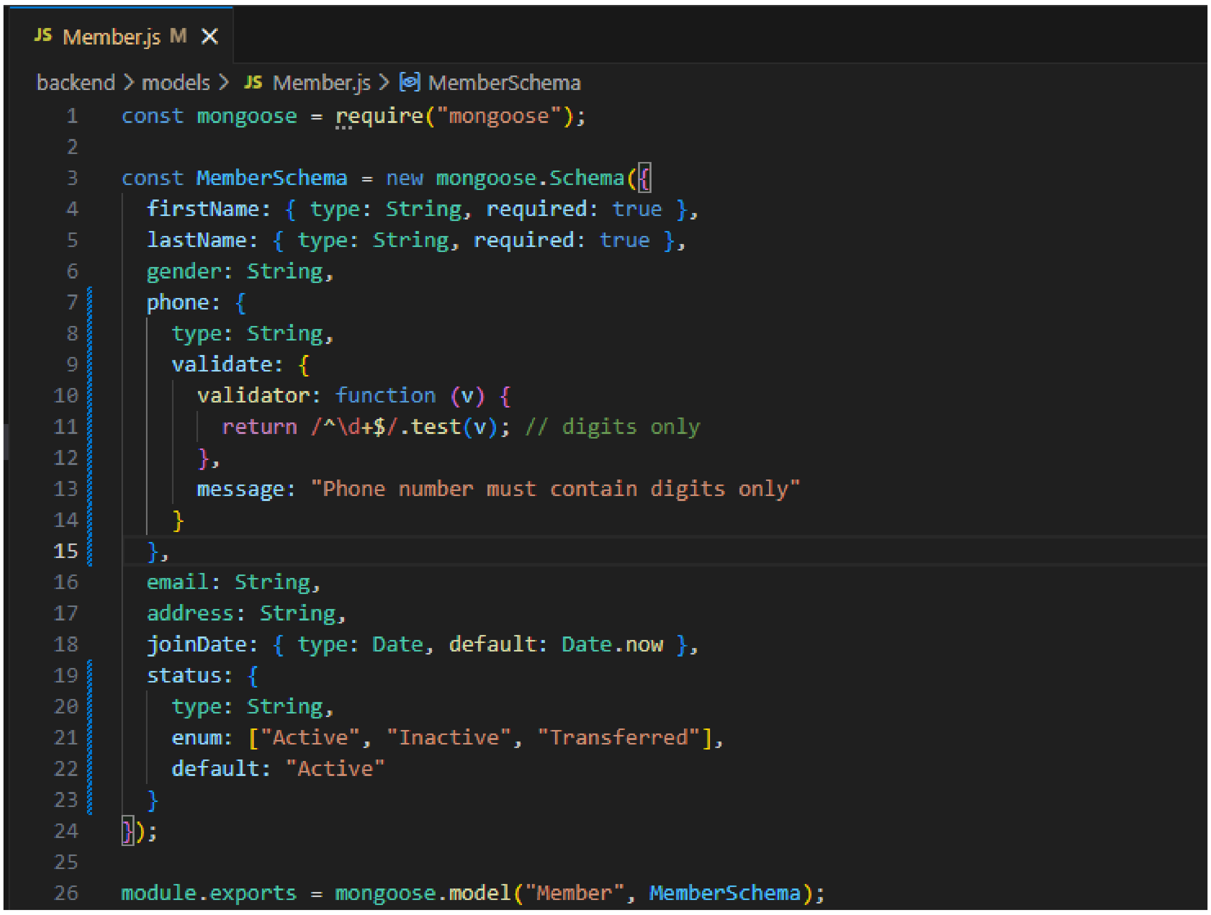
Fields:

- 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).



```javascript
const mongoose = require("mongoose");

const MemberSchema = new mongoose.Schema({
  firstName: { type: String, required: true },
  lastName: { type: String, required: true },
  gender: String,
  phone: {
    type: String,
    validate: {
      validator: function (v) {
        return /^\d+$/.test(v); // digits only
      },
      message: "Phone number must contain digits only"
    }
  },
  email: String,
  address: String,
  joinDate: { type: Date, default: Date.now },
  status: {
    type: String,
    enum: ["Active", "Inactive", "Transferred"],
    default: "Active"
  }
});

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

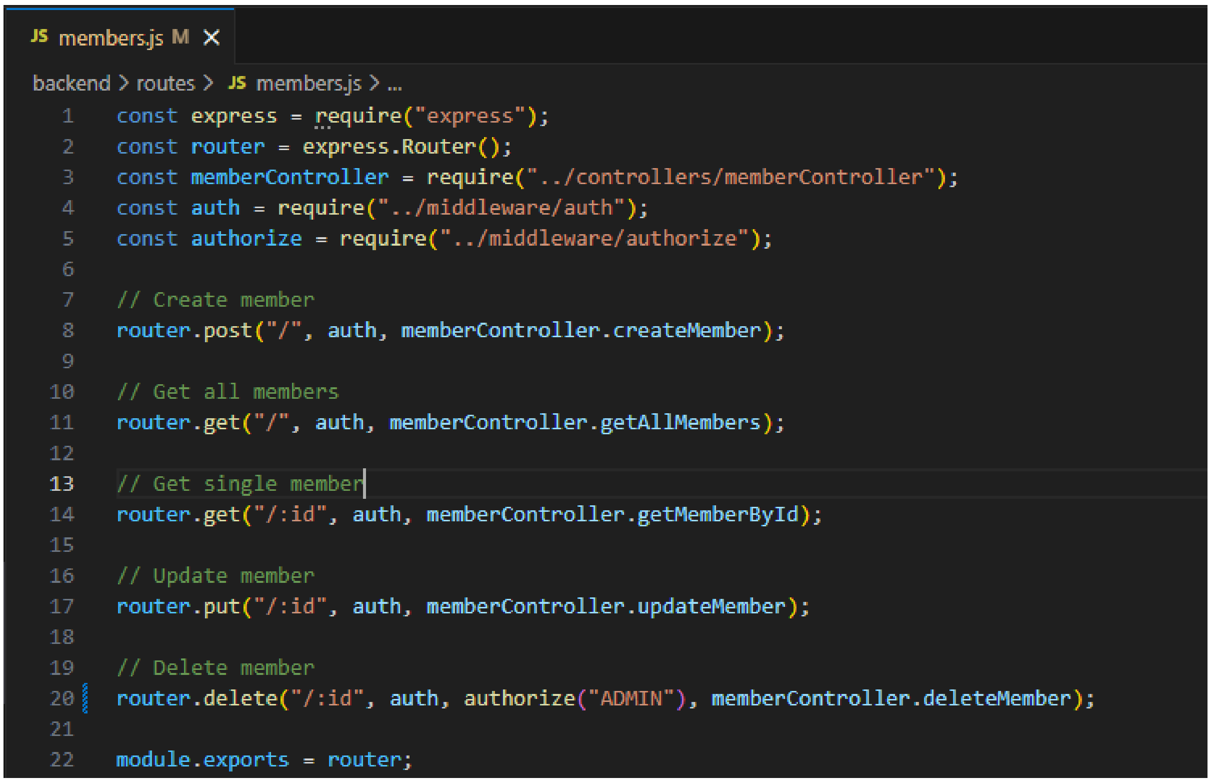
Endpoints:

- 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).



members.js M

backend > routes > members.js > ...

```javascript
const express = require("express");
const router = express.Router();
const memberController = require("../controllers/memberController");
const auth = require("../middleware/auth");
const authorize = require("../middleware/authorize");

// Create member
router.post("/", auth, memberController.createMember);

// Get all members
router.get("/", auth, memberController.getAllMembers);

// Get single member
router.get("/:id", auth, memberController.getMemberById);

// Update member
router.put("/:id", auth, memberController.updateMember);

// Delete member
router.delete("/:id", auth, authorize("ADMIN"), memberController.deleteMember);

module.exports = router;
```

routes/users.js

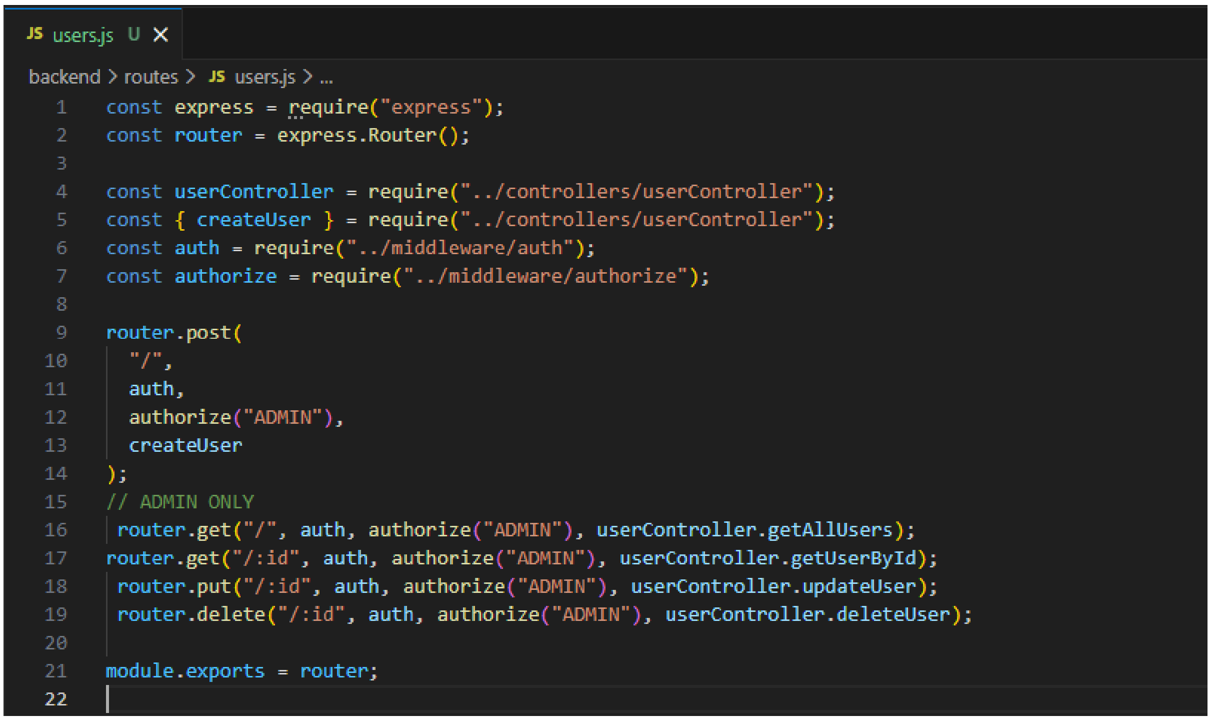
Endpoints:

- 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).



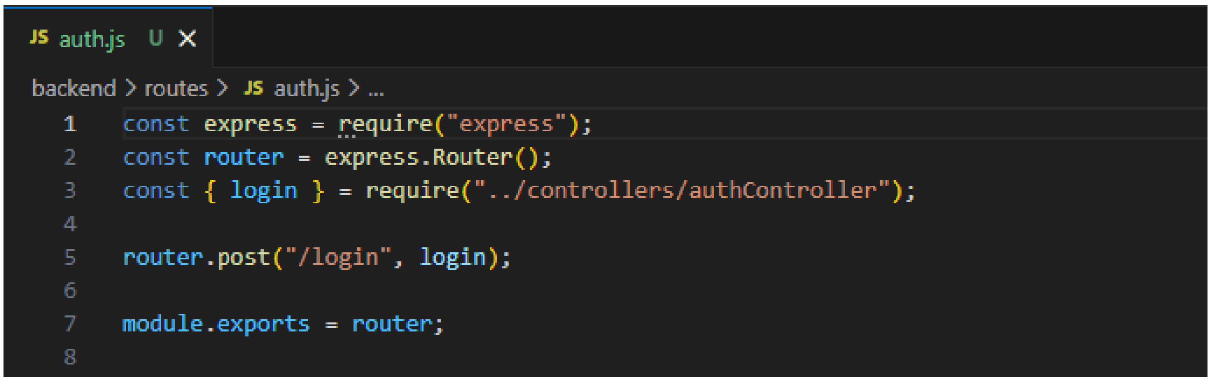
```javascript
const express = require("express");
const router = express.Router();

const userController = require("../controllers/userController");
const { createUser } = require("../controllers/userController");
const auth = require("../middleware/auth");
const authorize = require("../middleware/authorize");

router.post(
  "/",
  auth,
  authorize("ADMIN"),
  createUser
);
// ADMIN ONLY
 router.get("/", auth, authorize("ADMIN"), userController.getAllUsers);
router.get("/:id", auth, authorize("ADMIN"), userController.getUserById);
 router.put("/:id", auth, authorize("ADMIN"), userController.updateUser);
 router.delete("/:id", auth, authorize("ADMIN"), userController.deleteUser);

module.exports = router;

```

routes/auth.js

Endpoints:

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

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



```javascript
const express = require("express");
const router = express.Router();
const { login } = require("../controllers/authController");

router.post("/login", login);

module.exports = router;

```

routes/chatbot.js

Endpoints:

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

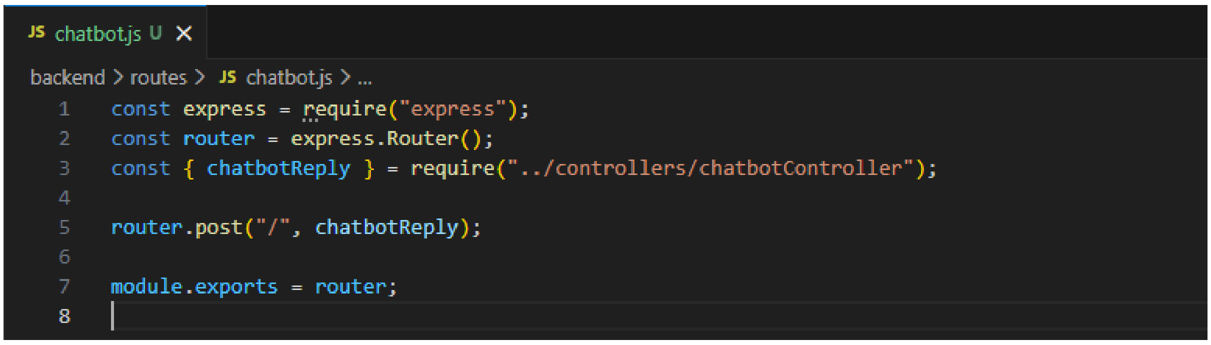
Middleware Applied:

- Auth

Mapped Controller:

- chatbotController.js

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



```javascript
const express = require("express");
const router = express.Router();
const { chatbotReply } = require("../controllers/chatbotController");

router.post("/", chatbotReply);

module.exports = router;
```

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

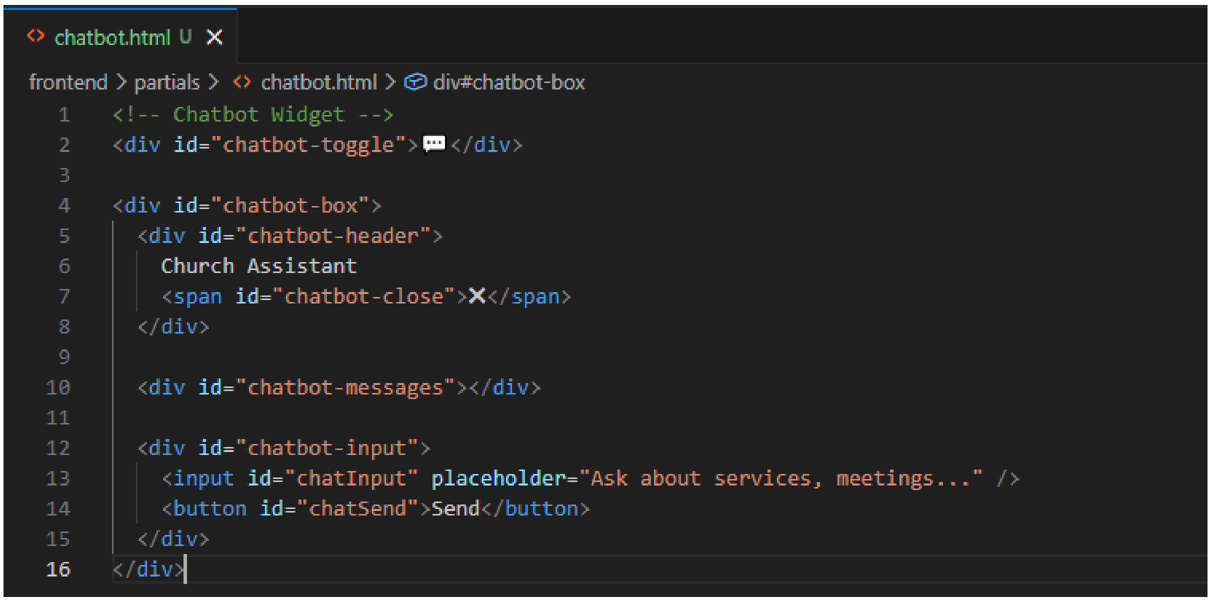
Purpose:

- 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).



```html
<!-- Chatbot Widget -->
<div id="chatbot-toggle">💬</div>

<div id="chatbot-box">
  <div id="chatbot-header">
    Church Assistant
    <span id="chatbot-close">✖</span>
  </div>

  <div id="chatbot-messages"></div>

  <div id="chatbot-input">
    <input id="chatInput" placeholder="Ask about services, meetings..." />
    <button id="chatSend">Send</button>
  </div>
</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:

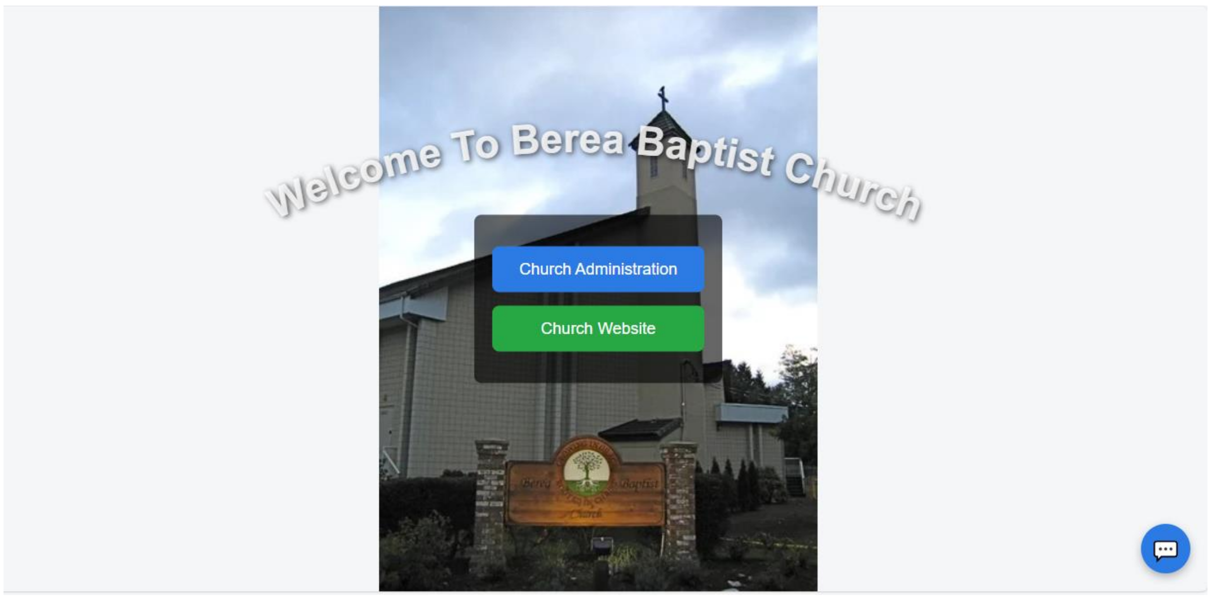
- Add-member
- Edit-member
- Create-user
- Edit-user

Features:

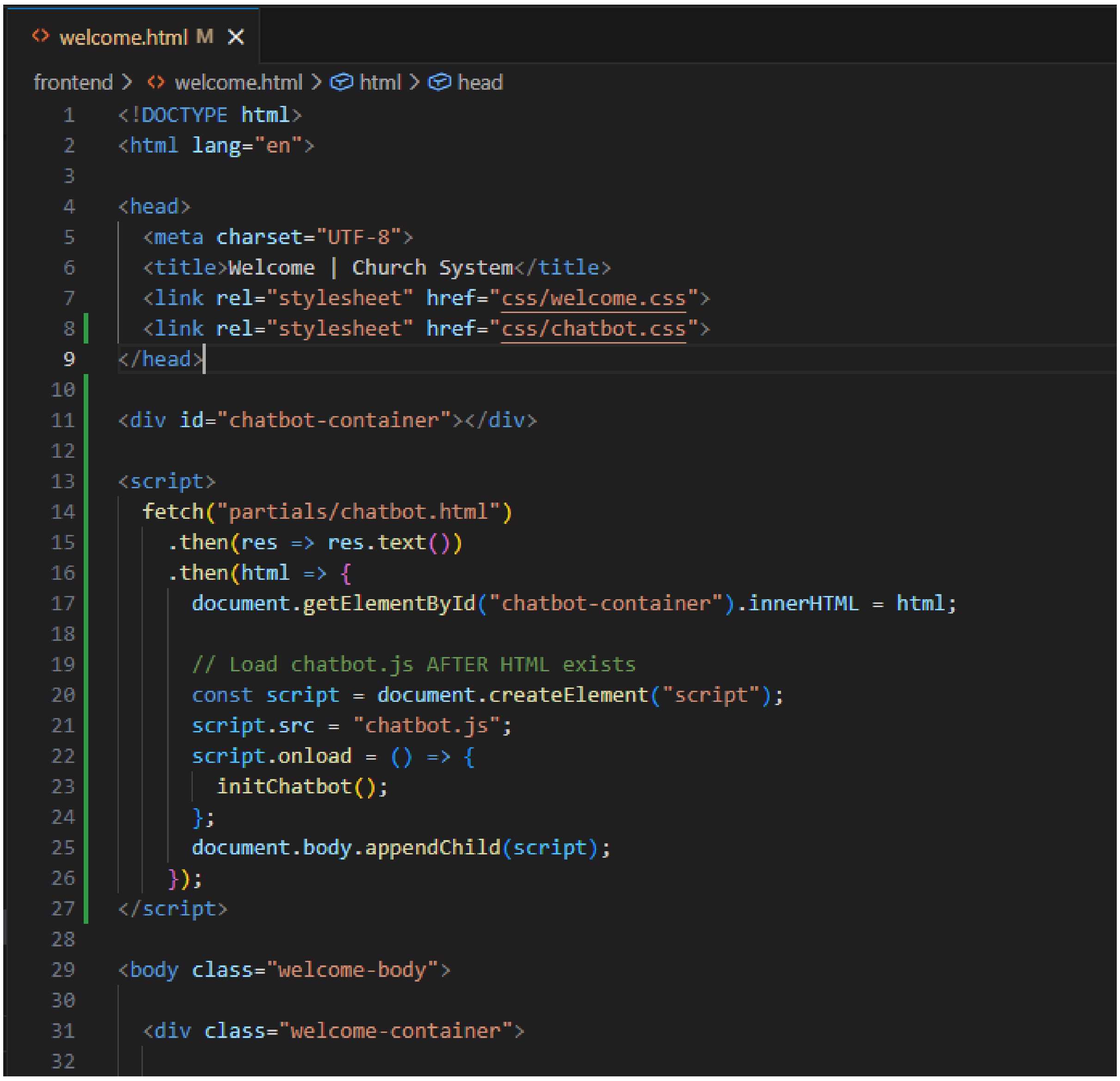
- Input validation
- Controlled fields
- API integration

5.6 Welcome Screen (Frontend)

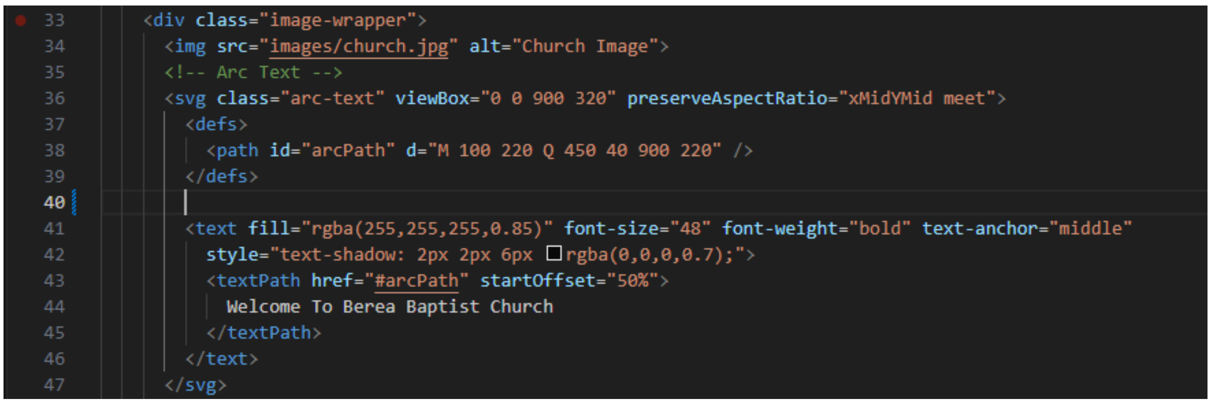
Implemented a welcome screen with two buttons, to allow the user to choose whether they want to do administration or to go to the church website. Included on the welcome screen is a chatbot for assistance with church information.



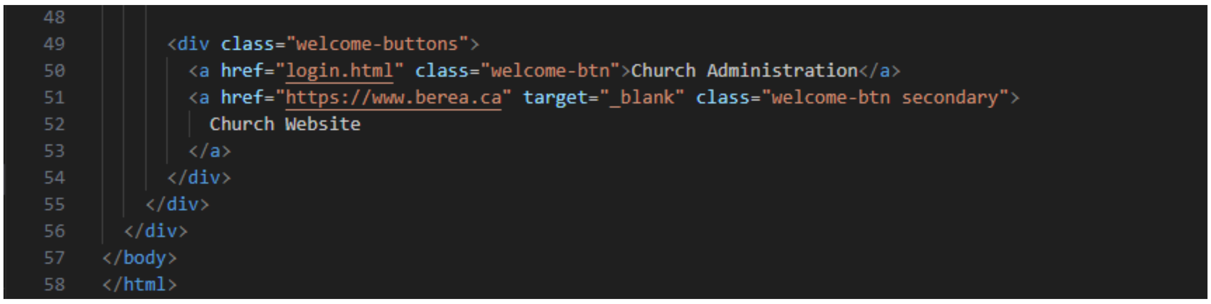
Below is the implemented welcome.html code for the welcome screen.



```html
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <title>Welcome | Church System</title>
  <link rel="stylesheet" href="css/welcome.css">
  <link rel="stylesheet" href="css/chatbot.css">
</head>

<div id="chatbot-container"></div>

<script>
  fetch("partials/chatbot.html")
    .then(res => res.text())
    .then(html => {
      document.getElementById("chatbot-container").innerHTML = html;

      // Load chatbot.js AFTER HTML exists
      const script = document.createElement("script");
      script.src = "chatbot.js";
      script.onload = () => {
        initChatbot();
      };
      document.body.appendChild(script);
    });
</script>

<body class="welcome-body">

  <div class="welcome-container">

```



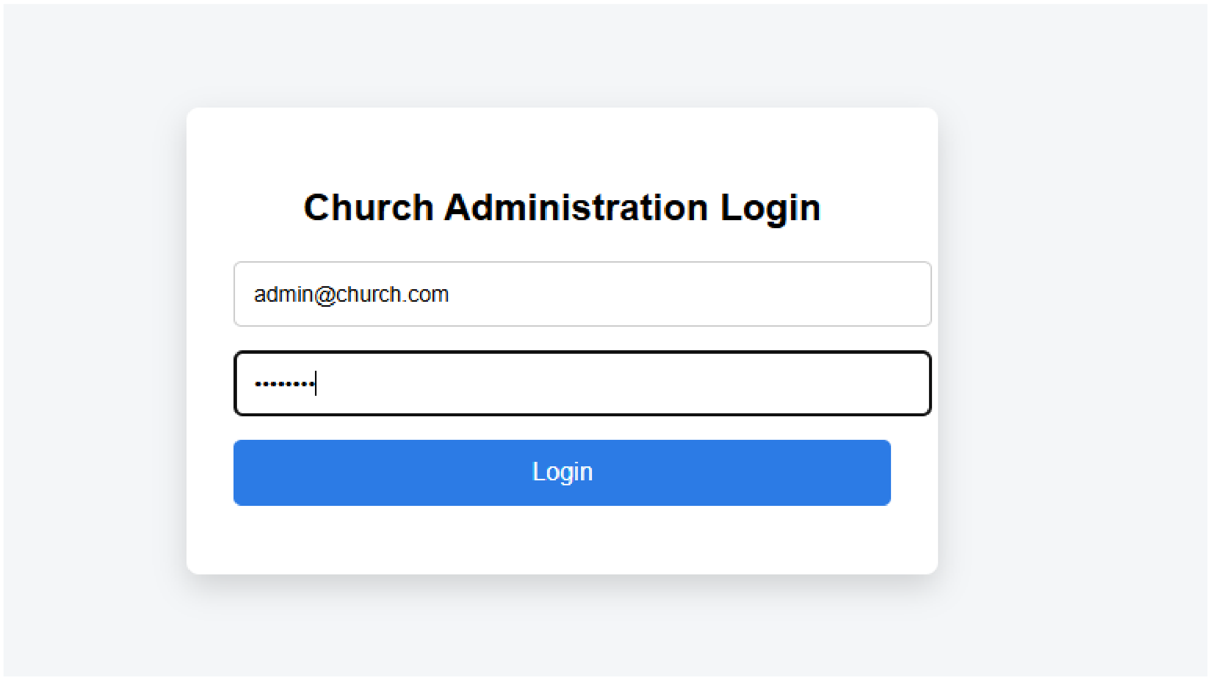
```html
    <div class="image-wrapper">
      <img src="images/church.jpg" alt="Church Image">
      <!-- Arc Text -->
      <svg class="arc-text" viewBox="0 0 900 320" preserveAspectRatio="xMidYMid meet">
        <defs>
          <path id="arcPath" d="M 100 220 Q 450 40 900 220" />
        </defs>

        <text fill="rgba(255,255,255,0.85)" font-size="48" font-weight="bold" text-anchor="middle"
          style="text-shadow: 2px 2px 6px rgba(0,0,0,0.7);">
          <textPath href="#arcPath" startOffset="50%">
            Welcome To Berea Baptist Church
          </textPath>
        </text>
      </svg>
```



```

            <div class="welcome-buttons">
              <a href="login.html" class="welcome-btn">Church Administration</a>
              <a href="https://www.berea.ca" target="_blank" class="welcome-btn secondary">
                Church Website
              </a>
            </div>
          </div>
        </div>
      </body>
      </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.

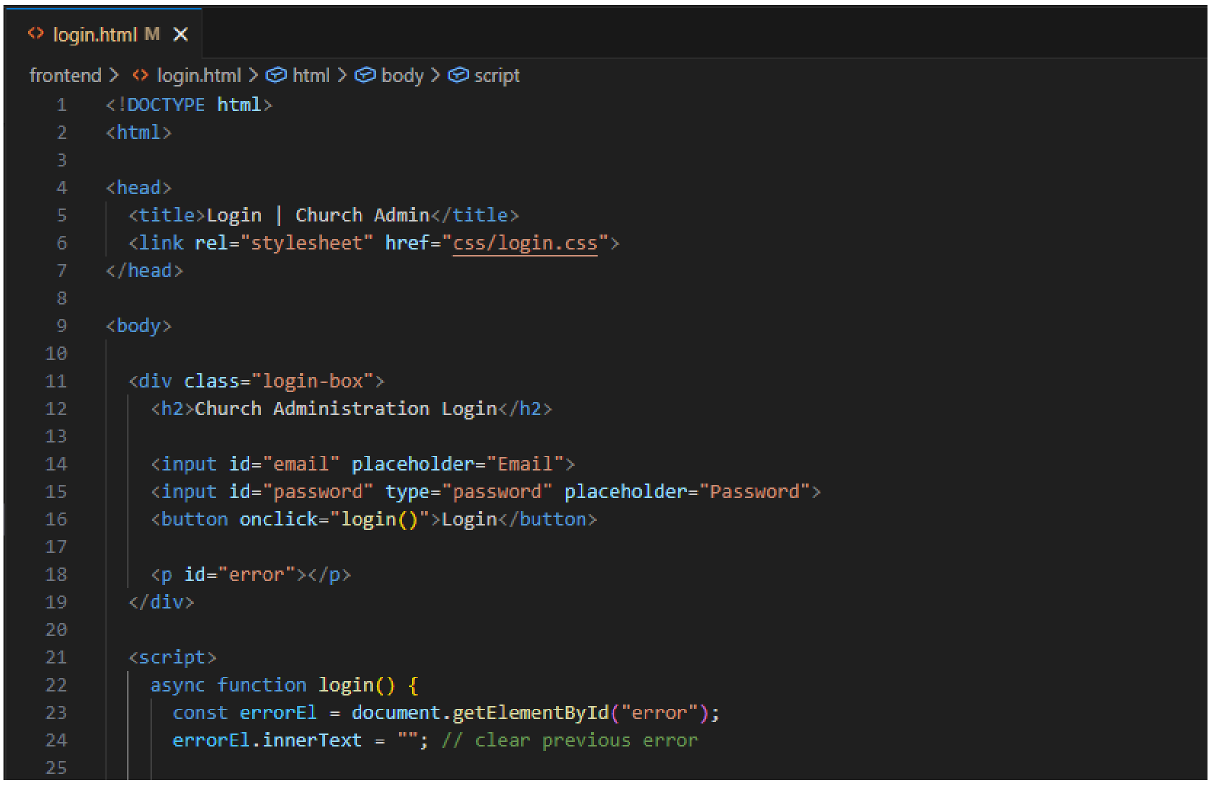


Church Administration Login

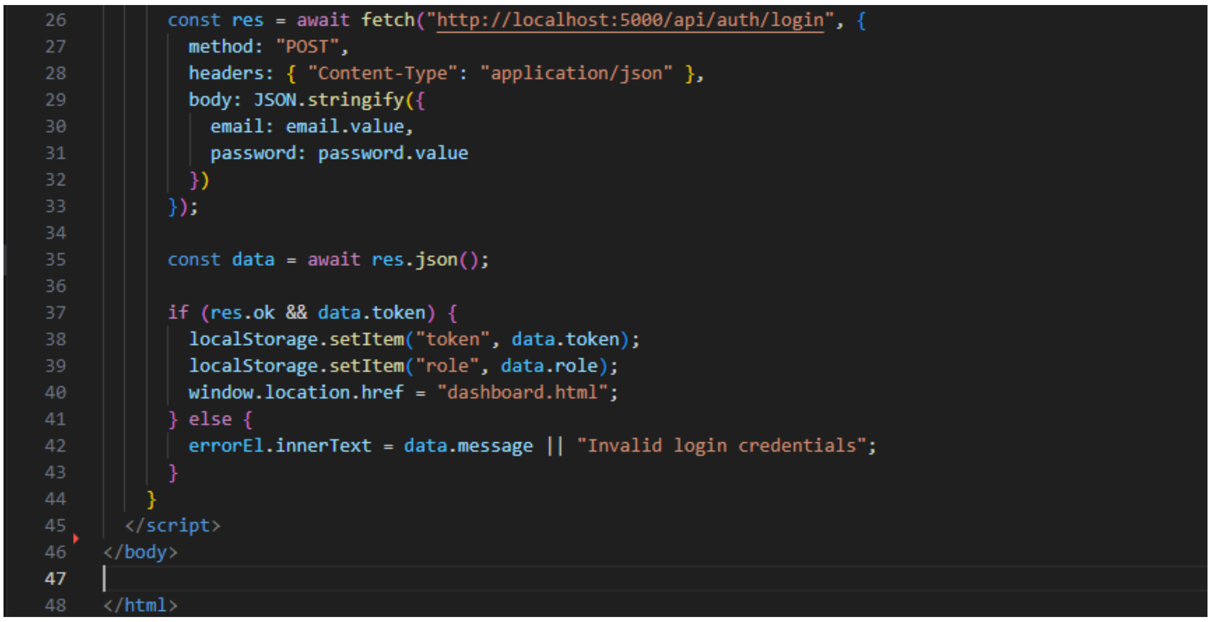
admin@church.com

••••••••

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

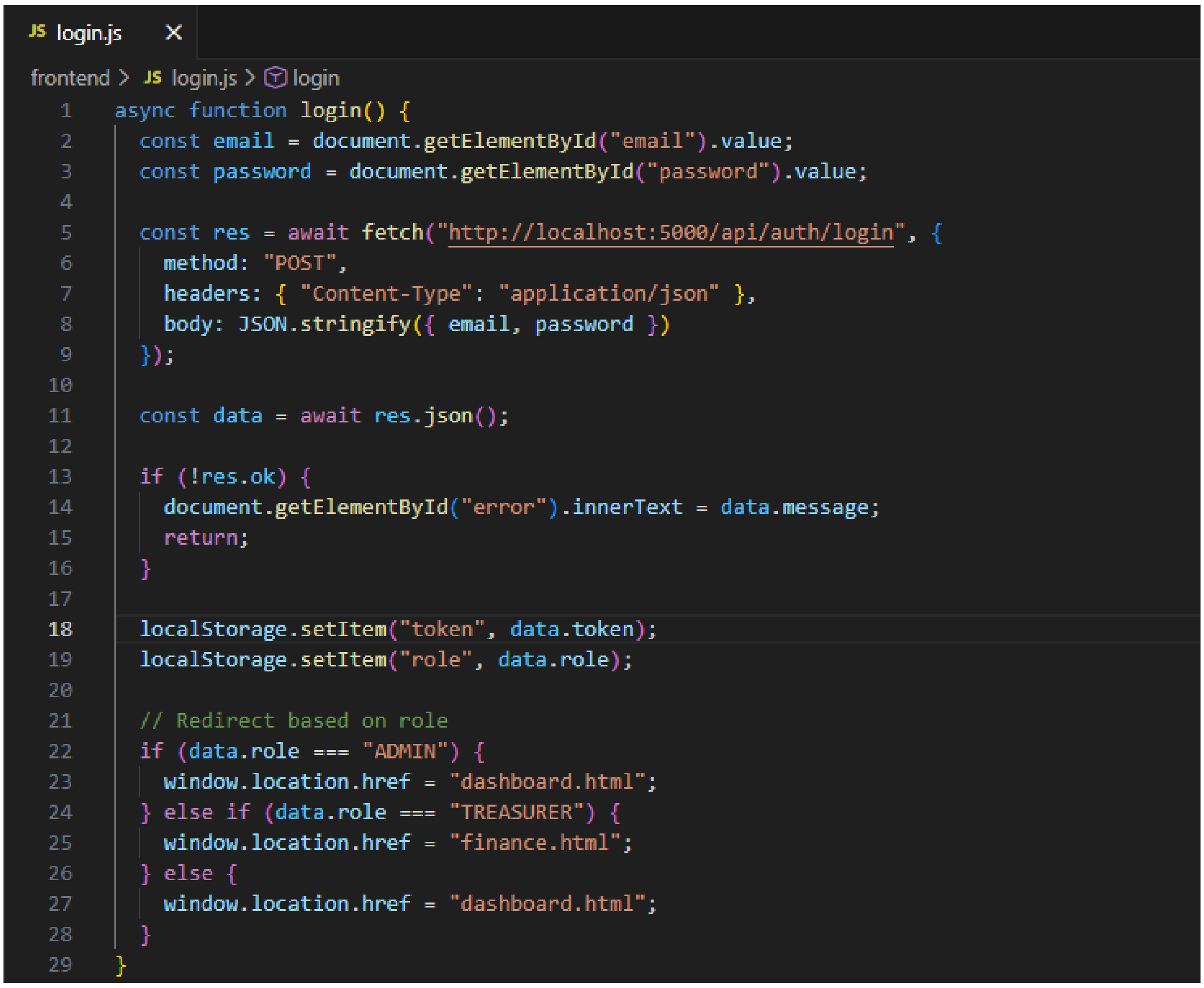


```html
<!DOCTYPE html>
<html>

<head>
  <title>Login | Church Admin</title>
  <link rel="stylesheet" href="css/login.css">
</head>

<body>

  <div class="login-box">
    <h2>Church Administration Login</h2>

    <input id="email" placeholder="Email">
    <input id="password" type="password" placeholder="Password">
    <button onclick="login()">Login</button>

    <p id="error"></p>
  </div>

  <script>
    async function login() {
      const errorEl = document.getElementById("error");
      errorEl.innerText = ""; // clear previous error

```



```html
      const res = await fetch("http://localhost:5000/api/auth/login", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({
          email: email.value,
          password: password.value
        })
      });

      const data = await res.json();

      if (res.ok && data.token) {
        localStorage.setItem("token", data.token);
        localStorage.setItem("role", data.role);
        window.location.href = "dashboard.html";
      } else {
        errorEl.innerText = data.message || "Invalid login credentials";
      }
    }
  </script>
</body>

</html>
```

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

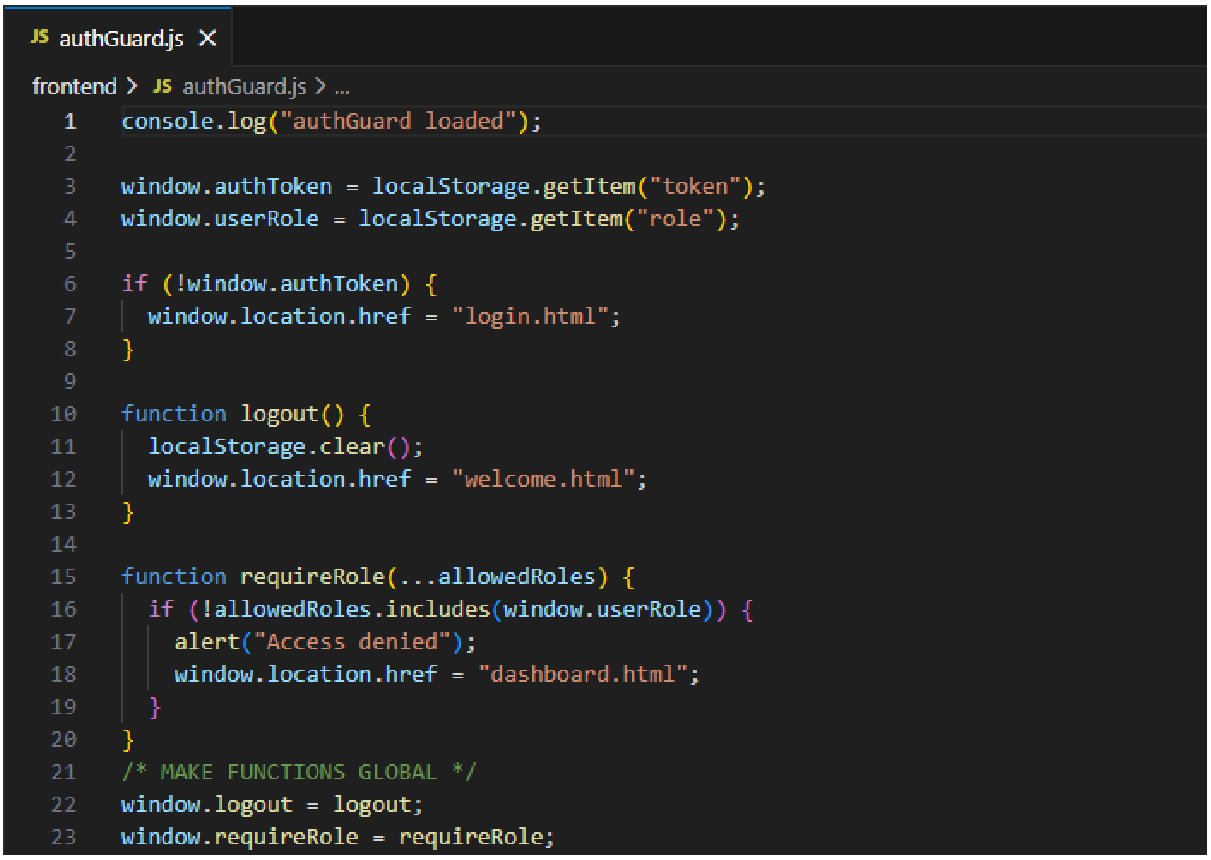


```javascript
async function login() {
  const email = document.getElementById("email").value;
  const password = document.getElementById("password").value;

  const res = await fetch("http://localhost:5000/api/auth/login", {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({ email, password })
  });

  const data = await res.json();

  if (!res.ok) {
    document.getElementById("error").innerText = data.message;
    return;
  }

  localStorage.setItem("token", data.token);
  localStorage.setItem("role", data.role);

  // Redirect based on role
  if (data.role === "ADMIN") {
    window.location.href = "dashboard.html";
  } else if (data.role === "TREASURER") {
    window.location.href = "finance.html";
  } else {
    window.location.href = "dashboard.html";
  }
}
```

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.



```javascript
console.log("authGuard loaded");

window.authToken = localStorage.getItem("token");
window.userRole = localStorage.getItem("role");

if (!window.authToken) {
  window.location.href = "login.html";
}

function logout() {
  localStorage.clear();
  window.location.href = "welcome.html";
}

function requireRole(...allowedRoles) {
  if (!allowedRoles.includes(window.userRole)) {
    alert("Access denied");
    window.location.href = "dashboard.html";
  }
}
/* MAKE FUNCTIONS GLOBAL */
window.logout = logout;
window.requireRole = requireRole;
```

5.9 User Management (Frontend)

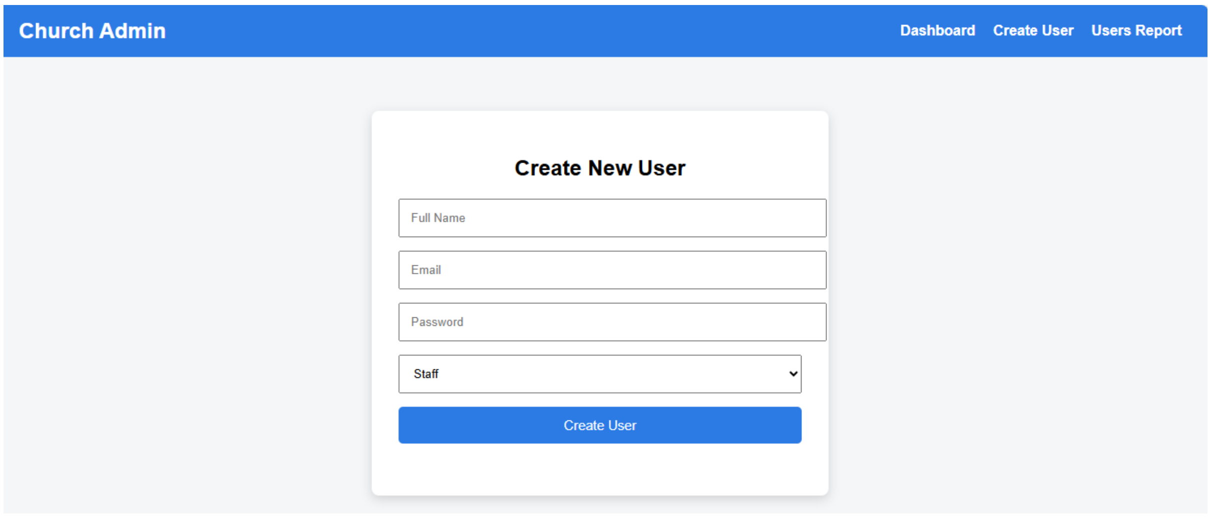
Modules implemented include:

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

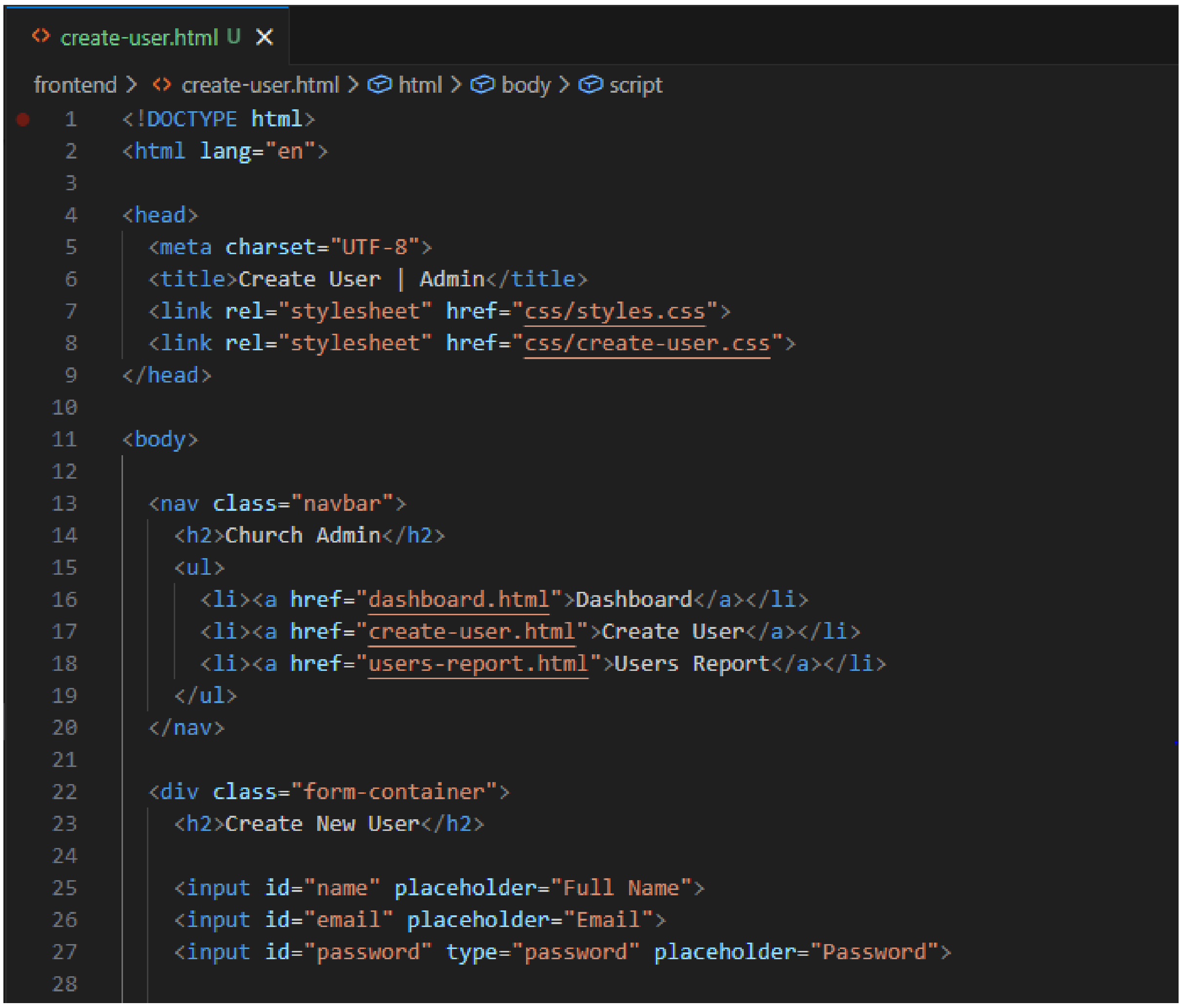
These modules allow administrators to manage system users securely.

5.9(i) create-user

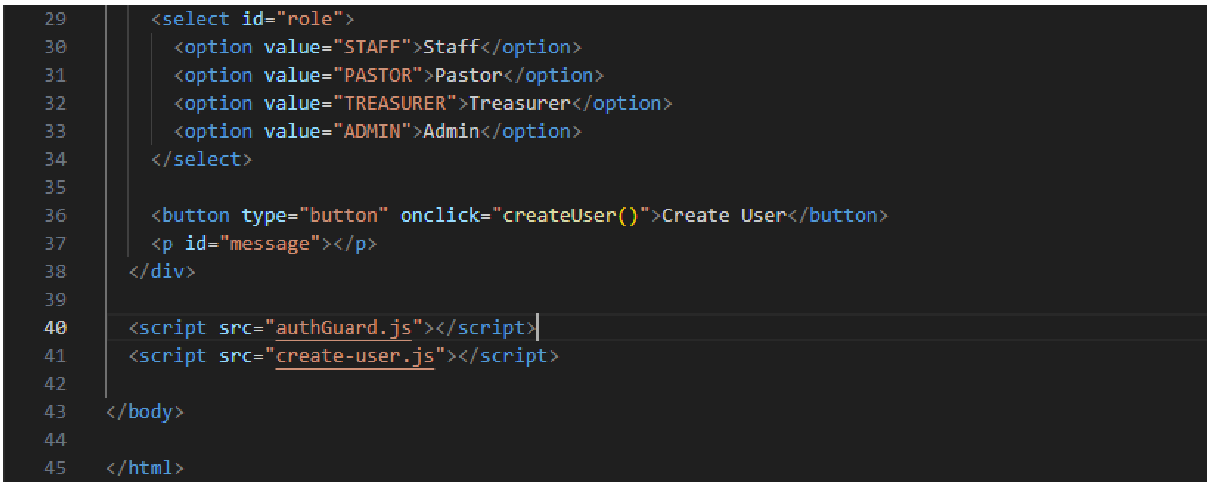
Below is the create-user screen, visible to the admin user.



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

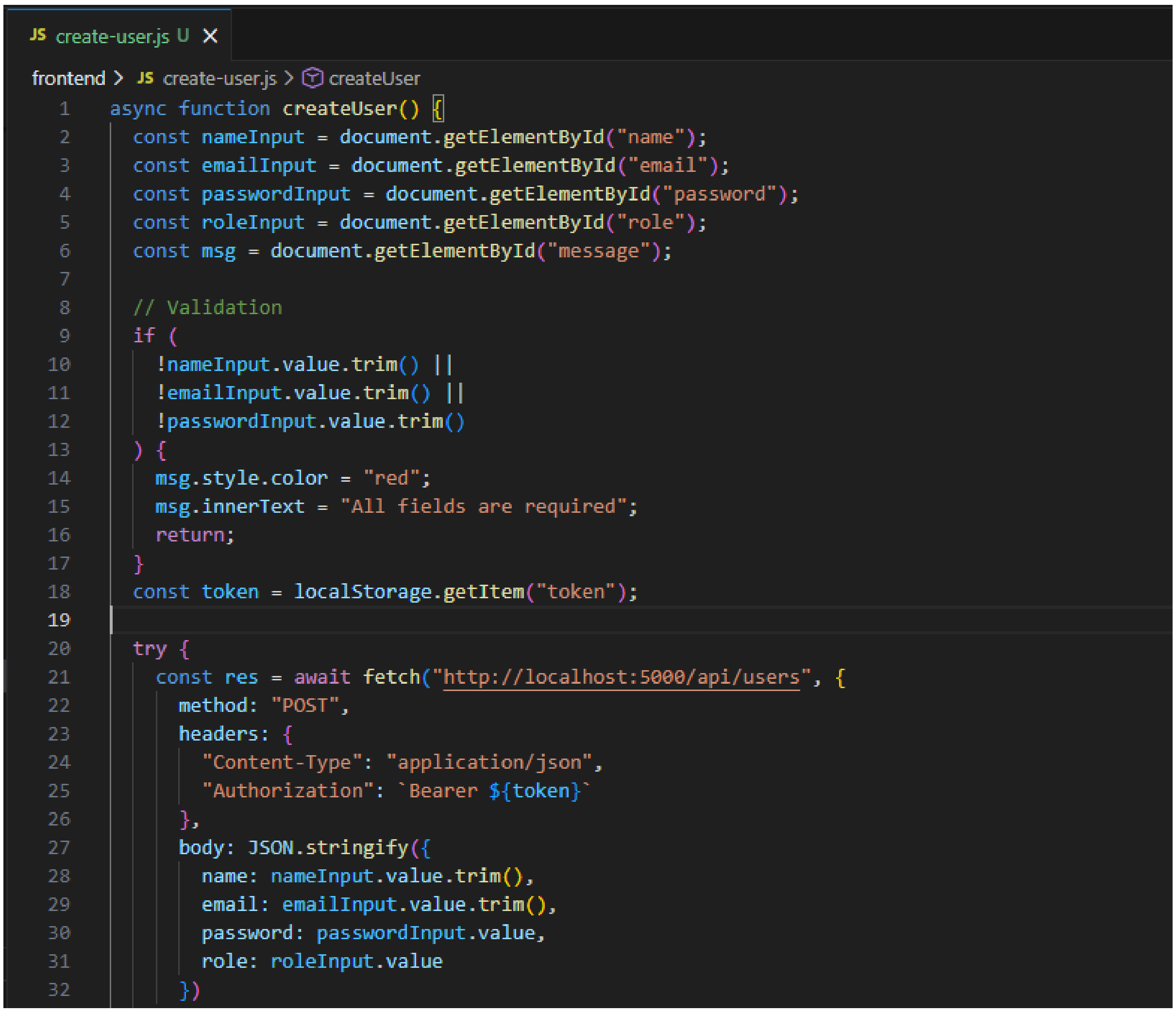


```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <title>Create User | Admin</title>
  <link rel="stylesheet" href="css/styles.css">
  <link rel="stylesheet" href="css/create-user.css">
</head>

<body>

  <nav class="navbar">
    <h2>Church Admin</h2>
    <ul>
      <li><a href="dashboard.html">Dashboard</a></li>
      <li><a href="create-user.html">Create User</a></li>
      <li><a href="users-report.html">Users Report</a></li>
    </ul>
  </nav>

  <div class="form-container">
    <h2>Create New User</h2>

    <input id="name" placeholder="Full Name">
    <input id="email" placeholder="Email">
    <input id="password" type="password" placeholder="Password">

```

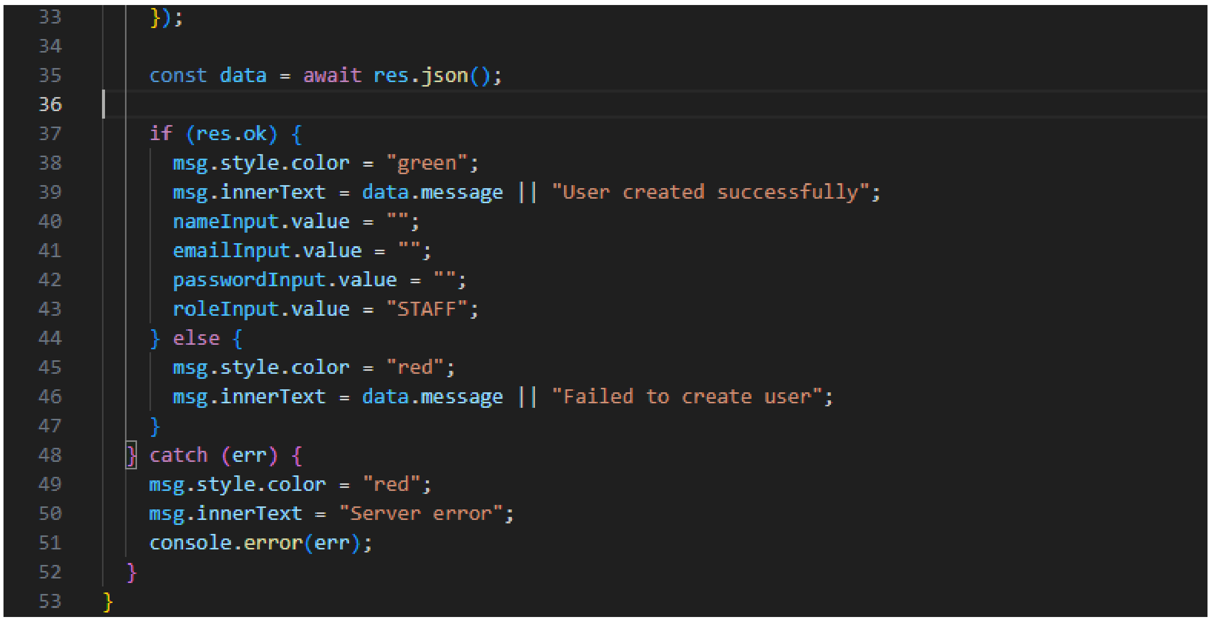


```html
      <select id="role">
        <option value="STAFF">Staff</option>
        <option value="PASTOR">Pastor</option>
        <option value="TREASURER">Treasurer</option>
        <option value="ADMIN">Admin</option>
      </select>

      <button type="button" onclick="createUser()">Create User</button>
      <p id="message"></p>
    </div>

    <script src="authGuard.js"></script>
    <script src="create-user.js"></script>

  </body>

  </html>
```

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



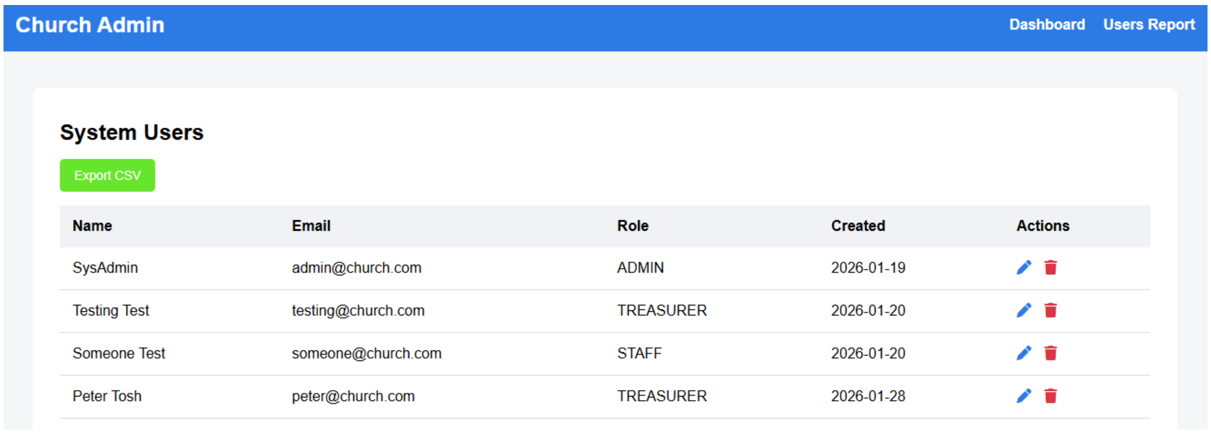
```javascript
async function createUser() {
  const nameInput = document.getElementById("name");
  const emailInput = document.getElementById("email");
  const passwordInput = document.getElementById("password");
  const roleInput = document.getElementById("role");
  const msg = document.getElementById("message");

  // Validation
  if (
    !nameInput.value.trim() ||
    !emailInput.value.trim() ||
    !passwordInput.value.trim()
  ) {
    msg.style.color = "red";
    msg.innerText = "All fields are required";
    return;
  }
  const token = localStorage.getItem("token");

  try {
    const res = await fetch("http://localhost:5000/api/users", {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
        "Authorization": `Bearer ${token}`
      },
      body: JSON.stringify({
        name: nameInput.value.trim(),
        email: emailInput.value.trim(),
        password: passwordInput.value,
        role: roleInput.value
      })
```



```
    });

    const data = await res.json();

    if (res.ok) {
      msg.style.color = "green";
      msg.innerText = data.message || "User created successfully";
      nameInput.value = "";
      emailInput.value = "";
      passwordInput.value = "";
      roleInput.value = "STAFF";
    } else {
      msg.style.color = "red";
      msg.innerText = data.message || "Failed to create user";
    }
  } catch (err) {
    msg.style.color = "red";
    msg.innerText = "Server error";
    console.error(err);
  }
}
```

5.9(ii) users-report

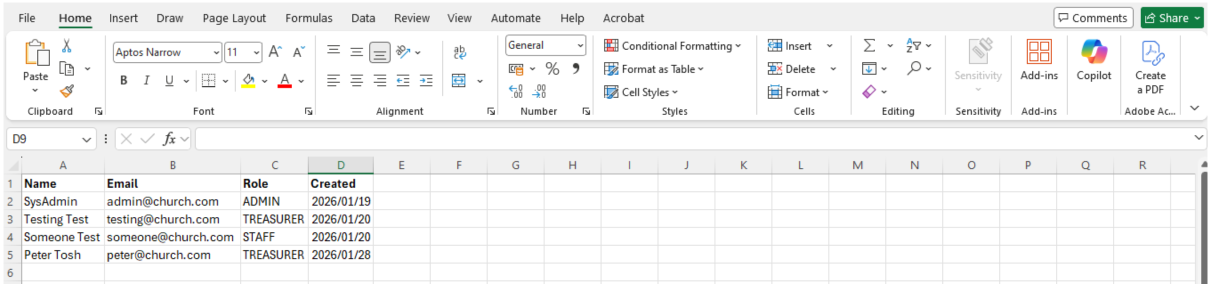
Below is the users report. It is visible to the admin user, and displays all users of the system. The admin user is able to delete or edit a user by clicking the icons in the actions column.



System Users

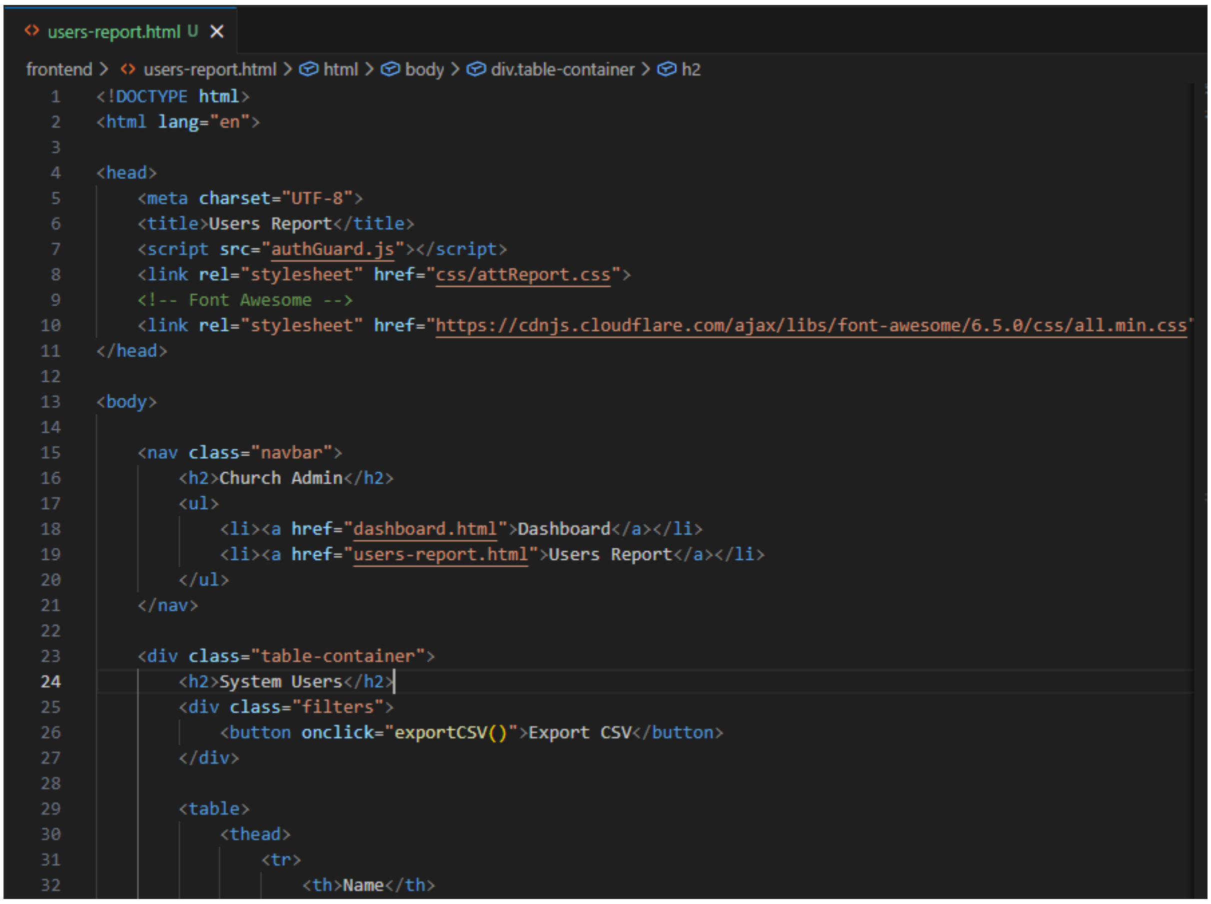
Name	Email	Role	Created	Actions
SysAdmin	admin@church.com	ADMIN	2026-01-19	✎ 🗑
Testing Test	testing@church.com	TREASURER	2026-01-20	✎ 🗑
Someone Test	someone@church.com	STAFF	2026-01-20	✎ 🗑
Peter Tosh	peter@church.com	TREASURER	2026-01-28	✎ 🗑

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.

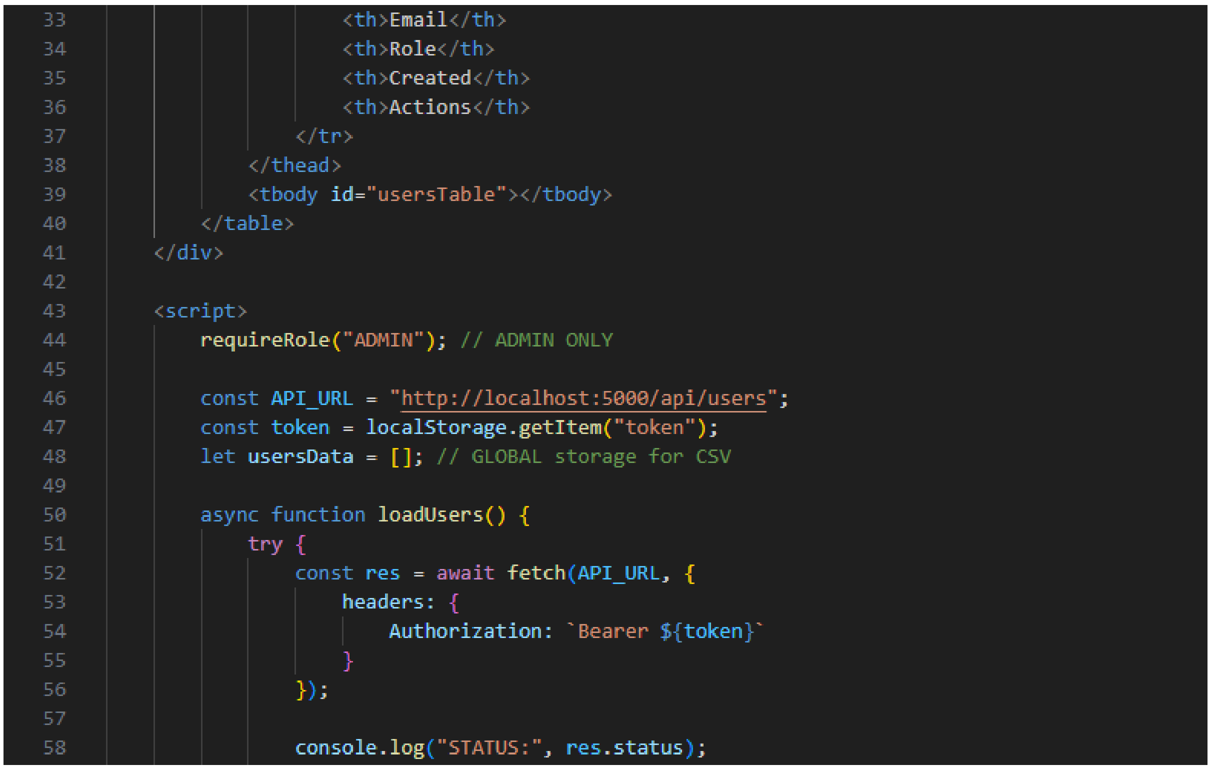


	A	B	C	D
1	Name	Email	Role	Created
2	SysAdmin	admin@church.com	ADMIN	2026/01/19
3	Testing Test	testing@church.com	TREASURER	2026/01/20
4	Someone Test	someone@church.com	STAFF	2026/01/20
5	Peter Tosh	peter@church.com	TREASURER	2026/01/28
6				

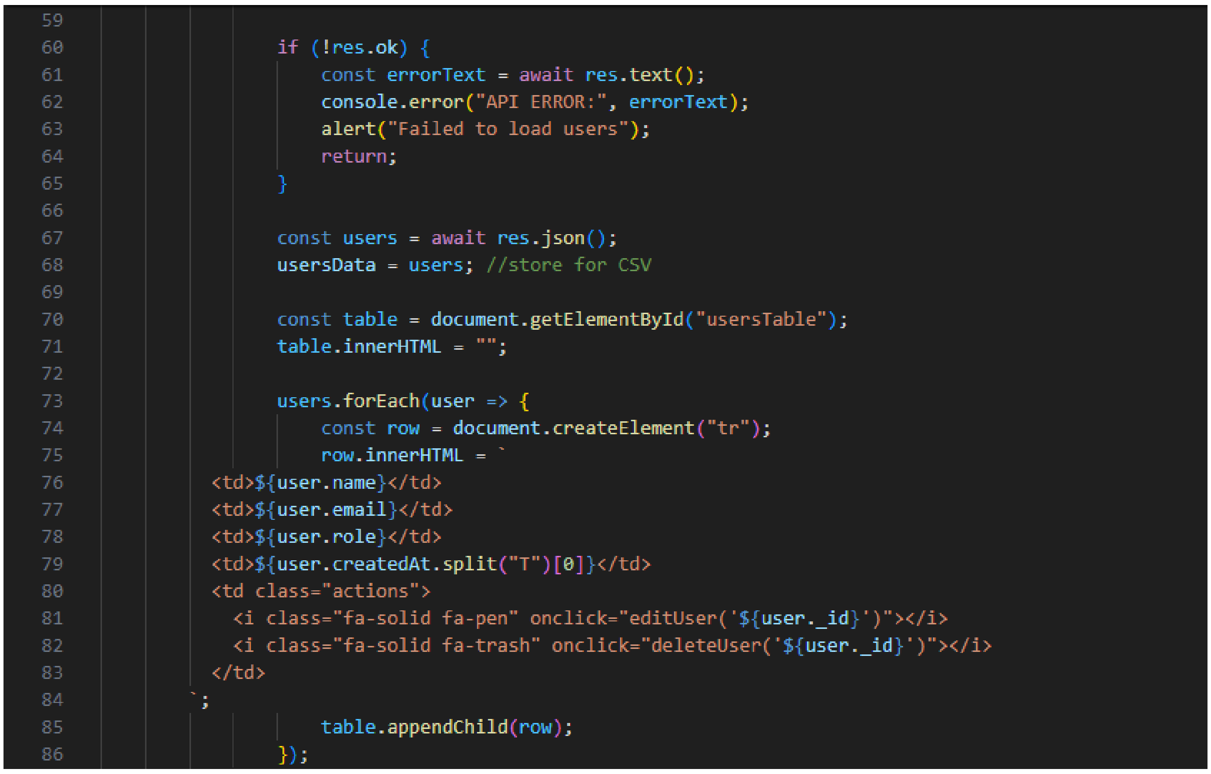
Below is the implemented users-report.html code to display the users report, and export the report to a csv file after clicking the button [Export CSV].


users-report.html U
frontend > users-report.html > html > body > div.table-container > h2

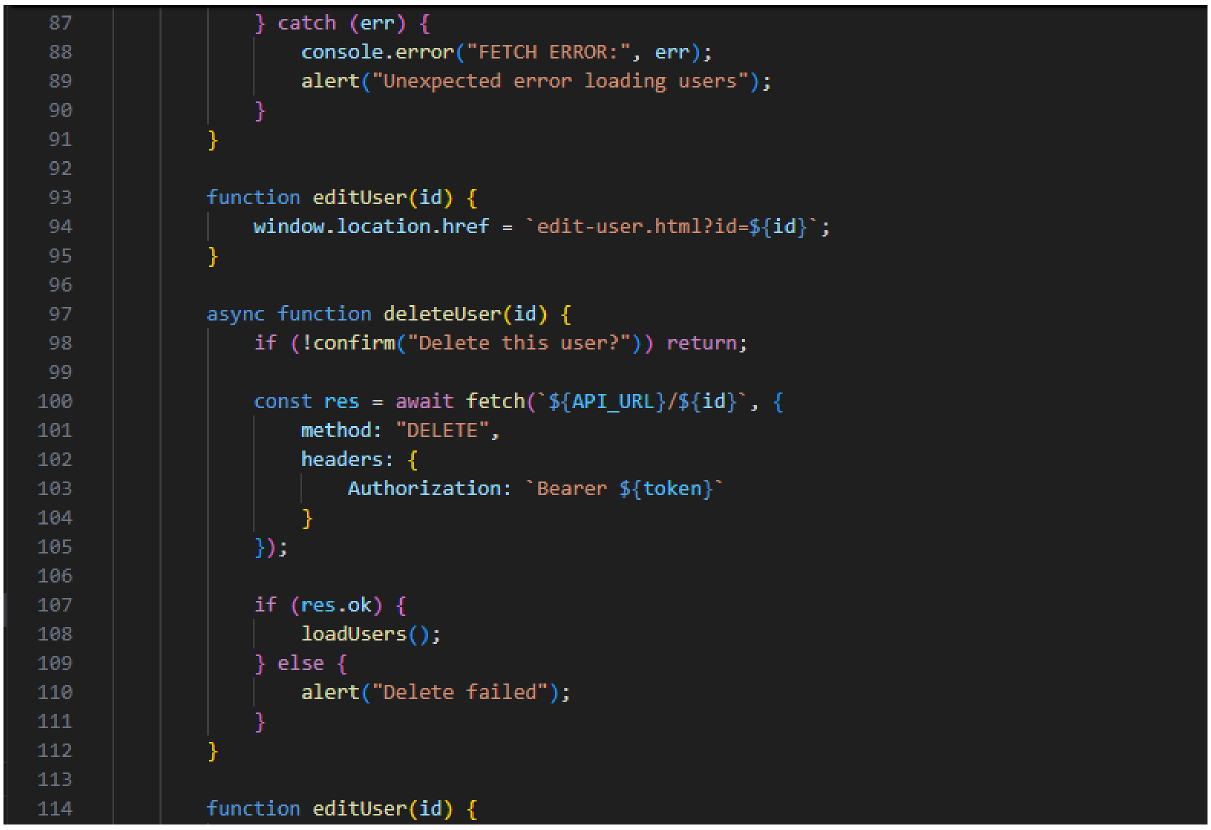
```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>Users Report</title>
    <script src="authGuard.js"></script>
    <link rel="stylesheet" href="css/attReport.css">
    <!-- Font Awesome -->
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.5.0/css/all.min.css'
</head>

<body>

    <nav class="navbar">
        <h2>Church Admin</h2>
        <ul>
            <li><a href="dashboard.html">Dashboard</a></li>
            <li><a href="users-report.html">Users Report</a></li>
        </ul>
    </nav>

    <div class="table-container">
        <h2>System Users</h2>
        <div class="filters">
            <button onclick="exportCSV()">Export CSV</button>
        </div>

        <table>
            <thead>
                <tr>
                    <th>Name</th>
```



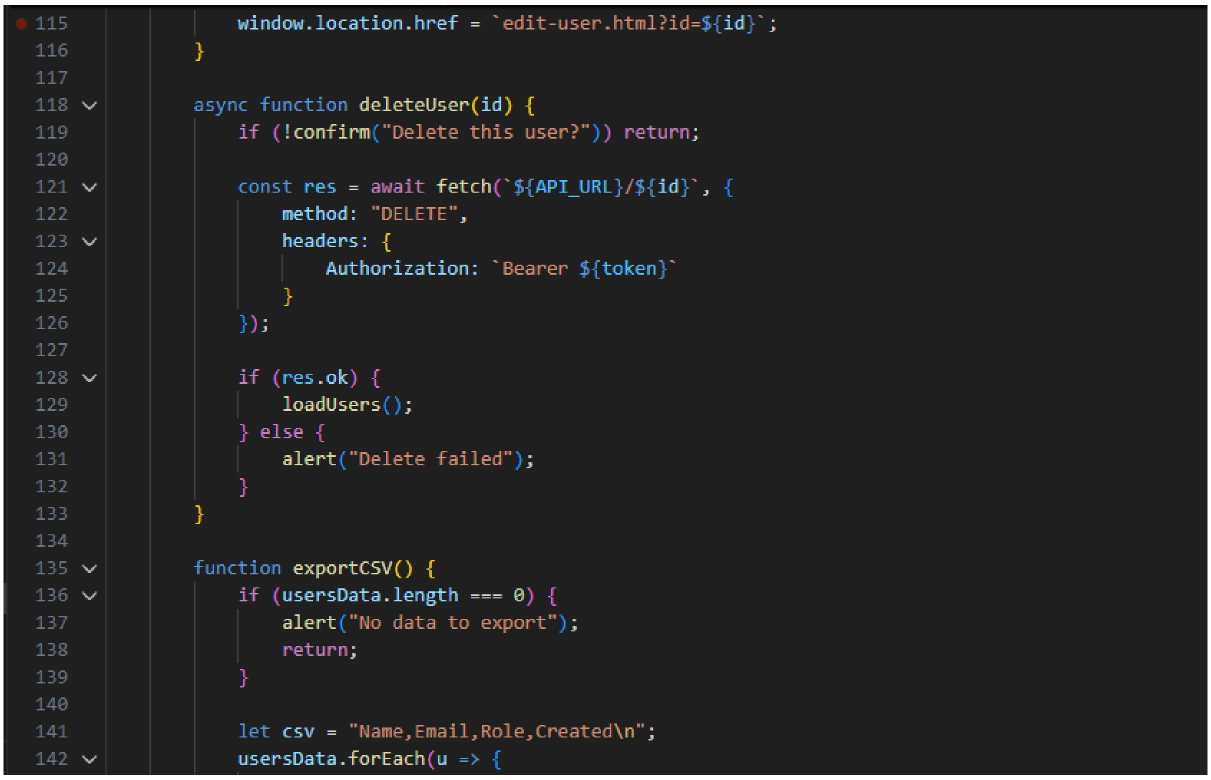
```html
                    <th>Email</th>
                    <th>Role</th>
                    <th>Created</th>
                    <th>Actions</th>
                </tr>
            </thead>
            <tbody id="usersTable"></tbody>
        </table>
    </div>

    <script>
        requireRole("ADMIN"); // ADMIN ONLY

        const API_URL = "http://localhost:5000/api/users";
        const token = localStorage.getItem("token");
        let usersData = []; // GLOBAL storage for CSV

        async function loadUsers() {
            try {
                const res = await fetch(API_URL, {
                    headers: {
                        Authorization: `Bearer ${token}`
                    }
                });

                console.log("STATUS:", res.status);
```



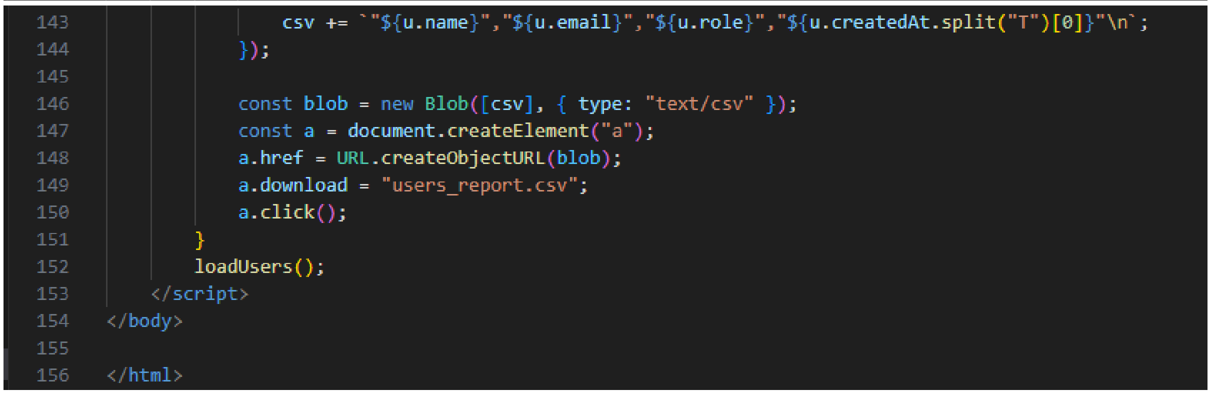
```
                if (!res.ok) {
                    const errorText = await res.text();
                    console.error("API ERROR:", errorText);
                    alert("Failed to load users");
                    return;
                }

                const users = await res.json();
                usersData = users; //store for CSV

                const table = document.getElementById("usersTable");
                table.innerHTML = "";

                users.forEach(user => {
                    const row = document.createElement("tr");
                    row.innerHTML = `
          <td>${user.name}</td>
          <td>${user.email}</td>
          <td>${user.role}</td>
          <td>${user.createdAt.split("T")[0]}</td>
          <td class="actions">
            <i class="fa-solid fa-pen" onclick="editUser('${user._id}')"></i>
            <i class="fa-solid fa-trash" onclick="deleteUser('${user._id}')"></i>
          </td>
        `;
                    table.appendChild(row);
                });
```



```
            } catch (err) {
                console.error("FETCH ERROR:", err);
                alert("Unexpected error loading users");
            }
        }

        function editUser(id) {
            window.location.href = `edit-user.html?id=${id}`;
        }

        async function deleteUser(id) {
            if (!confirm("Delete this user?")) return;

            const res = await fetch(`${API_URL}/${id}`, {
                method: "DELETE",
                headers: {
                    Authorization: `Bearer ${token}`
                }
            });

            if (res.ok) {
                loadUsers();
            } else {
                alert("Delete failed");
            }
        }

        function editUser(id) {
```



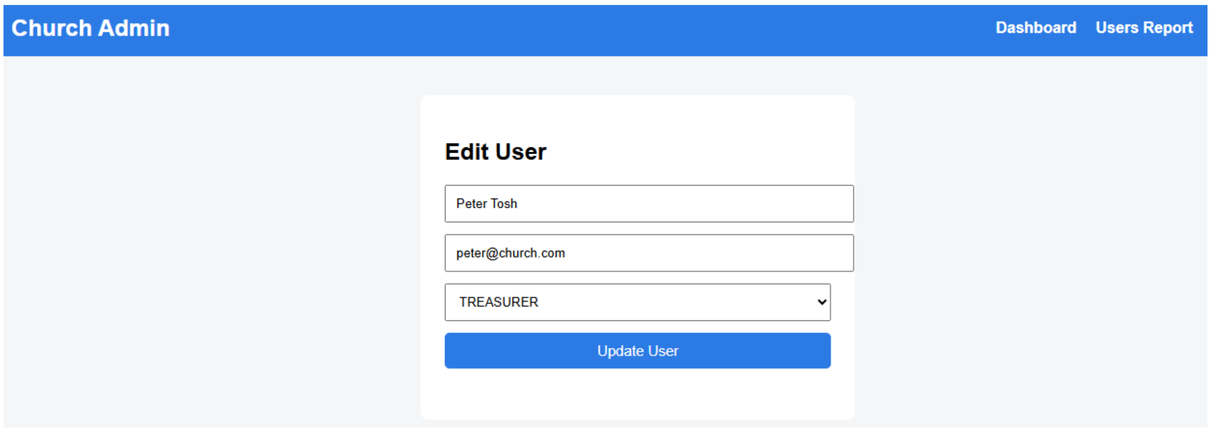
```
            window.location.href = `edit-user.html?id=${id}`;
        }

        async function deleteUser(id) {
            if (!confirm("Delete this user?")) return;

            const res = await fetch(`${API_URL}/${id}`, {
                method: "DELETE",
                headers: {
                    Authorization: `Bearer ${token}`
                }
            });

            if (res.ok) {
                loadUsers();
            } else {
                alert("Delete failed");
            }
        }

        function exportCSV() {
            if (usersData.length === 0) {
                alert("No data to export");
                return;
            }

            let csv = "Name,Email,Role,Created\n";
            usersData.forEach(u => {
```



```
                csv += `"${u.name}","${u.email}","${u.role}","${u.createdAt.split("T")[0]}"\n`;
            });

            const blob = new Blob([csv], { type: "text/csv" });
            const a = document.createElement("a");
            a.href = URL.createObjectURL(blob);
            a.download = "users_report.csv";
            a.click();
        }
        loadUsers();
    </script>
</body>

</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.



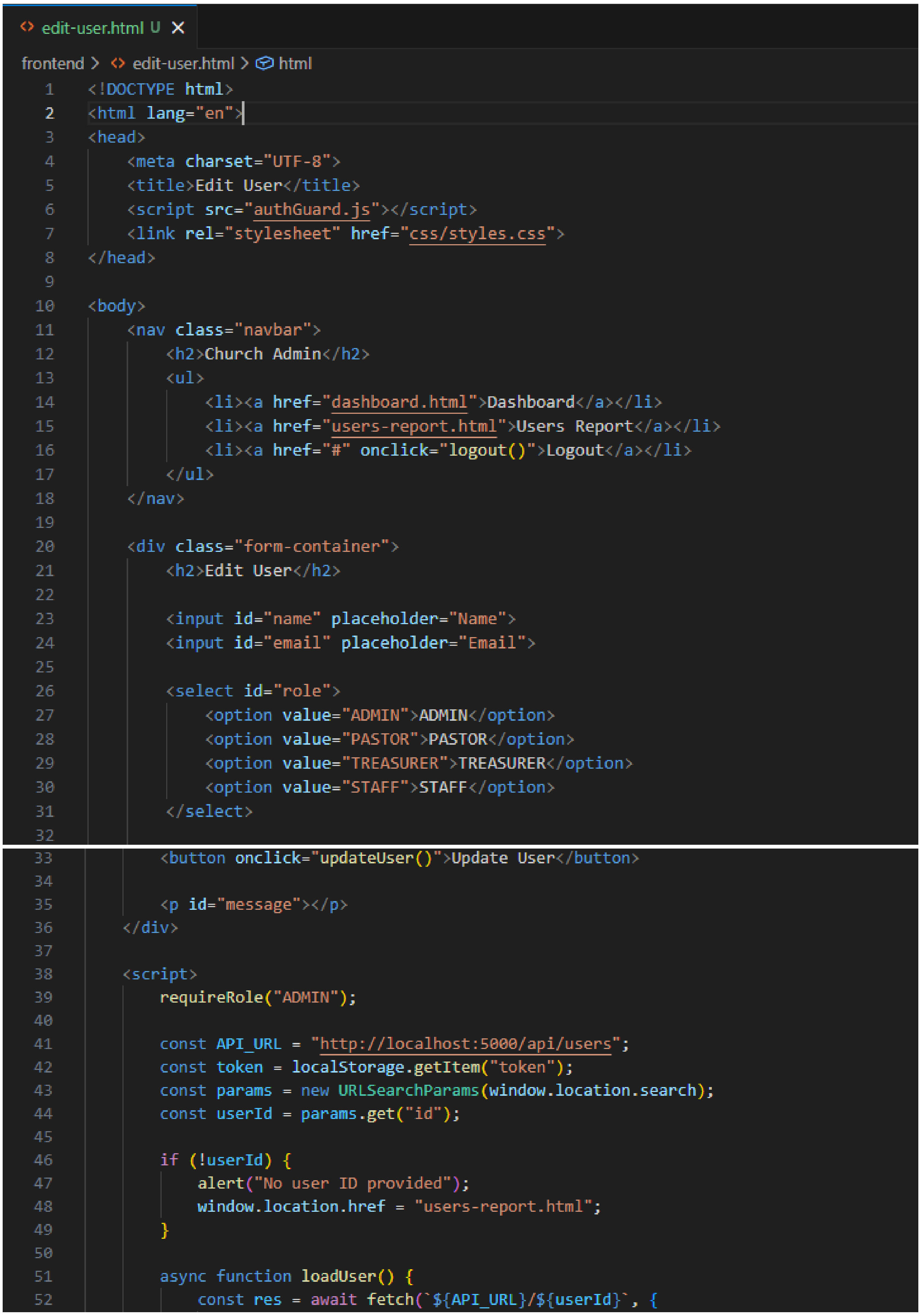
Edit User

Peter Tosh

peter@church.com

TREASURER

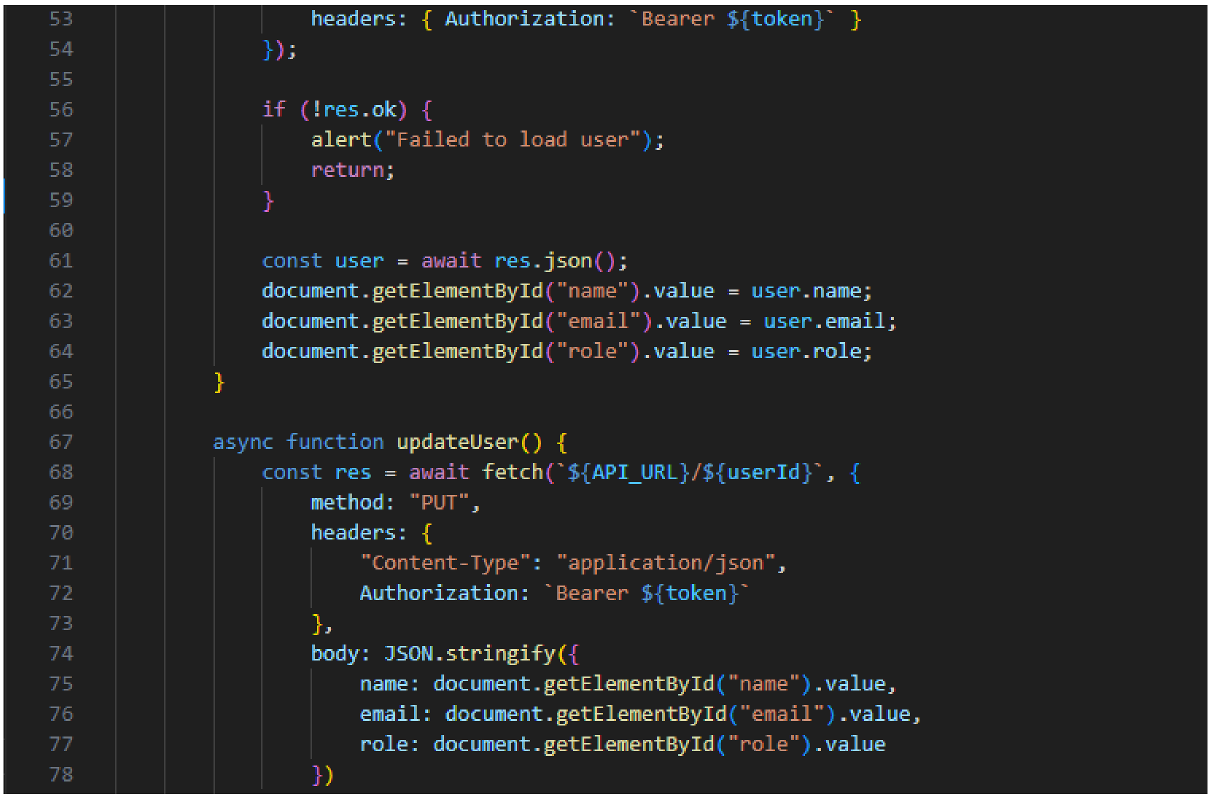
Below is the implemented edit-user.html code to display the above screen, and update user details.



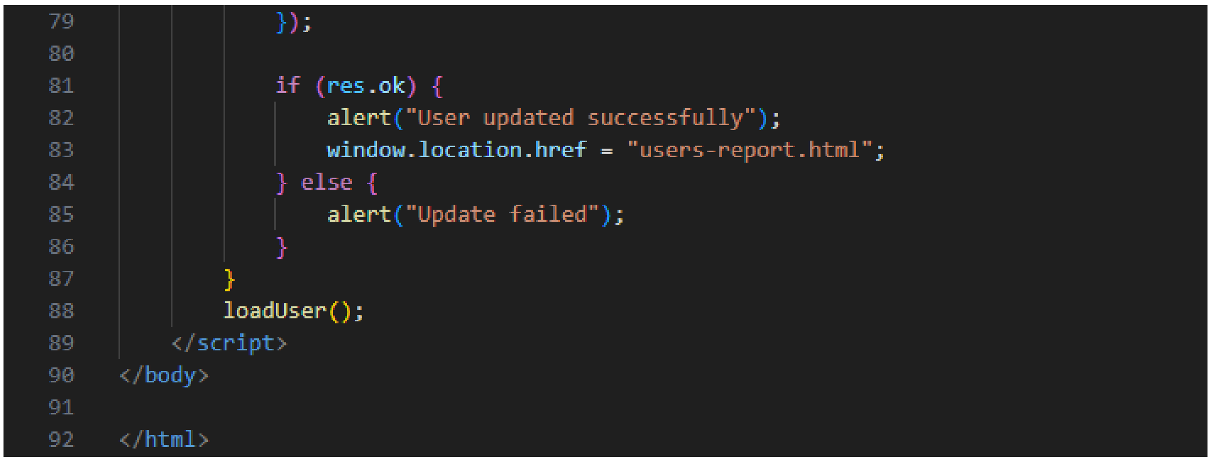
edit-user.html U ×

frontend > edit-user.html > html

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Edit User</title>
    <script src="authGuard.js"></script>
    <link rel="stylesheet" href="css/styles.css">
</head>

<body>
    <nav class="navbar">
        <h2>Church Admin</h2>
        <ul>
            <li><a href="dashboard.html">Dashboard</a></li>
            <li><a href="users-report.html">Users Report</a></li>
            <li><a href="#" onclick="logout()">Logout</a></li>
        </ul>
    </nav>

    <div class="form-container">
        <h2>Edit User</h2>

        <input id="name" placeholder="Name">
        <input id="email" placeholder="Email">

        <select id="role">
            <option value="ADMIN">ADMIN</option>
            <option value="PASTOR">PASTOR</option>
            <option value="TREASURER">TREASURER</option>
            <option value="STAFF">STAFF</option>
        </select>

        <button onclick="updateUser()">Update User</button>

        <p id="message"></p>
    </div>

    <script>
        requireRole("ADMIN");

        const API_URL = "http://localhost:5000/api/users";
        const token = localStorage.getItem("token");
        const params = new URLSearchParams(window.location.search);
        const userId = params.get("id");

        if (!userId) {
            alert("No user ID provided");
            window.location.href = "users-report.html";
        }

        async function loadUser() {
            const res = await fetch(`${API_URL}/${userId}`, {
```



```
                headers: { Authorization: `Bearer ${token}` }
            });

            if (!res.ok) {
                alert("Failed to load user");
                return;
            }

            const user = await res.json();
            document.getElementById("name").value = user.name;
            document.getElementById("email").value = user.email;
            document.getElementById("role").value = user.role;
        }

        async function updateUser() {
            const res = await fetch(`${API_URL}/${userId}`, {
                method: "PUT",
                headers: {
                    "Content-Type": "application/json",
                    Authorization: `Bearer ${token}`
                },
                body: JSON.stringify({
                    name: document.getElementById("name").value,
                    email: document.getElementById("email").value,
                    role: document.getElementById("role").value
                })
```



```
            });

            if (res.ok) {
                alert("User updated successfully");
                window.location.href = "users-report.html";
            } else {
                alert("Update failed");
            }
        }
        loadUser();
    </script>
</body>

</html>
```

5.10 Member Management

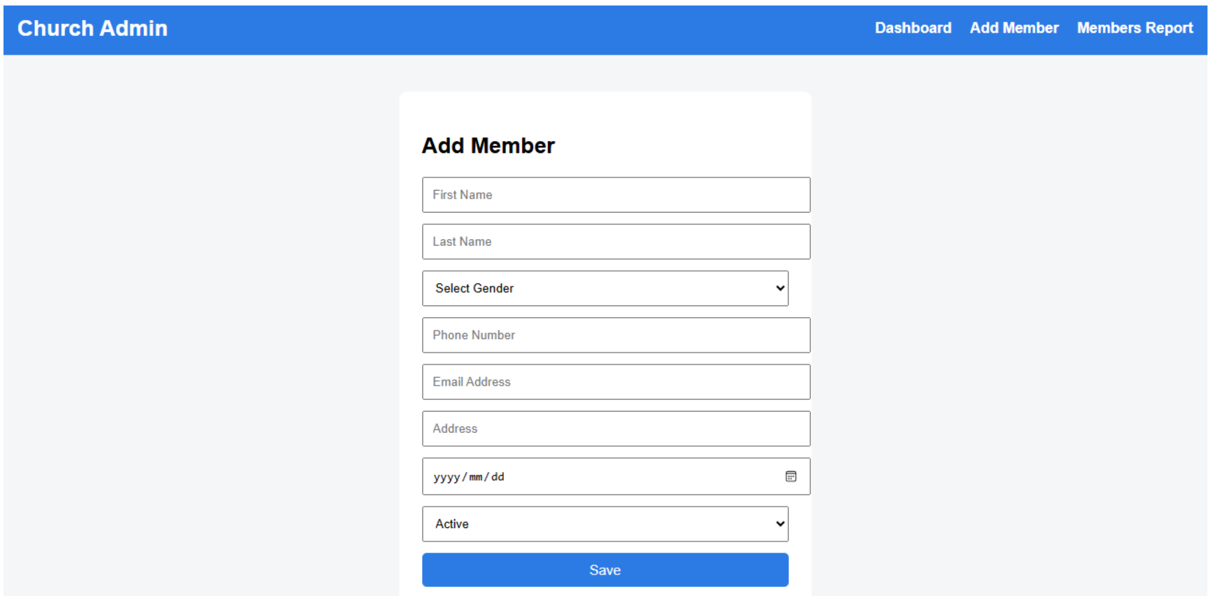
Implemented modules include:

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

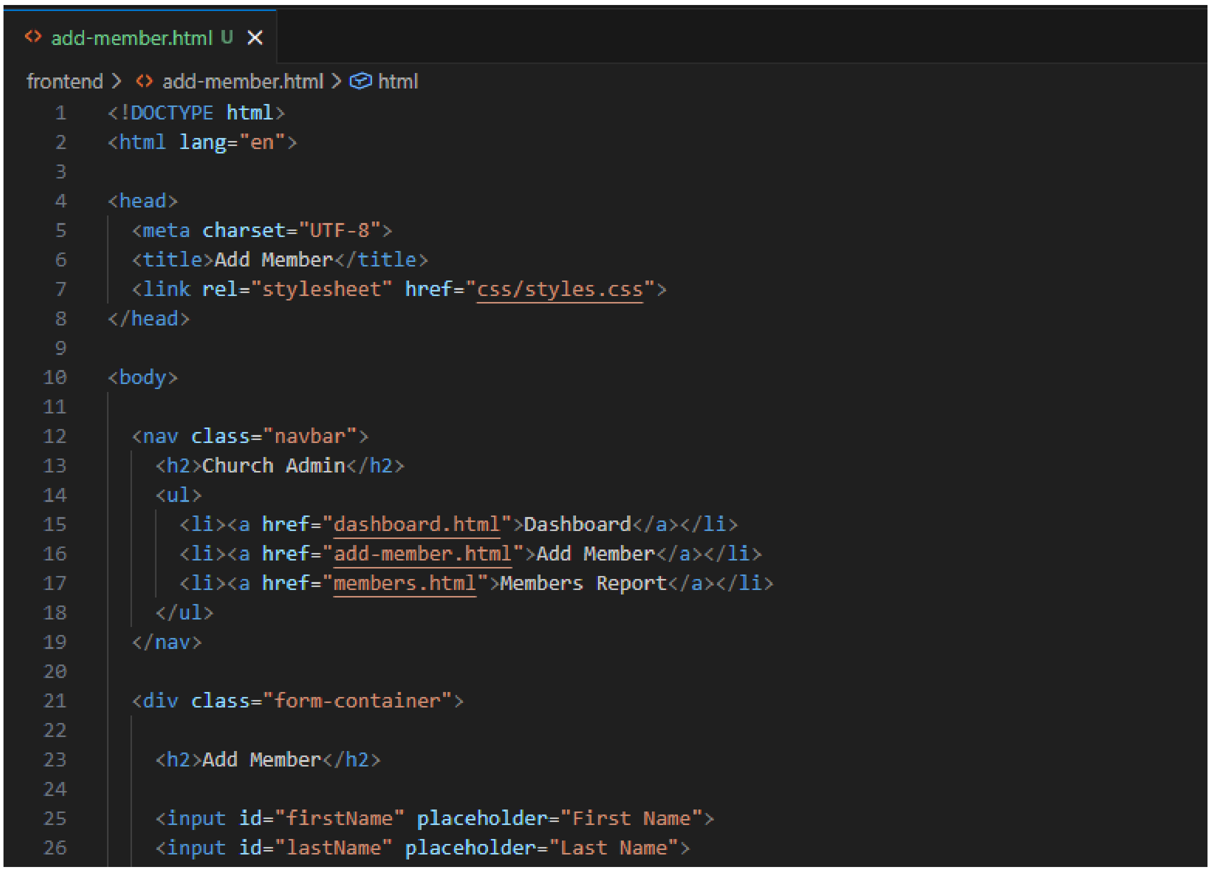
These features enable CRUD operations on church member records.

5.10(i) add-member

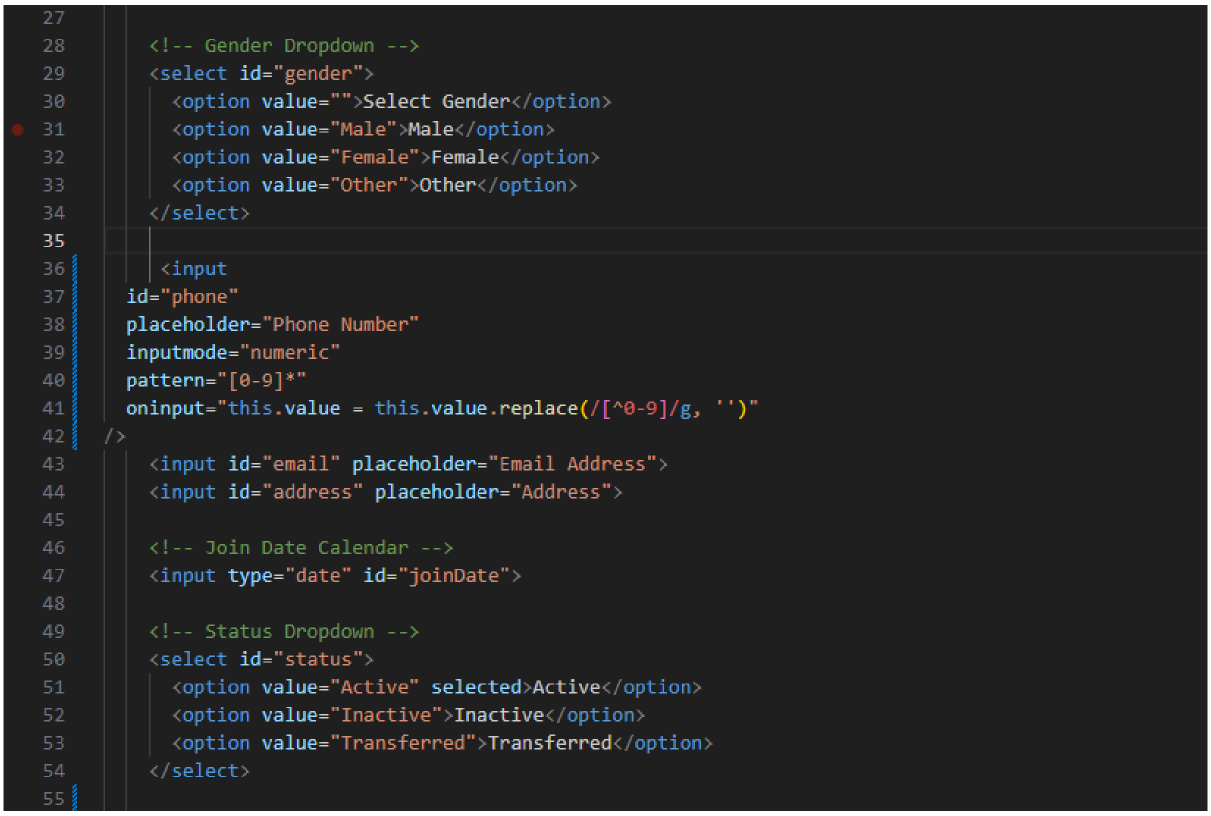
Below is the add-member screen, visible to the admin user.



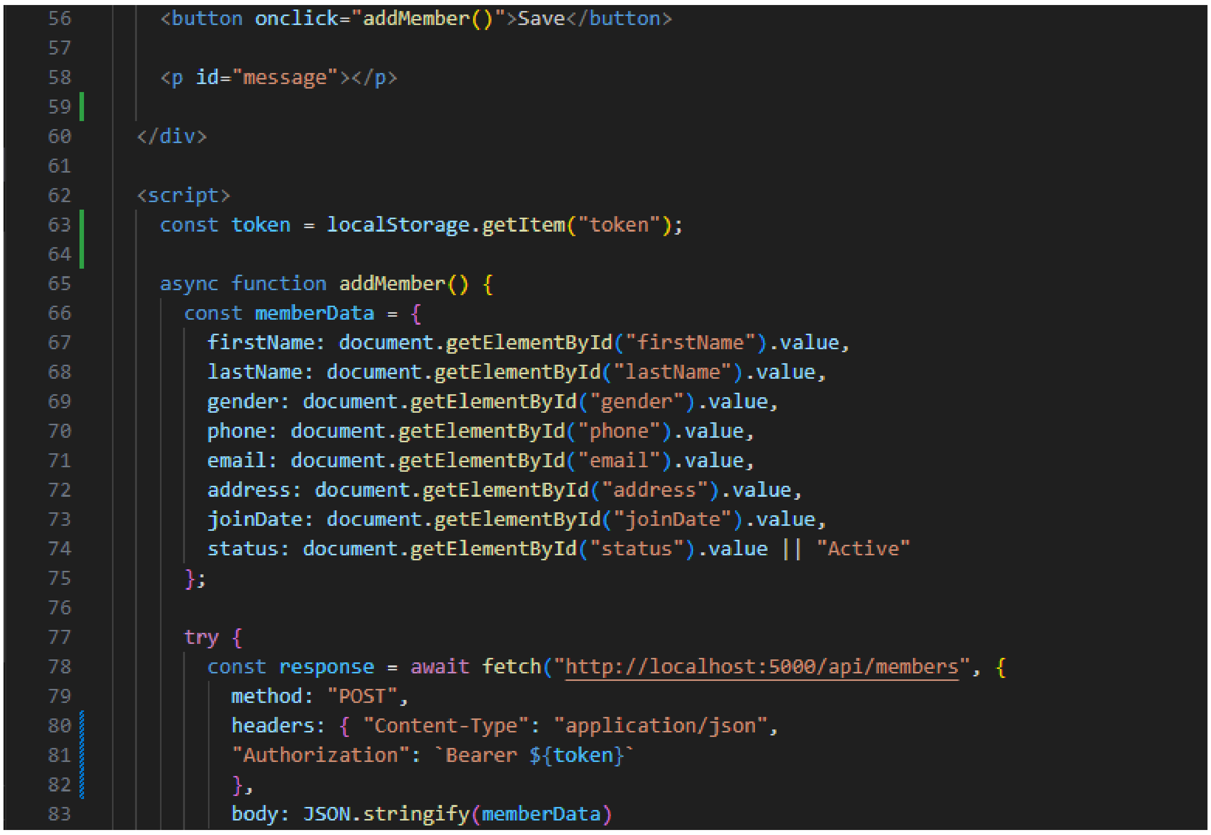
Below is the implemented add-member.html code to display the above screen, and post the entered details to the mongoDB database.



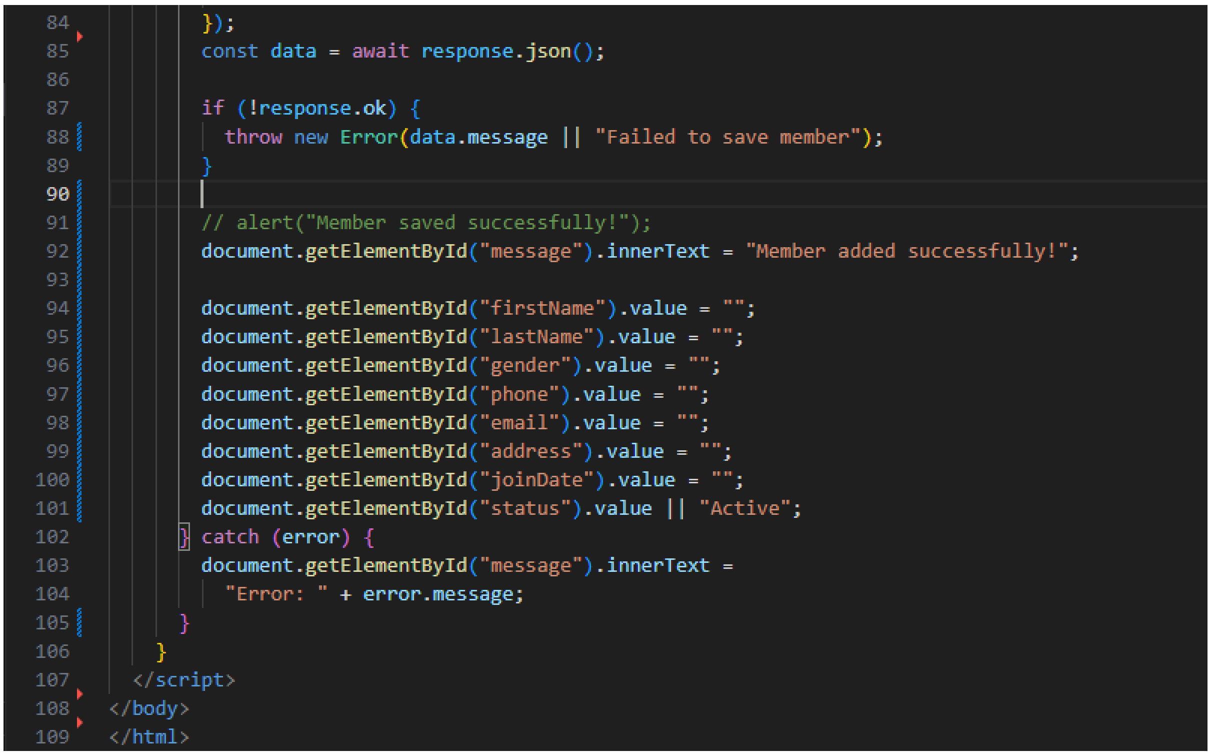
```html
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <title>Add Member</title>
  <link rel="stylesheet" href="css/styles.css">
</head>

<body>

  <nav class="navbar">
    <h2>Church Admin</h2>
    <ul>
      <li><a href="dashboard.html">Dashboard</a></li>
      <li><a href="add-member.html">Add Member</a></li>
      <li><a href="members.html">Members Report</a></li>
    </ul>
  </nav>

  <div class="form-container">

    <h2>Add Member</h2>

    <input id="firstName" placeholder="First Name">
    <input id="lastName" placeholder="Last Name">
```



```html

    <!-- Gender Dropdown -->
    <select id="gender">
      <option value="">Select Gender</option>
      <option value="Male">Male</option>
      <option value="Female">Female</option>
      <option value="Other">Other</option>
    </select>

     <input
  id="phone"
  placeholder="Phone Number"
  inputmode="numeric"
  pattern="[0-9]*"
  oninput="this.value = this.value.replace(/[^0-9]/g, '')"
/>
    <input id="email" placeholder="Email Address">
    <input id="address" placeholder="Address">

    <!-- Join Date Calendar -->
    <input type="date" id="joinDate">

    <!-- Status Dropdown -->
    <select id="status">
      <option value="Active" selected>Active</option>
      <option value="Inactive">Inactive</option>
      <option value="Transferred">Transferred</option>
    </select>

```

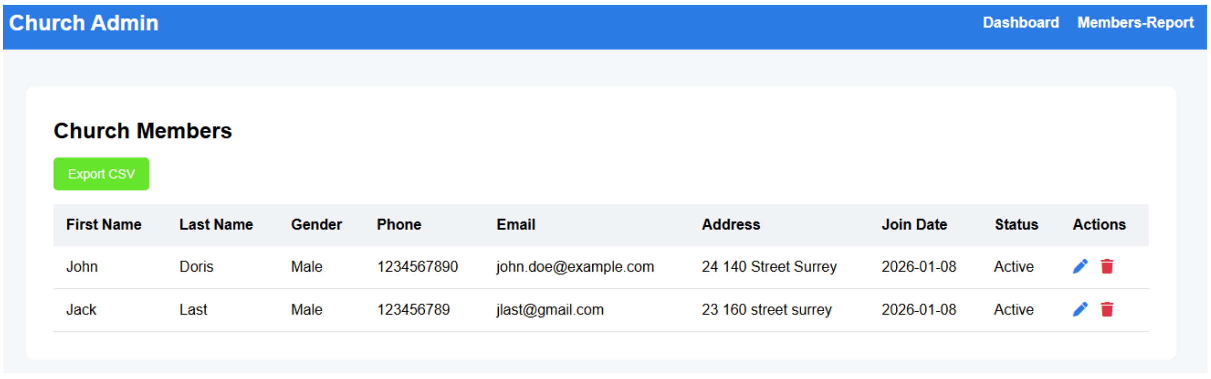


```html
    <button onclick="addMember()">Save</button>

    <p id="message"></p>

  </div>

  <script>
    const token = localStorage.getItem("token");

    async function addMember() {
      const memberData = {
        firstName: document.getElementById("firstName").value,
        lastName: document.getElementById("lastName").value,
        gender: document.getElementById("gender").value,
        phone: document.getElementById("phone").value,
        email: document.getElementById("email").value,
        address: document.getElementById("address").value,
        joinDate: document.getElementById("joinDate").value,
        status: document.getElementById("status").value || "Active"
      };

      try {
        const response = await fetch("http://localhost:5000/api/members", {
          method: "POST",
          headers: { "Content-Type": "application/json",
          "Authorization": `Bearer ${token}`
          },
          body: JSON.stringify(memberData)
```


```
        });
        const data = await response.json();

        if (!response.ok) {
          throw new Error(data.message || "Failed to save member");
        }

        // alert("Member saved successfully!");
        document.getElementById("message").innerText = "Member added successfully!";

        document.getElementById("firstName").value = "";
        document.getElementById("lastName").value = "";
        document.getElementById("gender").value = "";
        document.getElementById("phone").value = "";
        document.getElementById("email").value = "";
        document.getElementById("address").value = "";
        document.getElementById("joinDate").value = "";
        document.getElementById("status").value || "Active";
      } catch (error) {
        document.getElementById("message").innerText =
          "Error: " + error.message;
      }
    }
  </script>
</body>
</html>
```

5.10(ii) members-report

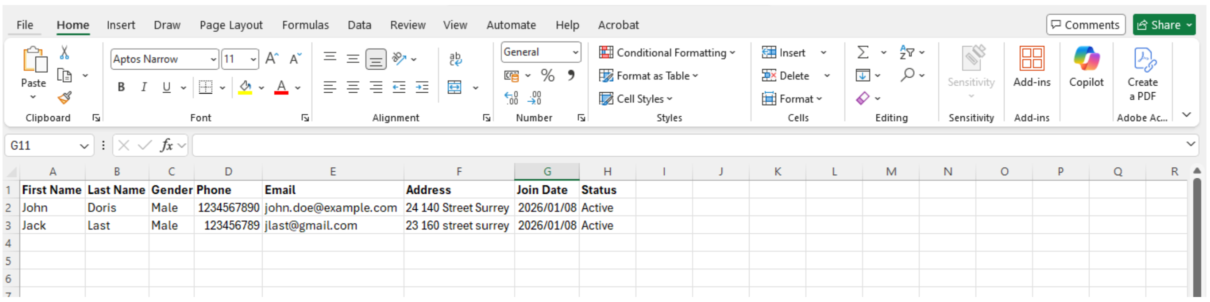
Below is the members report. It is visible to the admin user, and displays all members of the church. The admin user is able to delete or edit any member details by clicking the icons in the actions column.



Church Members

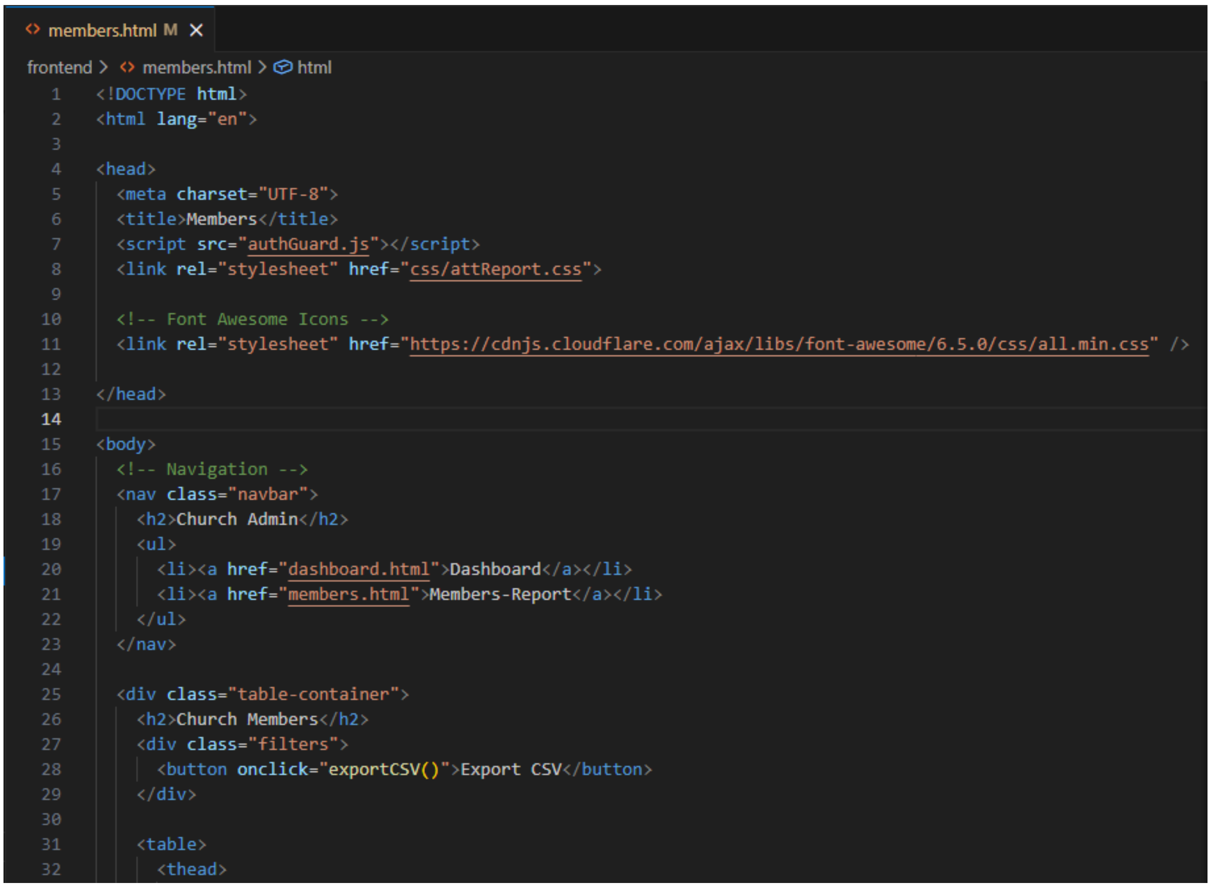
First Name	Last Name	Gender	Phone	Email	Address	Join Date	Status	Actions
John	Doris	Male	1234567890	john.doe@example.com	24 140 Street Surrey	2026-01-08	Active	✎ 🗑
Jack	Last	Male	123456789	jlast@gmail.com	23 160 street surrey	2026-01-08	Active	✎ 🗑

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.

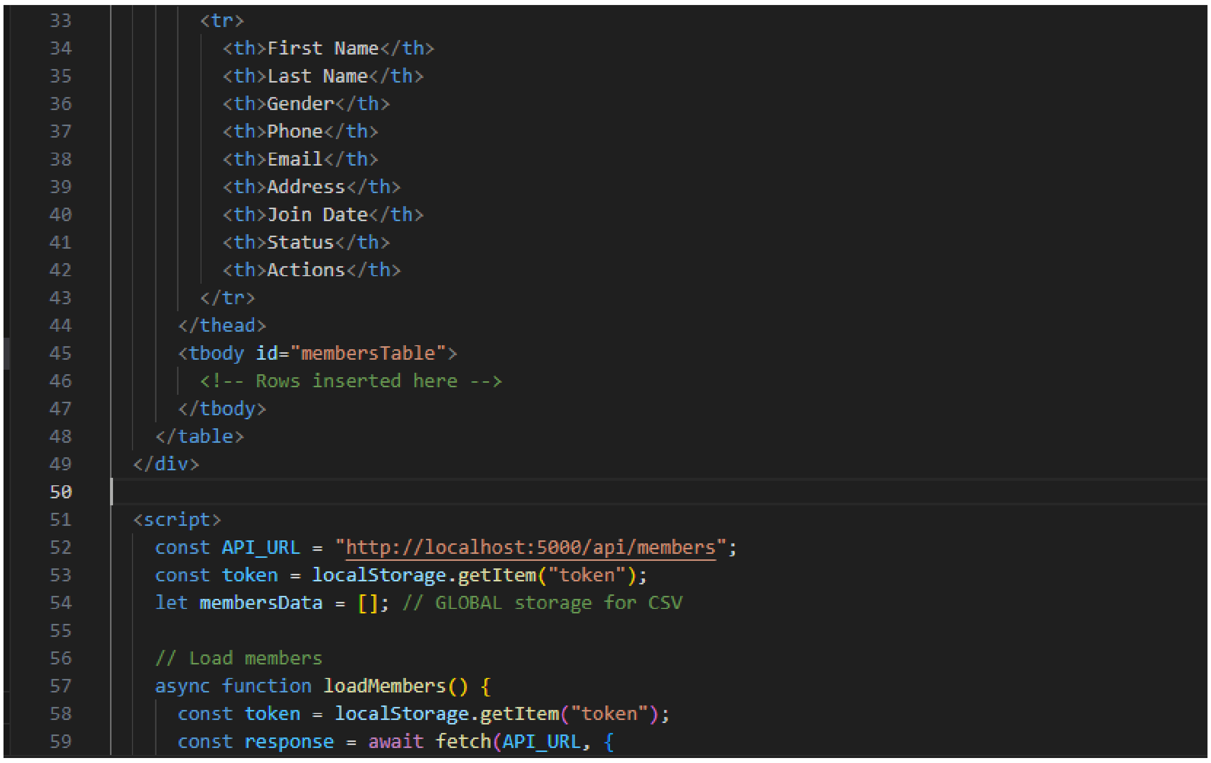


First Name	Last Name	Gender	Phone	Email	Address	Join Date	Status
John	Doris	Male	1234567890	john.doe@example.com	24 140 Street Surrey	2026/01/08	Active
Jack	Last	Male	123456789	jlast@gmail.com	23 160 street surrey	2026/01/08	Active

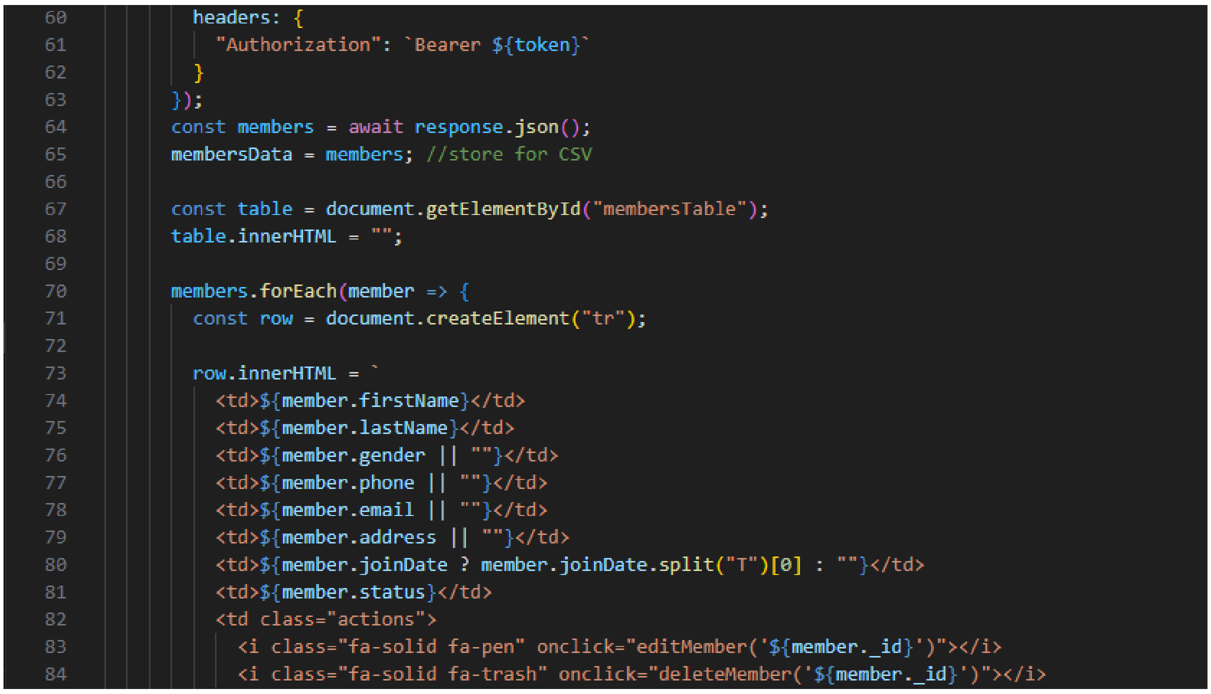
Below is the implemented members.html code to display the members report, and export the report to a csv file after clicking the button [Export CSV].



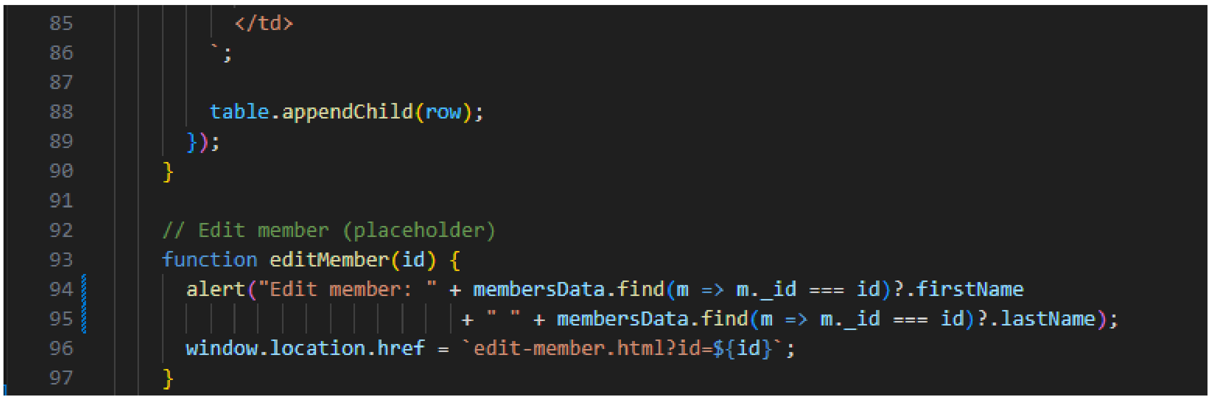
```html
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <title>Members</title>
  <script src="authGuard.js"></script>
  <link rel="stylesheet" href="css/attReport.css">

  <!-- Font Awesome Icons -->
  <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.5.0/css/all.min.css" />

</head>

<body>
  <!-- Navigation -->
  <nav class="navbar">
    <h2>Church Admin</h2>
    <ul>
      <li><a href="dashboard.html">Dashboard</a></li>
      <li><a href="members.html">Members-Report</a></li>
    </ul>
  </nav>

  <div class="table-container">
    <h2>Church Members</h2>
    <div class="filters">
      <button onclick="exportCSV()">Export CSV</button>
    </div>

    <table>
      <thead>
```



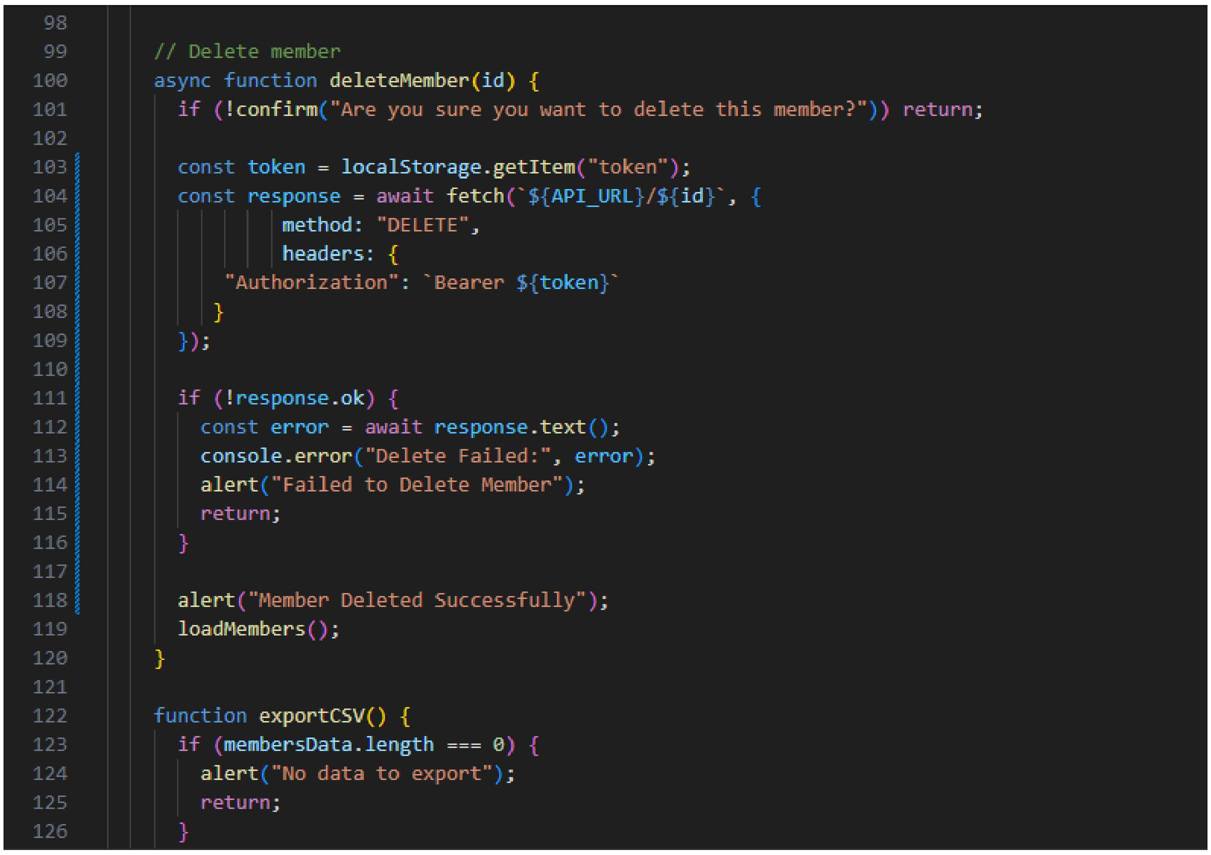
```html
      <tr>
        <th>First Name</th>
        <th>Last Name</th>
        <th>Gender</th>
        <th>Phone</th>
        <th>Email</th>
        <th>Address</th>
        <th>Join Date</th>
        <th>Status</th>
        <th>Actions</th>
      </tr>
    </thead>
    <tbody id="membersTable">
      <!-- Rows inserted here -->
    </tbody>
  </table>
</div>

<script>
  const API_URL = "http://localhost:5000/api/members";
  const token = localStorage.getItem("token");
  let membersData = []; // GLOBAL storage for CSV

  // Load members
  async function loadMembers() {
    const token = localStorage.getItem("token");
    const response = await fetch(API_URL, {
```



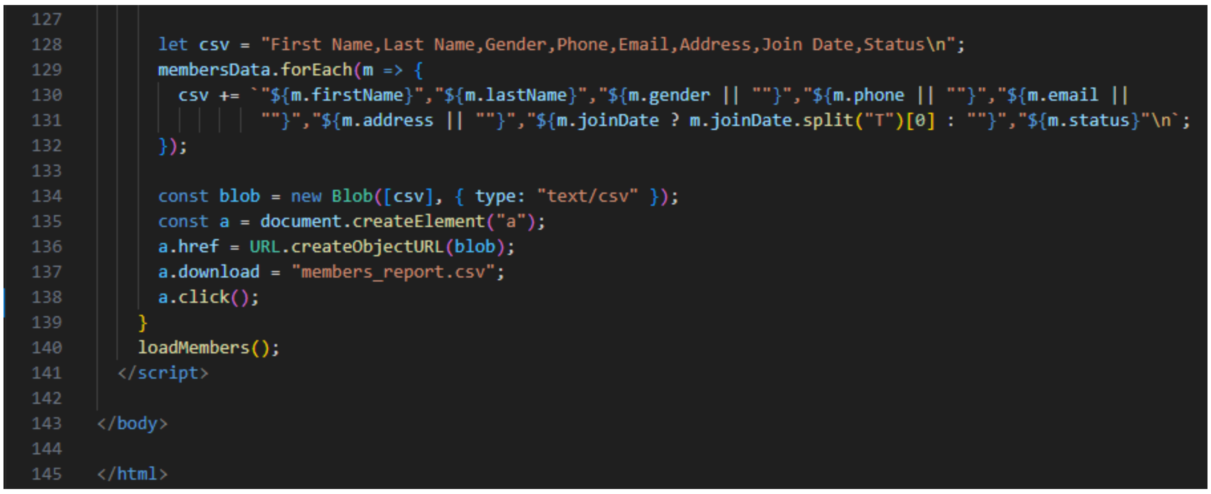
```javascript
      headers: {
        "Authorization": `Bearer ${token}`
      }
    });
    const members = await response.json();
    membersData = members; //store for CSV

    const table = document.getElementById("membersTable");
    table.innerHTML = "";

    members.forEach(member => {
      const row = document.createElement("tr");

      row.innerHTML = `
        <td>${member.firstName}</td>
        <td>${member.lastName}</td>
        <td>${member.gender || ""}</td>
        <td>${member.phone || ""}</td>
        <td>${member.email || ""}</td>
        <td>${member.address || ""}</td>
        <td>${member.joinDate ? member.joinDate.split("T")[0] : ""}</td>
        <td>${member.status}</td>
        <td class="actions">
          <i class="fa-solid fa-pen" onclick="editMember('${member._id}')"></i>
          <i class="fa-solid fa-trash" onclick="deleteMember('${member._id}')"></i>
```



```javascript
        </td>
      `;

      table.appendChild(row);
    });
  }

  // Edit member (placeholder)
  function editMember(id) {
    alert("Edit member: " + membersData.find(m => m._id === id)?.firstName
                             + " " + membersData.find(m => m._id === id)?.lastName);
    window.location.href = `edit-member.html?id=${id}`;
  }
```



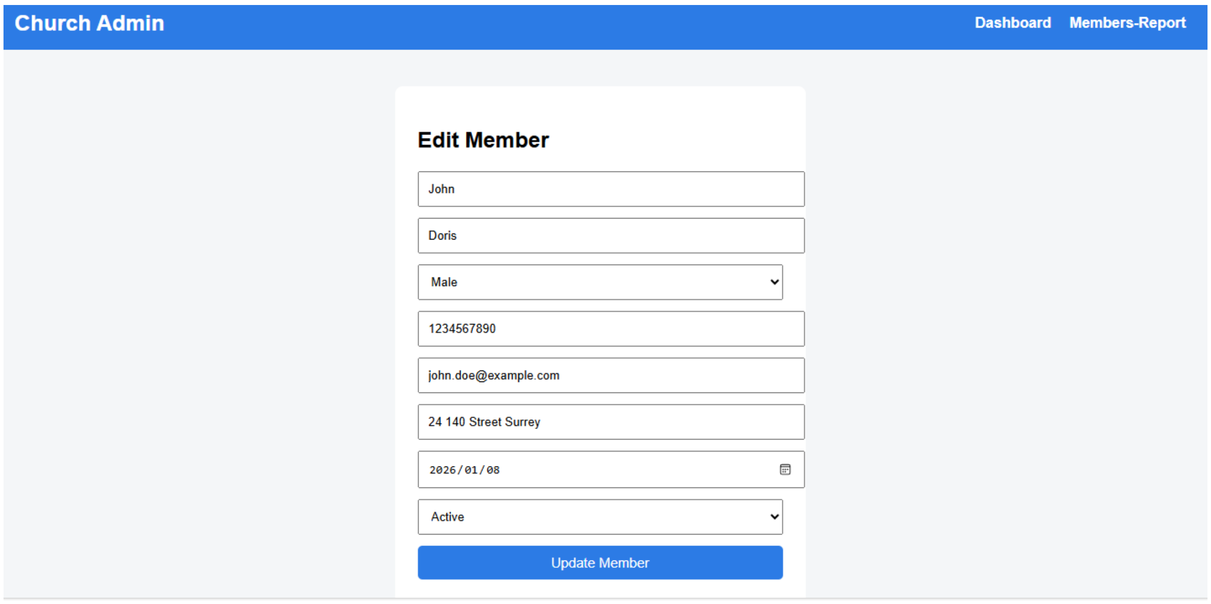
```javascript

    // Delete member
    async function deleteMember(id) {
      if (!confirm("Are you sure you want to delete this member?")) return;

      const token = localStorage.getItem("token");
      const response = await fetch(`${API_URL}/${id}`, {
              method: "DELETE",
              headers: {
          "Authorization": `Bearer ${token}`
        }
      });

      if (!response.ok) {
        const error = await response.text();
        console.error("Delete Failed:", error);
        alert("Failed to Delete Member");
        return;
      }

      alert("Member Deleted Successfully");
      loadMembers();
    }

    function exportCSV() {
      if (membersData.length === 0) {
        alert("No data to export");
        return;
      }
```



```javascript

      let csv = "First Name,Last Name,Gender,Phone,Email,Address,Join Date,Status\n";
      membersData.forEach(m => {
        csv += `"${m.firstName}","${m.lastName}","${m.gender || ""}","${m.phone || ""}","${m.email ||
                ""}","${m.address || ""}","${m.joinDate ? m.joinDate.split("T")[0] : ""}","${m.status}"\n`;
      });

      const blob = new Blob([csv], { type: "text/csv" });
      const a = document.createElement("a");
      a.href = URL.createObjectURL(blob);
      a.download = "members_report.csv";
      a.click();
    }
    loadMembers();
  </script>

</body>

</html>
```

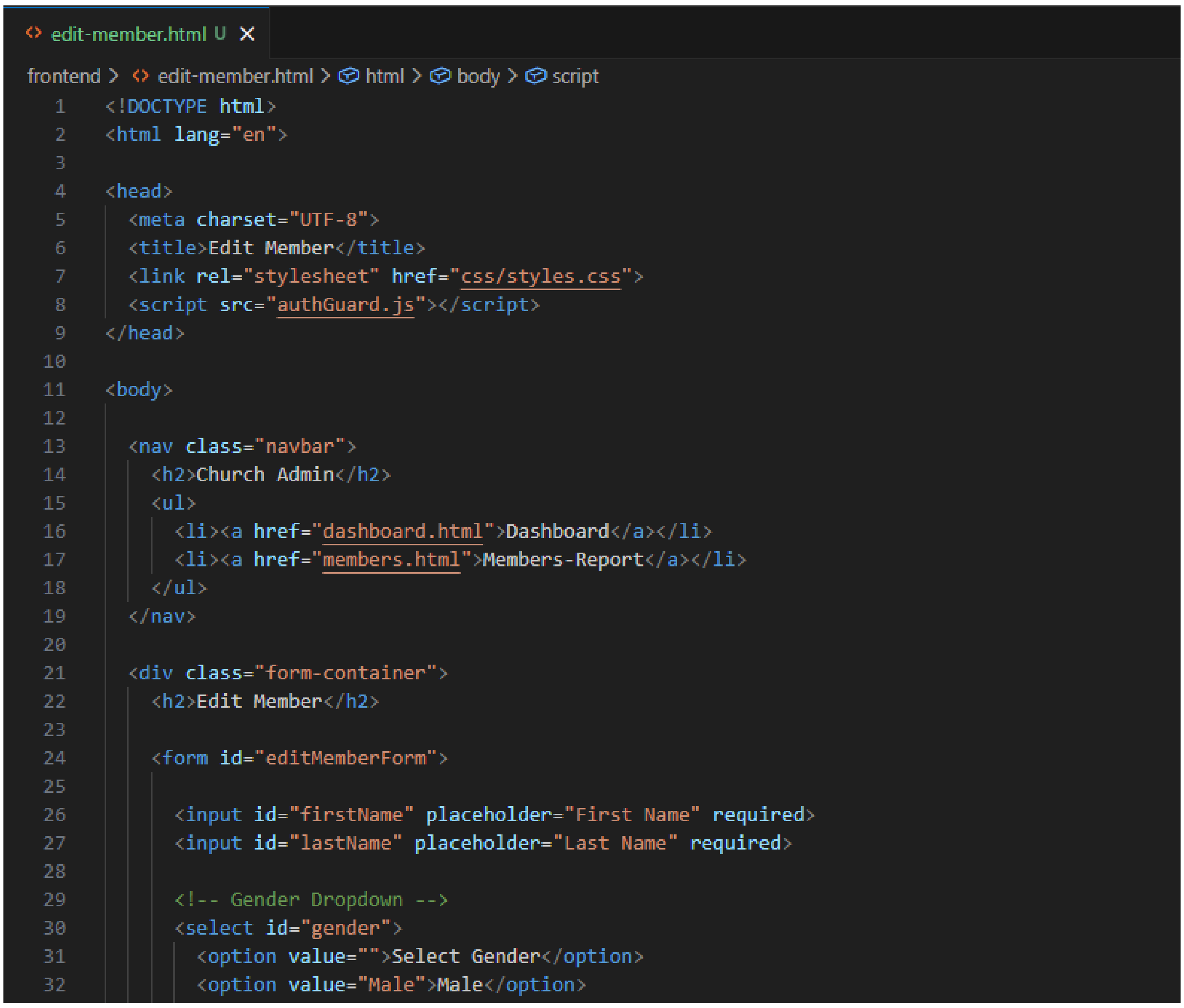
5.10(iii) edit-member

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.

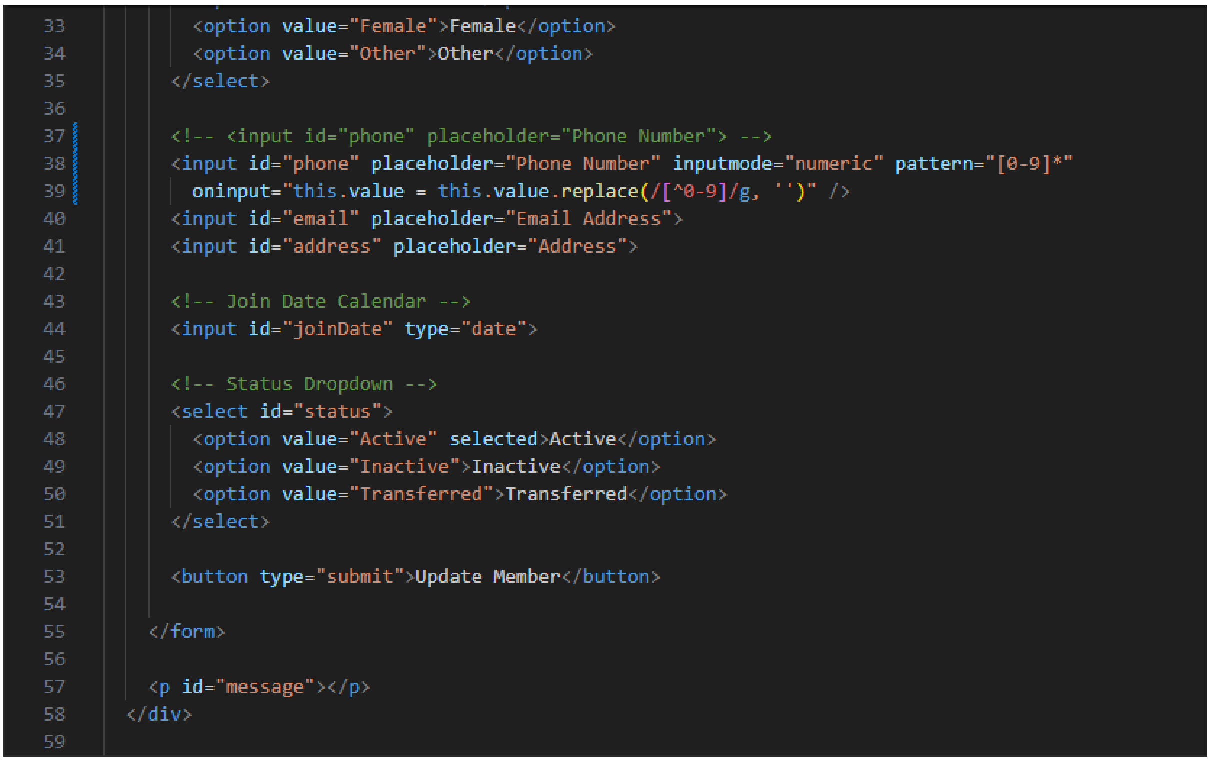


John
Doris
Male
1234567890
john.doe@example.com
24 140 Street Surrey
2026/01/08
Active

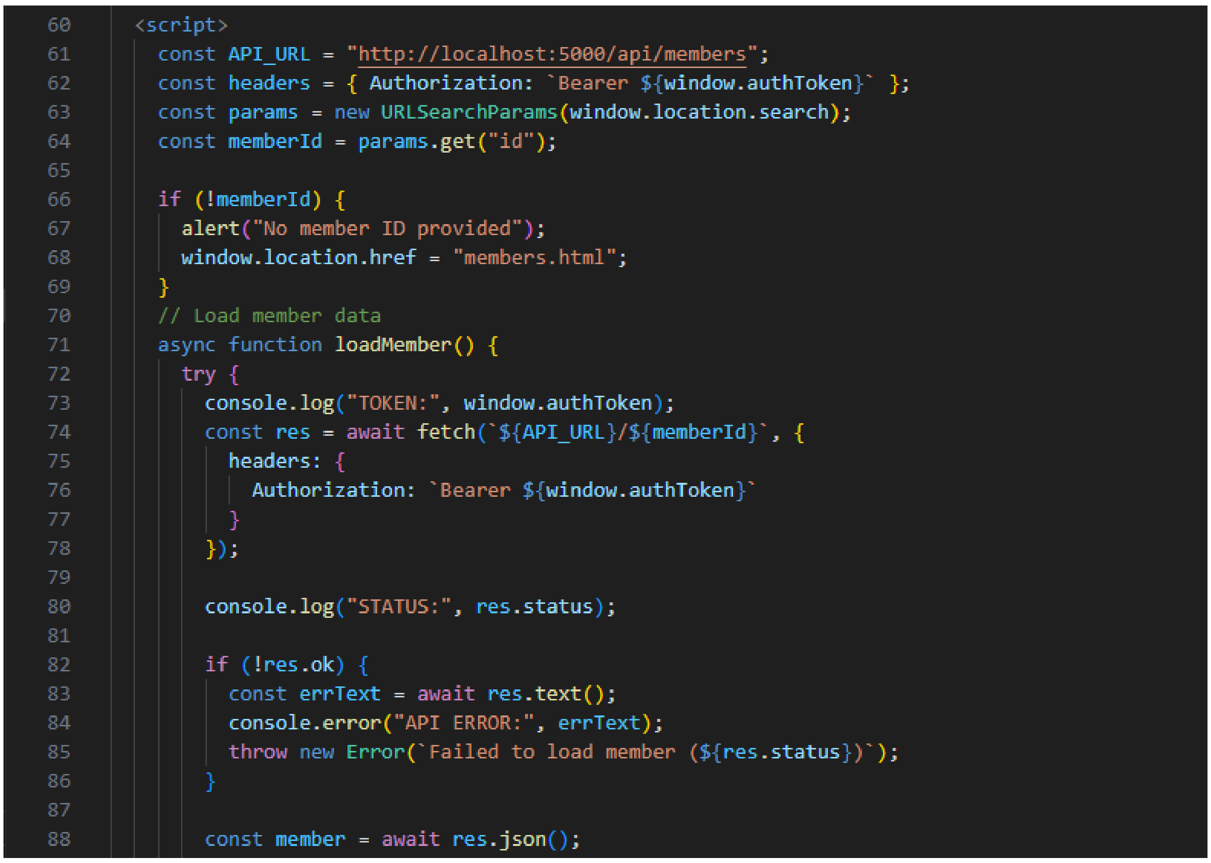
Below is the implemented edit-member.html code to display the above screen, and update member details.



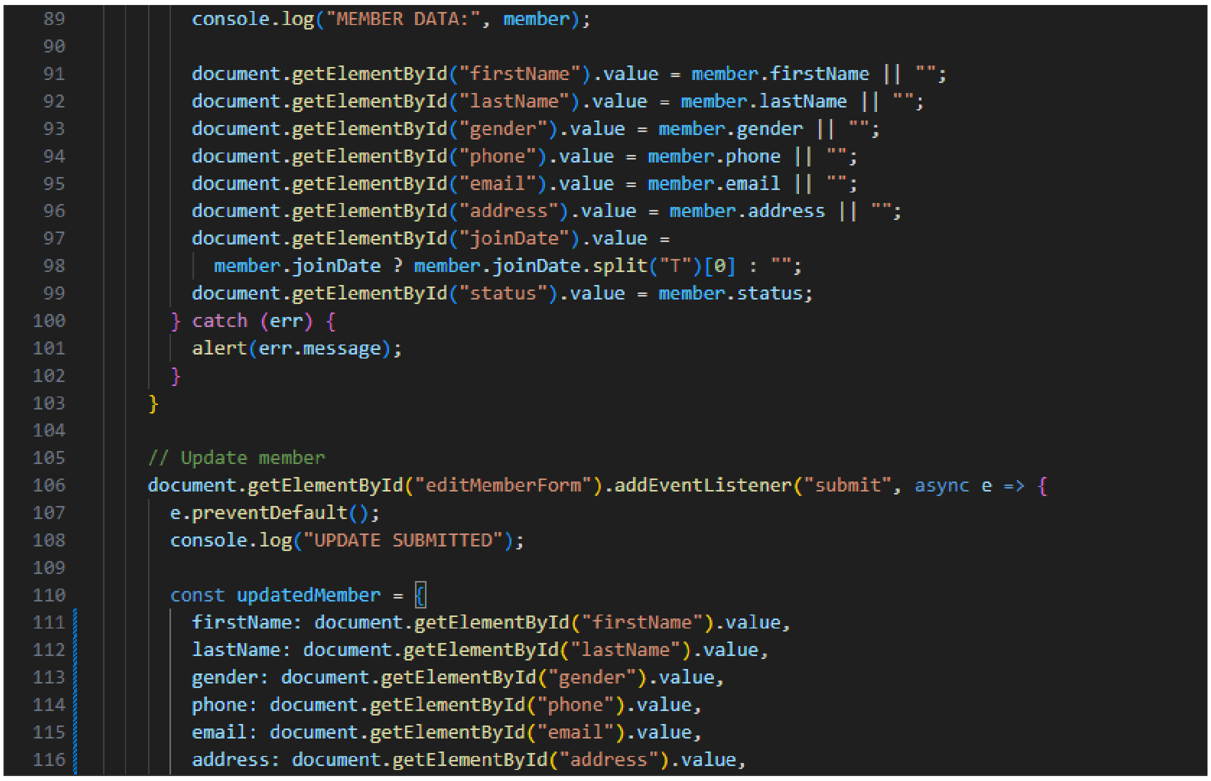
```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>Edit Member</title>
    <link rel="stylesheet" href="css/styles.css">
    <script src="authGuard.js"></script>
</head>

<body>

    <nav class="navbar">
        <h2>Church Admin</h2>
        <ul>
            <li><a href="dashboard.html">Dashboard</a></li>
            <li><a href="members.html">Members-Report</a></li>
        </ul>
    </nav>

    <div class="form-container">
        <h2>Edit Member</h2>

        <form id="editMemberForm">

            <input id="firstName" placeholder="First Name" required>
            <input id="lastName" placeholder="Last Name" required>

            <!-- Gender Dropdown -->
            <select id="gender">
                <option value="">Select Gender</option>
                <option value="Male">Male</option>
```



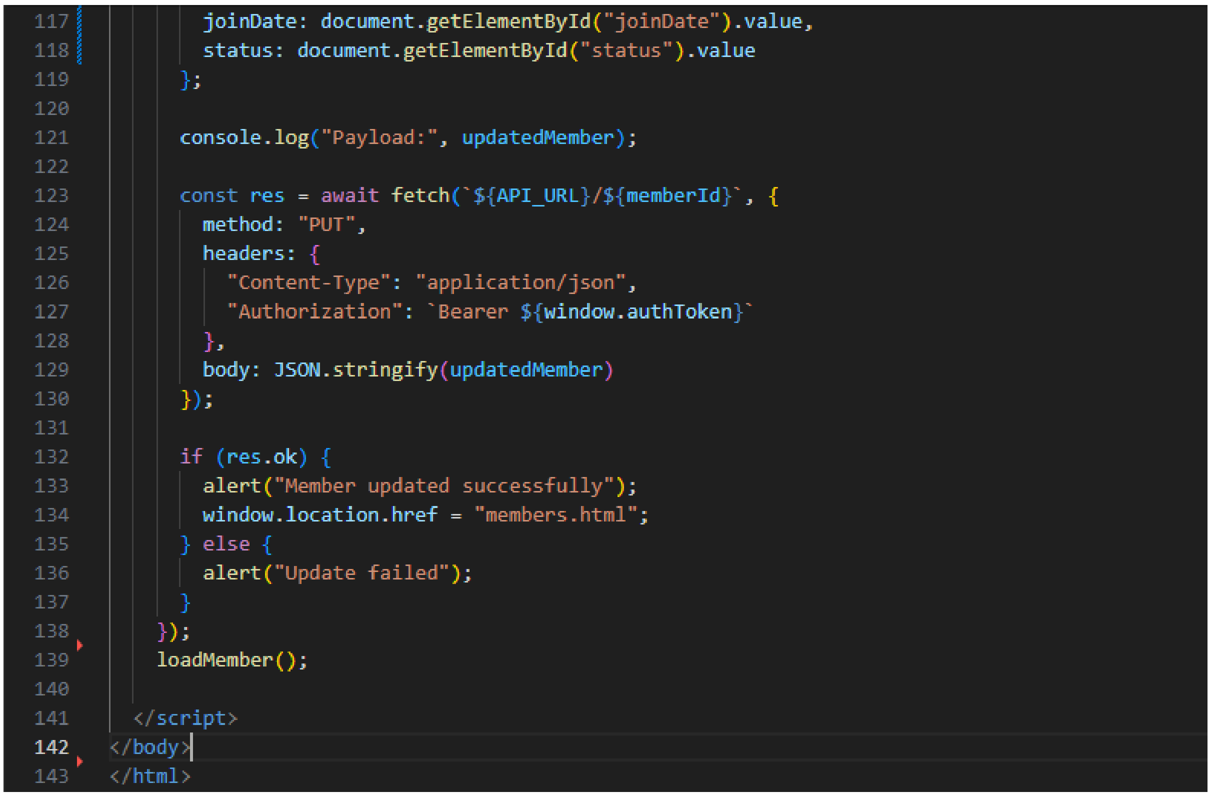
```html
        <option value="Female">Female</option>
        <option value="Other">Other</option>
      </select>

      <!-- <input id="phone" placeholder="Phone Number"> -->
      <input id="phone" placeholder="Phone Number" inputmode="numeric" pattern="[0-9]*"
        oninput="this.value = this.value.replace(/[^0-9]/g, '')" />
      <input id="email" placeholder="Email Address">
      <input id="address" placeholder="Address">

      <!-- Join Date Calendar -->
      <input id="joinDate" type="date">

      <!-- Status Dropdown -->
      <select id="status">
        <option value="Active" selected>Active</option>
        <option value="Inactive">Inactive</option>
        <option value="Transferred">Transferred</option>
      </select>

      <button type="submit">Update Member</button>

    </form>

    <p id="message"></p>
  </div>

```



```javascript
  <script>
    const API_URL = "http://localhost:5000/api/members";
    const headers = { Authorization: `Bearer ${window.authToken}` };
    const params = new URLSearchParams(window.location.search);
    const memberId = params.get("id");

    if (!memberId) {
      alert("No member ID provided");
      window.location.href = "members.html";
    }
    // Load member data
    async function loadMember() {
      try {
        console.log("TOKEN:", window.authToken);
        const res = await fetch(`${API_URL}/${memberId}`, {
          headers: {
            Authorization: `Bearer ${window.authToken}`
          }
        });

        console.log("STATUS:", res.status);

        if (!res.ok) {
          const errText = await res.text();
          console.error("API ERROR:", errText);
          throw new Error(`Failed to load member (${res.status})`);
        }

        const member = await res.json();
```



```
        console.log("MEMBER DATA:", member);

        document.getElementById("firstName").value = member.firstName || "";
        document.getElementById("lastName").value = member.lastName || "";
        document.getElementById("gender").value = member.gender || "";
        document.getElementById("phone").value = member.phone || "";
        document.getElementById("email").value = member.email || "";
        document.getElementById("address").value = member.address || "";
        document.getElementById("joinDate").value =
          member.joinDate ? member.joinDate.split("T")[0] : "";
        document.getElementById("status").value = member.status;
      } catch (err) {
        alert(err.message);
      }
    }

    // Update member
    document.getElementById("editMemberForm").addEventListener("submit", async e => {
      e.preventDefault();
      console.log("UPDATE SUBMITTED");

      const updatedMember = {
        firstName: document.getElementById("firstName").value,
        lastName: document.getElementById("lastName").value,
        gender: document.getElementById("gender").value,
        phone: document.getElementById("phone").value,
        email: document.getElementById("email").value,
        address: document.getElementById("address").value,
```

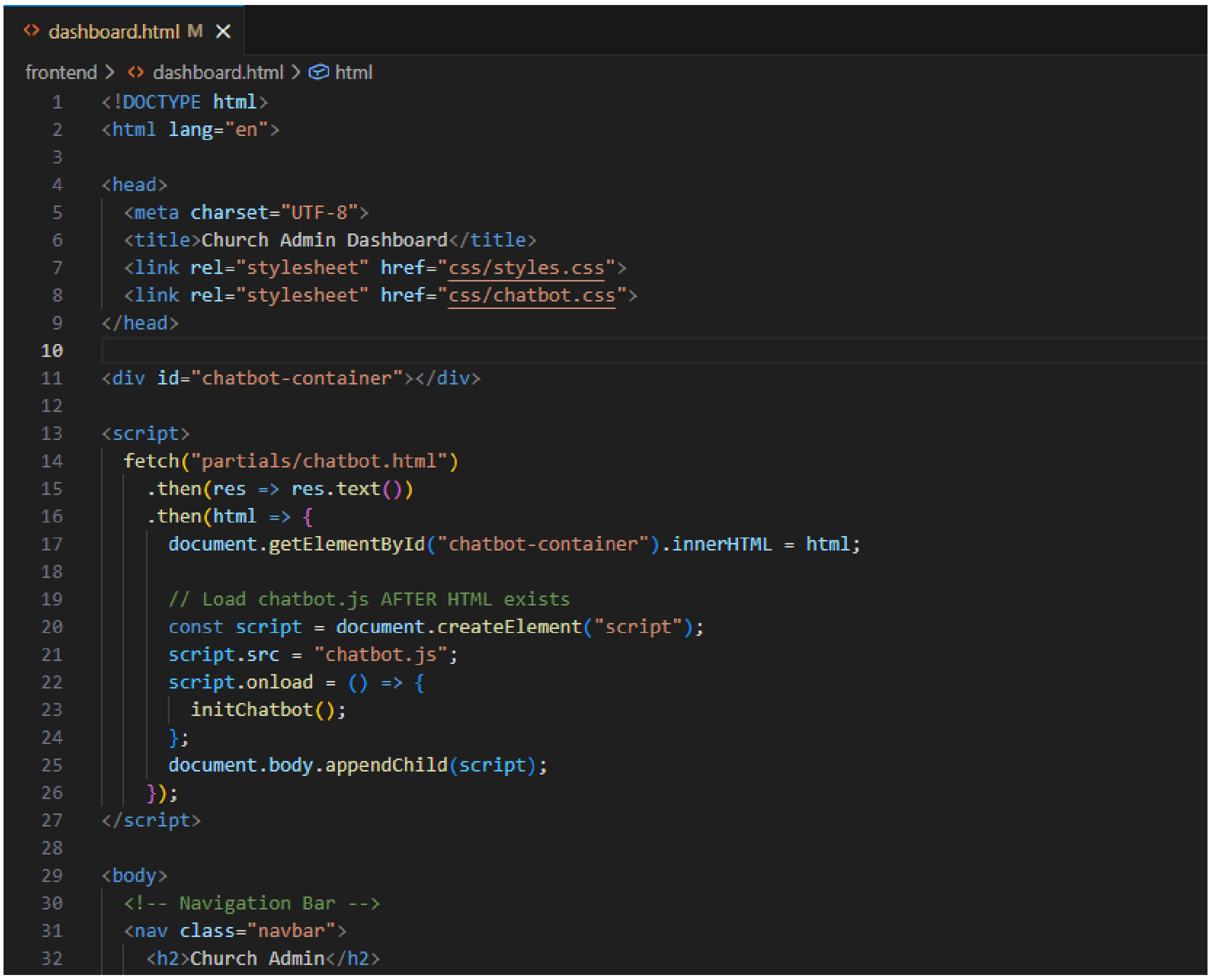


```
        joinDate: document.getElementById("joinDate").value,
        status: document.getElementById("status").value
      };

      console.log("Payload:", updatedMember);

      const res = await fetch(`${API_URL}/${memberId}`, {
        method: "PUT",
        headers: {
          "Content-Type": "application/json",
          "Authorization": `Bearer ${window.authToken}`
        },
        body: JSON.stringify(updatedMember)
      });

      if (res.ok) {
        alert("Member updated successfully");
        window.location.href = "members.html";
      } else {
        alert("Update failed");
      }
    });
    loadMember();

  </script>
</body>
</html>
```

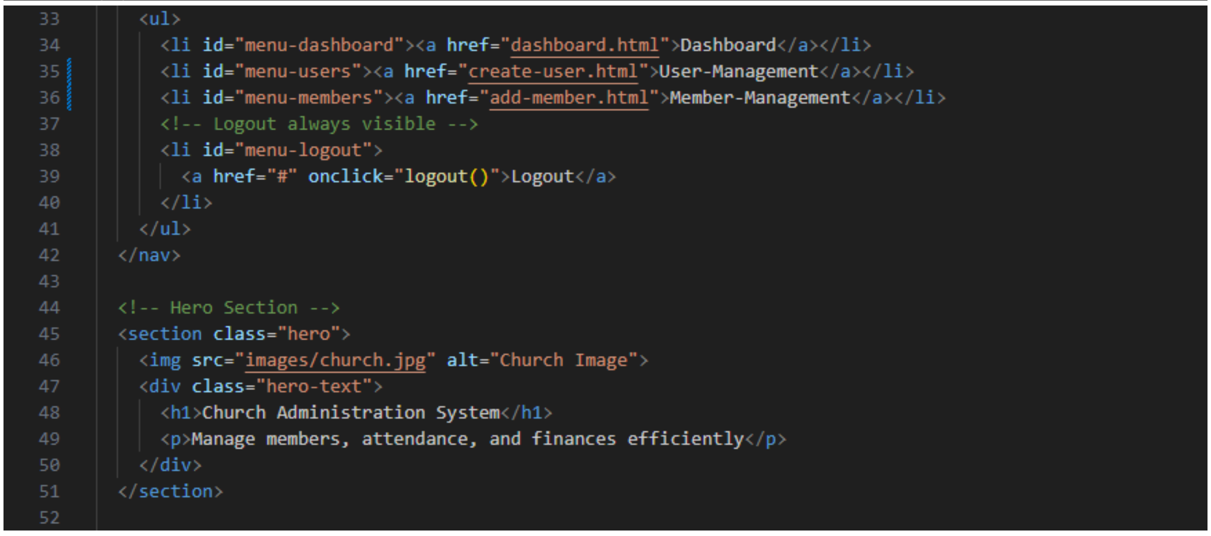
5.11 Dashboard (Frontend)

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

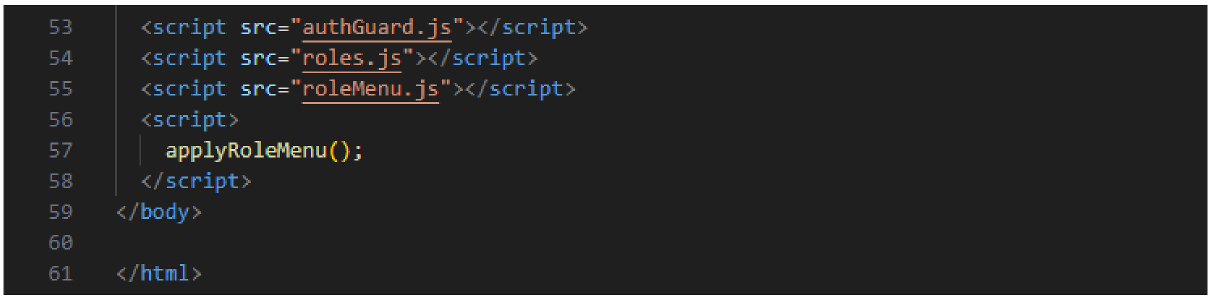
Below is the dashboard.html code that displays the dashboard of the system.



```html
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <title>Church Admin Dashboard</title>
  <link rel="stylesheet" href="css/styles.css">
  <link rel="stylesheet" href="css/chatbot.css">
</head>

<div id="chatbot-container"></div>

<script>
  fetch("partials/chatbot.html")
    .then(res => res.text())
    .then(html => {
      document.getElementById("chatbot-container").innerHTML = html;

      // Load chatbot.js AFTER HTML exists
      const script = document.createElement("script");
      script.src = "chatbot.js";
      script.onload = () => {
        initChatbot();
      };
      document.body.appendChild(script);
    });
</script>

<body>
  <!-- Navigation Bar -->
  <nav class="navbar">
    <h2>Church Admin</h2>
```



```html
    <ul>
      <li id="menu-dashboard"><a href="dashboard.html">Dashboard</a></li>
      <li id="menu-users"><a href="create-user.html">User-Management</a></li>
      <li id="menu-members"><a href="add-member.html">Member-Management</a></li>
      <!-- Logout always visible -->
      <li id="menu-logout">
        <a href="#" onclick="logout()">Logout</a>
      </li>
    </ul>
  </nav>

  <!-- Hero Section -->
  <section class="hero">
    <img src="images/church.jpg" alt="Church Image">
    <div class="hero-text">
      <h1>Church Administration System</h1>
      <p>Manage members, attendance, and finances efficiently</p>
    </div>
  </section>

```

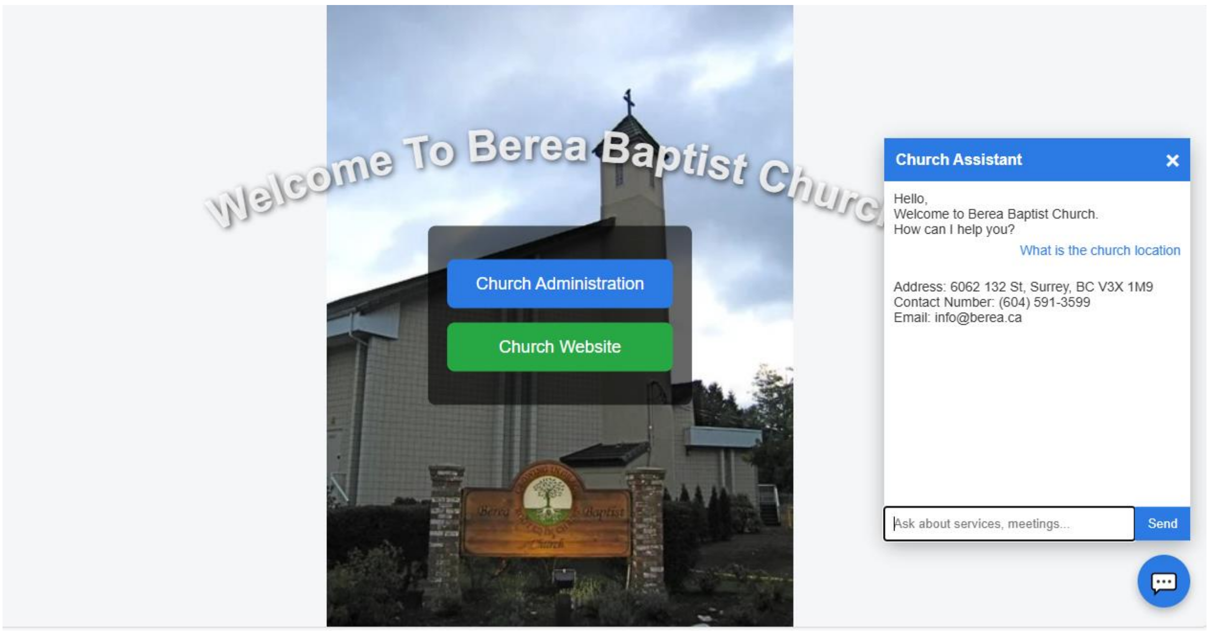


```
    <script src="authGuard.js"></script>
    <script src="roles.js"></script>
    <script src="roleMenu.js"></script>
    <script>
      applyRoleMenu();
    </script>
  </body>

  </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.



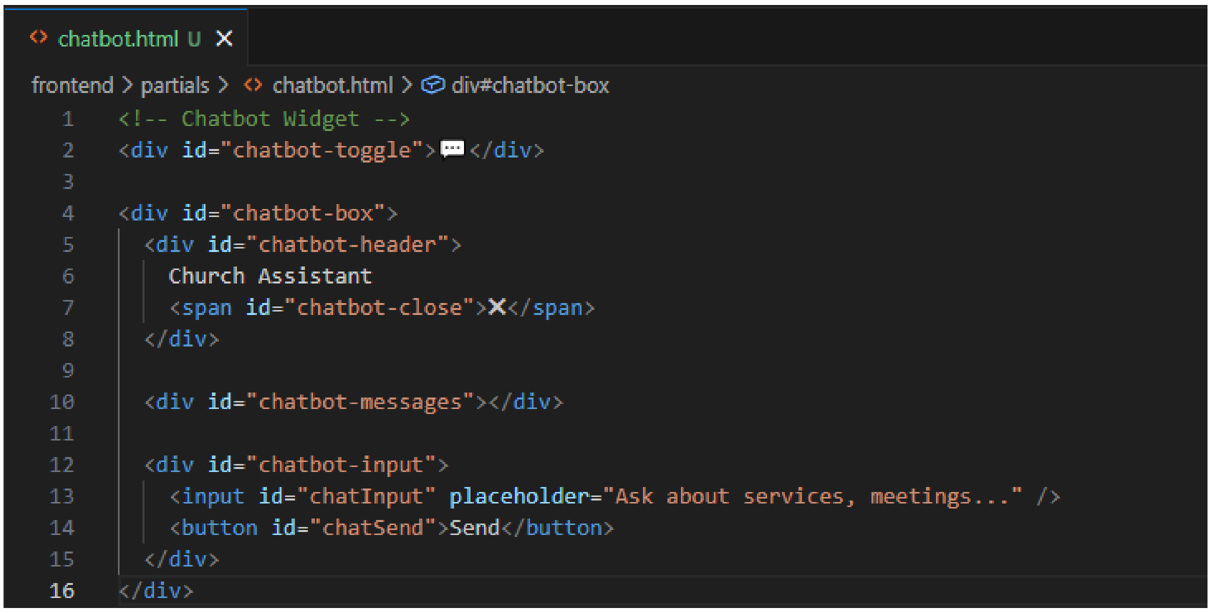
Welcome To Berea Baptist Churc
Church Administration
Church Website
Church Assistant
Ask about services, meetings...
Send

Hello,
Welcome to Berea Baptist Church.
How can I help you?

What is the church location

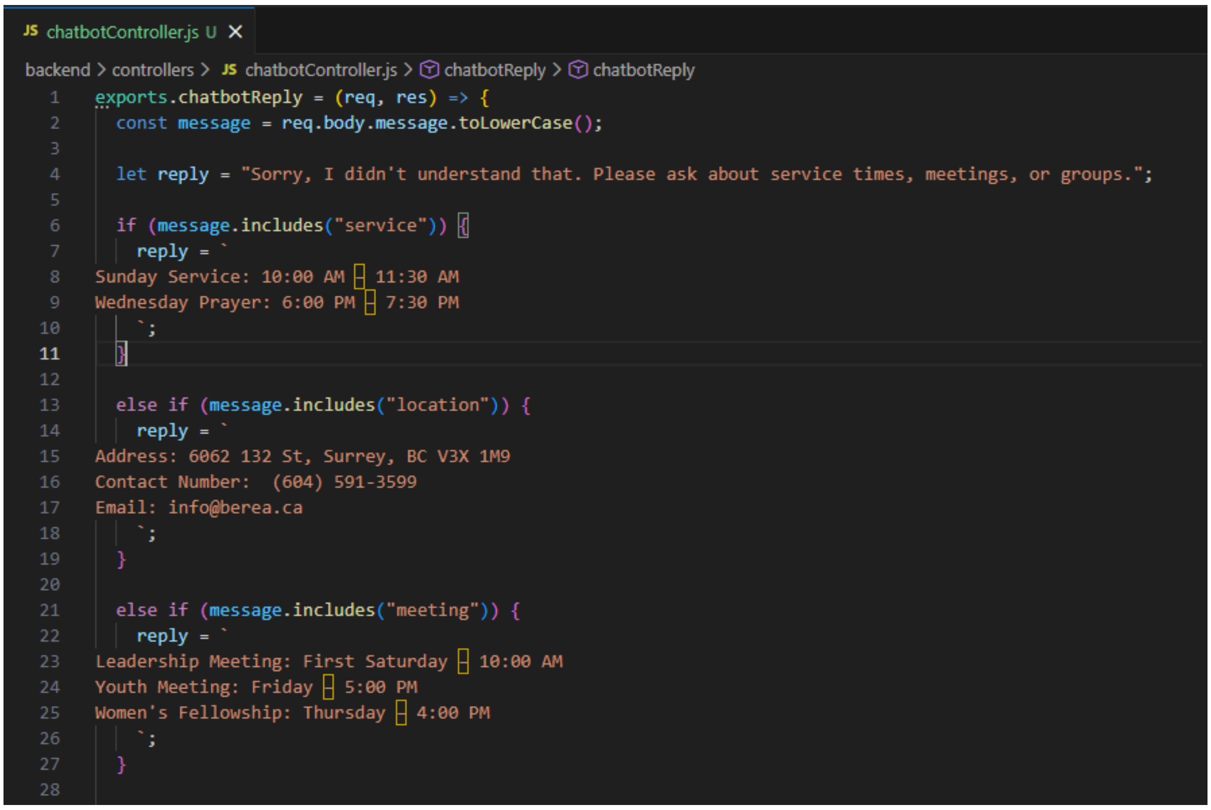
Address: 6062 132 St, Surrey, BC V3X 1M9
Contact Number: (604) 591-3599
Email: info@berea.ca

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

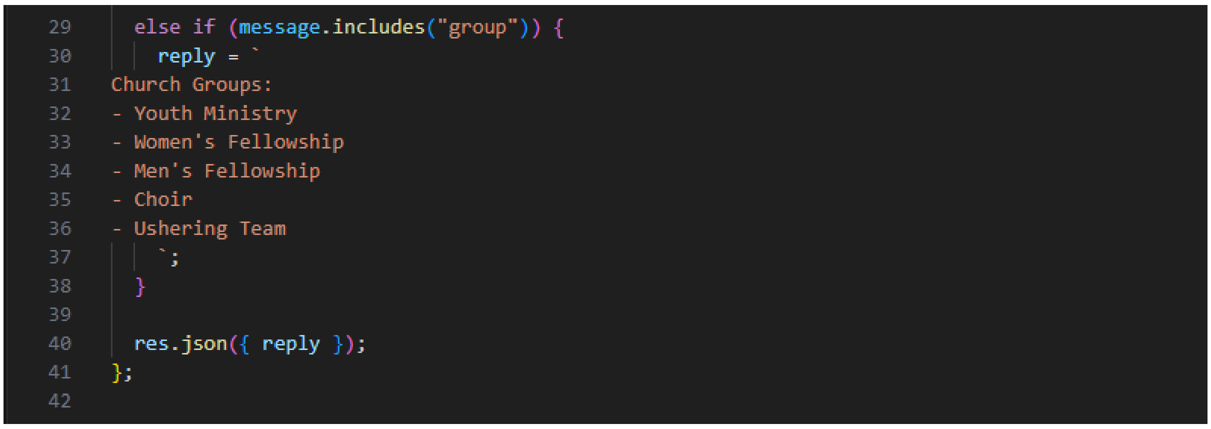


```html
<!-- Chatbot Widget -->
<div id="chatbot-toggle">💬</div>

<div id="chatbot-box">
  <div id="chatbot-header">
    Church Assistant
    <span id="chatbot-close">✖</span>
  </div>

  <div id="chatbot-messages"></div>

  <div id="chatbot-input">
    <input id="chatInput" placeholder="Ask about services, meetings..." />
    <button id="chatSend">Send</button>
  </div>
</div>
```

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


chatbotController.js U
backend > controllers > chatbotController.js > chatbotReply > chatbotReply

```javascript
exports.chatbotReply = (req, res) => {
  const message = req.body.message.toLowerCase();

  let reply = "Sorry, I didn't understand that. Please ask about service times, meetings, or groups.";

  if (message.includes("service")) {
    reply = `
Sunday Service: 10:00 AM □ 11:30 AM
Wednesday Prayer: 6:00 PM □ 7:30 PM
    `;
  }

  else if (message.includes("location")) {
    reply = `
Address: 6062 132 St, Surrey, BC V3X 1M9
Contact Number:  (604) 591-3599
Email: info@berea.ca
    `;
  }

  else if (message.includes("meeting")) {
    reply = `
Leadership Meeting: First Saturday □ 10:00 AM
Youth Meeting: Friday □ 5:00 PM
Women's Fellowship: Thursday □ 4:00 PM
    `;
  }

```



```
  else if (message.includes("group")) {
    reply = `
Church Groups:
- Youth Ministry
- Women's Fellowship
- Men's Fellowship
- Choir
- Ushering Team
    `;
  }

  res.json({ reply });
};
```

5.13 Application Core / Server Configuration Module

server.js

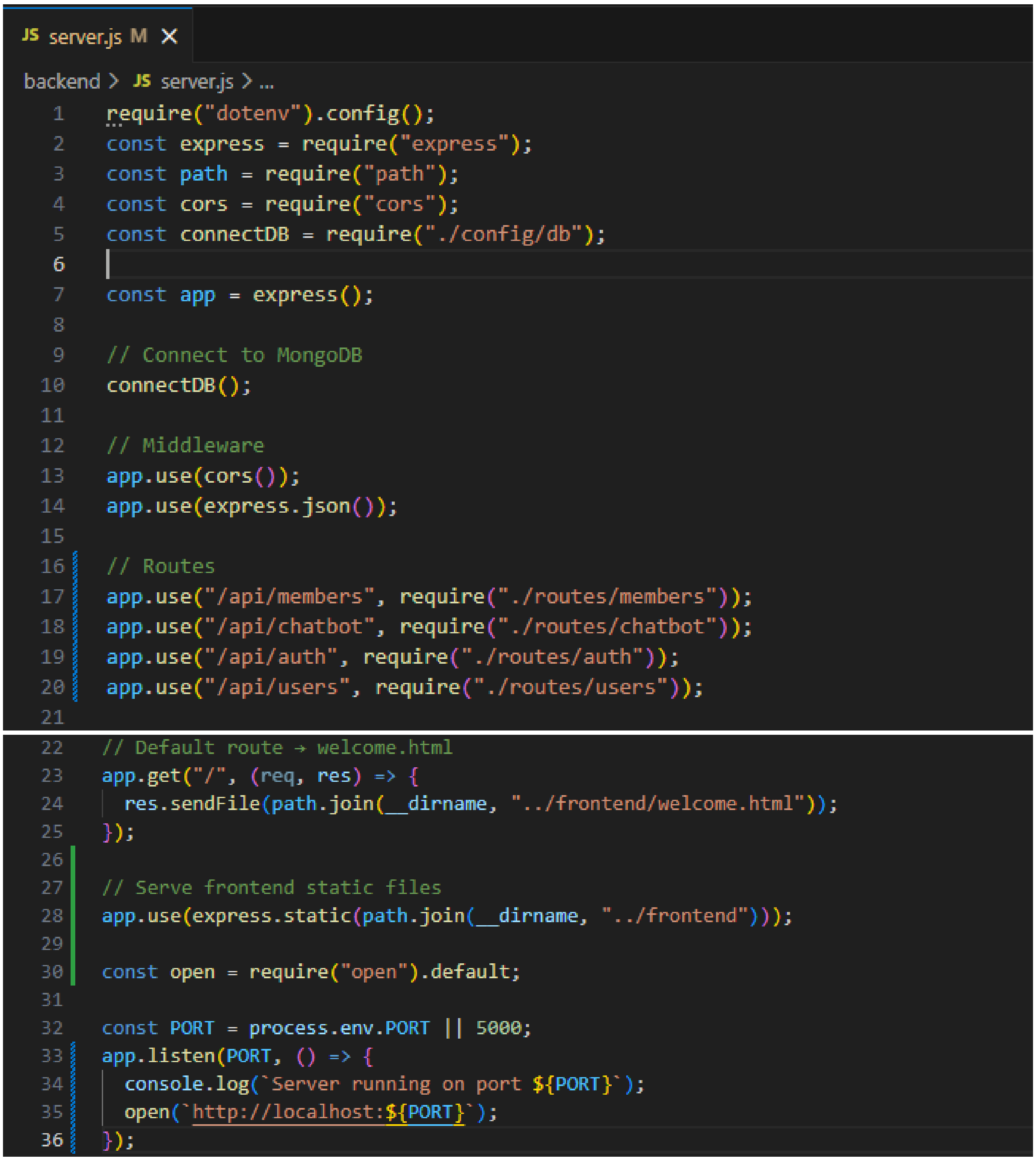
The server.js file serves as the main entry point of the backend application. It is responsible for initializing the Express server, configuring middleware, establishing the database connection, and registering all application routes. This file coordinates the interaction between different system modules 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.



```javascript
require("dotenv").config();
const express = require("express");
const path = require("path");
const cors = require("cors");
const connectDB = require("./config/db");

const app = express();

// Connect to MongoDB
connectDB();

// Middleware
app.use(cors());
app.use(express.json());

// Routes
app.use("/api/members", require("./routes/members"));
app.use("/api/chatbot", require("./routes/chatbot"));
app.use("/api/auth", require("./routes/auth"));
app.use("/api/users", require("./routes/users"));

// Default route → welcome.html
app.get("/", (req, res) => {
  res.sendFile(path.join(__dirname, "../frontend/welcome.html"));
});

// Serve frontend static files
app.use(express.static(path.join(__dirname, "../frontend")));

const open = require("open").default;

const PORT = process.env.PORT || 5000;
app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
  open(`http://localhost:${PORT}`);
});
```

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.