**EMERGENCY RESPONSE NETWORK**

The ERN backend is built using Node.js, Express.js, and MongoDB (Mongoose) to handle emergency requests, assign responders and volunteers, and manage real-time data flow. APIs are developed to allow seamless integration with mobile applications, web dashboards, and IoT devices.

**Key functionalities include:**

* Emergency request registration
* Real-time alerts to response teams
* Automated assignment of responders based on proximity and availability
* Incident status updates

**Scenario Based Case Study**

Scenario: Fire Emergency in a Residential Area

* A fire breaks out in a residential area. A resident reports the emergency via a mobile app.
* The request is received by the backend and logged into the database.
* The system identifies the nearest fire stations and available fire trucks.
* Responders are notified with location tracking.
* Authorities receive live updates on the incident status.

This case study demonstrates how ERN streamlines emergency response through efficient data handling.

**Technical Architecture**

The system follows a Microservices-based REST API architecture, ensuring modularity and scalability.

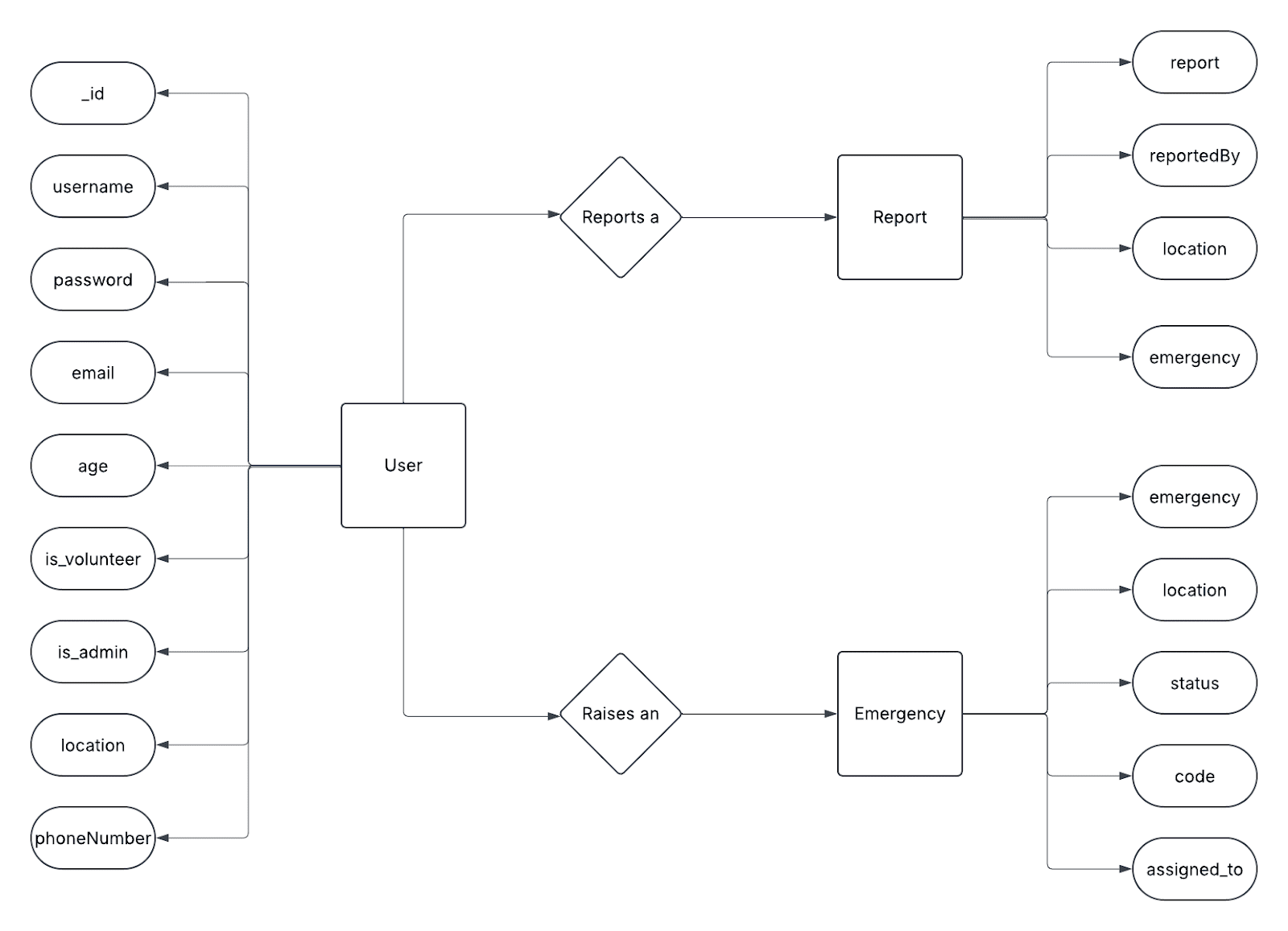
**Technology Stack:**

* Backend: Node.js, Express.js
* Database: MongoDB (Mongoose ODM)
* Authentication: JWT-based authentication
* API Testing: Postman / Thunderclient

**ER-Diagram**

The **Entity-Relationship Diagram (ERD)** represents database structure, including entities like:

* **Users:** Stores users, victims, volunteers and admin details
* **Emergencies:** Stores reported emergencies
* **Reports:** Stores reported incidents



**Roles and Responsibility**

The Emergency Response Network (ERN) involves three primary roles: Users, Volunteers, and Admins. Each role has distinct responsibilities to ensure efficient emergency handling and response coordination.

1.      Users (Emergency Reporters)

These are individuals who report emergencies, such as victims, witnesses, or concerned citizens.

Responsibilities:

·         Register and authenticate via the system.

·         Report emergencies by providing details such as location, type of incident, and severity.

2.      Volunteers (Responders & Emergency Personnel)

Volunteers are first responders such as medical professionals, firefighters, police officers, or trained community helpers.

Responsibilities:

·         Register and authenticate in the system as verified responders.

·         Receive emergency alerts based on location and availability.

3.      Admins (System Managers & Authorities)

Admins oversee the system's operation, ensure data integrity, and manage users and volunteers.

            Responsibilities:

* Manage system users (approve, block, or verify responders).
* Oversee emergency reports and ensure appropriate response.

**Project Setup and Configuration**

* Initialize the project with Node.js & Express.js.
* Set up package.json and install necessary dependencies (express, mongoose, dotenv, jsonwebtoken, mongoose, cookie-parser, bcrypt).

A screen shot of a computer program

AI-generated content may be incorrect.

* Configure environment variables (JWT\_SECRET, MONGODB\_URI) and basic routing structure.

**Backend Development**

* Create RESTful APIs for handling emergency requests, user authentication, and incident updates.

A screen shot of a computer code

AI-generated content may be incorrect.

* Implement JWT-based authentication for security.
* Define controllers and middleware for data validation and request handling.

**Database**

* Design MongoDB schema for users, reports and emergencies.

A screen shot of a computer program

AI-generated content may be incorrect.

* Users Collection in MongoDB Compass

A screenshot of a computer

AI-generated content may be incorrect.

* Emergencies Collection in MongoDB Compass

A screenshot of a computer

AI-generated content may be incorrect.

* Reports Collection in MongoDB Compass

A screenshot of a computer

AI-generated content may be incorrect.

* Implement Mongoose models and database operations (CRUD).
* Set up MongoDB Atlas (if using cloud storage).

**1) alertController.js :-**

const { getIo } = require('../socket');

const User = require('../models/userModel');

const Emergency = require('../models/emergencyModel');

// Controller to get alerts based on the user's location

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

try {

// Fetch the user's details using the user ID from the JWT token

const user = await User.findById(req.user.id);

if (!user) {

return res.status(404).json({ error: "User not found" });

}

const userLocation = user.location; // Assuming user has a location field in the User schema

// Fetch all emergencies with the same location as the user

const emergencies = await Emergency.find({ location: userLocation, status: 'pending' });

// Return the fetched emergencies as alerts

res.status(200).json({ alerts: emergencies });

} catch (error) {

console.error("Error fetching alerts:", error);

res.status(500).json({ message: "Server error while fetching alerts" });

}

};

**2) emergencyController.js :-**

const Emergency = require('../models/emergencyModel');

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

    try {

        const emergency = await Emergency.create(req.body);

        res.status(201).json(emergency);

    } catch (error) {

        res.status(500).json({ error: error.message });

    }

};

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

    const emergencies = await Emergency.find().populate('assignedTo');

    res.json(emergencies);

};

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

    const { emergencyId } = req.params; // Get the emergencyId from the URL

    const { description, location, emergency } = req.body; // Get the fields to update

    try {

        // Find the emergency by its ID and update it

        const updatedEmergency = await Emergency.findByIdAndUpdate(

            emergencyId,

            {

                description,

                location,

                emergency // Emergency type (e.g., RED, YELLOW, GREEN)

            },

            { new: true } // Return the updated document

        );

        if (!updatedEmergency) {

            return res.status(404).json({ message: 'Emergency not found' });

        }

        // Respond with the updated emergency

        res.status(200).json(updatedEmergency);

    } catch (error) {

        console.error('Error updating emergency:', error);

        res.status(500).json({ message: 'Server error while updating emergency' });

    }

};

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

    const { location } = req.params;  // Get the location from the route parameter

    try {

        // Query the database to find emergencies near the provided location

        const emergencies = await Emergency.find({ location: location });

        // If no emergencies are found

        if (emergencies.length === 0) {

            return res.status(404).json({ message: 'No emergencies found near this location' });

        }

        // Return the emergencies found

        res.status(200).json(emergencies);

    } catch (error) {

        console.error('Error fetching emergencies:', error);

        res.status(500).json({ message: 'Server error while fetching emergencies' });

    }

};

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

    try {

        // Extract query parameters from the request

        const query = req.query;

        // Build the filter object dynamically

        let filterConditions = {};

        // Check for specific filters and add them to the filter conditions

        if (query.severity) {

            filterConditions.severity = query.severity;

        }

        if (query.status) {

            filterConditions.status = query.status;

        }

        if (query.location) {

            filterConditions.location = { $regex: query.location, $options: 'i' };  // Case-insensitive search

        }

        // Fetch filtered emergencies from the database

        const emergencies = await Emergency.find(filterConditions);

        // If no emergencies are found

        if (emergencies.length === 0) {

            return res.status(404).json({ message: 'No emergencies found matching the filter' });

        }

        // Return the filtered emergencies

        res.status(200).json(emergencies);

    } catch (error) {

        console.error('Error fetching filtered emergencies:', error);

        res.status(500).json({ message: 'Server error while filtering emergencies' });

    }

};

**3) reportController.js :-**

const Report = require('../models/reportModel');

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

    try {

        // Destructure the required fields from the request body

        const { report, location, emergency } = req.body;

        // Validate if the required fields are provided

        if (!report || !location || !emergency) {

            return res.status(400).json({ error: "Report, location, and emergency level are required." });

        }

        // Create a new report instance

        const newReport = new Report({

            report,

            reportedBy: req.user.id,  // The user ID from the JWT token

            location,

            emergency

        });

        // Save the report to the database

        await newReport.save();

        // Return a success response with the created report

        res.status(201).json({ message: 'Report created successfully', report: newReport });

    } catch (error) {

        // Catch any errors and respond with a 500 error

        console.error('Error creating report:', error);

        res.status(500).json({ error: 'Error creating report', message: error.message });

    }

};

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

    try {

        const reports = await Report.find().populate('reportedBy');

        res.json(reports);

    } catch (error) {

        res.status(500).json({ error: error.message });

    }

};

// Controller Method to Get Nearby Reports

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

    const { location } = req.params; // Capture the location from the URL parameter

    try {

        // Search reports with a case-insensitive match to the location parameter

        const reports = await Report.find({

            location: { $regex: location, $options: 'i' } // 'i' for case-insensitive search

        });

        if (reports.length === 0) {

            return res.status(404).json({ message: 'No reports found for this location.' });

        }

        // Return the filtered reports

        res.status(200).json(reports);

    } catch (error) {

        console.error("Error fetching nearby reports:", error);

        res.status(500).json({ error: "Server error while fetching reports" });

    }

};

// Controller Method to Filter Reports

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

    const { emergency, location, reportedBy, startDate, endDate } = req.query;

    // Build the query object dynamically based on the presence of query parameters

    let filter = {};

    if (emergency) {

        filter.emergency = emergency;

    }

    if (location) {

        filter.location = { $regex: location, $options: 'i' }; // Case-insensitive search

    }

    if (reportedBy) {

        filter.reportedBy = reportedBy;

    }

    if (startDate || endDate) {

        filter.createdAt = {};

        if (startDate) {

            filter.createdAt.$gte = new Date(startDate);

        }

        if (endDate) {

            filter.createdAt.$lte = new Date(endDate);

        }

    }

    try {

        const reports = await Report.find(filter);

        if (reports.length === 0) {

            return res.status(404).json({ message: 'No reports found for the given filters.' });

        }

        res.status(200).json(reports);

    } catch (error) {

        console.error('Error filtering reports:', error);

        res.status(500).json({ error: 'Server error while filtering reports' });

    }

};

**4) userController.js :-**

const User = require('../models/userModel');

const bcrypt = require('bcryptjs');

const jwt = require('jsonwebtoken');

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

    try {

        const user = await User.create(req.body);

        res.status(201).json({ token: jwt.sign({ id: user.\_id, role: user.role }, process.env.JWT\_SECRET, { expiresIn: '7d' }) });

    } catch (error) {

        res.status(400).json({ error: error.message });

    }

};

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

    const { email, username, password } = req.body;

    try {

        // Find user by either email or username

        const user = await User.findOne({

            $or: [{ email }, { username }]

        });

        if (!user) {

            return res.status(401).json({ message: "Invalid credentials" });

        }

        // Compare hashed password

        const isMatch = await bcrypt.compare(password, user.password);

        if (!isMatch) {

            return res.status(401).json({ message: "Invalid credentials" });

        }

        // Generate JWT Token

        const token = jwt.sign(

            { id: user.\_id, role: user.is\_admin ? 'admin' : 'user' },

            process.env.JWT\_SECRET,

            { expiresIn: "1h" }

        );

        res.json({ token, userId: user.\_id, role: user.is\_admin ? 'admin' : 'user' });

    } catch (error) {

        console.error("Login error:", error);

        res.status(500).json({ message: "Server error" });

    }

};

// Update a user's profile data

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

    try {

        // Ensure that the user is authenticated and retrieve the user's ID from the token

        const userId = req.user.id; // The user ID is populated from the JWT token

        // Find the user in the database by their ID

        const user = await User.findById(userId);

        if (!user) {

            return res.status(404).json({ error: "User not found" });

        }

        // Destructure the data we want to update from the request body

        const { username, email, location, password } = req.body;

        // Validate if the fields are provided

        if (!username && !email && !location && !password) {

            return res.status(400).json({ error: "No data provided to update" });

        }

        // Prepare the update data

        const updatedData = {};

        if (username) {

            updatedData.username = username;

        }

        if (email) {

            updatedData.email = email;

        }

        if (location) {

            updatedData.location = location;

        }

        if (password) {

            // If password is provided, hash it before saving

            const salt = await bcrypt.genSalt(10);

            updatedData.password = await bcrypt.hash(password, salt);

        }

        // Update the user in the database with the new data

        const updatedUser = await User.findByIdAndUpdate(userId, updatedData, { new: true });

        // Return the updated user data

        res.status(200).json({

            message: "User profile updated successfully",

            user: updatedUser

        });

    } catch (error) {

        console.error("Error updating profile:", error);

        res.status(500).json({ error: "Server error" });

    }

};

// userController.js

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

    try {

        // The token is client-side, and we can't invalidate it server-side.

        // But, we can send a response instructing the client to delete the token.

        res.status(200).json({

            message: "User logged out successfully. Please delete the token from your client."

        });

    } catch (error) {

        res.status(500).json({ error: 'Server error while logging out' });

    }

};

**5) authMiddleware.js :-**

const jwt = require('jsonwebtoken');

const verifyToken = (req, res, next) => {

    const token = req.header('Authorization')?.replace('Bearer ', ''); // Extract the token from Authorization header

    if (!token) {

        //no token provided, user is not logged in.

        return res.status(401).json({ message: 'Unauthorized. Please log in.' });

    }

    try {

        const decoded = jwt.verify(token, process.env.JWT\_SECRET); // Replace 'SecretKey' with your actual secret

        req.user = decoded;  // Attach the decoded user info to the request object

        next();  // Pass to the next middleware or route handler

    } catch (error) {

        //If token verification fails, user is not logged in

        return res.status(401).json({ message: 'Unauthorized. Invalid token.' });

    }

};

module.exports = { verifyToken };

**6) is\_admin\_middleware.js :-**

exports.isAdmin = (req, res, next) => {

    if (req.user.role !== 'admin') {

        return res.status(403).json({ error: "Access Denied: Admins only" });

    }

    next();

};

**7) emergencyModel.js :-**

const mongoose = require('mongoose');

const emergencySchema = new mongoose.Schema({

    emergency: { type: String, required: true },

    status: { type: String, enum: ['pending', 'resolved', 'in-progress'], default: 'pending' },

    location: { type: String, required: true },

    code: { type: String, enum: ['RED', 'YELLOW', 'GREEN'], required: true }, // RED = High Priority

    assignedTo: { type: String, default: null }, // Username of responder

    created\_by: { type: mongoose.Schema.Types.ObjectId, ref: 'User'}

}, { timestamps: true });

module.exports = mongoose.model('Emergency', emergencySchema);

**8) reportModel.js :-**

const mongoose = require('mongoose');

const reportSchema = new mongoose.Schema({

    report: { type: String, required: true },

    reportedBy: { type: mongoose.Schema.Types.ObjectId, ref: 'User', required: true }, // User ID

    location: { type: String, required: true },

    emergency: { type: String, enum: ['RED', 'YELLOW', 'GREEN'], required: true }

}, { timestamps: true });

module.exports = mongoose.model('Report', reportSchema);

**9) userModel.js :-**

const mongoose = require('mongoose');

const bcrypt = require('bcryptjs');

const userSchema = new mongoose.Schema({

    username: { type: String, required: true },

    password: { type: String, required: true },

    email: { type: String, required: true, unique: true },

    age: { type: Number, required: true },

    volunteer: { type: Boolean, default: false },

    location: { type: String, required: true },

    phoneNumber: { type: String, required: true },

    is\_admin: { type: Boolean, default: false }

}, { timestamps: true });

// Hash password before saving

userSchema.pre('save', async function (next) {

    if (!this.isModified('password')) return next();

    const salt = await bcrypt.genSalt(10);

    this.password = await bcrypt.hash(this.password, salt);

    next();

});

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

**10) alertRoute.js :-**

const express = require('express');

const { getAlerts } = require('../controllers/alertController');

const { verifyToken } = require('../middleware/authMiddleware');

const router = express.Router();

router.get('/any', verifyToken, getAlerts);

module.exports = router;

**11) emergencyRoute.js :-**

const express = require('express');

const { createEmergency, getEmergencies, updateEmergency,getNearbyEmergencies, filterEmergencies } = require('../controllers/emergencyController');

const { verifyToken } = require('../middleware/authMiddleware');

const router = express.Router();

router.post('/create', verifyToken, createEmergency);

router.get('/all', verifyToken, getEmergencies);

router.put('/update/:emergencyId', verifyToken, updateEmergency);

router.get('/near/:location', verifyToken, getNearbyEmergencies);

router.get('/filter', filterEmergencies);

module.exports = router;

**12) reportRoute.js :-**

const express = require('express');

const { createReport, getReports, getNearbyReports, filterReports } = require('../controllers/reportController');

const { verifyToken } = require('../middleware/authMiddleware');

const router = express.Router();

router.post('/create', verifyToken, createReport);

router.get('/all', verifyToken, getReports);

router.get('/near/:location', verifyToken, getNearbyReports);

router.get('/filter', verifyToken, filterReports);

module.exports = router;

**13) userRoute.js :-**

const express = require('express');

const { registerUser, loginUser, updateUserProfile, logout } = require('../controllers/userController');

const { verifyToken } = require('../middleware/authMiddleware');

const router = express.Router();

router.post('/register', registerUser);

router.post('/login', loginUser);

router.post('/update', verifyToken, updateUserProfile);

router.post('/logout', logout);

module.exports = router;

**14) .env :-**

PORT=5000

MONGO\_URI=mongodb+srv://syedrajak:rajakR935c7@cluster0.w9akd.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0

JWT\_SECRET='SecretKey'

**15) jwt.js :-**

const jwt = require('jsonwebtoken');

exports.generateToken = (user) => {

    return jwt.sign({ id: user.\_id, role: user.role }, process.env.JWT\_SECRET, { expiresIn: '7d' });

};

exports.verifyToken = (req, res, next) => {

    const token = req.header('Authorization');

    if (!token) return res.status(401).json({ error: "Access Denied" });

    try {

        const decoded = jwt.verify(token.split(" ")[1], process.env.JWT\_SECRET);

        req.user = decoded;

        next();

    } catch (error) {

        res.status(400).json({ error: "Invalid Token" });

    }

};

**16) server.js :-**

require('dotenv').config();

const express = require('express');

const mongoose = require('mongoose');

const cors = require('cors');

const morgan = require('morgan');

const http = require('http');

const { initializeSocket } = require('./socket');

const app = express();

const server = http.createServer(app);

// Middleware

app.use(cors());

app.use(express.json());

app.use(morgan('dev'));

// Routes

app.use('/user', require('./routes/userRoute'));

app.use('/report', require('./routes/reportRoute'));

app.use('/emergency', require('./routes/emergencyRoute'));

app.use('/alert', require('./routes/alertRoute'));

// Database Connection

const connectDB = async () => {

    try {

        await mongoose.connect(process.env.MONGO\_URI );

        console.log('MongoDB Connected Successfully');

        // Start server only after DB is connected

        const PORT = process.env.PORT || 5000;

        server.listen(PORT, () => {

            console.log(`Server running on port ${PORT}`);

        });

        // Initialize Socket.io after DB connection

        initializeSocket(server);

    } catch (error) {

        console.error('MongoDB Connection Error:', error);

        process.exit(1);

    }

};

// Call the function to connect to MongoDB

connectDB();

**17) socket.js :-**

const socketIo = require('socket.io');

let io;

const initializeSocket = (server) => {

    io = socketIo(server);

    io.on('connection', (socket) => {

        console.log('A user connected');

        socket.on('disconnect', () => {

            console.log('User disconnected');

        });

    });

};

const getIo = () => {

    if (!io) {

        throw new Error('Socket.io not initialized!');

    }

    return io;

};

module.exports = { initializeSocket, getIo };

**API Testing**

* Use **Postman(Download Application) or Thunderclient(VS Code Extension)** to test API endpoints.
* Validate responses for:
  + Creating and updating reports, emergencies
  + Alerting volunteers and admins
  + Authentication and role-based access
* Debug issues and optimize performance.

**Outputs (API Testing)**

Add collection description…﻿

**POSThttp://localhost:5000/user/register**

[**Open request**](https://desktop.postman.com/?desktopVersion=11.35.0&userId=37401405&teamId=0&region=us)

http://localhost:5000/user/register

Method: POST URL: /user/register

Description: Register a new user

﻿

**Body**raw (json)

json

{

"username": "Syed",

"password": "123456789",

"email": "wxyy@gmail.com",

"age": 21,

"location": "kanigiri",

"phoneNumber": "9999999999"

}

**POSThttp://localhost:5000/user/login**

[**Open request**](https://desktop.postman.com/?desktopVersion=11.35.0&userId=37401405&teamId=0&region=us)

http://localhost:5000/user/login

Method: POST URL: /user/login

Description: Login a user

﻿

**Body**raw (json)

json

{

"username": "remo",

"password": "123456789"

}

**POSThttp://localhost:5000/report/create**

[**Open request**](https://desktop.postman.com/?desktopVersion=11.35.0&userId=37401405&teamId=0&region=us)

http://localhost:5000/report/create

Method: POST URL: /report/create

Description: Create a new report

﻿

**Request Headers**

**Authorization**

Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjY3YzNkMDJjNGFmN2ZkYWI3YzU2MTExZSIsInJvbGUiOiJ1c2VyIiwiaWF0IjoxNzQxMDA2NzIyLCJleHAiOjE3NDEwMTAzMjJ9.cAYl9XYFVKcYkCnwlqMZ7JeRdEJJrvpEXFTbXITfXeg

**Body**raw (json)

json

{

"report":"Earthquake",

"reportedBy": "ganesh",

"location": "ongole",

"emergency": "RED"

}

**GEThttp://localhost:5000/report/all**

[**Open request**](https://desktop.postman.com/?desktopVersion=11.35.0&userId=37401405&teamId=0&region=us)

http://localhost:5000/report/all

Method: GET URL: /report/all

Description: Get all reports

﻿

**Request Headers**

**Authorization**

Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjY3YzNkMDJjNGFmN2ZkYWI3YzU2MTExZSIsInJvbGUiOiJ1c2VyIiwiaWF0IjoxNzQxMDA2NzIyLCJleHAiOjE3NDEwMTAzMjJ9.cAYl9XYFVKcYkCnwlqMZ7JeRdEJJrvpEXFTbXITfXeg

**POSThttp://localhost:5000/emergency/create**

[**Open request**](https://desktop.postman.com/?desktopVersion=11.35.0&userId=37401405&teamId=0&region=us)

http://localhost:5000/emergency/create

Method: POST URL: /emergency/create

Description: Create an emergency

﻿

**Request Headers**

**Authorization**

Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjY3YzNkMDJjNGFmN2ZkYWI3YzU2MTExZSIsInJvbGUiOiJ1c2VyIiwiaWF0IjoxNzQxMDA2NzIyLCJleHAiOjE3NDEwMTAzMjJ9.cAYl9XYFVKcYkCnwlqMZ7JeRdEJJrvpEXFTbXITfXeg

**Body**raw (json)

json

{

"emergency": "Accident near sivanagar colony",

"location": "kanigiri",

"code": "RED"

}

**GEThttp://localhost:5000/emergency/all**

[**Open request**](https://desktop.postman.com/?desktopVersion=11.35.0&userId=37401405&teamId=0&region=us)

http://localhost:5000/emergency/all

Method: GET URL: /emergency/all

Description: Get all emergencies

﻿

**Request Headers**

**Authorization**

Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjY3YzNkMDJjNGFmN2ZkYWI3YzU2MTExZSIsInJvbGUiOiJ1c2VyIiwiaWF0IjoxNzQxMDA2NzIyLCJleHAiOjE3NDEwMTAzMjJ9.cAYl9XYFVKcYkCnwlqMZ7JeRdEJJrvpEXFTbXITfXeg

**GEThttp://localhost:5000/alert/any**

[**Open request**](https://desktop.postman.com/?desktopVersion=11.35.0&userId=37401405&teamId=0&region=us)

http://localhost:5000/alert/any

Method: GET URL: /alert/any

Description: Get all the alerts near by(based on requested user location)

﻿

**Request Headers**

**Authorization**

Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjY3YzNkMDJjNGFmN2ZkYWI3YzU2MTExZSIsInJvbGUiOiJ1c2VyIiwiaWF0IjoxNzQxMDA2NzIyLCJleHAiOjE3NDEwMTAzMjJ9.cAYl9XYFVKcYkCnwlqMZ7JeRdEJJrvpEXFTbXITfXeg

**POSThttp://localhost:5000/user/update**

[**Open request**](https://desktop.postman.com/?desktopVersion=11.35.0&userId=37401405&teamId=0&region=us)

http://localhost:5000/user/update

Method: POST URL: /user/update

Description: Update a user’s profile data

﻿

**Request Headers**

**Authorization**

Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjY3YzNkMDJjNGFmN2ZkYWI3YzU2MTExZSIsInJvbGUiOiJ1c2VyIiwiaWF0IjoxNzQxMDA2NzIyLCJleHAiOjE3NDEwMTAzMjJ9.cAYl9XYFVKcYkCnwlqMZ7JeRdEJJrvpEXFTbXITfXeg

**Body**raw (json)

json

{

"username": "remo"

}

**POSThttp://localhost:5000/user/logout**

[**Open request**](https://desktop.postman.com/?desktopVersion=11.35.0&userId=37401405&teamId=0&region=us)

http://localhost:5000/user/logout

Method: POST URL: /user/logout

Description: Logout a user

﻿

**Request Headers**

**Authorization**

Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjY3YzNkMDJjNGFmN2ZkYWI3YzU2MTExZSIsInJvbGUiOiJ1c2VyIiwiaWF0IjoxNzQxMDA2NzIyLCJleHAiOjE3NDEwMTAzMjJ9.cAYl9XYFVKcYkCnwlqMZ7JeRdEJJrvpEXFTbXITfXeg

**GEThttp://localhost:5000/report/all**

[**Open request**](https://desktop.postman.com/?desktopVersion=11.35.0&userId=37401405&teamId=0&region=us)

http://localhost:5000/report/all

Method: GET URL: /report/all

Description: Get all reports

﻿

**Request Headers**

**Authorization**

Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjY3YzNkMDJjNGFmN2ZkYWI3YzU2MTExZSIsInJvbGUiOiJ1c2VyIiwiaWF0IjoxNzQxMDA2NzIyLCJleHAiOjE3NDEwMTAzMjJ9.cAYl9XYFVKcYkCnwlqMZ7JeRdEJJrvpEXFTbXITfXeg

**Query Params**

**GEThttp://localhost:5000/report/near/ongole**

[**Open request**](https://desktop.postman.com/?desktopVersion=11.35.0&userId=37401405&teamId=0&region=us)

http://localhost:5000/report/near/ongole

Method: GET URL: /report/near/

Description: Get nearby reports(pass the location in url)

﻿

**Request Headers**

**Authorization**

Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjY3YzNkMDJjNGFmN2ZkYWI3YzU2MTExZSIsInJvbGUiOiJ1c2VyIiwiaWF0IjoxNzQxMDA2NzIyLCJleHAiOjE3NDEwMTAzMjJ9.cAYl9XYFVKcYkCnwlqMZ7JeRdEJJrvpEXFTbXITfXeg

**GEThttp://localhost:5000/report/filter?location=ongole**

[**Open request**](https://desktop.postman.com/?desktopVersion=11.35.0&userId=37401405&teamId=0&region=us)

http://localhost:5000/report/filter?location=ongole

Method: GET URL: /report/filter?

Description: Filter reports based on query

﻿

**Request Headers**

**Authorization**

Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjY3YzNkMDJjNGFmN2ZkYWI3YzU2MTExZSIsInJvbGUiOiJ1c2VyIiwiaWF0IjoxNzQxMDA2NzIyLCJleHAiOjE3NDEwMTAzMjJ9.cAYl9XYFVKcYkCnwlqMZ7JeRdEJJrvpEXFTbXITfXeg

**Query Params**

**location**

ongole

**PUThttp://localhost:5000/emergency/update/67c3e1a26c2c8e3244601a6a**

[**Open request**](https://desktop.postman.com/?desktopVersion=11.35.0&userId=37401405&teamId=0&region=us)

http://localhost:5000/emergency/update/67c3e1a26c2c8e3244601a6a

Method: PUT URL: /emergency/update

Description: Update an emergency and give the specific user id to update

﻿

**Request Headers**

**Authorization**

Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjY3YzNkMDJjNGFmN2ZkYWI3YzU2MTExZSIsInJvbGUiOiJ1c2VyIiwiaWF0IjoxNzQxMDA2NzIyLCJleHAiOjE3NDEwMTAzMjJ9.cAYl9XYFVKcYkCnwlqMZ7JeRdEJJrvpEXFTbXITfXeg

**Body**raw (json)

json

{

"description": "updated",

"emergency": "accident ",

"status": "in-progress",

"location": "kanigiri"

}

**GEThttp://localhost:5000/emergency/near/kanigiri**

[**Open request**](https://desktop.postman.com/?desktopVersion=11.35.0&userId=37401405&teamId=0&region=us)

http://localhost:5000/emergency/near/kanigiri

Method: GET URL: /emergency/near/

Description: Get emergencies near a location; mention location in url

﻿

**Request Headers**

**Authorization**

Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjY3YzNkMDJjNGFmN2ZkYWI3YzU2MTExZSIsInJvbGUiOiJ1c2VyIiwiaWF0IjoxNzQxMDA2NzIyLCJleHAiOjE3NDEwMTAzMjJ9.cAYl9XYFVKcYkCnwlqMZ7JeRdEJJrvpEXFTbXITfXeg

**GEThttp://localhost:5000/emergency/filter?code=RED**

[**Open request**](https://desktop.postman.com/?desktopVersion=11.35.0&userId=37401405&teamId=0&region=us)

http://localhost:5000/emergency/filter?code=RED

Method: GET URL: /emergency/filter?<query=value>

Description: Filter emergencies based on query parameter values

﻿

**Query Params**

**code**

RED