System Specifications Document

Criminal Record Management System

Nirban Roy, Bhaskar Saha, Debanjan Adak, Srijan Deb, Shreya Tiwari

Contents

1 Introduction

1.1 Purpose

This document provides a detailed specification for the **Criminal Record Management System**. It outlines the system's functionalities, architecture, security measures, and deployment processes. The document serves as a guide for developers, system administrators, and other technical stakeholders involved in the system's development and maintenance.

1.2 Scope

The **Criminal Record Management System** is designed to streamline the management of criminal records within a law enforcement framework. It offers tailored dashboards for various roles, including Judges, Police Officers, Superintendents, and CBI Officers, along with Admin Dashboards for user and database management. The system ensures secure access, data integrity, and efficient workflows across all user roles.

1.3 Definitions, Acronyms, and Abbreviations

- **RBAC**: Role-Based Access Control
- **JWT**: JSON Web Token
- **API**: Application Programming Interface
- **CBI**: Central Bureau of Investigation
- **UI**: User Interface
- **UX**: User Experience
- **CRUD**: Create, Read, Update, Delete
- **ER Diagram**: Entity-Relationship Diagram

1.4 References

1 IEEE Standard for Software Requirements Specifications.

2 Node.js Documentation. https://nodejs.org/en/docs/

3 Express.js Documentation. https://expressjs.com/

4 SQLite3 Documentation. https://www.sqlite.org/docs.html

5 Tailwind CSS Documentation. https://tailwindcss.com/docs

6 Bcrypt Documentation. https://www.npmjs.com/package/bcrypt

7 JSON Web Tokens (JWT). https://jwt.io/

2 System Overview

The **Criminal Record Management System** is a web-based application comprising a frontend built with HTML, CSS (Tailwind CSS), and JavaScript, and a backend developed using Node.js with Express.js. The system employs SQLite3 as its database and integrates security measures such as JWT for authentication and bcrypt for password hashing. Additionally, Groq SDK is utilized for advanced data querying capabilities.

3 Functional Requirements

3.1 User Authentication

- Users can log in using their credentials.
- Passwords are securely hashed using bcrypt.
- JWTs are issued upon successful authentication for session management.
- Implement secure logout functionality to invalidate JWTs.
- Enforce strong password policies during user registration and password changes.

3.2 Role-Based Dashboards

- **Judge Dashboard**: Manage and oversee case details.
- **Police Officer Dashboard**: Register criminals, assign jails, and transfer criminals.
- **Superintendent Dashboard**: Comprehensive criminal management including biometrics and health records.
- **CBI Dashboard**: File case reports and manage evidence.
- **Admin Dashboards**: Create, update, and delete users; manage database entries; assign roles to users.

3.3 Case Management

- Create, update, and delete cases.
- Assign judges to cases and schedule hearings.
- Link cases to criminals and track case progress.
- Generate case reports and summaries.

3.4 Criminal Management

- Register new criminals with detailed information including personal and biometric data.
- Record crimes committed by criminals.
- Collect and update biometric data.
- Maintain location details including jail assignments and cell numbers.
- Record meetings between criminals and outsiders.
- Update health conditions of criminals.
- Assign works or tasks to criminals.
- Monitor and update criminal status (e.g., incarcerated, released, transferred).

3.5 User Management

- Admins can create, update, and delete user accounts.
- Assign roles to users to control access levels.
- Reset user passwords and manage user permissions.
- View and audit user activity logs.

3.6 Security Measures

- Implement RBAC to restrict access based on user roles.
- Use HTTPS for secure data transmission.
- Implement rate limiting to prevent abuse.
- Sanitize and validate all user inputs to prevent SQL injection and XSS attacks.
- Encrypt sensitive data both in transit and at rest.
- Regularly update and patch system components to protect against vulnerabilities.

4 Non-Functional Requirements

4.1 Performance

- The system should handle up to 100 concurrent users without performance degradation.
- Response time for any API request should not exceed 2 seconds under normal load.
- Optimize database queries to ensure efficient data retrieval and storage.

4.2 Scalability

- The system architecture should support easy scaling to accommodate more users and data.
- Design the database to handle large volumes of records efficiently.
- Implement load balancing and distributed computing if necessary.

4.3 Security

- All sensitive data must be encrypted in transit and at rest.
- Implement robust authentication and authorization mechanisms.
- Conduct regular security audits and penetration testing.
- Ensure compliance with relevant data protection regulations.

4.4 Usability

- The user interface should be intuitive and easy to navigate.
- Provide responsive design to ensure compatibility across various devices and screen sizes.
- Offer training materials and support for users.
- Implement accessibility features to accommodate users with disabilities.

4.5 Maintainability

- Codebase should follow best practices and be well-documented.
- Implement automated testing to ensure system reliability.
- Modularize the code to facilitate easy updates and maintenance.
- Use version control systems to manage code changes effectively.

5 System Architecture

5.1 Overview

The **Criminal Record Management System** follows a client-server architecture with a clear separation between the frontend and backend. The frontend interacts with users, while the backend handles data processing, business logic, and database interactions. This separation ensures scalability, maintainability, and security.

5.2 Components

- **Frontend**:
  - Built with HTML, Tailwind CSS, and JavaScript.
  - Provides responsive and user-friendly dashboards tailored to each role.
  - Implements client-side validation for user inputs.
- **Backend**:
  - Developed using Node.js and Express.js.
  - Manages API endpoints, authentication, and data processing.
  - Implements server-side validation and error handling.
- **Database**:
  - Utilizes SQLite3 for relational data storage.
  - Stores user information, criminal records, cases, and other pertinent data.
  - Ensures data integrity through relational mappings and constraints.
- **Security**:
  - Employs JWT for secure authentication.
  - Uses bcrypt for password hashing.
  - Implements middleware for authorization and input sanitization.
- **Groq SDK**:
  - Utilized for advanced data querying and management.
  - Enhances the system's ability to perform complex searches and data manipulations.

5.3 Data Flow Diagram

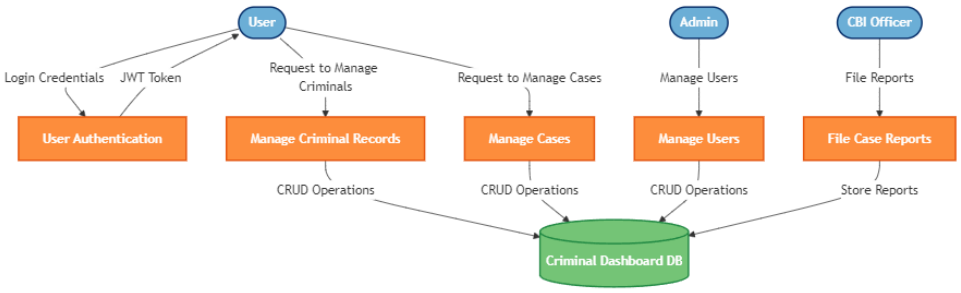

Figure 1: Data Flow Diagram

5.4 Entity-Relationship Diagram

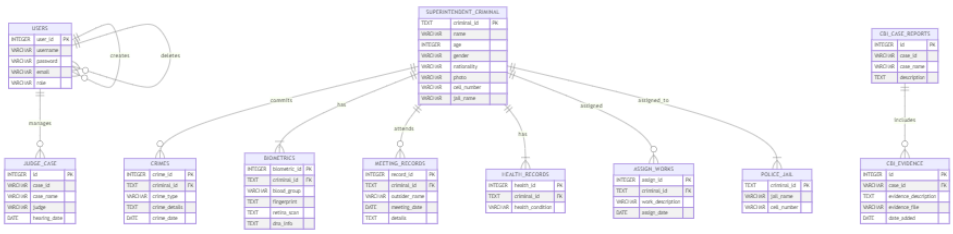

Figure 2: Entity-Relationship Diagram

6 Technologies Used

6.1 Frontend

- **HTML5**: Structure of web pages.
- **Tailwind CSS**: Utility-first CSS framework for rapid UI development.
- **JavaScript**: Interactive functionalities and dynamic content.

- **Responsive Design**: Ensures compatibility across various devices and screen sizes.

6.2 Backend

- **Node.js**: JavaScript runtime for server-side development.
- **Express.js**: Web framework for building API endpoints.
- **SQLite3**: Lightweight relational database for data storage.
- **Multer**: Middleware for handling file uploads.
- **bcrypt**: Library for hashing passwords.
- **JSON Web Tokens (JWT)**: Secure token-based authentication.
- **Groq SDK**: Advanced data querying capabilities.
- **dotenv**: Manages environment variables.
- **CORS**: Cross-Origin Resource Sharing to control resource access.

6.3 Security

- **HTTPS**: Secure data transmission between client and server.
- **RBAC**: Role-Based Access Control to restrict user access.
- **Rate Limiting**: Prevents abuse by limiting the number of requests.
- **Input Validation and Sanitization**: Protects against SQL injection and XSS attacks.
- **Encryption**: Ensures sensitive data is encrypted at rest and in transit.

7 Database Schema

7.1 Overview

The database is structured to support various functionalities across different dashboards. It ensures efficient data management and retrieval while maintaining data integrity through relational mappings.

7.2 Tables

Table Name	Description
USERS	Stores user credentials and roles, including username, hashed password, email, and assigned role.
SUPERINTENDENT_CRIMINAL	Contains criminal personal details such as ID, name, age, gender, nationality, photo, cell number, and jail name.
CRIMES	Records crimes associated with criminals, including crime type, details, and date. Linked to criminals via criminal ID.
BIOMETRICS	Stores biometric data of criminals, including photos, blood group, fingerprint, retina scan, and DNA information.
MEETING_RECORDS	Logs meetings between criminals and outsiders, capturing names, dates, and meeting details.
HEALTH_RECORDS	Maintains health condition records for each criminal, including any medical updates or conditions.
ASSIGN_WORKS	Assigns work-related tasks to criminals, detailing work descriptions and assignment dates.

Continued on next page

Table 1 Continued from previous page

Table Name	Description
JUDGE_CASE	Manages court cases handled by judges, including case IDs, names, assigned judges, and hearing dates.
POLICE_JAIL	Associates criminals with their jail assignments and cell numbers, facilitating location management.
CBI_CASE_REPORTS	Stores case reports filed by CBI officers, including case IDs, names, and detailed descriptions.
CBI_EVIDENCE	Manages evidence related to CBI case reports, including descriptions, file references, and dates added.

8 API Endpoints

The system provides a set of API endpoints to facilitate communication between the frontend and backend. These endpoints handle user authentication, user management, criminal management, case management, and more, ensuring seamless operations across different roles.

8.1 Authentication

POST `/api/login`

- **Description**: Authenticates users and issues JWTs.
- **Request Body**:

```
{
  "username": "string",
  "password": "string"
}

```

- **Response**:

```
{
  "token": "JWT_TOKEN"
}

```

8.2 User Management

GET `/api/users`

- **Description**: Retrieves a list of all users.
- **Headers**: Authorization: Bearer `JWT_TOKEN`
- **Response**:

```
{
  "users": [
    {
      "user_id": 1,
      "username": "admin",
      "role": "Admin_Judge"
    },
    ...
  ]
}

```

POST `/api/users`

- **Description**: Creates a new user (Admins only).
- **Headers**: Authorization: Bearer `JWT_TOKEN`

- **Request Body**:

```
{
  "username": "string",
  "password": "string",
  "role": "string"
}
```

- **Response**:

```
{
  "message": "User created successfully.",
  "user_id": 2
}
```

DELETE `/api/users/:userId`

- **Description**: Deletes a user by ID (Admins only).
- **Headers**: Authorization: Bearer `JWT_TOKEN`
- **Parameters**:
  - `userId`: *integer* - ID of the user to delete.
- **Response**:

```
{
  "message": "User deleted successfully."
}
```

8.3 Case Management

POST `/api/cases`

- **Description**: Adds a new case (Judges only).
- **Headers**: Authorization: Bearer `JWT_TOKEN`
- **Request Body**:

```
{
  "caseId": "string",
  "caseName": "string",
  "judge": "string",
  "hearingDate": "YYYY-MM-DD"
}
```

- **Response**:

```
{
  "case": {
    "id": 1,
    "caseId": "C123",
    "caseName": "Case Name",
    "judge": "Judge Name",
    "hearingDate": "2024-12-01"
  }
}
```

GET `/api/cases`

- **Description**: Retrieves all cases.
- **Headers**: Authorization: Bearer `JWT_TOKEN`

- **Response**:

```
{
  "cases": [
    {
      "id": 1,
      "case_id": "C123",
      "case_name": "Case Name",
      "judge": "Judge Name",
      "hearing_date": "2024-12-01"
    },
    ...
  ]
}

```

PUT `/api/cases/:id`

- **Description**: Updates a case by ID (Judges only).
- **Headers**: Authorization: Bearer `JWT_TOKEN`
- **Parameters**:
  - `id`: *integer* - ID of the case to update.
- **Request Body**:

```
{
  "case_id": "string",
  "case_name": "string",
  "judge": "string",
  "hearing_date": "YYYY-MM-DD"
}

```

- **Response**:

```
{
  "message": "Case updated successfully."
}

```

DELETE `/api/cases/:id`

- **Description**: Deletes a case by ID (Judges only).
- **Headers**: Authorization: Bearer `JWT_TOKEN`
- **Parameters**:
  - `id`: *integer* - ID of the case to delete.
- **Response**:

```
{
  "message": "Case deleted successfully."
}

```

8.4 Criminal Management

POST `/api/register-criminal`

- **Description**: Registers a new criminal (Superintendents and Admins).
- **Headers**: Authorization: Bearer `JWT_TOKEN`
- **Form Data**:

```
criminal_id: string
name: string
age: integer
gender: string
nationality: string
photo: file (optional)

```

- **Response**:

```
{
  "criminal": {
    "criminal_id": "C123",
    "name": "John Doe",
    "age": 35,
    "gender": "Male",
    "nationality": "Country",
    "photo": "filename.jpg"
  }
}

```

GET `/api/criminal-info/:criminal_id`

- **Description**: Retrieves comprehensive information about a criminal.
- **Headers**: Authorization: Bearer `JWT_TOKEN`
- **Parameters**:
  - `criminal_id`: *string* - ID of the criminal.
- **Response**:

```
{
  "basicInfo": {
    "criminal_id": "C123",
    "name": "John Doe",
    "age": 35,
    "gender": "Male",
    "nationality": "Country",
    "photo": "filename.jpg"
  },
  "biometrics": {
    "blood_group": "O+",
    "fingerprint": "fingerprint_data",
    "retina_scan": "retina_scan_data",
    "dna_info": "dna_info_data"
  },
  "crimes": [
    {
      "crime_id": 1,
      "crime_type": "Theft",
      "crime_details": "Details of the theft",
      "crime_date": "2023-05-20"
    },
    ...
  ],
  "meetings": [
    {
      "record_id": 1,
      "outsider_name": "Jane Smith",
      "meeting_date": "2023-06-15",
      "details": "Details of the meeting"
    },
    ...
  ],
  "health": {
    "health_condition": "Healthy"
  },
  "works": [
    {
```

```
      "assign_id": 1,
      "work_description": "Cleaning duties",
      "assign_date": "2023-07-01"
    },
    ...
  ],
  "cbiCases": [
    {
      "case_id": "CBI001",
      "case_name": "Case Name",
      "description": "Case Description"
    },
    ...
  ],
  "cbiEvidence": [
    {
      "id": 1,
      "case_id": "CBI001",
      "evidence_description": "Evidence Description",
      "evidence_file": "evidence_file.jpg",
      "date_added": "2023-07-10"
    },
    ...
  ]
}
```

PUT `/api/criminal/:criminal_id`

- **Description**: Updates criminal details (Superintendents and Admins).
- **Headers**: Authorization: Bearer `JWT_TOKEN`
- **Parameters**:
  - `criminal_id`: *string* - ID of the criminal to update.
- **Request Body**:

```
{
  "name": "string",
  "age": integer,
  "gender": "string",
  "nationality": "string",
  "photo": "file (optional)"
}
```

- **Response**:

```
{
  "message": "Criminal updated successfully."
}
```

DELETE `/api/criminal/:criminal_id`

- **Description**: Deletes a criminal record by ID (Superintendents and Admins).
- **Headers**: Authorization: Bearer `JWT_TOKEN`
- **Parameters**:
  - `criminal_id`: *string* - ID of the criminal to delete.
- **Response**:

```
{
  "message": "Criminal deleted successfully."
}
```

8.5 File Case Reports (CBI Dashboard)

POST `/api/file-case-report`

- **Description**: Files a new case report (CBI Officers and Admins).
- **Headers**: Authorization: Bearer `JWT_TOKEN`
- **Request Body**:

```
{
  "case_id": "string",
  "case_name": "string",
  "description": "string"
}

```

- **Response**:

```
{
  "case_report": {
    "id": 1,
    "case_id": "CBI001",
    "case_name": "Case Name",
    "description": "Case Description"
  },
  "message": "Case report filed successfully."
}

```

POST `/api/add-evidence`

- **Description**: Adds evidence to a case report (CBI Officers and Admins).
- **Headers**: Authorization: Bearer `JWT_TOKEN`
- **Form Data**:

```
case_id: string
evidence_description: string
evidence_file: file

```

- **Response**:

```
{
  "evidence": {
    "id": 1,
    "case_id": "CBI001",
    "evidence_description": "Evidence Description",
    "evidence_file": "evidence_file.jpg",
    "date_added": "2024-04-25T10:20:30Z"
  },
  "message": "Evidence added successfully."
}

```

8.6 Additional Endpoints

Example: Assign Jail to Criminal (Police Officer Dashboard)

- **POST** `/api/assign-jail`
  - **Description**: Assigns a jail and cell number to a criminal (Police Officers and Admins).
  - **Headers**: Authorization: Bearer `JWT_TOKEN`
  - **Request Body**:

```
{
  "criminal_id": "string",
  "jail_name": "string",
  "cell_number": "string"
}
```

– **Response**:

```
{
  "message": "Jail assigned successfully."
}
```

9 Security Measures

9.1 Authentication and Authorization

- Utilize JWTs for secure session management.
- Implement RBAC to restrict access based on user roles.
- Passwords are hashed using bcrypt before storage.
- Enforce token expiration and implement token refresh mechanisms.

9.2 Data Protection

- Use HTTPS to encrypt data in transit.
- Sanitize and validate all user inputs to prevent SQL injection and XSS attacks.
- Store sensitive data securely with encryption at rest.
- Implement proper access controls to ensure data is only accessible to authorized users.

9.3 Rate Limiting and Throttling

- Implement rate limiting to protect against brute-force attacks and API abuse.
- Use middleware to monitor and control the number of requests from individual IP addresses.

9.4 Regular Security Audits

- Conduct regular security audits and penetration testing to identify and mitigate vulnerabilities.
- Keep all dependencies and libraries up-to-date to patch known vulnerabilities.
- Implement logging and monitoring to detect and respond to security incidents promptly.

10 User Roles and Permissions

10.1 Roles

- **Admin_Judge**: Administer Judges and manage related data.
- **Admin_Police**: Administer Police Officers and manage related data.
- **Admin_Superintendent**: Administer Superintendents and manage related data.
- **Admin_CBI**: Administer CBI Officers and manage related data.
- **Judge**: Manage cases and oversee judicial processes.
- **Police Officer**: Register criminals, assign jails, and manage criminal data.
- **Superintendent**: Comprehensive criminal management including biometrics and health records.
- **CBI Officer**: File case reports and manage evidence.

10.2 Permissions

- **Admins**:
  - Create, update, and delete users.
  - Assign roles to users to control access levels.
  - Access and modify all data related to their respective roles.
  - Oversee and manage database entries to ensure data integrity.
- **Judges**:
  - Manage and oversee case details.
  - Assign judges to cases and schedule hearings.
  - Access detailed criminal information relevant to cases.
  - Update case statuses and track progress.
- **Police Officers**:
  - Register criminals and record crimes.
  - Assign jails and transfer criminals.
  - Manage criminal location details.
  - Update criminal statuses.
- **Superintendents**:
  - Comprehensive criminal management.
  - Collect and update biometric data.
  - Maintain location and health records.
  - Assign works or tasks to criminals.
- **CBI Officers**:
  - File case reports.
  - Manage and add evidence.
  - Access detailed case information and associated evidence.

11 Deployment Instructions

11.1 Prerequisites

- **Node.js** and **npm** installed on the server.
- **SQLite3** installed for database management.
- A web server (e.g., **Nginx** or **Apache**) to serve the frontend and proxy API requests.
- SSL certificates for HTTPS to ensure secure data transmission.
- **Groq SDK** API key for advanced data querying.

11.2 Setup Steps

1. **Clone the Repository**

```
git clone https://github.com/yourusername/criminal-record-management.git
cd criminal-record-management

```

Note: As there is no GitHub repository at this time, ensure you have the project files locally on your machine.

2. **Install Dependencies**

```
npm install

```

3. **Configure Environment Variables**

- Create a '.env' file in the root directory.
- Add the following variables:

```
PORT=3000
JWT_SECRET=your_jwt_secret_key
GROQ_API_KEY=your_groq_api_key

```

4. **Initialize the Database**

```
node init_db.js

```

5. **Start the Server**

```
node backend.js

```

Note: Ensure that 'backend.js' is your main server file. Adjust accordingly if different.

6. **Access the Application**

- Open a web browser and navigate to http://localhost:3000

7. **Deploy Frontend**

- Serve the frontend files using a web server (e.g., Nginx).
- Ensure the frontend can communicate with the backend APIs securely.

8. **Secure the Application**

- Configure HTTPS with valid SSL certificates.
- Set up firewall rules to allow necessary traffic.
- Regularly update and patch system components.

9. **Monitor and Maintain**

- Implement logging and monitoring tools to track system performance and security.
- Schedule regular backups of the database to prevent data loss.
- Set up alerts for critical issues such as server downtime or security breaches.

12 System Maintenance

12.1 Regular Updates

- Keep all dependencies and libraries up-to-date to patch known vulnerabilities.
- Monitor security advisories related to the technologies used.
- Update the system regularly to incorporate new features and improvements.

12.2 Backup and Recovery

- Schedule regular backups of the SQLite3 database.
- Implement a recovery plan to restore data in case of system failures.
- Store backups in secure, off-site locations.

12.3 Monitoring

- Use monitoring tools to track system performance and uptime.
- Set up alerts for critical issues such as server downtime or security breaches.
- Analyze logs regularly to identify and address potential issues.

13 Code Listings

13.1 Database Initialization Script

File: `init_db.js`

```
const sqlite3 = require('sqlite3').verbose();
const path = require('path');

const dbPath = path.resolve(__dirname, 'criminal_dashboard.db');
const db = new sqlite3.Database(dbPath, (err) => {
  if (err) {
    console.error('Error opening database:', err.message);
  } else {
    console.log('Connected to SQLite database.');
  }
});

// Execute schema creation for Superintendent, Judge, Police, and CBI dashboards
const initSchema = () => {
  const schema = `
    -- Superintendent Criminal Table
    CREATE TABLE IF NOT EXISTS SUPERINTENDENT_CRIMINAL (
        criminal_id TEXT PRIMARY KEY,
        name TEXT NOT NULL,
        age INTEGER NOT NULL,
        gender TEXT NOT NULL,
        nationality TEXT NOT NULL,
        photo TEXT,
        cell_number TEXT,
        jail_name TEXT
    );

    -- Crimes Table
    CREATE TABLE IF NOT EXISTS CRIMES (
        crime_id INTEGER PRIMARY KEY AUTOINCREMENT,
        criminal_id TEXT NOT NULL,
        crime_type TEXT NOT NULL,
        crime_details TEXT NOT NULL,
        crime_date TEXT NOT NULL,
        FOREIGN KEY(criminal_id) REFERENCES SUPERINTENDENT_CRIMINAL(criminal_id)
    );

    -- Biometrics Table
    CREATE TABLE IF NOT EXISTS BIOMETRICS (
        biometric_id INTEGER PRIMARY KEY AUTOINCREMENT,
        criminal_id TEXT UNIQUE NOT NULL,
        photo_front TEXT,
        photo_side TEXT,
        photo_back TEXT,
        blood_group TEXT NOT NULL,
        fingerprint TEXT,
        retina_scan TEXT,
        dna_info TEXT NOT NULL,
        FOREIGN KEY(criminal_id) REFERENCES SUPERINTENDENT_CRIMINAL(criminal_id)
```

```
    );

    -- Meeting Records Table
    CREATE TABLE IF NOT EXISTS MEETING_RECORDS (
        record_id INTEGER PRIMARY KEY AUTOINCREMENT,
        criminal_id TEXT NOT NULL,
        outsider_name TEXT NOT NULL,
        meeting_date TEXT NOT NULL,
        details TEXT NOT NULL,
        FOREIGN KEY(criminal_id) REFERENCES SUPERINTENDENT_CRIMINAL(criminal_id)
    );

    -- Health Records Table
    CREATE TABLE IF NOT EXISTS HEALTH_RECORDS (
        health_id INTEGER PRIMARY KEY AUTOINCREMENT,
        criminal_id TEXT UNIQUE NOT NULL,
        health_condition TEXT NOT NULL,
        FOREIGN KEY(criminal_id) REFERENCES SUPERINTENDENT_CRIMINAL(criminal_id)
    );

    -- Assign Works Table
    CREATE TABLE IF NOT EXISTS ASSIGN_WORKS (
        assign_id INTEGER PRIMARY KEY AUTOINCREMENT,
        criminal_id TEXT NOT NULL,
        work_description TEXT NOT NULL,
        assign_date TEXT NOT NULL,
        FOREIGN KEY(criminal_id) REFERENCES SUPERINTENDENT_CRIMINAL(criminal_id)
    );

    -- Users Table (Updated to include 'email' and 'role')
    CREATE TABLE IF NOT EXISTS USERS (
        user_id INTEGER PRIMARY KEY AUTOINCREMENT,
        username TEXT UNIQUE NOT NULL,
        password TEXT NOT NULL,
        email TEXT,
        role TEXT NOT NULL
    );

    -- Judge Case Table
    CREATE TABLE IF NOT EXISTS JUDGE_CASE (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        case_id TEXT NOT NULL UNIQUE,
        case_name TEXT NOT NULL,
        judge TEXT NOT NULL,
        hearing_date TEXT NOT NULL
    );

    -- Police Jail Table
    CREATE TABLE IF NOT EXISTS POLICE_JAIL (
        criminal_id TEXT PRIMARY KEY,
        jail_name TEXT NOT NULL,
        cell_number TEXT NOT NULL,
        FOREIGN KEY(criminal_id) REFERENCES SUPERINTENDENT_CRIMINAL(criminal_id)
    );

    -- CBI Case Reports Table
    CREATE TABLE IF NOT EXISTS CBI_CASE_REPORTS (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        case_id TEXT UNIQUE NOT NULL,
        case_name TEXT NOT NULL,
        description TEXT NOT NULL
    );

    -- CBI Evidence Table
    CREATE TABLE IF NOT EXISTS CBI_EVIDENCE (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        case_id TEXT NOT NULL,
        evidence_description TEXT NOT NULL,
        evidence_file TEXT NOT NULL,
        date_added TEXT DEFAULT CURRENT_TIMESTAMP,
        FOREIGN KEY(case_id) REFERENCES CBI_CASE_REPORTS(case_id)
    );
  `;
```

```
123
124   db.exec(schema, (err) => {
125     if (err) {
126       console.error('Error creating tables:', err.message);
127     } else {
128       console.log('Database schema initialized.');
129     }
130   });
131 };
132
133 // Initialize the database schema
134 initSchema();
135
136 // Export the database connection for use in other modules
137 module.exports = db;
```

13.2 Credentials Configuration

File: credentials.js

```
1  const credentials = {
2      admin: {
3          username: "admin",
4          password: "admin123",
5          redirect: "admin.html"
6      },
7      officer: {
8          username: "officer",
9          password: "officer123",
10         redirect: "police.html"
11     },
12     judge: {
13         username: "judge",
14         password: "judge123",
15         redirect: "judge.html"
16     },
17     jailsuperintendent: {
18         username: "jailsup",
19         password: "jailsup123",
20         redirect: "jailsuperintendent.html"
21     },
22     CBI: {
23         username: "cbi",
24         password: "cbi123",
25         redirect: "CBI.html" % Corrected the redirect path
26     }
27 };
28
29 // Export credentials for use in authentication
30 module.exports = credentials;
```

13.3 Login Endpoint

File: backend.js

```
1  // Required Modules
2  const express = require('express');
3  const bcrypt = require('bcrypt');
4  const jwt = require('jsonwebtoken');
5  const cors = require('cors');
6  const dotenv = require('dotenv');
7  const db = require('./init_db'); // Database connection
8  const credentials = require('./credentials'); // User credentials
9
10 dotenv.config();
11
12 const app = express();
13
14 // Middleware
15 app.use(cors());
16 app.use(express.json());
```

```
17
18 // Login Endpoint
19 app.post('/api/login', async (req, res) => {
20     const { username, password } = req.body;
21
22     // Input validation
23     if (!username || !password) {
24         return res.status(400).json({ error: 'Username and password are required.' });
25     }
26
27     try {
28         // Query to find the user by username
29         const query = `SELECT * FROM USERS WHERE username = ?`;
30         db.get(query, [username], async (err, user) => {
31             if (err) {
32                 console.error('Error querying the database:', err.message);
33                 return res.status(500).json({ error: 'Internal server error.' });
34             }
35
36             if (!user) {
37                 // User not found
38                 return res.status(401).json({ error: 'Invalid username or password.' });
39             }
40
41             try {
42                 // Compare the provided password with the hashed password in the
    database
43                 const match = await bcrypt.compare(password, user.password);
44
45                 if (match) {
46                     // Passwords match, authentication successful
47
48                     // Create JWT payload
49                     const payload = {
50                         user_id: user.user_id,
51                         username: user.username,
52                         role: user.role
53                     };
54
55                     // Sign the token
56                     const token = jwt.sign(payload, process.env.JWT_SECRET, { expiresIn:
     '8h' });
57
58                     // Send the token to the client
59                     res.json({ token });
60                 } else {
61                     // Passwords do not match
62                     res.status(401).json({ error: 'Invalid username or password.' });
63                 }
64             } catch (error) {
65                 console.error('Error comparing passwords:', error);
66                 res.status(500).json({ error: 'Internal server error.' });
67             }
68         });
69     } catch (error) {
70         console.error('Unexpected error:', error);
71         res.status(500).json({ error: 'Internal server error.' });
72     }
73 });
74
75 // Start the Server
76 const port = process.env.PORT || 3000;
77 app.listen(port, () => {
78     console.log(`Server is running on http://localhost:${port}`);
79 });
```

14 Security Measures

14.1 Authentication and Authorization

- Utilize JWTs for secure session management.

- Implement RBAC to restrict access based on user roles.
- Passwords are hashed using bcrypt before storage.
- Enforce token expiration and implement token refresh mechanisms.
- Protect sensitive endpoints by requiring valid JWTs.

14.2 Data Protection

- Use HTTPS to encrypt data in transit.
- Sanitize and validate all user inputs to prevent SQL injection and XSS attacks.
- Store sensitive data securely with encryption at rest.
- Implement proper access controls to ensure data is only accessible to authorized users.

14.3 Rate Limiting and Throttling

- Implement rate limiting to protect against brute-force attacks and API abuse.
- Use middleware to monitor and control the number of requests from individual IP addresses.

14.4 Regular Security Audits

- Conduct regular security audits and penetration testing to identify and mitigate vulnerabilities.
- Keep all dependencies and libraries up-to-date to patch known vulnerabilities.
- Implement logging and monitoring to detect and respond to security incidents promptly.

14.5 Input Validation and Sanitization

- Validate all incoming data on both client and server sides.
- Use libraries like ‘validator.js‘ to sanitize and validate inputs beyond basic checks.
- Implement parameterized queries to prevent SQL injection.

14.6 Data Encryption

- Encrypt sensitive data such as personal information and biometrics in the database.
- Use secure encryption algorithms and manage encryption keys securely.

15 User Roles and Permissions

15.1 Roles

- **Admin_Judge**: Administer Judges and manage related data.
- **Admin_Police**: Administer Police Officers and manage related data.
- **Admin_Superintendent**: Administer Superintendents and manage related data.
- **Admin_CBI**: Administer CBI Officers and manage related data.
- **Judge**: Manage cases and oversee judicial processes.
- **Police Officer**: Register criminals, assign jails, and manage criminal data.
- **Superintendent**: Comprehensive criminal management including biometrics and health records.
- **CBI Officer**: File case reports and manage evidence.

15.2 Permissions

- **Admins**:
  - Create, update, and delete users.
  - Assign roles to users to control access levels.
  - Access and modify all data related to their respective roles.
  - Oversee and manage database entries to ensure data integrity.
- **Judges**:
  - Manage and oversee case details.
  - Assign judges to cases and schedule hearings.
  - Access detailed criminal information relevant to cases.
  - Update case statuses and track progress.
- **Police Officers**:
  - Register criminals and record crimes.
  - Assign jails and transfer criminals.
  - Manage criminal location details.
  - Update criminal statuses.
- **Superintendents**:
  - Comprehensive criminal management.
  - Collect and update biometric data.
  - Maintain location and health records.
  - Assign works or tasks to criminals.
- **CBI Officers**:
  - File case reports.
  - Manage and add evidence.
  - Access detailed case information and associated evidence.

16 Deployment Instructions

16.1 Prerequisites

- **Node.js** and **npm** installed on the server.
- **SQLite3** installed for database management.
- A web server (e.g., **Nginx** or **Apache**) to serve the frontend and proxy API requests.
- SSL certificates for HTTPS to ensure secure data transmission.
- **Groq SDK** API key for advanced data querying.

16.2 Setup Steps

1. **Clone the Repository**

```
# Since there is no GitHub repository at this time, ensure you have the project files locally.
# Place all project files in a designated directory, e.g., /var/www/criminal-record-management

```

2. **Install Dependencies**

```
cd /path/to/criminal-record-management
npm install

```

3. **Configure Environment Variables**

   - Create a '.env' file in the root directory.
   - Add the following variables:

```
PORT=3000
JWT_SECRET=your_jwt_secret_key
GROQ_API_KEY=your_groq_api_key

```

- Replace 'your$_{j}wt_{s}ecret_{k}ey$'$with a strong secret key. Replace 'your_{g}roq_{a}pi_{k}ey'$ with your Groq SDK API key.

4. **Initialize the Database**

```
node init_db.js

```

   Note: This will create the SQLite3 database and necessary tables as defined in 'init$_{d}b.js$'.

5. **Start the Server**

```
node backend.js

```

   Note: Ensure that 'backend.js' is your main server file. Adjust accordingly if different.

6. **Access the Application**

   - Open a web browser and navigate to http://localhost:3000

7. **Deploy Frontend**

   - Serve the frontend files using a web server (e.g., Nginx).
   - Ensure the frontend can communicate with the backend APIs securely.

8. **Secure the Application**

   - Configure HTTPS with valid SSL certificates.
   - Set up firewall rules to allow necessary traffic (e.g., port 443 for HTTPS).
   - Regularly update and patch system components to protect against vulnerabilities.

9. **Monitor and Maintain**

   - Implement logging and monitoring tools to track system performance and security.
   - Schedule regular backups of the database to prevent data loss.
   - Set up alerts for critical issues such as server downtime or security breaches.

17 System Maintenance

17.1 Regular Updates

- Keep all dependencies and libraries up-to-date to patch known vulnerabilities.
- Monitor security advisories related to the technologies used.
- Update the system regularly to incorporate new features and improvements.

17.2 Backup and Recovery

- Schedule regular backups of the SQLite3 database.
- Implement a recovery plan to restore data in case of system failures.
- Store backups in secure, off-site locations.

17.3 Monitoring

- Use monitoring tools to track system performance and uptime.
- Set up alerts for critical issues such as server downtime or security breaches.
- Analyze logs regularly to identify and address potential issues.

18 Code Listings

18.1 Backend Server Configuration

File: `backend.js`

```
// Required Modules
const express = require('express');
const bcrypt = require('bcrypt');
const jwt = require('jsonwebtoken');
const cors = require('cors');
const dotenv = require('dotenv');
const db = require('./init_db'); // Database connection
const credentials = require('./credentials'); // User credentials

dotenv.config();

const app = express();

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

// Authentication Middleware
const authenticateJWT = (req, res, next) => {
    const authHeader = req.headers.authorization;

    if (authHeader) {
        const token = authHeader.split(' ')[1];

        jwt.verify(token, process.env.JWT_SECRET, (err, user) => {
            if (err) {
                return res.sendStatus(403); // Invalid token
            }

            req.user = user;
            next();
        });
    } else {
        res.sendStatus(401); // No token provided
    }
};

```

```
// Authorization Middleware
const authorizeRoles = (...allowedRoles) => {
    return (req, res, next) => {
        if (!allowedRoles.includes(req.user.role)) {
            return res.sendStatus(403); // Forbidden
        }
        next();
    };
};

// Login Endpoint (Already defined in previous section)

// Example: User Creation Endpoint (Admins only)
app.post('/api/users', authenticateJWT, authorizeRoles('Admin_Judge', 'Admin_Police', 'Admin_Superintendent', 'Admin_CBI'), async (req, res) => {
    const { username, password, role, email } = req.body;

    if (!username || !password || !role) {
        return res.status(400).json({ error: 'Username, password, and role are required.' });
    }

    try {
        const hashedPassword = await bcrypt.hash(password, 10); // Hash the password
        const query = `INSERT INTO USERS (username, password, email, role) VALUES (?, ?, ?, ?)`;
        db.run(query, [username, hashedPassword, email || null, role], function (err) {
            if (err) {
                console.error('Error creating user:', err.message);
                if (err.message.includes('UNIQUE constraint failed')) {
                    res.status(400).json({ error: 'Username already exists.' });
                } else {
                    res.status(500).json({ error: 'Internal server error' });
                }
            } else {
                res.json({ message: 'User created successfully.', user_id: this.lastID });
            }
        });
    } catch (error) {
        console.error('Error creating user:', error);
        res.status(500).json({ error: 'Internal server error' });
    }
});

// Additional endpoints follow the same structure...

// Start the Server
const port = process.env.PORT || 3000;
app.listen(port, () => {
    console.log(`Server is running on http://localhost:${port}`);
});
```

18.2 User Credentials Configuration

File: `credentials.js`

```
// credentials.js
const credentials = {
    admin: {
        username: "admin",
        password: "admin123",
        redirect: "admin.html"
    },
    officer: {
        username: "officer",
        password: "officer123",
        redirect: "police.html"
    },
    judge: {
        username: "judge",
        password: "judge123",
```

```
        redirect: "judge.html"
    },
    jailsuperintendent: {
        username: "jailsup",
        password: "jailsup123",
        redirect: "jailsuperintendent.html"
    },
    CBI: {
        username: "cbi",
        password: "cbi123",
        redirect: "CBI.html" // Corrected the redirect path
    }
};

// Export credentials for use in authentication
module.exports = credentials;
```

18.3 Sample API Endpoint: Update Case (Judge Dashboard)

File: backend.js

```
// Update Case Endpoint (Judge Dashboard)
app.put('/api/cases/:id', authenticateJWT, authorizeRoles('Judge'), (req, res) => {
    const { id } = req.params;
    const { case_id, case_name, judge, hearing_date } = req.body;

    if (!case_id && !case_name && !judge && !hearing_date) {
        return res.status(400).json({ error: 'At least one field is required for update.' });
    }

    let fields = [];
    let params = [];

    if (case_id) {
        fields.push('case_id = ?');
        params.push(case_id);
    }
    if (case_name) {
        fields.push('case_name = ?');
        params.push(case_name);
    }
    if (judge) {
        fields.push('judge = ?');
        params.push(judge);
    }
    if (hearing_date) {
        fields.push('hearing_date = ?');
        params.push(hearing_date);
    }

    const query = `UPDATE JUDGE_CASE SET ${fields.join(', ')} WHERE id = ?`;
    params.push(id);

    db.run(query, params, function (err) {
        if (err) {
            console.error('Error updating case:', err.message);
            res.status(500).json({ error: 'Internal server error.' });
        } else if (this.changes === 0) {
            res.status(404).json({ error: 'Case not found.' });
        } else {
            res.json({ message: 'Case updated successfully.' });
        }
    });
});
```

18.4 Sample API Endpoint: Delete Case (Judge Dashboard)

File: backend.js

```
// Delete Case Endpoint (Judge Dashboard)
```

```
app.delete('/api/cases/:id', authenticateJWT, authorizeRoles('Judge'), (req, res) => {
    const { id } = req.params;

    const query = `DELETE FROM JUDGE_CASE WHERE id = ?`;
    const params = [id];

    db.run(query, params, function (err) {
        if (err) {
            console.error('Error deleting case:', err.message);
            res.status(500).json({ error: 'Internal server error.' });
        } else if (this.changes === 0) {
            res.status(404).json({ error: 'Case not found.' });
        } else {
            res.json({ message: 'Case deleted successfully.' });
        }
    });
});
```

19 Security Measures

19.1 Authentication and Authorization

- **JWT Authentication**: Users must log in with valid credentials to receive a JWT, which must be included in the `Authorization` header of subsequent requests.
- **Role-Based Access Control (RBAC)**: Defines permissions based on user roles to restrict access to certain endpoints and functionalities.
- **Password Hashing**: Uses bcrypt to securely hash passwords before storing them in the database.
- **Token Expiration**: JWTs have an expiration time to minimize the risk of token theft.
- **Secure Storage**: Sensitive information such as JWT secrets and database paths are stored securely using environment variables.

19.2 Data Protection

- **HTTPS**: All data transmitted between the client and server is encrypted using HTTPS.
- **Input Sanitization**: All user inputs are sanitized and validated to prevent SQL injection and Cross-Site Scripting (XSS) attacks.
- **Data Encryption**: Sensitive data, especially biometric information, is encrypted at rest in the database.
- **Access Controls**: Implement strict access controls to ensure that only authorized users can access or modify sensitive data.

19.3 Rate Limiting and Throttling

- Implement rate limiting using middleware such as `express-rate-limit` to restrict the number of requests from a single IP address within a specific timeframe.
- Helps protect against brute-force attacks and denial-of-service (DoS) attacks.

19.4 Regular Security Audits

- Conduct regular security audits and penetration testing to identify and fix vulnerabilities.
- Use tools like OWASP ZAP or Snyk to scan for security issues in dependencies and codebase.
- Maintain a security log to monitor and track security-related events.

19.5 Input Validation and Sanitization

- Validate all incoming data on both client and server sides using libraries like ‘validator.js‘.
- Ensure that data conforms to expected formats and constraints.
- Reject or sanitize any inputs that contain malicious content.

19.6 Data Encryption

- Encrypt sensitive data such as personal information and biometrics using secure encryption algorithms.
- Manage encryption keys securely, ensuring they are not exposed or hard-coded.
- Use environment variables to store encryption keys and secrets.

20 User Roles and Permissions

20.1 Roles

- **Admin_Judge**: Administer Judges and manage related data.
- **Admin_Police**: Administer Police Officers and manage related data.
- **Admin_Superintendent**: Administer Superintendents and manage related data.
- **Admin_CBI**: Administer CBI Officers and manage related data.
- **Judge**: Manage cases and oversee judicial processes.
- **Police Officer**: Register criminals, assign jails, and manage criminal data.
- **Superintendent**: Comprehensive criminal management including biometrics and health records.
- **CBI Officer**: File case reports and manage evidence.

20.2 Permissions

- **Admins**:
  - Create, update, and delete user accounts.
  - Assign roles to users to control access levels.
  - Access and modify all data related to their respective roles.
  - Oversee and manage database entries to ensure data integrity.
- **Judges**:
  - Manage and oversee case details.
  - Assign judges to cases and schedule hearings.
  - Access detailed criminal information relevant to cases.
  - Update case statuses and track progress.
- **Police Officers**:
  - Register criminals and record crimes.
  - Assign jails and transfer criminals.
  - Manage criminal location details.
  - Update criminal statuses.
- **Superintendents**:

- Comprehensive criminal management.
- Collect and update biometric data.
- Maintain location and health records.
- Assign works or tasks to criminals.

- **CBI Officers**:
  - File case reports.
  - Manage and add evidence.
  - Access detailed case information and associated evidence.

21 Conclusion

The **Criminal Record Management System** is a robust and secure solution designed to meet the diverse needs of various administrative roles within a law enforcement context. By implementing Role-Based Access Control, secure authentication mechanisms, and comprehensive data management practices, the system ensures efficient and secure operations. This document serves as a foundation for the development, deployment, and maintenance of the system, ensuring clarity and alignment among all technical stakeholders.

For further assistance or inquiries, please contact the development team at support@yourcompany.com.

22 Appendices

22.1 Appendix A: Sample Database Queries

Example: Registering a New User

```
INSERT INTO USERS (username, password, email, role)
VALUES ('newuser', '$2b$10$hashedpassword', 'newuser@example.com', 'Police Officer');
```

Example: Fetching Criminal Information

```
SELECT * FROM SUPERINTENDENT_CRIMINAL
WHERE criminal_id = 'C123';
```

22.2 Appendix B: Sample Environment Variables

File: `.env`

```
PORT=3000
JWT_SECRET=your_jwt_secret_key
GROQ_API_KEY=your_groq_api_key
DATABASE_PATH=./criminal_dashboard.db
```

22.3 Appendix C: Sample API Request and Response

Endpoint: `/api/login`

Request:

```
{
  "username": "admin",
  "password": "admin123"
}
```

Response:

```
{
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."
}
```

22.4 Appendix D: Code Snippets

Example: Middleware for Authentication and Authorization

```
// authMiddleware.js
const jwt = require('jsonwebtoken');
const dotenv = require('dotenv');

dotenv.config();

const authenticateJWT = (req, res, next) => {
    const authHeader = req.headers.authorization;

    if (authHeader) {
        const token = authHeader.split(' ')[1];

        jwt.verify(token, process.env.JWT_SECRET, (err, user) => {
            if (err) {
                return res.sendStatus(403); // Invalid token
            }

            req.user = user;
            next();
        });
    } else {
        res.sendStatus(401); // No token provided
    }
};

const authorizeRoles = (...allowedRoles) => {
    return (req, res, next) => {
        if (!allowedRoles.includes(req.user.role)) {
            return res.sendStatus(403); // Forbidden
        }
        next();
    };
};

module.exports = { authenticateJWT, authorizeRoles };
```

22.5 Appendix E: Deployment Checklist

- ✓ Install Node.js and npm.
- ✓ Install SQLite3.
- ✓ Clone or transfer project files to the server.
- ✓ Install project dependencies using ‘npm install‘.
- ✓ Configure environment variables in the ‘.env‘ file.
- ✓ Initialize the database using ‘node init_db.js‘.
- ✓ Start the backend server using ‘node backend.js‘.
- ✓ Set up a web server (e.g., Nginx) to serve the frontend.
- ✓ Configure HTTPS with SSL certificates.
- ✓ Implement monitoring and logging.
- ✓ Schedule regular backups.