**Project Documentation**

The Heart Track Application is a low-cost IoT-enabled web application designed to monitor and visualize heart rate and blood oxygen (SpO2) data in real-time. The system integrates a heart rate sensor connected to a microcontroller (Particle Photon), which sends data to a cloud-hosted backend. This data is stored in a MongoDB database and visualized on a web-based user interface, allowing users to monitor their health effectively.

**Account Creation and Management**

**Backend Implementation**

User Schema:  
The User schema in MongoDB is implemented using Mongoose and contains the following fields:

* Endpoints:  
  RESTful APIs are used for account management:
  + POST /users/register: Creates a new user account, hashes the password using bcrypt, and registers the provided device.
  + POST /users/login: Authenticates the user by validating their email and password.
  + PUT /users/update: Allows users to update account information (except email).
  + POST /users/add-device: Adds a new device to the user's account.
  + DELETE /users/remove-device: Removes a specific device from the user's account.
* Token-Based Authentication:
  + JWT (JSON Web Token) is used to secure endpoints and manage user sessions.
  + Tokens are issued during login and verified in protected routes to ensure only authenticated users can access their data.

**Frontend Implementation**

* Account Creation Page (createaccount.html):
  + The user inputs an email, password, and device ID to create an account.
  + Password strength validation ensures secure inputs.
* Login Page (login.html):
  + Users log in using their credentials.
  + A password visibility toggle is implemented for user convenience.
* Device Management:
  + Users can add or remove devices from their account.
  + Buttons for managing devices trigger AJAX requests to the backend.

**Embedded Device Implementation**

* Device Registration:
  + Users manually input the device ID (coreid) during account creation.
  + The device ID is later used to authenticate and associate incoming data with the correct user.

**Server**

Backend Implementation

The backend server is implemented using Node.js and Express.js with the following features:

1. RESTful API Design
   * Endpoints adhere to RESTful conventions. Examples include:
     + POST /data: Accepts IoT sensor data from the Particle device.
     + GET /data/device/:coreid: Fetches data for a specific device.
     + GET /data/device/:coreid/summary: Provides a weekly summary of average, minimum, and maximum heart rate values.
2. API Key Validation
   * The server requires an API Key to authenticate incoming POST requests from IoT devices.
3. Database
   * MongoDB is used to store user accounts and sensor data.

**Web Application**

Frontend Implementation

The web interface is developed using HTML, CSS, and JavaScript with Chart.js for graph visualization.

1. Login and Logout:
   * Users can log in securely. A logout button redirects them to the login page.
2. Real-Time Visualization:
   * Daily View: Plots the most recent 10 heart rate (AvgBPM) and blood oxygen (SpO2) values as line charts.
   * Weekly Summary: Displays average, minimum, and maximum AvgBPM for the past 7 days in a bar chart.
3. Dropdown Selection:
   * Users can switch between different chart views (AvgBPM, SpO2, Weekly Summary) using a dropdown selector.
4. Responsive Design:
   * CSS media queries ensure that the web application is accessible on both mobile and desktop devices.

**Heart Track IoT Device**

Embedded Device Implementation

The IoT component uses a Particle Photon microcontroller with a heart rate sensor.

* + - Measurement Prompt: RGB LED flashes blue to prompt the user to take a reading.
    - Data Recording: Reads heart rate (AvgBPM) and SpO2 values.
    - Wi-Fi Check:
      * Connected: Sends data to the server immediately and flashes the LED green on success.
      * Not Connected: Locally stores data for up to 24 hours and flashes yellow.

1. Data Transmission
   * Data is sent in JSON format to the backend via HTTP POST requests.
2. Accuracy Optimization
   * First measurements are inaccurate, so the system will gather a couple of measurements before displaying readings for accurate measures.

**Files description**

1.**Public Directory**

* index.html  
  This is the landing page for the application. It provides an overview of the project, a "Meet the Team" section, and an embedded video showcasing the Heart Rate Monitor project. It includes navigation to the login page.
* login.html  
  Provides a login interface where users enter their email and password to access the application.
* createaccount.html  
  Allows users to create an account. Users input their email, password, and device ID. Password strength validation is implemented for security.
* myaccount.html  
  The main user dashboard that displays:
  + Heart Rate (AvgBPM) over time.
  + Blood Oxygen (SpO2) over time.
  + Weekly summary charts with average, minimum, and maximum heart rate values.  
    A drop down allows users to toggle between the three charts.
* CSS Files:
  + index.css: Styling for the index and team pages with responsive design for both mobile and desktop.
  + login.css: Styles the login and create account pages.
  + myaccount.css: Styles the user dashboard and charts for responsive display.

**2. Routes Directory**

The routes folder contains all server-side route definitions.

* data.js  
  Handles incoming IoT data from devices and provides user-specific and summary endpoints:
  + POST /data: Saves IoT sensor data (AvgBPM, SpO2, coreid, timestamp) to the database.
  + GET /data/device/:coreid: Fetches the most recent 10 data points for a specific device.
  + GET /data/device/:coreid/summary: Calculates the average, minimum, and maximum values of heart rate for the past 7 days.
* users.js  
  Handles user account management endpoints:
  + POST /users/register: Allows users to create an account with email, password, and device.
  + POST /users/login: Authenticates users with their credentials.
  + POST /users/add-device: Adds a new device to the user's account.
  + DELETE /users/delete: Deletes a user by email.
  + GET /users/readAll: Retrieves all users for admin management.

**3. Models Directory**

The models folder defines MongoDB schemas using Mongoose.

data.js  
Defines the schema for IoT data sent by the devices. Fields include:  
Javascript

user.js  
Defines the schema for user accounts with fields for email, password, and registered devices:  
javascript

**4. JavaScript Directory**

The javascripts folder contains client-side scripts for frontend interactivity and AJAX calls.

* createaccount.js  
  Handles the account creation logic:
  + Collects user input (email, password, device ID).
  + Validates password strength and sends the data to the server using AJAX.
* myaccount.js  
  Fetches and visualizes data for the logged-in user:
  + Heart Rate (AvgBPM) and SpO2 data plotted using Chart.js.
  + Weekly summary displayed with bar charts showing average, maximum, and minimum BPM values.
  + Includes real-time chart updates using WebSockets for new incoming data.
* login.js  
  Handles the user login process, verifies credentials, and redirects users to their account dashboard.

**5. Backend and Configuration Files**

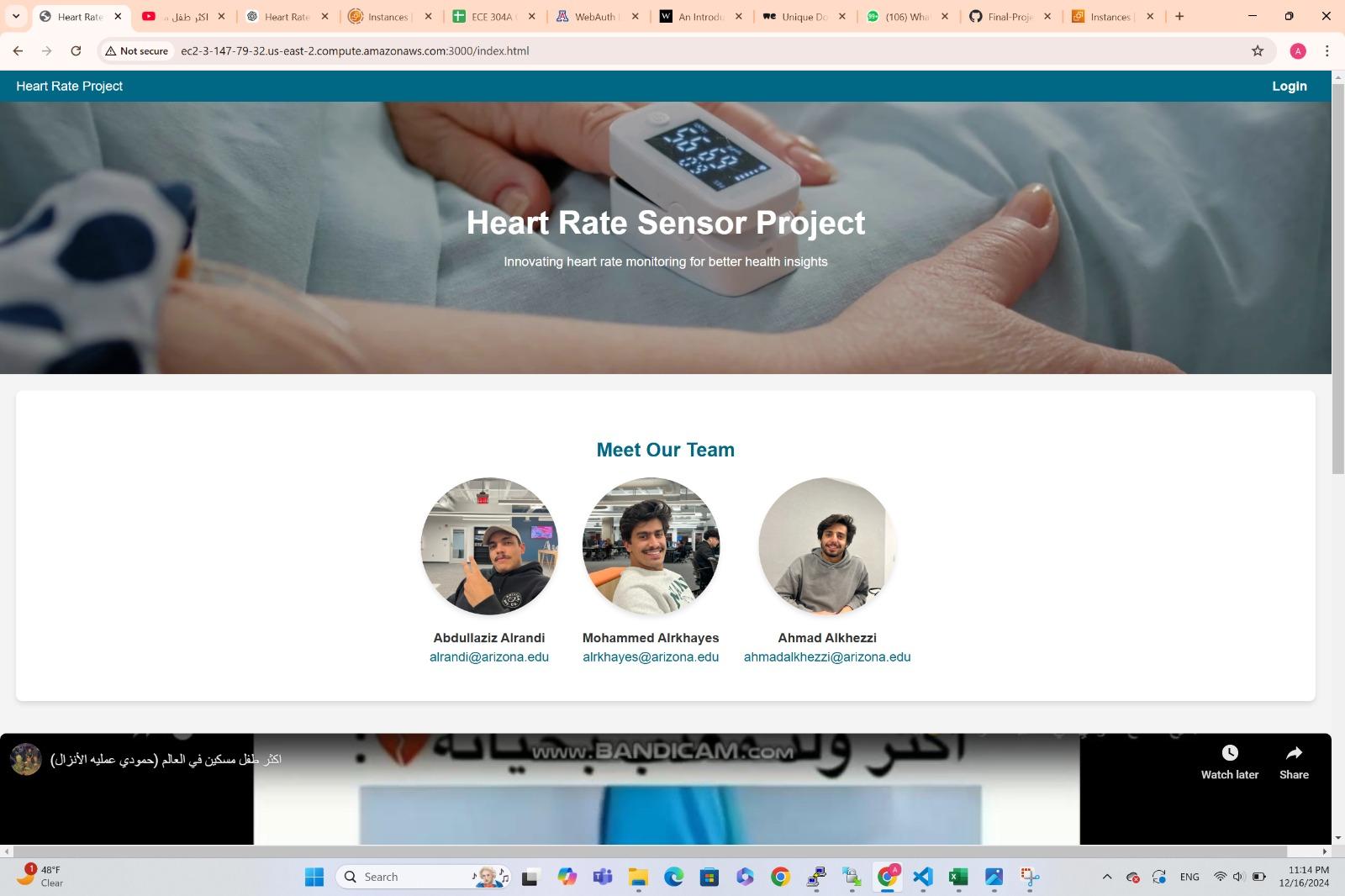
* app.js  
  The main entry point for the server:
  + Initializes the Express server.
  + Configures middlewares such as body-parser, CORS, and JSON parsing.
  + Imports and uses routes (data, users, admin).
  + Serves static files from the public directory.
* bin/www  
  Starts the server and listens on the specified port. Handles errors gracefully.
* db.js  
  Establishes the MongoDB database connection using Mongoose.

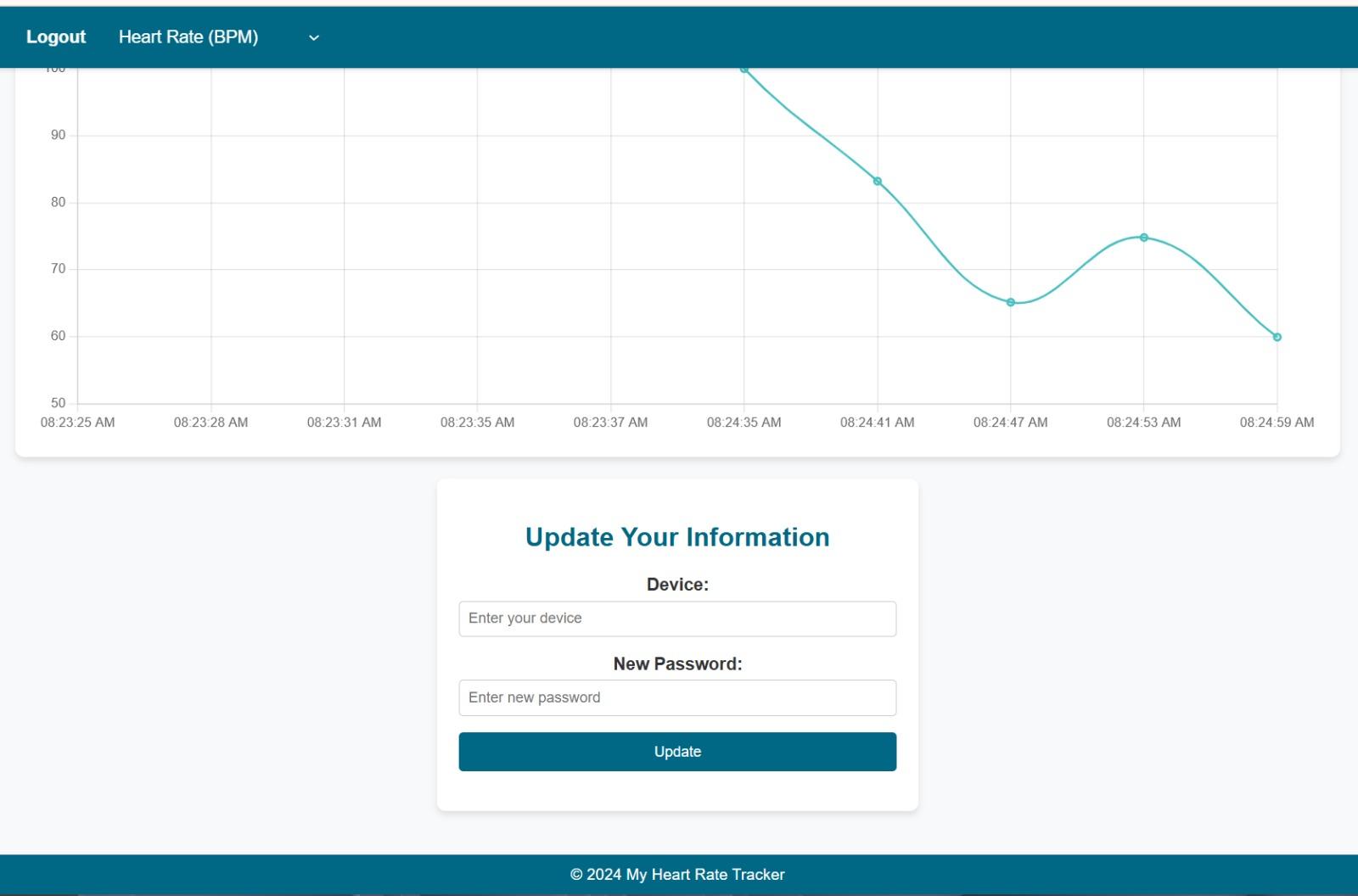
**6. Additional Files and Directories**

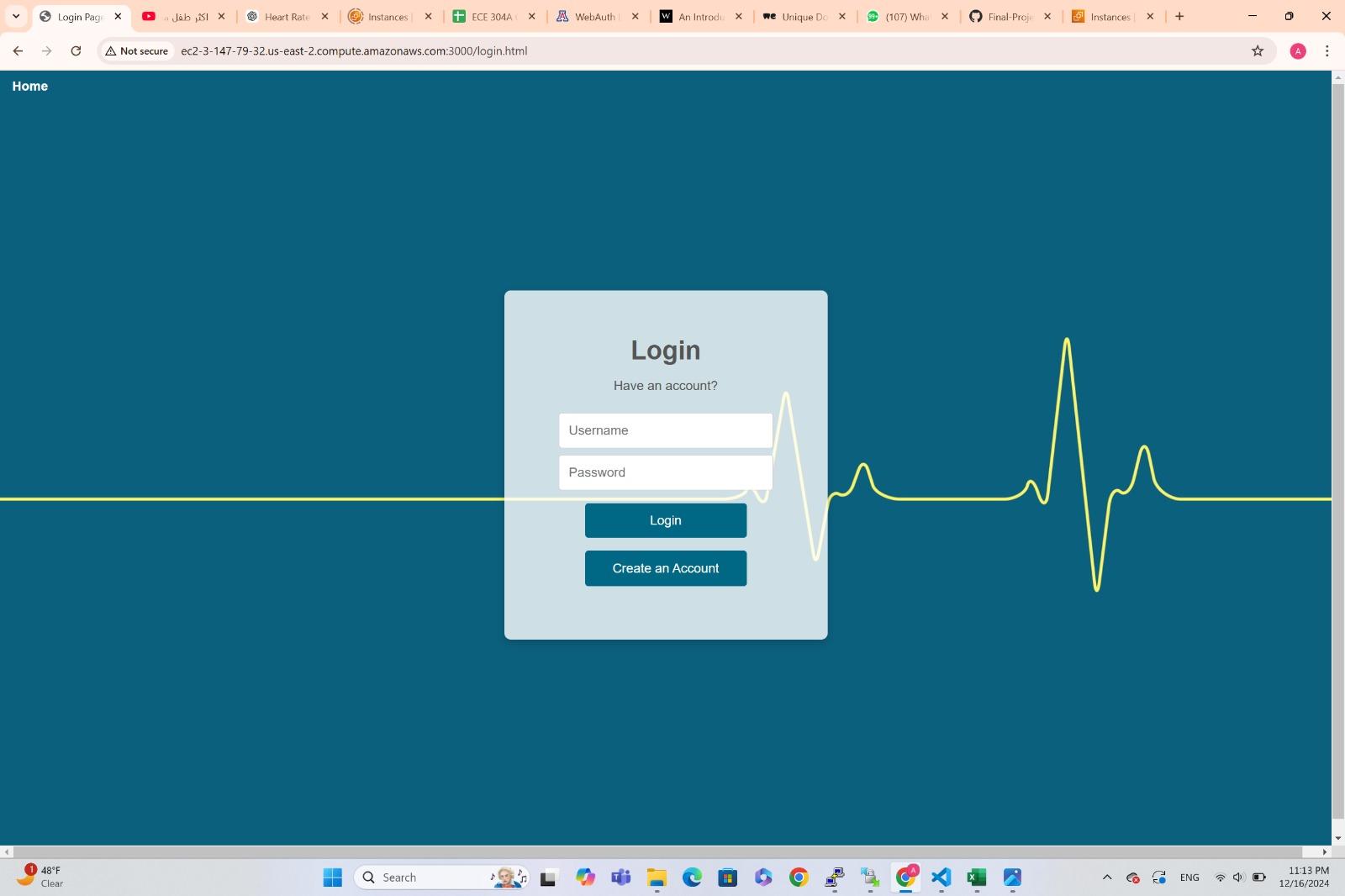
* package.json  
  Lists project dependencies like Express, Mongoose, bcrypt, and Chart.js.
* Images Directory  
  Contains team member profile pictures, project images, and background images used in the frontend.

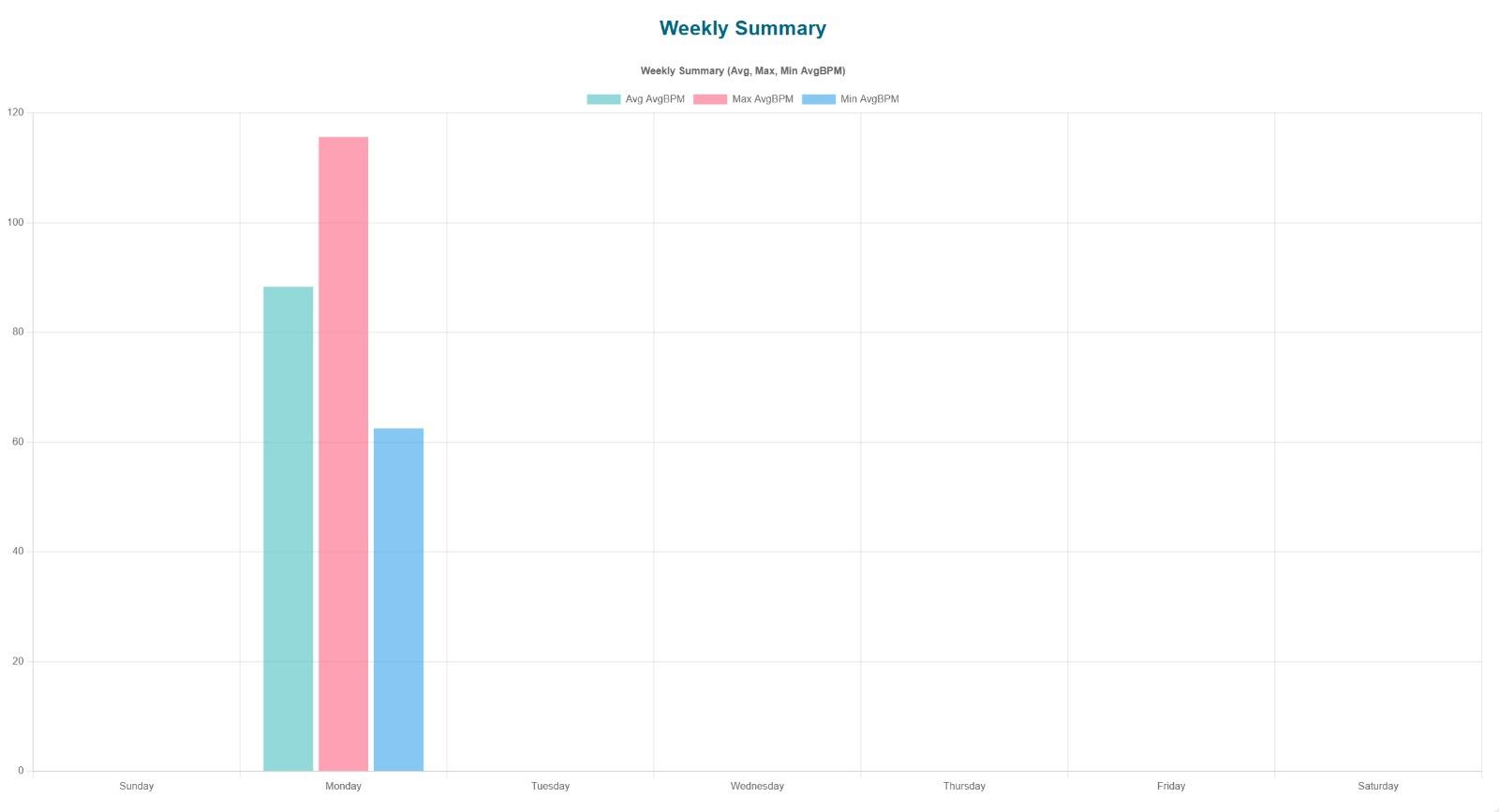
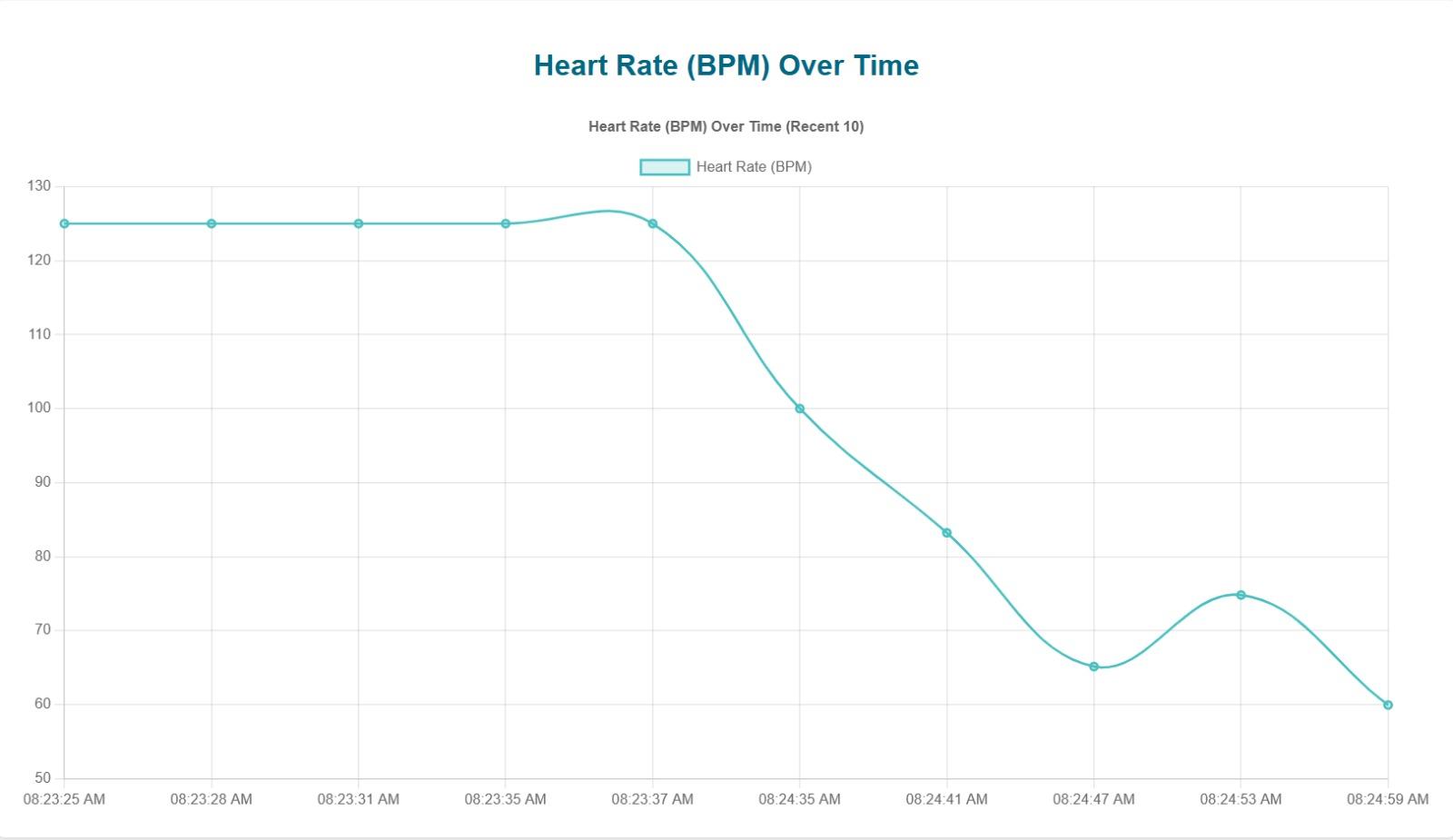
**Use of LLMs (Large Language Models) for secure implementation**

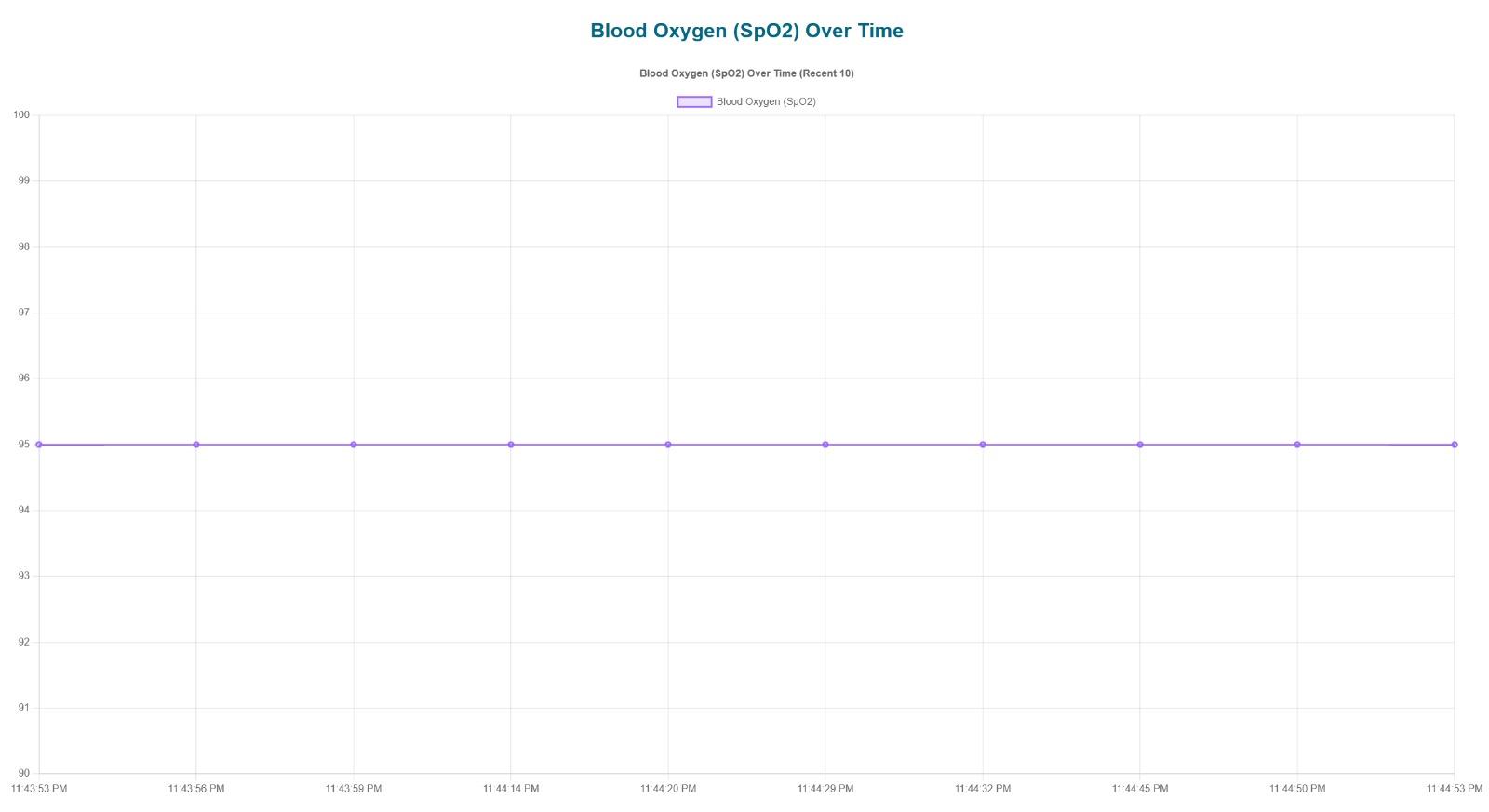
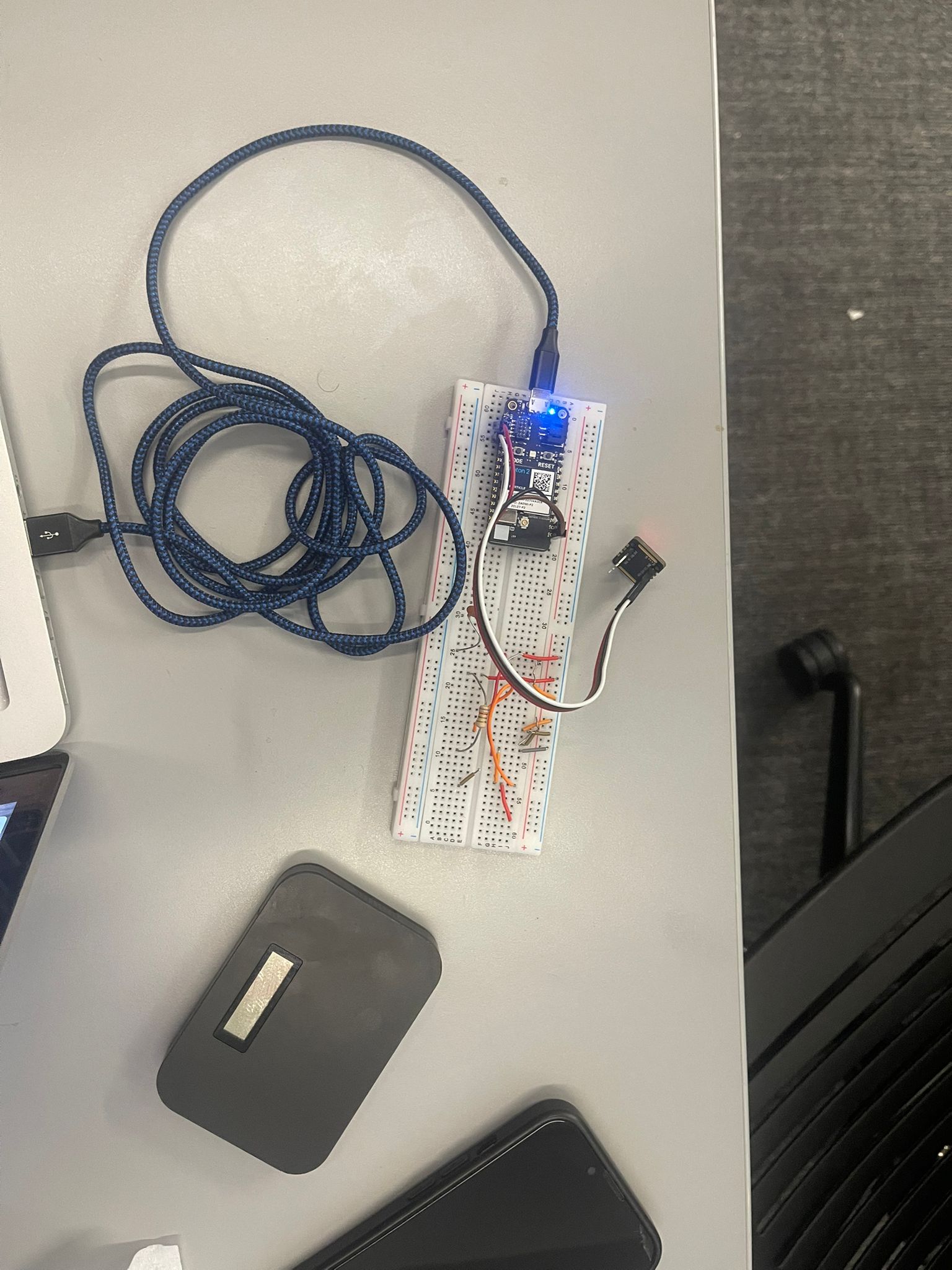
| **CWE-ID (Web Link)** | **Description** | **Domain** | **Detected** | **Mitigated** |
| --- | --- | --- | --- | --- |
| CWE-79 | Cross-site scripting (XSS): Input is improperly neutralized, allowing malicious scripts to be executed. | HTML, JS | Yes | Did not try mitigated code yet. |
| Vulnerabile Code: //student.js  $('#rxData').html(data.message); // Directly inserting unvalidated data  Here, data.message could contain a malicious script like <script>alert('Hacked!')</script> | | | | |
| Mitigated Code: const DOMPurify = require('dompurify'); // Use DOMPurify to sanitize user data  const cleanData = DOMPurify.sanitize(data.message);  $('#rxData').html(cleanData); // Insert sanitized input | | | | |
| CWE-20 | Improper input validation: Inputs like email, passwords, and API keys are not validated, leading to security risks. | JS, Node.js | Yes | Did not try mitigated code yet. |
| Vulnerabile Code: //student.js  const { email, password, device } = req.body;  if (!email || !password) {  return res.status(400).json({ message: "Fields are required" });  } | | | | |
| Mitigated Code:  const validator = require('validator');  const { email, password, device } = req.body;  if (!email || !password || !device) {  return res.status(400).json({ message: "All fields are required." });  }  if (!validator.isEmail(email)) {  return res.status(400).json({ message: "Invalid email format." });  }  const strongPasswordRegex = /^(?=.\*[A-Z])(?=.\*[a-z])(?=.\*\d)(?=.\*[!@#$%^&\*])[A-Za-z\d!@#$%^&\*]{8,}$/;  if (!strongPasswordRegex.test(password)) {  return res.status(400).json({ message: "Password must be at least 8 characters, include upper/lowercase letters, a number, and a special character." });  } | | | | |

**Results Section**









**Lessons Learned**

1. Full-Stack Development Integration  
   Implementing this project allowed us to understand how the frontend, backend, and database work together. We learned how to connect RESTful APIs with MongoDB to display real-time data on a web interface.
2. Data Visualization with Chart.js  
   We gained experience in visualizing real-time data using Chart.js, learning how to dynamically update line and bar charts with incoming data to make the results clear and user-friendly.
3. IoT Device Integration  
   Understanding how IoT devices communicate with servers using API Keys and HTTP requests was a major lesson. The implementation of state machines on the device side to handle Wi-Fi connectivity and data posting was valuable.
4. Security Best Practices  
   While working on account management, we learned the importance of secure password storage using bcrypt hashing and the need for API key validation to protect sensitive data.
5. Debugging and Problem-Solving  
   We developed a systematic approach to debugging code issues across different components, including asynchronous JavaScript issues in AJAX calls, data validation errors, and backend API failures.

**Challenges and Resolutions**

1. Managing Real-Time Data from IoT Devices  
   Challenge: Ensuring that real-time data from IoT devices is properly received and stored in MongoDB without duplication or data loss.  
   Resolution: Implemented API Key validation and careful database schema design to handle device coreid mapping efficiently.
2. Chart Responsiveness for Mobile and Desktop  
   Challenge: Making the Chart.js graphs responsive on both mobile and desktop screens.  
   Resolution: Used CSS media queries to adjust chart sizes and ensured that the container dynamically resizes based on the viewport width.
3. Backend Validation and Error Handling  
   Challenge: Ensuring valid inputs for the RESTful API endpoints and preventing crashes from unexpected data.  
   Resolution: Added comprehensive try-catch blocks in Express routes to handle invalid requests and return meaningful error messages.
4. Secure User Authentication  
   Challenge: Securely storing user passwords and implementing an efficient login mechanism.  
   Resolution: Used bcrypt to hash user passwords and added proper password comparison during login.
5. Device-User Matching  
   Challenge: Linking IoT device data to the correct user account using coreid and ensuring accurate retrieval of device-specific data.  
   Resolution: Matched the device ID (coreid) in the incoming IoT data with the device field in the user schema, enabling seamless data association.

**Team Member Contributions**

| Team Member | Frontend (%) | Backend (%) | IoT Device (%) | Documentation (%) | Overall Contribution (%) |
| --- | --- | --- | --- | --- | --- |
| Abdullaziz Alrandi | 10 | 10 | 70 | 10 | 100 |
| Mohammed Alrkhayes | 10 | 70 | 10 | 10 | 100 |
| Ahmad Alkhezzi | 70 | 10 | 10 | 10 | 100 |

**References**

Heart Rate and SpO2 Measurement GitHub Repository  
Kershrita. *"Heart Rate and SpO2 Measurement"*. GitHub Repository.<https://github.com/kershrita/Heart-Rate-and-SpO2-measurement>

MongoDB Mongoose Guide  
Mongoose Contributors. *"Mongoose ODM Documentation"*. https://mongoosejs.com/docs/

ChatGPT Assistance  
OpenAI. *"ChatGPT by OpenAI"*. AI-based text generation for guidance, code explanations, and content creation. [https://chat.openai.co](https://chat.openai.com)m