Project Documentation

ECE 413

Abdulwahab Al-Qenaei

Ahmad Malik

Rashed AlTammar

Table of contents:

1. backend, frontend, and embedded device implementations
2. file description
3. Results
4. Lessons Learned
5. Challenges
6. Team’s contribution
7. References
8. frontend, backend, and embedded device implementations

(add a broad explanation of the project)

* 1. Frontend:
* The front-end integration is an easy to navigate site that allows users to register, login, create a device, and view the device readings.
* The front end was implemented using HTML, CSS, and JavaScript files. These files work together to give the user a seamless experience while reading data from the device.
  1. Backend:
* The backend is all the work that takes place behind the stage. This includes the utilization of Putty, Amazon AWS, MongoDB, WinSCP, and Json. This allows the server to receive, store data from the device an user input, in addition to showing data from the device to the user. Moreover, all routing and saved data is accessed through the backend.
  1. Embedded device implementations:

The project utilized two embedded devices to achieve seamless data acquisition and transfer:

* The Argon Particle microcontroller transfers data from the sensors to the webpage. It acts as an intermediary, collecting data from the MAX30102 sensor and transmitting it to the backend server via a Wi-Fi connection. The device is programmed to ensure efficient communication and real-time updates to the server, enabling users to view device readings on the frontend.
* The MAX30102 is a sensor used for monitoring heart rate and oxygen levels. This device collects data through photoplethysmography technology. The Argon Particle processes the sensor's data and transmits it to the backend, where it is stored and made accessible for users to view on the frontend.

1. file description:

* HTML:
  + index.html: the main page of the project, which introduces the project and the team members.
  + signup.html: provides the interface for the sign-up page where the user creates their account.
  + login.html: provides the interface for the user to login into their personal account
  + references.html: Lists external resources, libraries, and tools used in the Heart Track project
  + deviceDetails.html: Displays detailed weekly and daily charts for heart rate and oxygen saturation using a date picker and dynamic Chart.js visualizations.
  + devices.html: Manages user devices by allowing them to add, view, and interact with a list of devices dynamically​.
  + settings.html: Allows users to configure measurement settings such as time range and frequency for heart rate data collection​.
* CSS:
  + style.css: This file contains a dedicated stylesheet used to define the visual appearance and layout of the web application. It controls the presentation layer of the front end, ensuring a cohesive and visually appealing design
  + devices.css: Styles the device management page with clean layouts, form elements, and buttons for adding, displaying, and interacting with devices.
* JavaScript:
  + signup.js: Facilitates user registration by validating inputs and creating a new user account through an API call.
  + login.js: Handles user login by validating credentials, generating a JWT, and redirecting users to the devices page.
  + devicedetails.js: Fetches and displays device-specific heart rate and oxygen saturation data, including weekly summaries and detailed daily charts.
  + devices.js: Manages user devices, including listing, adding, and deleting devices, with navigation to detailed device views.
* Middleware:
  + authenticatetoken.js: Middleware that verifies JWT tokens and ensures authenticated access to protected routes.
* Models
  + Device.js: Defines the Mongoose schema and model for IoT devices, storing readings, user preferences, and API keys.
  + User.js: Defines the Mongoose schema and model for users, including email, hashed passwords, and active JWT tokens.
* Routes
  + Auth.js: Handles user registration and login by hashing passwords, validating credentials, and issuing JWT tokens.
  + Devices.js: (Back-end): Provides API endpoints for managing user devices, updating preferences, and storing IoT sensor data.
* App.js: configures the Express server, integrates authentication and device routes, and handles errors.
* package.json: Specifies the project's metadata, dependencies (e.g., Express, bcrypt, JWT), and script configurations for running the Node.js application​.
* package-lock.json: Locks the exact versions of all installed Node.js dependencies to ensure consistency across different environments​
* final.ino: Contains the Arduino code responsible for reading and transmitting heart rate and oxygen saturation data from the sensor.

1. Results

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedThis is the index.html which is the home page

This is the references.html page where we listed any outside sources we used

A screenshot of a computer

Description automatically generatedThis is the login.html page where a user can login in to their account to access their data

A screenshot of a computer

Description automatically generatedThis is the signup page where a user can create an account

A screenshot of a computer

Description automatically generatedThis is what the users sees after they login in, they can register a new device and see their data

A graph with red lines

Description automatically generated

After entering the device the user can see the daily heart rate graph

A screen shot of a graph

Description automatically generatedThe user will also see the weekly heart rate graph and can also select what day they want

A screen shot of a graph

Description automatically generatedThe user can also see the daily blood oxygen saturation graph

1. Lessons Learned

* Working as a team really showed us how important it is to be on the same page and give updates on a timely manner.
* One thing we learned was time management and how to make sure that everyone has a clear idea of what they need to do.
* We learned how to efficiently store and receive user data and device data using MongoDB.
* We learned about RESTful API design and how to create well structured endpoints. Creating them also helped us understand the importance of them for maintainability of backends
* We also learned how to integrate the front and back end systems and how it requires careful planning to get consistent data formats and smooth communication via AJAX requests.

5.0) Challenges

* One challenge we faced was with storing user credentials securely and we resolved them by using password hashing with bcrypt.
* We also faced a challenge where we could not log in after creating a account and we fixed it by looking over the JavaScript and routing where we realized we had the wrong code.
* One challenge that was easily fixed was with the transfer of files. The naes of the files were different and we fixed it by sitting as a group and going over everything making sure the names were all correct.
* One challenge we faced early on was with installing mongodb on our AWS server. We tried to install it using putyy but everytime we wanted to run it we got an error. We resolved this issue by looking over the class slides and tutorial and following the exact steps which inturn made it work.
* One more challenge we faced was that the particle argon was not connecting to the internet. We resolved this issue by connecting the antenna to the device which inturn allowed us to connect to the internet.

6.0) Team’s contribution

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Frontend | Backend | Embedded device implementation | Documenataion |
| Ahmad Malik | 75% | 3% | 0% | 40% |
| Rashed AlTammar | 5% | 90% | 95% | 20% |
| Abdulwahab Al-Qenai | 20% | 7% | 5% | 40% |

7.0) References

* Chart.js - <https://www.chartjs.org/>
* Bootstrap CSS - <https://getbootstrap.com/>
* Particle Cloud - <https://docs.particle.io/>
* MAX30102 Heart Rate Sensor Library - <https://github.com/sparkfun/MAX30105_Breakout>
* Node.js - <https://nodejs.org/>
* Express.js - <https://expressjs.com/>
* Mongoose (MongoDB ODM) - <https://mongoosejs.com/>
* JSON Web Tokens (JWT) - <https://jwt.io/>
* Particle Webhooks - <https://docs.particle.io/webhooks/>
* Particle Photon/Argon - <https://store.particle.io/>

|  |  |  |
| --- | --- | --- |
| **CWE-ID (Web Link)** | **Description** | **Domain (HTML/CSS/JS/Firmware)** |
| CWE-20 | Improper Input Validation | signup.js |
| □ **Detected;** □ **Mitigated;** | | |
| * Please provide the code snippet including the vulnerability and explain your approach for mitigating the weakness.   if (email === "") {  window.alert("Invalid email!");  return;  }   * This only checks if the email field is empty, but **doesn't validate the format of the email**. * The server may also lack proper sanitization and validation. * If you are using LLM to do it, please provide the prompt and explain how you make LLM work. * Chatgpt prompt: Try to identify only one vulnerability in my codes based on the following list and suggest a mitigation method: CWE-352: Cross-Site Request Forgery (CSRF) When a web server is designed to receive a request from a client without any mechanism for verifying that it was intentionally sent, then it might be possible for an attacker to trick a client into making an unintentional request to the web server which will be treated as an authentic request. This can be done via a URL, image load, XMLHttpRequest, etc. and can result in exposure of data or unintended code execution. CWE-232: Improper Handling of Undefined Values The product does not handle or incorrectly handles when a value is not defined or supported for the associated parameter, field, or argument name. CWE-20: Improper Input Validation Input validation is a frequently-used technique for checking potentially dangerous inputs in order to ensure that the inputs are safe for processing within the code, or when communicating with other components. When software does not validate input properly, an attacker is able to craft the input in a form that is not expected by the rest of the application. This will lead to parts of the system receiving unintended input, which may result in altered control flow, arbitrary control of a resource, or arbitrary code execution. CWE-311: Missing Encryption of Sensitive Data The lack of proper data encryption passes up the guarantees of confidentiality, integrity, and accountability that properly implemented encryption conveys. CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting' - XSS) Cross-site scripting (XSS) involves injecting malicious scripts into web applications, leading to unauthorized execution of code in victims' browsers, potentially compromising their data and actions, with three main types: Reflected, Stored, and DOM-Based XSS. The product does not neutralize or incorrectly neutralizes user-controllable input before it is placed in output that is used as a web page that is served to other users. CWE-94: Improper Control of Generation of Code ('Code Injection') When a product allows a user's input to contain code syntax, it might be possible for an attacker to craft the code in such a way that it will alter the intended control flow of the product. Such an alteration could lead to arbitrary code execution. * Above is what I asked chatgpt to do, along with this prompt I included our JS files so it can view them and detect the problems * Mitigation: * Use regular expressions or input patterns to ensure correct formats. * Example for validating an email:   const emailPattern = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;  if (!emailPattern.test(email)) {  alert("Invalid email format!");  return;  }  if (password.length < 6) {  alert("Password must be at least 6 characters.");  return;  } | | |