Robert Tkaczyk, Ryan Raad, Emma Halferty

ECE 413: Internet of Things

12/16/2024

Project Documentation

Project Implementation:

Front End:

The frontend of the project contains the HTML and CSS we developed for the web application. The purpose of the front end exists as a way for the user to interact with their data and alter account information as they pleased. We utilized built-in functionality within HTML to manipulate our data to interface to the user. However, we did have to import an external library (Chart.JS) to implement our graphs for the user. For most of the user interaction we utilized “inputs” to extract the values the user would input. Other than inputs, we also used password fields for our login and signup features. Another feature we employed was the button nodes to link to our backend for functionality when signing up, logging in, etc. Our HTML featured a main navigation bar that let the user navigate around our HTML pages. One thing to note in our navigation bar was that we had a feature that would hide the “login” tab when the user was login and then display a new “account” tab that is not present when initially logging in. This account display featured a main portion of our functionality, allowing the user to manipulate details in their account (excluding email change). All of our navigation tabs were linked using “a href’s” to allow the user to switch between HTML’s. The account page’s manipulation functionalities all linked to other HTML that allowed the user to change information at their will. Our index HTML contained our team information displayed according to the rubric, with a short project description. Our signup page included the user entering an email or username and a password. The password had to be “strong” and had requirements for the user to incorporate in their password for added security. Finally we had a reference HTML which simply showed some external resources we utilized to implement our project. The frontend of the project was not too complex and most of the functionality and implementation lies in the connection between our HTML and JS where the actual functionality is carried out. Next, we will discuss how we implemented this functionality in our backend.

Backend:

To begin, we set up our server using Amazon's server-side tools (AWS). We followed this setup according to guides already given to us earlier in the semester by our TA and also ZyBooks. Our app.js file within our server had to be modified to include our routes, models, and to enable cross origin access. Once we established our AWS, we installed MongoDB on its console so we could store account information and sensor data in a separate database. We installed Mongo according to guides given to us earlier in the semester by our TA. To connect to Mongo, we had a simple "db.js" file that connected to our local database using mongoose. Another thing to note before we continue is our account schema for our Mongo. This is found in our models folder and includes the schema for an account. Our account schema had a username, a password, devices, and an array of data. Our index and reference HTML did not have any specific functionality, thus we will not discuss them in this section. Moving to our signup HTML, we implemented this in our JS by routing the valid credentials to our "accounts" route. This route carries out all of our functionality and handles our connections to mongo and will be discussed often as we move forward. The route for signup included checking if an account had existed using the credentials the user had inputted. The route would then add an account if none had existed, or return errors if the account had existed. An important thing to note is that the password the user had inputted would be encrypted using a one way hash and be stored in this form. This was to ensure we were not storing passwords as text and added security to our page. THis was also a key requirement in the project, and we had to implement it to move forward. Next, we implemented our "login" javascript, which was linked to our login HTML. This would route to our accounts route and check if the credentials were valid by comparing the encrypted passwords together and ensuring it was correct. This login functionality would create a token for the user and store it in localStorage for continued use of the website (such as accessing the charts). The login functionality would also alter HTML elements in the navigation bar to remove the login tab and add the account tab. Next we will cover the "accountjava" javascript, which includes functionality for manipulating information in a user's account. This javascript carried out four main functions: adding devices, removing devices, changing passwords, and logging out. Adding and removing devices were implemented in the accounts route, and would add or remove devices according to the user's input to the mongo collection associated with their account. The change password function would access the users account information and alter their existing password a newly inputted and encrypted password. Finally, the logout feature included removing the users token and hiding the account tab and redisplaying the login tab. Next we will move to our dashboard javascript which implements our charts and data for the user to observe. To enter the dashboard page, the user had to be logged in and hold a token in their localStorage. This enforced token based authorization for our web application portion of the project. The dashboard javascript had two main functionalities, which contained a weekly view and a daily view. The data for both these functions were accessed with an API call that

would link to our accounts route and store any data that was not a duplicate. The weekly view simply called the accounts route and parsed through the users data to return data that was only within the last seven days. The route would take this data and calculate an average, maximum, and minimum heart rate and display it for the user (according to the rubric, no blood oxygen saturation had to be displayed for this weekly section). Our daily view was a little more complicated and featured a user interaction where the user could choose a day to view their data from. The chosen data would be sent to our accounts route where it would parse through the users data and only return data relevant to the date they selected. This data would be graphed using Charts.JS libraries to display both graphs that were noted in the rubric. This section excludes any implementation of the device, and that information will be covered next.

Embedded Device:

The embedded device aspect of the project encompasses our argon particle device and our MAX3010 heart rate and blood oxygen level sensor. We collect data from the sensor via the particle device code, which is then sent through a webhook (we used ThingSpeak) to the cloud, where it can be accessed by our javascript files and uploaded to our Mongo database. The first thing we did was setup our particle device; this can be done online in a web browser or through an app. Once our particle device was setup, we downloaded Particle CLI, which includes the drivers for our device. Using an integrated Particle terminal in Visual Studios code, we made sure our drivers and firmware for our device were up to date and that we were ready to begin coding. Next we initiated the sensor through our files heartRate.cpp, heartRate.h, MAX30105.cpp, MAX30105.h, spo2_algorithm.cpp, and spo2_alogrithm.h. The file heartRate.cpp processes data from our MAX30105 sensor to detect heartbeats by analyzing changes in infrared signal patterns. The MAX30105.cpp file provides a library for configuring and using the MAX30105 optical smoke and heart rate detection sensor, enabling communication via I2C to control settings, read sensor data, and manage interrupts. Finally, the spo2_algorithm.cpp file contains an algorithm for calculating heart rate and SpO2 levels by processing infrared and red sensor data, identifying signal peaks, and deriving metrics based on the AC and DC components of the signals. These files are all combined in our final embedded device file called Argon.ino (which generates its own .cpp automatically as well). The Argon.ino file serves as the main control program that integrates all the other files and functionality to create a working heart rate and SpO2 monitoring system using the MAX30105 sensor and a Particle Argon device. The device will publish the data to ThingSpeak and flash a green LED on the device if the data was published successfully. The device only works between the hours of 6am to 10pm in the Arizona time zone, but this can be adjusted by the user. This device also implements a user reminder to collect data by flashing a blue LED on the particle device every 30 mins since the last reading.

<u>File Description:</u>

Our project folder contains the following folders (with sub files):

To reduce some redundancy it should be noted that each HTML file contains a navigation section to implement our navigation bar. This functionality uses "a hrefs" to link to other HTML pages.

-app: Includes our app.js file used for our AWS

- app.js: This file was modified to enable our cross origin access for mongodb. Along with this, it sets up our AWS server to run correctly. It also was modified to include the use of our new "accounts" route and to include our "account" schema.

-html: Includes all of our HTML pages

- account.html: This HTML is the display for our account page where the user may alter information of their account. This HTML contains buttons for adding a device, removing a device, changing a password, or logging out.
- adddevice.html: This HTML is used for when the user clicks on the add device button on the account html. It links to a new page where the user may enter a device name and id to add to their account.
- changepassword.html: This HTML is used for when the user clicks on the change password button on the account html. It links to a new page where the user may enter a new password according to the password criteria.
- dashboard.html: This HTML is used for displaying weekly and daily data for the user. The user may select a date to display for the daily data, and corresponding graphs are generated. The weekly data is displayed when entering the html and only displays the previous seven days data. It should be noted access to this html is controlled with token based authorization, ensuring the user must be logged in to view this html.
- index.html: This HTML includes the display for our team information and a short description of the project.
- login.html: This HTML includes fields for the user to enter a valid username and password to gain access to the web application. The login html displays when a successful or unsuccessful login is carried out and updates the navigation bar accordingly.
- references.html: This HTML includes a short section of references and resources we utilized to finish the project
- removedevice.html: This HTML is used for when the user clicks on the remove device button on the account html. It links to a new page where the user may enter a device name to remove from their account.
- signup.html: This HTML includes two fields where the user may enter a username and password to sign up. The password criteria is listed below these fields to ensure the user enters a "strong" password.

-IoT device

- Argon.ino: This file serves as the main control program for our argon particle device and sensor to create our full functioning heart rate and SpO2 monitor. It sends all the successful data collection to our webhook (ThingSpeak). It also implements user measurement reminder every 30 mins by flashing a blue LED since the last measurement. It flashes the LED green when the data has been published successfully to ThingSpeak. It also implements the time the device will run, which is 6am-10pm (AZ time zone).
- Argon.cpp: This file is automatically generated in the project folder when running Argon.ino.
- heartRate.cpp: This file encompasses all the heart rate sensor initiation and setup. It processes data from our MAX30105 sensor to detect heartbeats by analyzing changes in infrared signal patterns.
- heartRate.h: The header file for our heartRate.cpp file: declares the functions, classes, and variables used in heartRate.cpp.
- MAX30105.cpp: This file provides the library for controlling the MAX30105 sensor. It handles communication, configuration, and data acquisition from the sensor.
- Max30105.h: The header file for our MAX30105.cpp file: declares the functions, classes, and variables used in MAX30105.cpp.
- spo2_algorithm.cpp: Implements the algorithm for calculating heart rate and SpO2 levels from the raw IR and red LED sensor data.
- spo2_algorithm.h: The header file for our spo2_algorithm.cpp file: declares the functions, classes, and variables used in spo2_algorithm.cpp.

-javascripts

- accountjava.js: This javascript file contains the post requests and functionality for buttons displayed on the account html page. This file either connects the user to other html's based on the buttons the user may input, or calls the routes folder to manipulate account information by creating POST requests.
- adddevice.js: This javascript file contains the functionality for the adddevice html page and creates a POST request to the routes folder to add a device to the users account
- changepassword.js: This javascript file contains the functionality for the changepassword html page and creates a POST request to the routes folder to change a password for the user as long as criteria for a new password is met.
- dashboard.js: This javascript file contains the functionality for the dashboard html page. This file calls the thingspeak API to retrieve recent data from the sensor. This data is passed to the routes folder and submitted as long as it is new or unique data. This file also calls the routes folder for weekly and daily views by filtering the users account based on the week or selected date and returns data for the dashboard.js file to use to graph and display the data.
- index.js: This javascript file simply contains functionality to hide or display

certain tabs on the navigation bar based on if a user is logged in or not (login or account tabs)
- references.js: This javascript file simply contains functionality to hide or display certain tabs on the navigation bar based on if a user is logged in or not (login or account tabs)
- removedevice.js: This javascript file contains the functionality for the removedevice html page and creates a POST request to the routes folder to remove a device from the users account
- signup.js: This javascript file contains the functionality for the signup html page and creates a POST request to the routes folder to create a new account model based on the users input and also check if this user has been created before.

-models
- account.js: This javascript file defines our mongo schema for our account. It includes a username, password, devices, and a data section

-mongo
- db.js: This javascript file connects to our mongo database using mongoose methods

-routes
- users.js: This route was established upon creating the AWS, no modifications were made
- index.js: This route was established upon creating the AWS, no modifications were made
- accounts.js: This route was created to handle all account requests. This route includes functionality for logging in, signing up, changing password, adding devices, removing devices, submitting data, displaying weekly data and displaying daily data. This route file is utilized to manipulate the mongo database based on the users account and returns data and tokens for the user.

-stylesheets
- styles.css: This CSS file defines the style of our webpage using ids and classes to manipulate HTML elements on our web application.

Results:

HTML/JS Results:

- Team Info
- References
- Signup
- Dashboard
- Account

Welcome to Our Project

Meet our team and learn about our innovative web application!

Our team has built a Heart Track application for monitoring heart rate and blood oxygen saturation level throughout the day of a user. We utilize a IoT device that connects to our web server to convey data of a user's health in a graph format. Our web application allows a user to define the time of day range and frequency at which measurements will be requested. We utilize a weekly and daily summary view for our measurements to allow a user to have a more general view (weekly) of their health measurements, or a more specific view (daily). We combine HTML, JS, and CSS concepts we learned in class to create the website for our web application. Along with this, we implemented a MongoDB database to store measurements and account information. We hope you enjoy our project and stay healthy!

Member 1: Robert Tkaczyk

Email: roberttkaczyk@arizona.edu

A simple and fun loving guy who loves to take walks and cook great food! Ready to graduate college and take on the big world :D

Member 2: Emma Halferty

Email: emmahalferty@arizona.edu

Team Info: Showing our team descriptions

References

Third-party API's:

Thingspeak API

Webhook API

Libraries:

Bcrypt Library

JWT Library

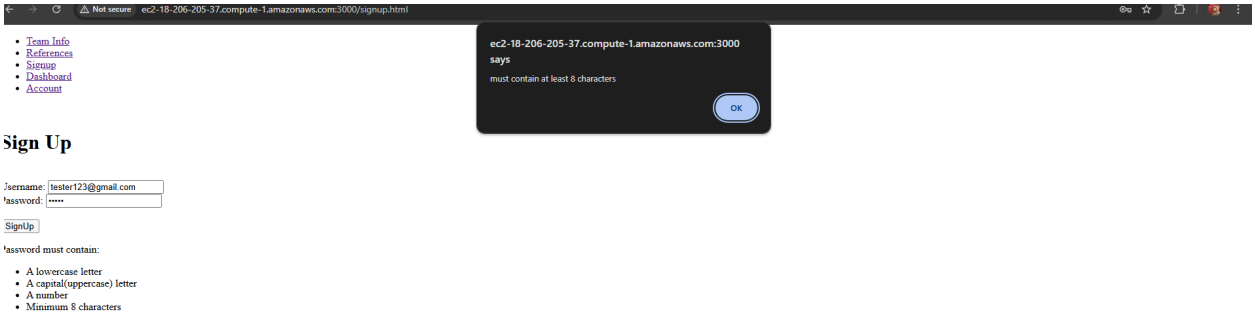
Chart JS Library

Code:

413 Example Code: MongoDB_Activites_Source-Code

413 Example Code: Authentication_Activities_Source-Code

References: Showing our references



Signup page: Failed signup with not strong password

Signup page: successful sign up for a user

- Team Info
- References
- Login
- Signup
- Dashboard

Log-In

Username: test@gmail.co
Password: ••••••••

Log In
login failed, please try again

Login page: Login failed, not valid

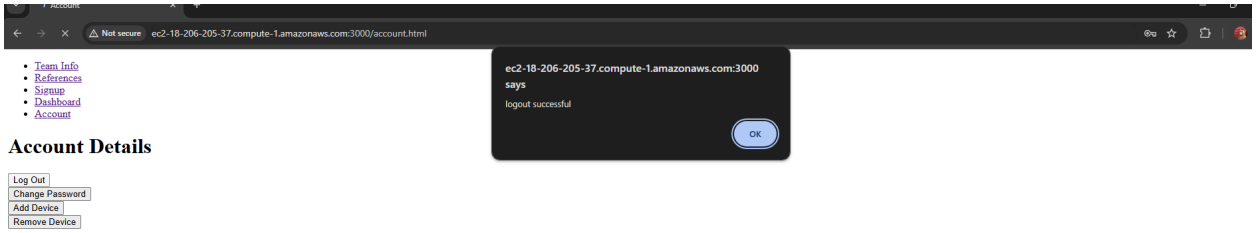
- Account

Log-In

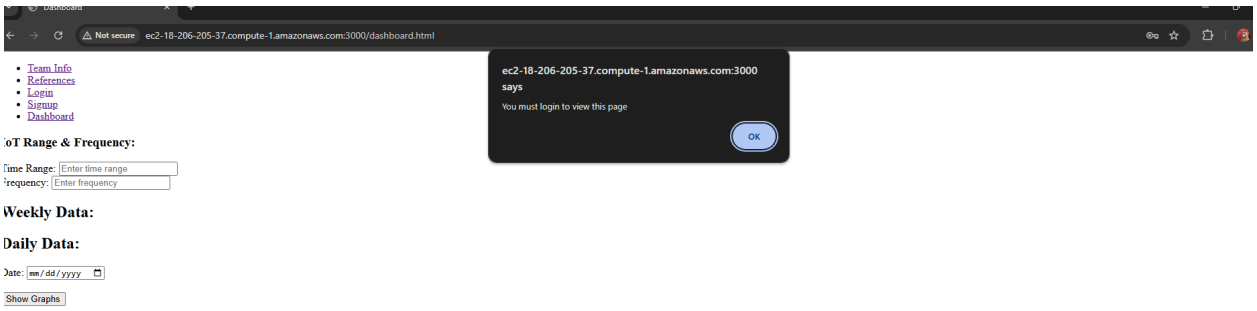
Username: tester@gmail.com
Password: •••••••••

Log In
Login success

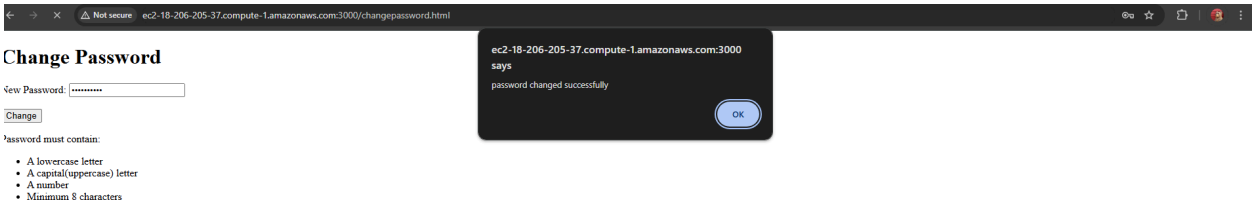
Login page: Login success, user enters



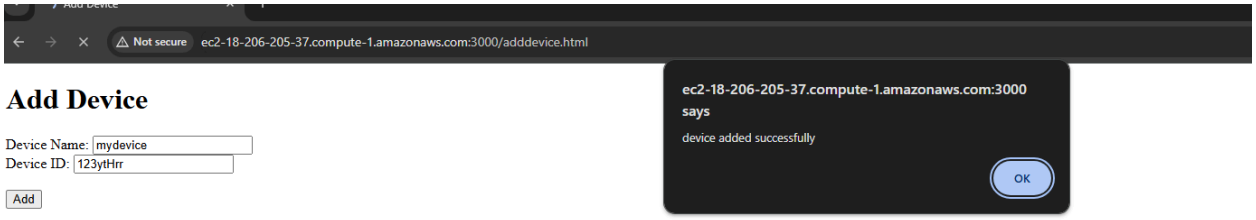
Account page: successful logout



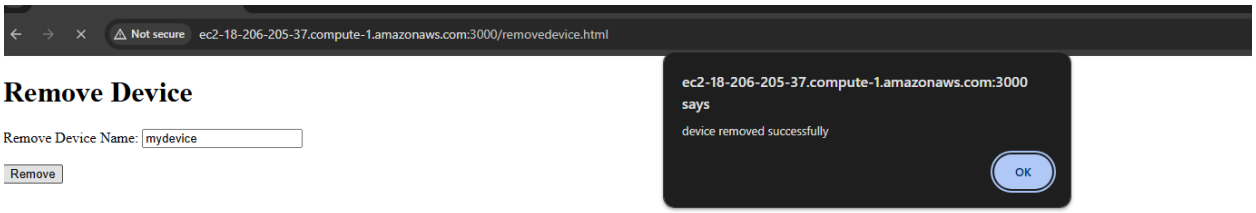
Dashboard page: user not logged in (token based authorization), rejection of entering dashboard page



Changepassword: user changes password successfully



Add device page: successfully added a device



Remove device page: successfully removed a device

- Team Info
- References
- Signup
- Dashboard
- Account

IoT Range & Frequency:

Time Range: Enter time range
Frequency: Enter frequency

Weekly Data:

Average Heart Rate: 133

Minimum Heart Rate: 28

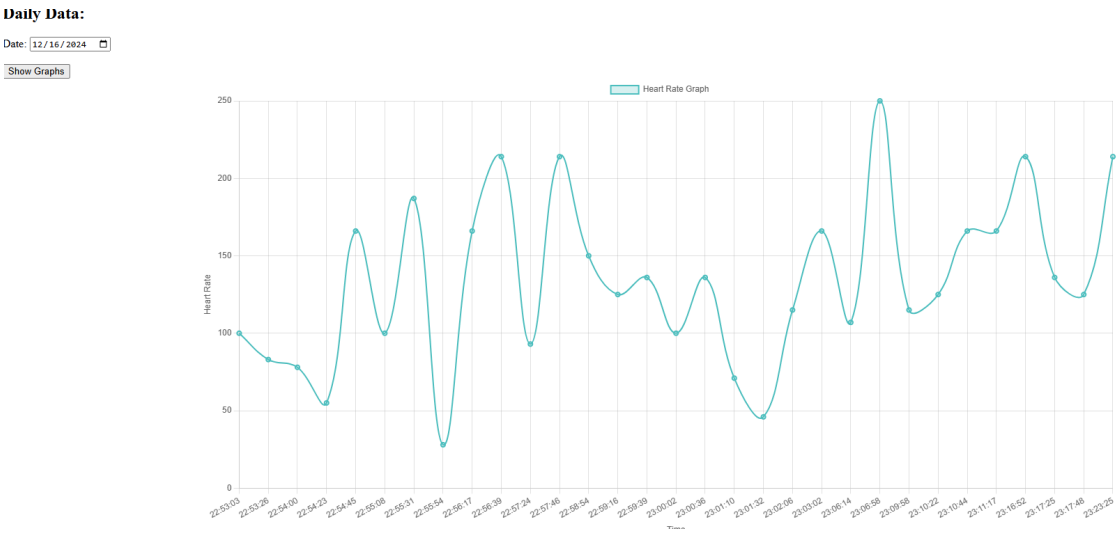
Maximum Heart Rate: 250

Daily Data:

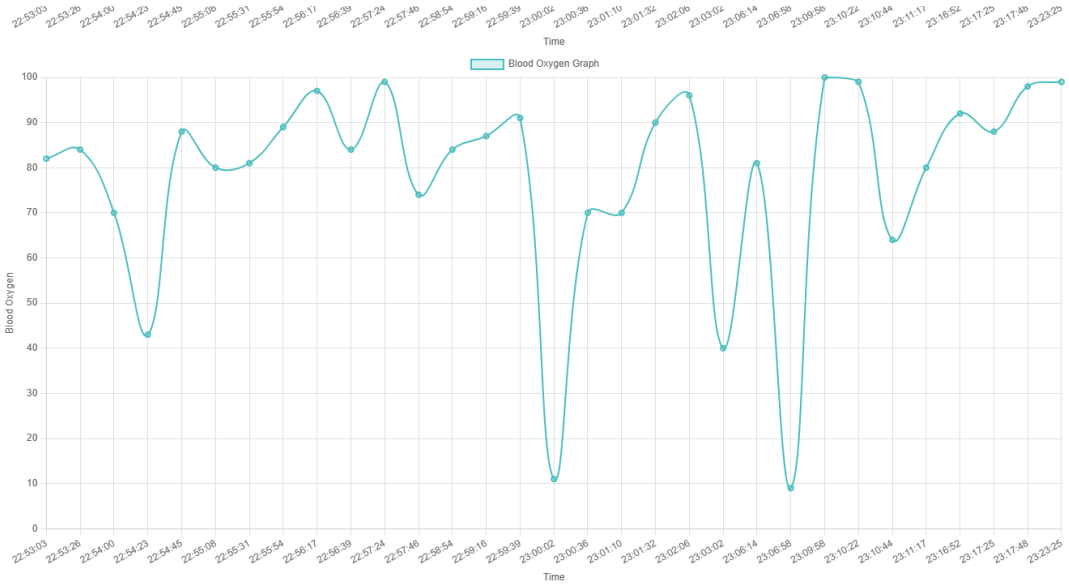
Date: mm/dd/yyyy

Show Graphs

Dashboard page: weekly date shown successfully



Dashboard page: successfully date input for heart rate graph

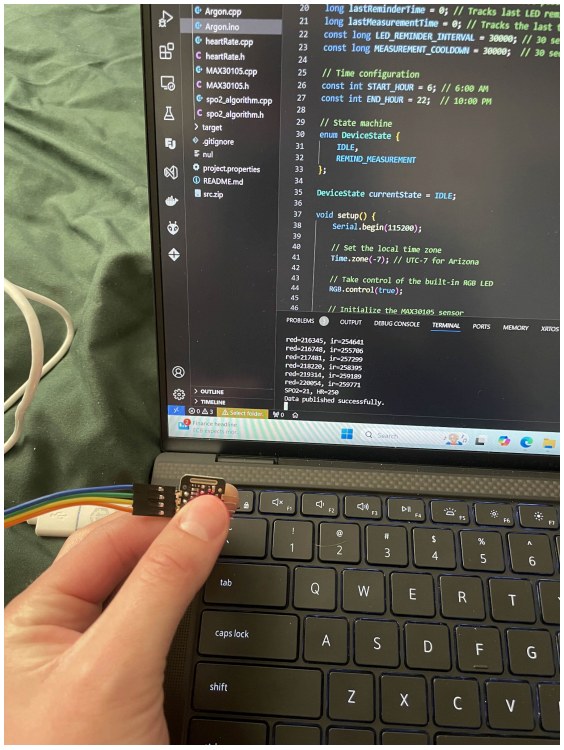


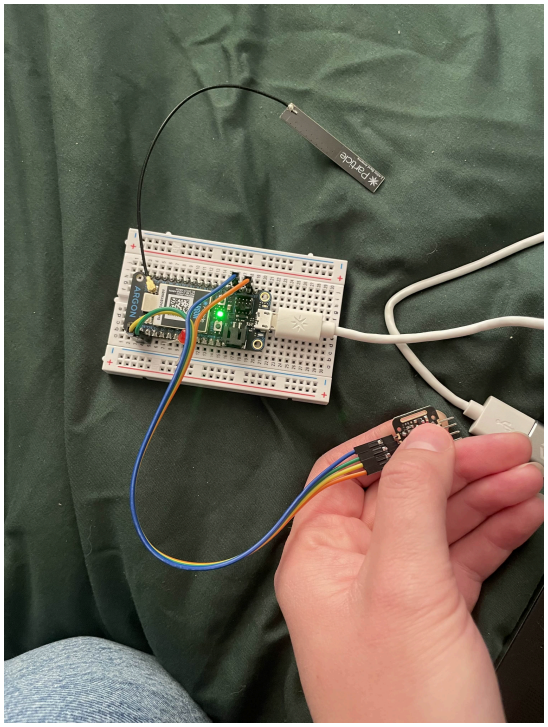
Dashboard page: successfully date input for blood oxygen graph

Embedded Device Results:

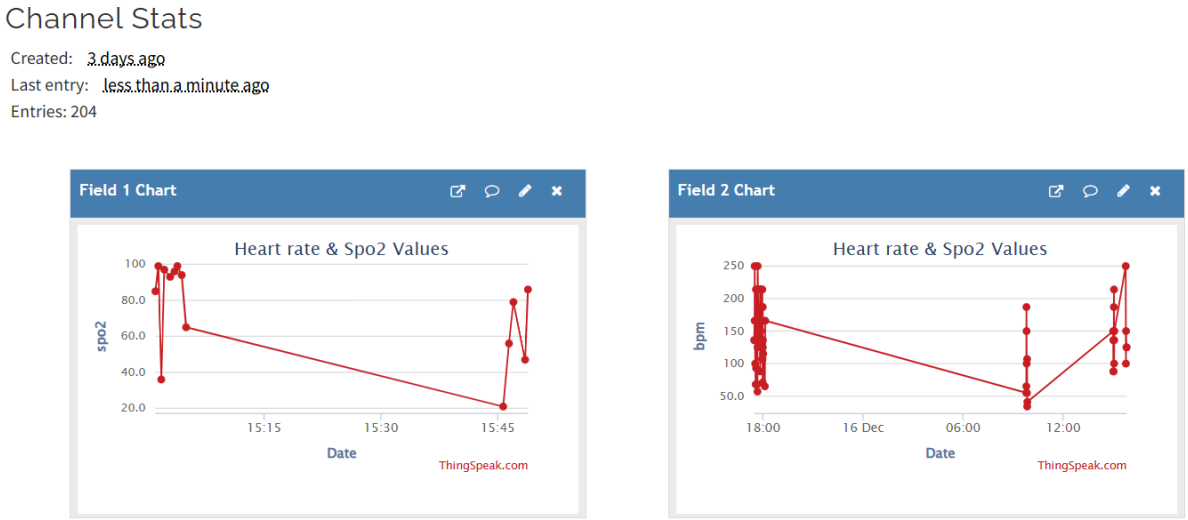
In our project, we successfully implemented an embedded system using the Argon Particle board and the MAX3010 sensor to monitor heart rate and SpO2 levels. The collected data is sent in real-time to a ThingSpeak webhook, enabling efficient cloud-based visualization and analysis. We included screenshots of the ThingSpeak dashboard, displaying the live heart rate and SpO2 measurements transmitted from the embedded device. Key functionalities include seamless data capture from the MAX3010 sensor every 10 seconds, reliable data transmission using the webhook, and dynamic

visualizations of the recorded metrics through ThingSpeak, as well as our html webpage.





Serial console displaying that the data collected was successfully sent to our WebHook and our particle device flashing a green LED to show the data was successfully sent.



The sensor heart rate and spo2 values uploading successfully to ThingSpeak, our webhook.

Lessons Learned:

- The first thing we learned through this class and project was how to connect and integrate front end and back end capabilities to make a website function correctly. By this I mean seamlessly combining HTML, CSS, and JS functionalities to ensure an ideal experience for a user who may be browsing our website. We had to learn how to modify HTML elements utilizing JS code so we could have an interactive website for our web application. This was an essential portion of our project and needed to be learned and understood to move forward.
- The next thing we learned was how to use token authentication to ensure only signed up users could access our web application. This was a tricky part for us to learn and took a little bit of time to understand before we had to implement it. We learned how to encrypt a password that the user had for their account and use this password and their email to generate a token that would grant access to the web application for the user. This was another essential part of the project, ensuring we had some security control over the web application, which is important for a website.
- The next thing we learned was how to use MongoDB to store our account information and the data we got from the sensor readings. We learned how to utilize CRUD operations to add and change accounts for the user, and also to make sure login information was valid when a user was logging in. This was another portion that took us a while to understand. We utilized the class lecture notes to help guide us through setting up mongo, and coding routes to be able to store data in it.
- The next thing we learned was how to use Git Repo to store all of our files and code for the project. This was an important thing for us to learn as Git Repo is commonly used outside of academic purposes as a way to store and monitor code and be able to commit changes with ease. This was also needed for the project so we could upload all of our code to the Git Repo (according to the rubric).
- One of the last things we learned was how to set up our IoT device to be able to read data and then send that data to our database for our web application purposes. Obviously this is the main part of the project, and also a main part of the class, thus we had to undergo some research and go over lectures to make sure we understood the IoT implementation. We had a lot of trouble setting up our IoT device and had to go to office hours to learn how to use it and connect it to our particle accounts. This was by far the greatest lesson we learned, as embedded devices are very common in the real world and this definitely helped us grasp some understanding of how they work.

Challenges Faced:

- The main problem we encountered was establishing a connection with the IoT device to our particle account and our vs code (i.e. flashing). We had this problem for like two weeks, and we simply could not connect our IoT device and have it configured properly. We finally solved this issue after coming into office hours on 12/9/2024 and 12/11/2024 and telling the TA what our issue was. The TA walked us through and guided us through setting up the IoT device and we were finally able to configure and flash the IoT device correctly, which was incredibly important for our web application.
- Another problem we encountered was connecting to our mongo database and submitting account information for login capabilities. We had a lot of issues with POST requests successfully submitting our information and connecting to the mongo database when attempting our sign up functionalities. We continued to get unauthorized access issues and even not found HTTP responses. We solved this issue by closely consulting the mongo source code provided in the course d2l page. After a lengthy review of the source code, we tweaked our code and were finally able to connect to our mongo database successfully and implemented our signup and login functionalities.
- Another problem was trying to figure out how to use Git Repo. None of our project team members were familiar with using Git Repo, so we all had to learn the platform from scratch. To resolve this, we spent a small portion of time researching Git Repo tutorials and even consulting the resources given in the project description document. Because of this, we had a good enough understanding of Git to be able to satisfy the project requirements.
- Another problem we had was that the initial sensor we used was faulty and was not functioning properly. After coming into office hours and having problems with our sensor, the TA recommended we try to use another sensor to locate where the main problem was. We switched the sensor with another sensor that came with our purchase of the IoT device and it ended up fixing the issues we had with reading data from our sensors. This was a frustrating section of our project to overcome as it was not a personal error and just an error with the actual hardware we were using.
- The last problem we had was problems with actually working as a team. Initially we all had very busy schedules considering senior design and other class responsibilities. Because of this we could not find a lot of time at the beginning to meet and felt as if we were falling behind in the project. This became easier towards the end of the semester and was resolved naturally as we all collectively began to finish our classes. To add on to this, we also had issues with where we were working on our codes. Our AWS server was set up on a single person's computer and we found it was difficult to push changes and alter code when all of our directories were located on this one team member's computer. We remedied this by making more of our meetings on discord so we could screen share and work together even when not in person with the team.

Extra Credit LLM Secure Implementation:

CWE-ID (Web Link)	Description	Domain (HTML/CSS/JS/Firmware)
CWE-352	A CSRF attack is a type of security vulnerability where an attacker tricks a user into performing actions on a web application without their knowledge or consent. They are able to do this by taking advantage of the fact that there is no authentication in the routes. So if a user is already authenticated and clicks on a link containing a CSRF attack, the request will be completed because the server takes the malicious request as a user request and processes it.	HTML, JS
Detected; Mitigated;		

- Please provide the code snippet including the vulnerability and explain your approach for mitigating the weakness.

In our account.js file we have routes for actions like adding devices, removing devices, changing password, etc. They are vulnerable to CSRF attacks because they do not include any mechanism to verify the authenticity of the request.

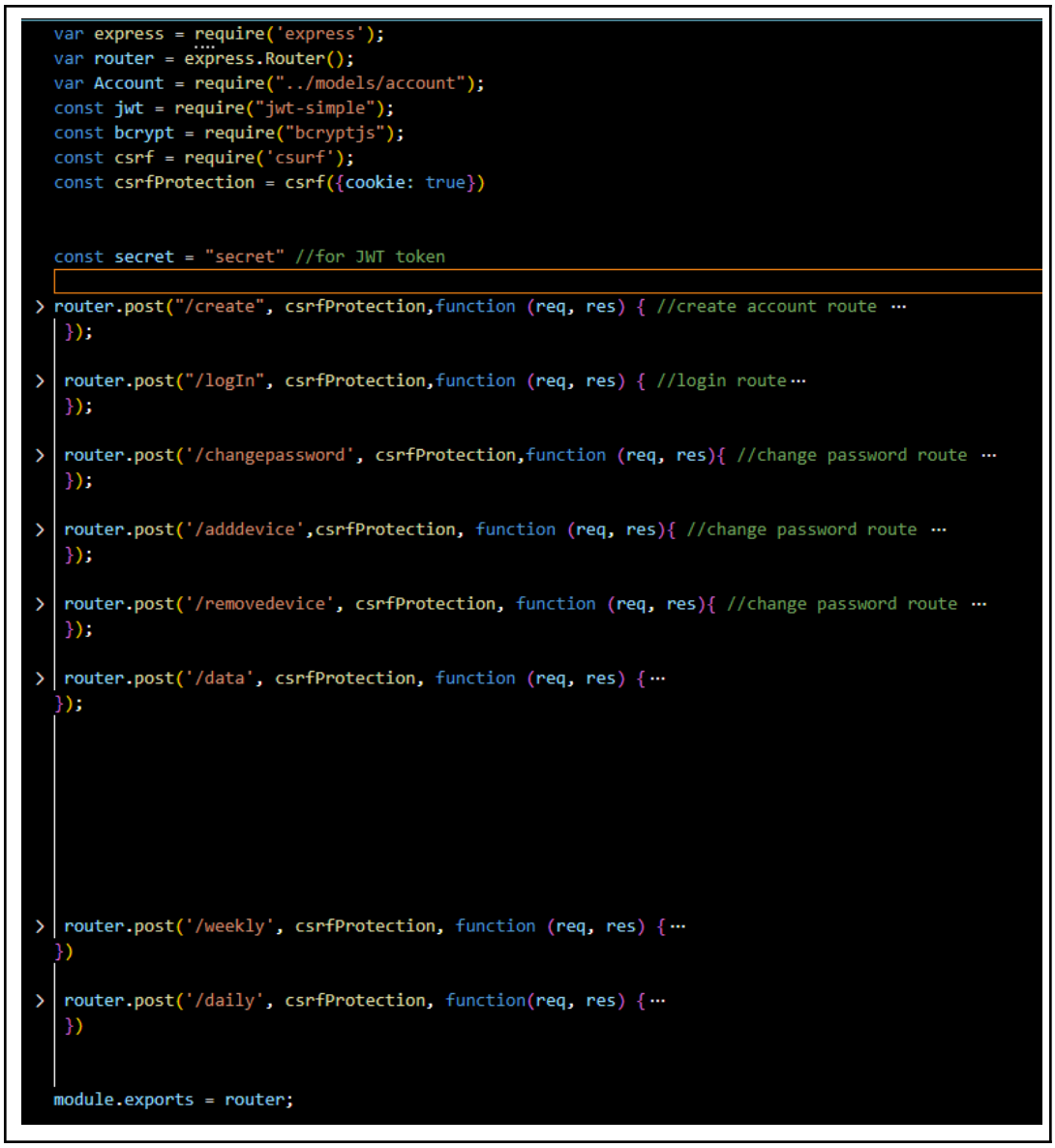
Vulnerable Code Snippet:

```
const secret = "secret" //for JWT token

router.post("/create", function (req, res) { //create account route ···
});

router.post("/logIn", function (req, res) { //login route···
});

router.post('/changepassword', function (req, res){ //change password route ···
});

router.post('/adddevice', function (req, res){ //change password route ···
});

router.post('/removedevice', function (req, res){ //change password route ···
});

router.post('/data', function (req, res) {···
})

router.post('/weekly', function (req, res) {···
})

router.post('/daily', function(req, res) {···
})
```

Mitigation Approach:
To mitigate CSRF vulnerabilities, we decided to implement CSRF tokens. These tokens ensure that the request is coming from an authenticated user and not from a malicious source.

Screenshots of CSRF Approach:

```
var express = require('express');
var router = express.Router();
var Account = require("../models/account");
const jwt = require("jwt-simple");
const bcrypt = require("bcryptjs");
const csrf = require('csurf');
const csrfProtection = csrf({cookie: true})
```



```
var express = require('express');
var router = express.Router();
var Account = require("../models/account");
const jwt = require("jwt-simple");
const bcrypt = require("bcryptjs");
const csrf = require('csurf');
const csrfProtection = csrf({cookie: true})


const secret = "secret" //for JWT token

router.post("/create", csrfProtection,function (req, res) { //create account route ···
});

router.post("/logIn", csrfProtection,function (req, res) { //login route···
});

router.post('/changepassword', csrfProtection,function (req, res){ //change password route ···
});

router.post('/adddevice',csrfProtection, function (req, res){ //change password route ···
});

router.post('/removedevice', csrfProtection, function (req, res){ //change password route ···
});

router.post('/data', csrfProtection, function (req, res) {···
});









router.post('/weekly', csrfProtection, function (req, res) {···
})

router.post('/daily', csrfProtection, function(req, res) {···
})


module.exports = router;
```

Example Mitigation Code

1. **Install CSRF Protection Middleware:**

```
npm install csurf
```

2. **Update Your Code to Use CSRF Tokens:**

```
const csrf = require('csurf');
const csrfProtection = csrf({ cookie: true });

// Apply CSRF protection to routes
router.post('/adddevice', csrfProtection, function (req, res) {
    // ... existing code ...
});

router.post('/removedevice', csrfProtection, function (req, res) {
    // ... existing code ...
});

router.post('/changepassword', csrfProtection, function (req, res) {
    // ... existing code ...
});
```

3. **Include CSRF Token in Forms:** In your HTML forms, include the CSRF token as a hidden input field.

```
<form action="/adddevice" method="POST">
    <input type="hidden" name="_csrf" value="<%= csrfToken %>">
    <!-- other form fields -->
</form>
```

- If you are using LLM to do it, please provide the prompt and explain how you make LLM work.

An initial prompt was sent to give the LLM context of the situation. From there I sent a string of other prompts with our code base, attached as files.

Prompt:

“TASK:
You have to identify 1 CWE and mitigate it. Detect and mitigate the CWEs within my HTML, CSS, and JavaScript codebase.
You can refer to the below websites for more information:
Detailed description of vulnerabilities (CVEs): https://nvd.nist.gov/vuln
Detailed description of weaknesses (CWEs): https://cwe.mitre.org/
Identify weaknesses, such as code injection, that have a high severity score according to the CVEs in your project, find the CWEs related to them, and mitigate them (Please refer to Appendix A at the end of this prompt).

Appendix A:
CWE-ID
CWE Description
Domain
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.
HTML, JS
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.
JS
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.
JS, Firmware
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.
JS, Firmware
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.
HTML, CSS
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.
JS, Firmware

I want you to look for one of the CWEs in the code I sent you. In the following prompts I will send you parts of our current code base. You do not need to respond to this prompt.”

Team Contribution:

	Components			
Team Members	*Front end*	*Back end*	*Device implementation*	*Documentation /videos/etc*
Robert Tkaczyk	33.33 %	33.33 %	33.33 %	33.33 %
Ryan Raad	33.33 %	33.33 %	33.33 %	33.33 %
Emma Halferty	33.33 %	33.33 %	33.33 %	33.33 %
Total:	**100%**	**100%**	**100%**	**100%**

References:

- Thingspeak API
- Webhook API
- Bcrypt Library
- JWT Library
- Chart JS Library
- 413 Example Code: MongoDB_Activities_Source-Code
- 413 Example Code: Authentication_Activities_Source-Code
- For heartRate.cpp, spo2_algoritm.cpp, and MAX30105.cpp files references from: sparkfun/SparkFun_MAX3010x_Sensor_Library: An Arduino Library for the MAX3015 particle sensor and MAX30102 Pulse Ox sensor