Milestone 5 Report

Engicoders
H. Bird, B. Karacelik, J. Peters, J. Ropotar, A. Rybka
Department of Computer Science, University of British Columbia
COSC 310: Software Engineering
Dr. Gema Rodriguez-Perez
April 12th, 2024

Project

The project is an IoT system to query, cleanse, filter, visualize, alert, and analyze manufacturing data. The project will function to empower engineers with a web application for the visualization and encapsulation of sensor data from IoT devices. Additionally, the system will use machine learning techniques to draw insight into the future states of sensors.

Project Description

The project is meant to take IoT data based on user preferences from an API. API data, of any variety, will be pulled from the IoT cloud platform “ThingSpeak” using its GET HTTP API. The data provided by the API can come in many forms such as pressure, thermometer, speedometer, height, and many more, as long as they are against time. The user demographic is manufacturing engineers seeking to utilize IoT data from sensors on a plant floor or that provide manufacturing data. This data is then cleansed, which includes a combination of filters and outlier removal. Following this Arima model, machine learning algorithms, based on user desires, are used to predict future data values and trends. Also, high and low-limit alarms with alarm management options can be added to the sensor to inform the user when the inputs go over, or under their desired value. The program will maintain an alarm log in conjunction with the alarm settings. Cleansed and analyzed data is displayed in three standard formats to provide insight into manufacturing operations. In addition to the main functionality, premium users will be able to import their data to cleanse, predict, alert, and visualize. Finally, the visualizations and alarm logs may be exported and shared by premium users to empower manufacturing engineers.

Overview Video

Youtube: https://youtu.be/S5huluvRdBY

Use-Case Diagram

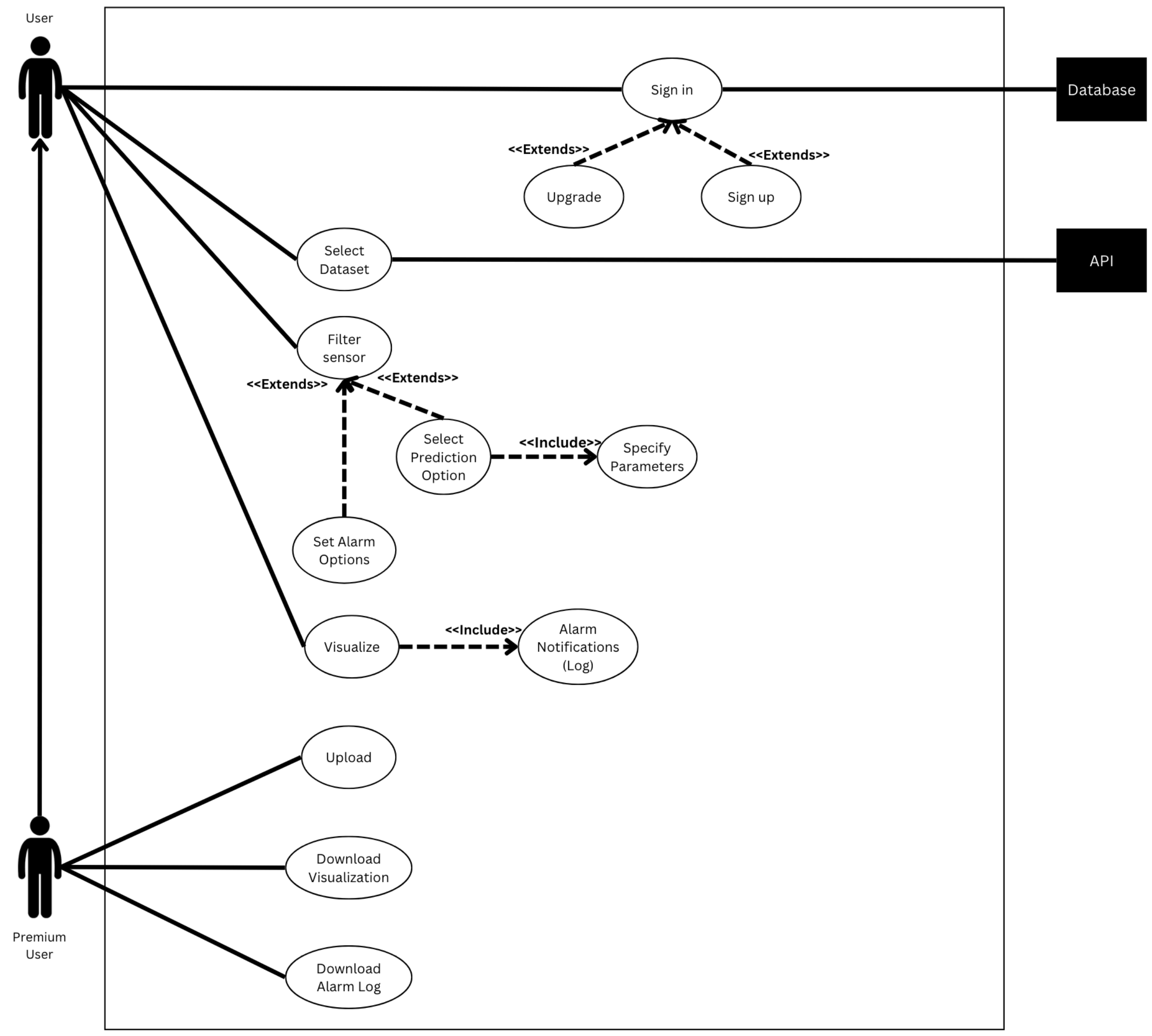


Figure 1. Visualize Data Use-Case Diagram

Use Case: Visualize a Dataset
Primary Actor: User
Description: Describe the process of how a user visualizes a dataset.
Precondition: The user must have successfully accessed the website.
Postcondition: If successful, the user will have a visualization of the data that they chose.

<u>Main scenario:</u>

1. The user signs into their account.
2. The user's credentials are verified.

3. The user selects a dataset to use from ThingSpeak with the channel ID, field ID, start date, end date, and increments in minutes.
4. The user filters the sensor by removing outliers and selecting the chart type.
5. Once all choices are made, the user can visualize the data they chose along with the alarm log (if any).

<u>Extensions:</u>

1 a. User doesn't exist

  1 a 1. The system prompts the user to sign up

1 b. The user selects to upgrade to premium user

  1 b 1. The system brings the user to a page to pay and upgrade their account.

4 a. User selects to add alarms to the sensor.

  4 a 1. The system prompts the user to input their high and low threshold along with a deadband option if they choose.

4 b. User selects to add predictions on the sensor.

  4 b 1. The system prompts the user to input the end date they want for the predictions.

<u>Premium user scenario</u>

1. The premium user signs into their account.
2. The credentials are verified.
3. The premium user can upload their dataset to analyze which brings them to the select dataset page with the uploaded dataset (they can still select a default dataset).
4. The premium user follows the same path as the main scenario steps 3-9.
5. Once the main scenario, the premium user can download/export the visualization onto their device.
6. Once the premium user makes the visualization while also implementing an alarm, they can download the alarm log.

Status of Software Implementation

Implemented Requirements

User Requirements:

- Common users will start by signing into their account
- If the user doesn't have an account, they will be prompted to make one
    - Users are also offered to upgrade their account for a price
- Users can select a dataset and sensor from an API and use the channel and route ID to get the data in the program
- Users can filter the dataset they chose as they see necessary
- Users select the type of visualization they want for the sensor
- Users will be able to create alarms to monitor the data
    - The user can set high and/or low limits to be alerted if they are breached
    - The user can set alarm management options to avoid alarm issues
- The user can select an advanced Machine Learning Algorithm to predict future values of a sensor using other sensor values which is accompanied by a confidence interval for all predicted data
- Results will be displayed for the user in Visualization
    - The premium user can export the visualization
- The user will be able to see the alarm log if any went off
    - The premium user can export the alarm log
- Users can Sign out at any time, which kills their current data
- Premium users will have all the functionality of the common user but be able to upload their dataset

Table 1. Functional and Non-Functional Requirements

Functional	Non-Functional
The program will allow new users to sign up for an account with a unique name	The transition between sending a pull request of data to the API and accessing the data will be seamless
The program will allow existing users to sign in	The webpage should run on any modern device, including mobile.
The program will allow users to upgrade their account to a premium user account	The prediction algorithm should not take more than 1 minute to process future data
The program will allow the users to sign out of their account	User input data must be securely stored, as it could include confidential information

The program will use the dataset retrieved from the API by the user	The webpage must be able to handle multiple user interactions at the same time
The program will use the dataset provided by the premium user	The UI will be intuitive and easy for the user to interact with
The program will allow the user to select between three different chart types	The application will be easy to install on a new device
The program will visualize data pre- and post-analysis for the user	
The program will allow the user to cleanse the data by removing outliers	
The program will predict future trends within the data the user provides	
The program will allow the user to add alarm thresholds to the data	
The program will alert the user if an alarm has been breached	
The program will keep a log of all alarm breaches	
The program will provide the user with data analysis	
The program will allow premium users to download the visualization and alarm log	
The program will throw errors if the user has mic input values or made a mistake	

The software implementation has successfully met the specified requirements, ensuring a seamless user experience. Users can easily sign up, sign in, and upgrade their accounts. The system allows for the use of data from a public API channel, enabling users to filter and visualize information according to their preferences. With high-low limits and alarm management functionalities in place, users can effectively monitor data fluctuations and set alerts accordingly. The prediction algorithm processes future data promptly, ensuring results are generated within a minute. Importantly, premium users' input data is securely stored to protect confidential information. The webpage is designed to be responsive and compatible with all modern devices, including mobile, facilitating accessibility for users on the go. Additionally, the system can handle multiple user interactions simultaneously, providing a smooth and efficient experience.

With features like exporting results and alarm logs, premium users have the flexibility to manage and analyze data effectively. Finally, users can securely sign out, and premium users can also upload data seamlessly.

A few initial requirements were not able to be met and they will be further discussed in the next sections. It is important to note that the unmet requirements do not cause any problems with the functionality of the program, rather they are things that were not feasible to implement for this program or are things that are not useful. This program meets all the requirements listed above in Table 1, but more importantly, meets all the minimum requirements set out by the professor and TAs.

Surprisingly, the initial requirements set at the start of the project were more than enough to sufficiently capture the details of the project. The requirements in the form of functional, non-functional, user, and user stories set our group on a good path to success for this project. The initial requirements adhered to the minimum requirements for the project and added a few more requirements, which allowed this group to create a robust, detailed, and functional program in the end. It is worth mentioning that some requirements were altered as the project progressed and that will also be touched on in the next sections.

Unimplemented Requirements

Initially, our project intended to allow users to input or select multiple sensors to visualize in the end. Unfortunately, we could not implement this into the program due to time constraints as well as it would cause difficulties with the way our project takes inputs from users. We also intended to have preset data from an API in a list for users to easily select, but now we make the users get the channel and route ID from a public channel in ThingSpeak to access data to use in the program. Finally, the users are not able to manipulate the axis of the charts since that can not be implemented with the selected chart import. Chart.js automatically chooses the best axis to view the input data anyway.

Backlogged Tasks

When problems arose with user stories and tasks that were either changed or not possible, the tasks were altered and implemented or marked as not feasible and closed respectively. This left no tasks in the backlog by the end of the project, but the notable unfinished or changed tasks are task #34 multiple sensors which was marked as unfeasible, task #109 and #132 which changes the preset data list to inputs from the user which gets data from ThingSpeak, and task #259 axis manipulation which was marked as unfeasible and unnecessary due to chart.js doing the work already.

System Architecture

The system is designed around 3 design patterns. Primarily, it focuses on an observer pattern between the data inputted and the alarms that observe it. Other classes operate around this central core. An alarm manager is a singleton that manages the alarms, keeping them present for modifications to prediction changes with the dataset. There can only be one alarm manager, as all alarms are treated as global. The reasoning for this was that the system can only process one data set at a time, so alarms, as created, should be maintained during changes in parameters on the set. There is another pattern implemented with the decorator pattern. This is used to dynamically add runtime functionality to the sensor class. It allows the sensor to extend functionality to do prediction and cleansing of the data.

There are two ways of adding data to the system. First data may be queried from the ThingSpeak API. This requires sending an HTTP GET request to the public ThingSpeak channel where the data exists. While one class handles these GET requests, a separate class handles the input of data from the user via a .xlsx file. Together these two classes represent the data acquisition functionality of the application.

Surrounding the data processing is the rest of the system. A login, registration, and user system was also created. Upon entering the website for the first time, or while not logged in, users are placed on a welcome splash that introduces the system. From there, they can log in or register to access the main page. This page is where data can be processed and alarms set. This main page has different features depending on user type, where regular users can only use public ThingSpeak channels, and premium users can upload their data. Upgrading is possible for a regular user, by entering a specific upgrades page. Furthermore, on the main page, users can set alarms in the sidebar.

An example flow for the system would be a user arriving on the welcome page, logging in, choosing a public sensor to use, or using their data if they're a premium user, then setting alarms and plotting their data. If they were a premium user, they would then be able to download their data.

Overall, our project outcome is very similar to what we had originally intended. Our flow pattern is exactly what we had hoped to achieve. Some small features ended up different, however the core of the system is the same. One change was that initially it was planned that there would be pre-generated datasets that could be selected, but it was decided that it would be a better user experience to allow for public sensors to be used. This gave users the ability to see how features worked on any public sensor they could find. Alarms also changed, with them shifting from real-time processing disrupts to post-process notices. This was justified as the data was always historical or predicted data, meaning there was no need for a real-time disruption. The core of the system stayed consistent as the focus was on taking a dataset, cleansing it, and predicting future trends in it.

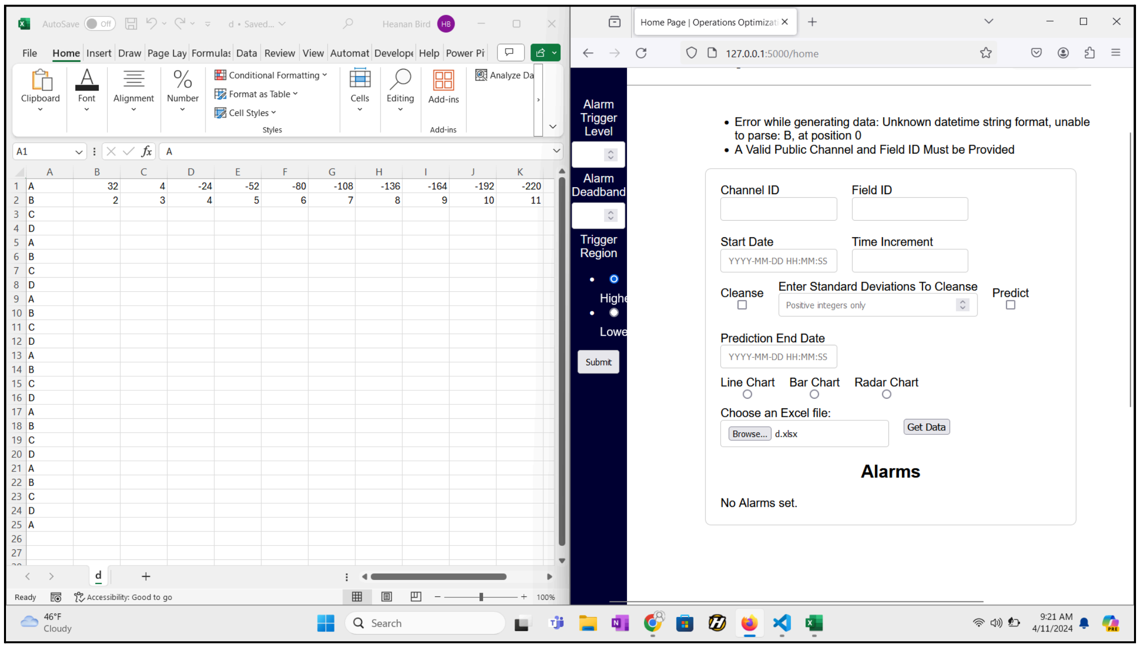
Code Reuse

We worked to ensure as much code reuse as possible throughout this project. Anything that could be reused would be. This included things like the decorator classes for the sensor being reused to implement the observer pattern between the alarm and the sensor. Our user class used the inheritance of the Flask user to avoid duplication of functionality and create an easier flow. All of our forms implemented Flask forms, which also aided in our code reuse. These features were used in these ways to make flow cleaner and to save time. It prevented different sections of code from doing essentially the same thing and allowed us to make good use of inheritance within our program. Overall, this code reuse should make it so the program is easy to follow on the code side and allow for minimal duplication of features.

<u>Known Bugs with Solution</u>

Improper Formatting of Uploaded Data - Double Error Display:

When data is uploaded that does not follow the proper format, one column of datetime parsable strings and one column of values, two errors are thrown, caught, and displayed. The first error "Error while generating data: Unknown datetime string format, unable to parse: B, at position 0" is correct and identifies the issue to the user. The second error "A Valid Public Channel and Field ID Must be Provided" does not relate to the uploaded data.



	A	B	C	D	E	F	G	H	I	J	K
1	A	32	4	-24	-52	-80	-108	-136	-164	-192	-220
2	B	2	3	4	5	6	7	8	9	10	11
3	C										
4	D										
5	A										
6	B										
7	C										
8	D										
9	A										
10	B										
11	C										
12	D										
13	A										
14	B										
15	C										
16	D										
17	A										
18	B										
19	C										
20	D										
21	A										
22	B										
23	C										
24	D										
25	A										

- Error while generating data: Unknown datetime string format, unable to parse: B, at position 0
- A Valid Public Channel and Field ID Must be Provided

Choose an Excel file: d.xlsx

Alarms

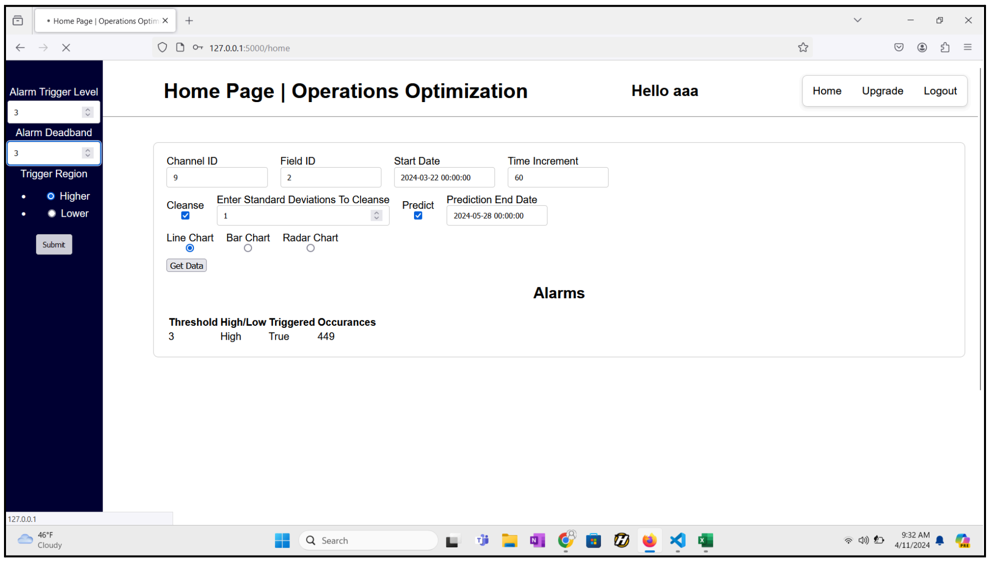
No Alarms set.

Figure 2. Excel data input logging

<u>Fix</u>: The "A Valid Public Channel and Field ID Must be Provided" error is thrown and displayed for unsuccessful plotting attempts regardless of the data source. This needs to be changed to specify if the error is from a data upload attempt vs a ThingSpeak API retrieval attempt.

Reset of Prediction Model Fitting Via Alarm Manager Request:

After data has been entered and a prediction is requested the web application will load as the model is fitted. If during this loading time, a value is entered to add an alarm to the alarm manager, the prediction will stop and no data will be displayed. Here the addition of the alarm is taking precedence over model fitting.



Home Page | Operations Optimization

Hello aaa

Alarm Trigger Level: 3

Alarm Deadband: 3

Trigger Region: [✓ Higher] Lower

Channel ID	Field ID	Start Date	Time Increment
9	2	2024-03-22 00:00:00	60

Cleanse [✓] Enter Standard Deviations To Cleanse: 1 Predict [✓] Prediction End Date: 2024-05-28 00:00:00

[✓ Line Chart] Bar Chart Radar Chart

Alarms

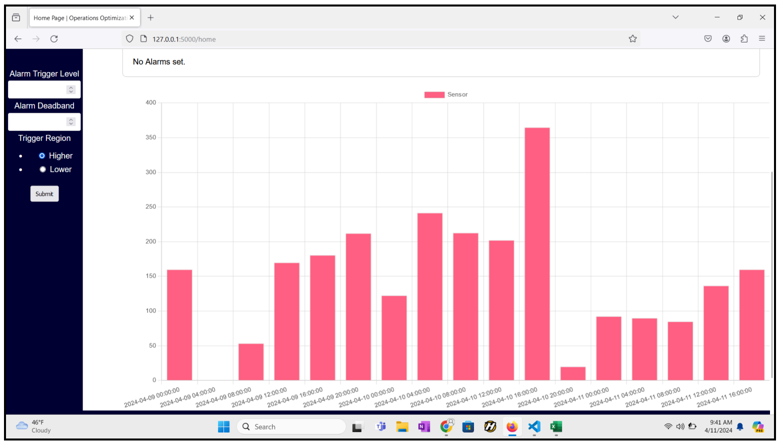
Threshold	High/Low	Triggered	Occurances
3	High	True	449

Figure 3. Setting parameters for the front-end application.

Fix: Both the Flask triggers for the Alarm and Data entry forms can be found in the same routing function. This allows for the interruption of one to occur via the submission of the other. To stop this each form submission and subsequent logic should be in a different routing form but routing to the same page. This would require refactoring and some minor logical changes of code.

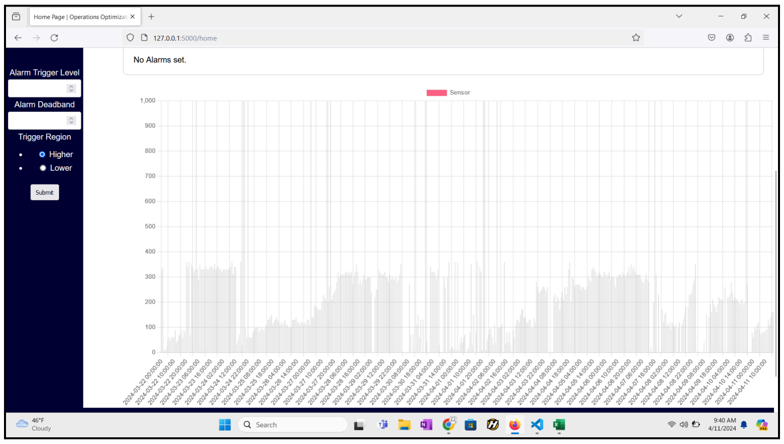
Excessive Data Display Overlap On Bar Chart:

When an excessive quantity of data is provided to the chart and a bar chart is selected as the means of display, a display bug occurs. This bug is a result of the chart.js attempting to display a bar of data for each data point. As the number of datapoints grows chart.js automatically resizes the bars. But for excessive quantities of data, the bars are forced to overlap and an overlapped bar greys itself out. This means that the small quantities of data will display correctly but excessive data quantities will be difficult to see and read. The point at which this bug occurs depends on the screen size being used.



No Alarms set.

Figure 4. Bar chart data with proper functionality



No Alarms set.

Figure 5. Bar chart data with improper functionality

Fix: A bar chart itself is useful for small and medium quantities of data. The best fix would be to limit the amount of data the bar chart selection can plot. Before the data query is sent to ThingSpeak the length of the query can be estimated. If too large an error can be thrown, caught, and displayed to the user restricting the use of the bar chart in the form input so as the stop the bug from occurring.

Project Handover - Based on README.md

To run this project, the repository must be downloaded/cloned from GitHub onto a machine with Python 3 installed.

To run the Flask application on your local machine, follow these steps:

1. Open your terminal in the folder where the repository has been cloned

2. Install the required libraries & dependencies by using
   ```
   pip install -r requirements.txt
   ```

3. Initialize the database via the use of the following command in the terminal
   ```
   pytest
   ```

4. Navigate to the project directory using
   ```
   cd src
   ```

5. Finally, start the Flask application with the command
   ```
   flask --app flaskr run
   ```

6. In your terminal, the Flask application will start on localhost **127.0.0.1:5000**

7. Copy **127.0.0.1:5000** into a **Chrome** or **Firefox** browser and the web app will be accessible

Potential Maintenance Issues:

When starting the application two things may happen. First, you may already be logged in as a regular user. At this point, you may log out, upgrade, or use the application as is. Two, you may not be logged in at all. At this point, you may register or log in, but will not be able to use the application until an account has been made.

Reflections

In managing our project, one of the significant challenges we faced was the diverse skill set within our team, stemming from varied programming backgrounds due to our engineering program's unique curriculum. While this diversity brought valuable perspectives, it also led to gaps in assumed knowledge, creating obstacles along the way. To overcome this, we had to engage in extensive communication to clarify project requirements effectively. While our approach proved effective, we would make one adjustment for future projects: initiating independent meetings earlier in the term to anticipate and address potential roadblocks. The division of duties, such as frontend and backend responsibilities, worked well, but we recognized the need for increased communication between these areas as the project progressed. More frequent group or scrum meetings would have facilitated problem-solving, a lesson we will implement by starting such meetings earlier next time. Additionally, we would establish clearer delineations of tasks and responsibilities and plan software usage. A clear project manager, with the ability and skills to work on each section, would be established. This would prevent one person from being delegated as a super contributor at the expense of their assigned work, and allow the whole team to benefit.

In terms of our initial requirements, we believe they were sufficiently detailed, especially considering that we adhered to the minimum requirements outlined for the project. There were few changes or overlooks in additional requirements throughout the process. Each requirement was well-considered and straightforward enough for us to accomplish within the given timeframe. Overall, our initial requirements provided a solid foundation for the project's execution, contributing to its successful completion.

Beyond the requirements, there were several crucial aspects that we missed in our initial planning. Firstly, we didn't adequately consider which programs would be necessary for the project, both in terms of technical tools and interpersonal communication for the project. This oversight resulted in some inefficiencies and delays in coordinating tasks and sharing information. No group members were working in virtual environments for the first half of the course, which led to some code not running properly on each machine. Another issue we faced was underestimating the importance of proper issue management. Without a robust system in place, we encountered difficulties in identifying, tracking, and resolving issues promptly, leading to some setbacks during the project's execution. Additionally, we overlooked the significance of establishing clear expectations for teamwork. Several group members under or over-contributed during the project due to this lack of communication. The lack of defined guidelines on individual contribution and team effort resulted in misunderstandings and last-minute complaints, revealing a communication gap that could have been addressed with better upfront planning.

Looking back on our testing approach for the project, we believe we handled it effectively as a team. We diligently executed unit testing, automation testing, and to some extent, black box testing, which ensured the functionality of our codebase. The implementation of a pipeline was particularly beneficial, guaranteeing that all merged code was functional before deployment. However, in hindsight, one area we could improve upon for future projects is ensuring that our tests are more comprehensive. While our existing tests covered significant aspects, enhancing their scope could further bolster the robustness of our code and minimize the risk of undiscovered bugs.

Our initial estimates for the project's efforts compared to its actual delivery were distant from each other. Initially, we underestimated the time required to grasp the background programming frameworks and concepts essential for the project, such as Flask, Python, HTML, and Jinga. The learning curve for these technologies proved to be more time-consuming than anticipated, delaying our programming start. Additionally, implementing alarms posed a significant challenge, consuming more time than expected due to its complexity. We also overestimated the feasibility of splitting duties among team members, as understanding each component's intricacies and dependencies required more collaboration and coordination than initially thought. Furthermore, the extent of testing and the time required for thorough PR reviews were underestimated, highlighting the criticality of these processes in ensuring quality deliverables. Moving forward, we will factor in these insights to provide more accurate estimations and allocate resources effectively for future projects.

One standout achievement that our team takes immense pride in is our ability to develop a functional system despite having limited prior experience in programming. While some team members possessed basic knowledge of Java or Python, the depth of our expertise significantly expanded throughout the project's duration. Witnessing the project come together and successfully integrating various imports and programs was a rewarding experience for all of us. Moreover, mastering the intricacies of the program, especially considering our relatively recent introduction to coding in Java, was a significant learning moment. Additionally, the creation of the radar chart, despite any reservations by our TA, stands out as a proud component of our project.