CS 307 | Software Engineering I
Group 12 | Project Design Document

ANDA: Application for Natural Disaster Avoidance

2/14/2025

Gleb Bereziuk, James Lamparski, Josh Rubow, Jinhoo Yoon

1. System

The goal of our system is to combine the Google Maps API and our ESP devices as well as weather data from trusted government agencies to relay data regarding local weather and road conditions while storing it in our own database, creating a real-time application software. The system will allow for users to use our online application in order to commute via the safest and most efficient pathways, and report road/weather conditions manually if they wish, resulting in safer transportation conditions for the general public.

Our application will differentiate itself through two main components. First, we will aggregate data from multiple geographic areas, ensuring a seamless transition for users travelling over long distances. Other similar competitors, such as Indy Snow Force Viewer, only report on localized areas, forcing users to use multiple applications for travelling purposes should they go across state lines. Additionally, functionality across different applications vary on the types of weather reporting, or do not even have an applicable product for users to access.

Furthermore, our aggregation of data means that users can easily access a standardized application from an array of different geographical locations. We believe ANDA will be a more efficient and accessible application by incorporating real-time sensor data using our proprietary devices and combining that with reporting from established government agencies, creating both a farther-reaching and more accurate product than competitors.

2. Design Outline

Design Decisions

Devices - ESP32 + Sensors

We are using the ESP32 because it includes all the necessary capabilities for our project:

- Bluetooth LE for our IoT mesh network
- 2.4ghz WiFi for connecting to our API
- GPIO pins for connecting various sensors
- 5v/3.3v outputs for sensors of different power needs

Backend - MySQL

We will use MySQL for storing user information, admin information and sensor information. MySQL is the best option for us because it is a free SQL server architecture. MySQL offers fast performance, scalability, user permissions and is open source. We chose this over other options (like MSSQL) because they are not open source and cost extra money that is not necessary for a minimum viable product.

API - Express.js

Express.js is a lightweight, fast and secure way to implement an API. This is great for our project since we are on a deadline and need to finish our API quickly. Express.js also offers middleware support which is great for robustness and a modular API like we will need. It also offers easy routing for different request endpoints, of which we will have a lot.

Frontend - React.js

We chose React because it is easy to modularize through the use of functional components, easy API integration and routing capabilities. React is an extremely popular frontend framework which is used by some of the largest companies in the world, this means that it is well tested and secure.

Interactions between Individual System Components

System Architecture Overview

- Cloud based backend for the data to be processed and stored
- Web front end for user friendly interfaces
- IoT devices and third party weather API's to collect weather data
- API for transfer of data between frontend, backend and IoT devices.

Interactions between Components

- IoT sensors
    - collect real-time environmental data such as temperature, road conditions, and flood levels **will communicate directly** with our custom API.
    - Will communicate directly with each other over Bluetooth LE to pass along sensor information.
- Frontend
    - **will use the Weather data API** that allows weather data from external sources such as The Weather Channel
    - **will communicate with our API** for the following purposes: Frontend to make reports, display alerts, map layers for safe route determination for general users as well as all necessary administrative tools.
- Backend server
    - to process the weather data, government alerts, user reports, run route calculations + optimizations algorithms, manage sign-ins. **Will communicate directly** with our SQL Server to retrieve information. It will also serve json to requests from the API.
- API
    - **Will communicate with our backend, our frontend and our devices.** It will communicate with json data.

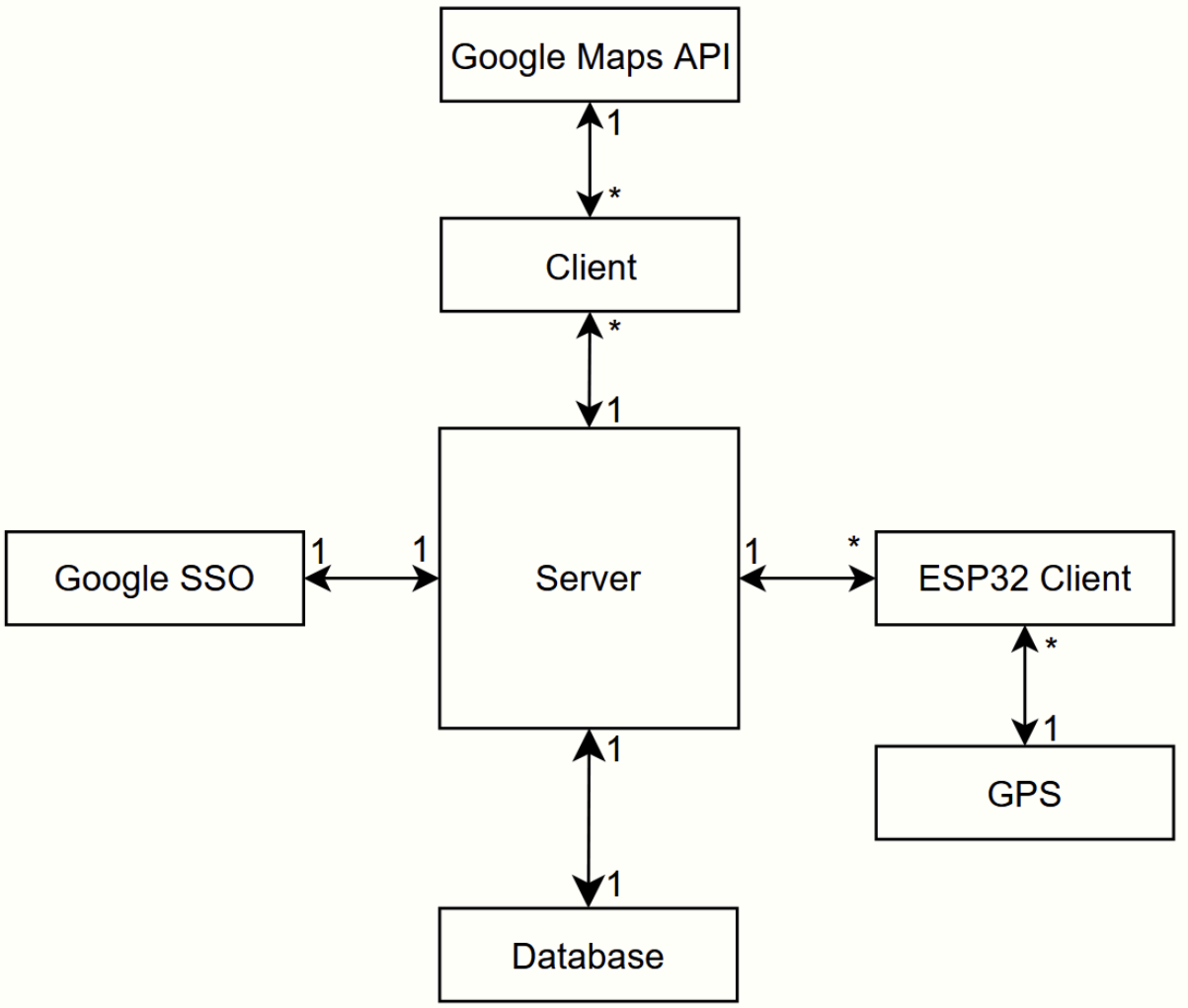

Figure 1: Basic UML system diagram

3. Design Issues

Functional

Issue 1: Real time route adjustments

- Option 1: Dynamic rerouting
- Option 2: Manual hazards report
- Option 3: Predictive modeling

Final choice: Option 1 + Option 2
Reasoning: Option 1 and 2 can work well together without interfering. The program will have to account for false manual reports, but can prioritize the data from the API and IoT's. The manual inputs will account for patterns the devices cannot account for, such as car crashes. The predictive modeling would be too much to implement when dynamic rerouting is sufficient enough.

Issue 2: Malicious user spam reports

- Option 1: Leave as is
- Option 2: Request other users to confirm
- Option 3: Allow authorities to remove inaccurate reports
- Option 4: Send every report to authorities to verify first

Final choice: Option 2 + Option 3
Reasoning: Option 1 will allow our service to be populated with inaccurate data which could be dangerous for users; Option 4 might cause our service to be less reliable given that government agencies will not always be available to confirm each and every report. Option 2 will let users "police" each other and inaccurate/outdated reports will be removed after multiple confirmatory reports; Option 3 allows reports to be removed without multiple confirmations and with official information.

Issue 3: Data privacy handling

- Option 1: Requiring password for users
- Option 2: Secure database model
- Option 3: Delete user data

Final choice: Option 1 + 2
Reasoning: This is the most secure way to handle user data and privacy. There is information such as location and preferences that should be kept safely. Requiring a strong password will make their account safer. Also a secure database will improve this safety. These both help each other and make the website that much more secure.

Non-Functional

Issue 4: Powering ESP32 devices

- Option 1: Solar power
- Option 2: AA/AAA batteries
- Option 3: LiPo batteries
- Option 3: Cables

Final choice: Option 2
Reasoning: Option 1 is objectively pricey; Option 3 is highly impractical; Option 3 is viable, but is also pricey for minimum viable projects. Option 2 is the most optimal given relatively low cost, accessibility, and ease of use.

Issue 5: Communicating with ESP32 devices

- Option 1: Each device is connected to mobile network
- Option 2: Set up signal boosters
- Option 3: Cables
- Option 4: Leave as is, use Bluetooth

Final choice: Option 4
Reasoning: Option 1 is an overkill for a small network like ours will be; Option 2 is pricey; Option 3 is impractical. Given ESP32 built-in Bluetooth modems, this is the most optimal option. While we are sacrificing range for cost-effectiveness, this is sufficient for a small test network.

Issue 6: Weathering to ESP32 Devices

- Option 1: Outside shell to keep device safe
- Option 2: Redundant sensors
- Option 3: Leave device unprotected
- Option 4: Put device in safe places

Final choice: Option 1
Reasoning: This is a secure way to keep the devices safe without needing to purchase redundant sensors. The redundancy is in having x amount of ESP32 devices available at all times. If one gets damaged, its highly likely that the on board redundant sensors would get damaged as well. This is why option 1 is the most secure way and economical way to do so.

Issue 7: API server dependency (Google Maps)

- Option 1: Include Apple Maps
- Option 2: Pre load areas of interest (big cities)
- Option 3: Monitor google Maps connection

Final choice: Option 3
Reasoning: This is the most economical and reasonable way to approach this problem. Using Google Maps to load a map of the area is what we will stick with. To include Apple maps would be too much redundancy for how reliable Google Maps is. Also preloading areas of interest would be a lot of data collection and could go out of date. This is why we chose option 3 to ensure we are always connected to Google Maps.

Issue 8: Conflicting data from multiple sources

- Option 1: prioritize IoTs
- Option 2: prioritize Weather Channel
- Option 3: prioritize user inputs
- Option 4: combine and create hierarchy of inputs

Final choice: Option 4
Reasoning: This allows the website to take multiple inputs of data to yield the most accurate depiction of the weather. IoTs will get in time readings, and the Weather Channel will be good at giving other weather conditions and predictions. While the user inputs will give inputs that we cannot predict or find. The Weather Channel will get the highest priority, then IoTs next, and then any user input that does not contradict the higher levels will be accepted. These will be monitored and altered if necessary.

Issue 9: Limited data from rural areas

- Option 1: Use IoT's
- Option 2: Rely on user inputs
- Option 3: Use weather channel
- Option 4: Use satellite data

Final choice: Option 1, 2, and 3
Reasoning: These options are what we plan to do for most areas, but there might not be as much coverage in rural areas. This is where user inputs would be more useful and the IoT's. The use of satellite data would be too much money, and the weather channel already uses satellite data. There might have to just be more emphasis on user input or IoT data depending on the area.

Issue 10: API rate limits

- Option 1: Cache data to limit calls
- Option 2: Stagger API requests
- Option 3: Increase API rate with paid plan

Final choice: Option 1
Reasoning: This option will allow fewer API calls without increasing the cost. Storing the data will allow the website to first access this information before calling the API again. Staggering the request would make the user experience much slower. Also paying for a better plan is out of the scope as we are on a low budget.

4. Design Details

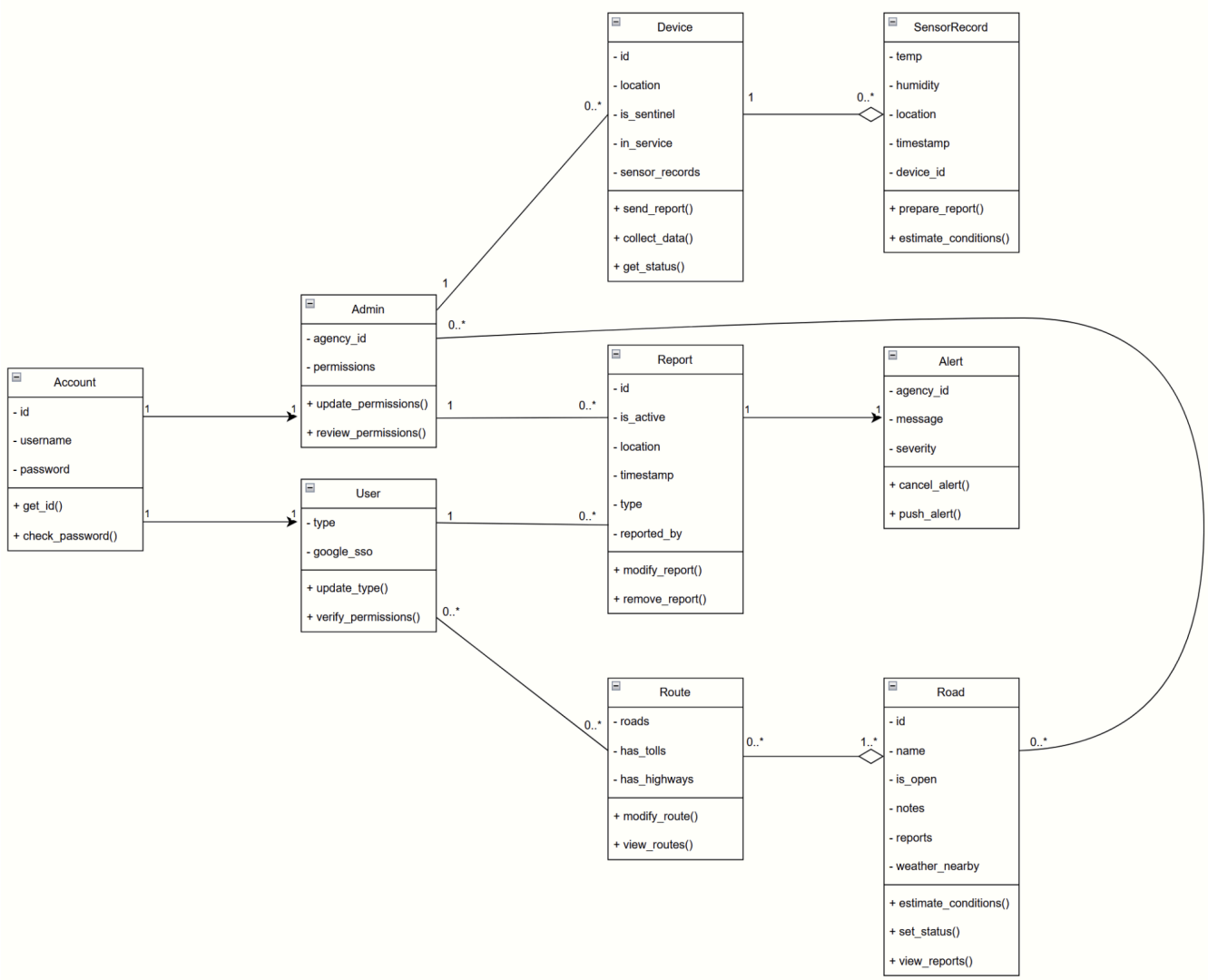

Figure 2: Class diagram

**Note: Class attributes might get modified as the system starts taking shape.*

Description

Account class: the parent class of two of our user account types. Contains the most basic attributes like id number (internal to the system), username and password.

- Admin class: one of two child classes of Account. Contains the agency identifier number as well as permissions within the system.
- User class: the other child of Account. Contains the user type as well as whether they are using Google SSO feature.

Device class: contains information about each ESP32 device in the system. Identified by ID, contains its location, service status, if it is a sentinel device and SensorRecord list of data recorded by its sensors,

- SensorRecord class: not a child of the Device class but a supplemental class that would contain all the data collected by its sensors at a single time point.

Report class: contains identifying information such as ID number and User/Admin who created for each specific report as well as the actual details and location of report

- Alert class: a type of Report, extends report. Only created and modified by Admin (with permissions). Contains severity identifier. Used for urgent official alerts/warnings.

Route class: uses Road objects and has supplemental information that is necessary for a route to be built. Built upon User request for a route

- Road class: an object with each road's important information like name, notes, whether it is open, and weather conditions nearby. Used for building routes (in Route class). Modified by Admin.

Relations between classes:

- Admin and User extend Account
- Device uses SensorRecord objects for organizing data
- Admin (with permissions) is able to access Device classes and SensorRecord data
- Alert extends Report
    - User able to create Report; Admin able to create and modify report (with permissions)
    - User able to view other Report and Alert details
    - Admin (with permissions) able to create and modify Alert
- Route uses Road
    - Road can be modified by Admin (with permissions)

Sample Sequence Diagrams

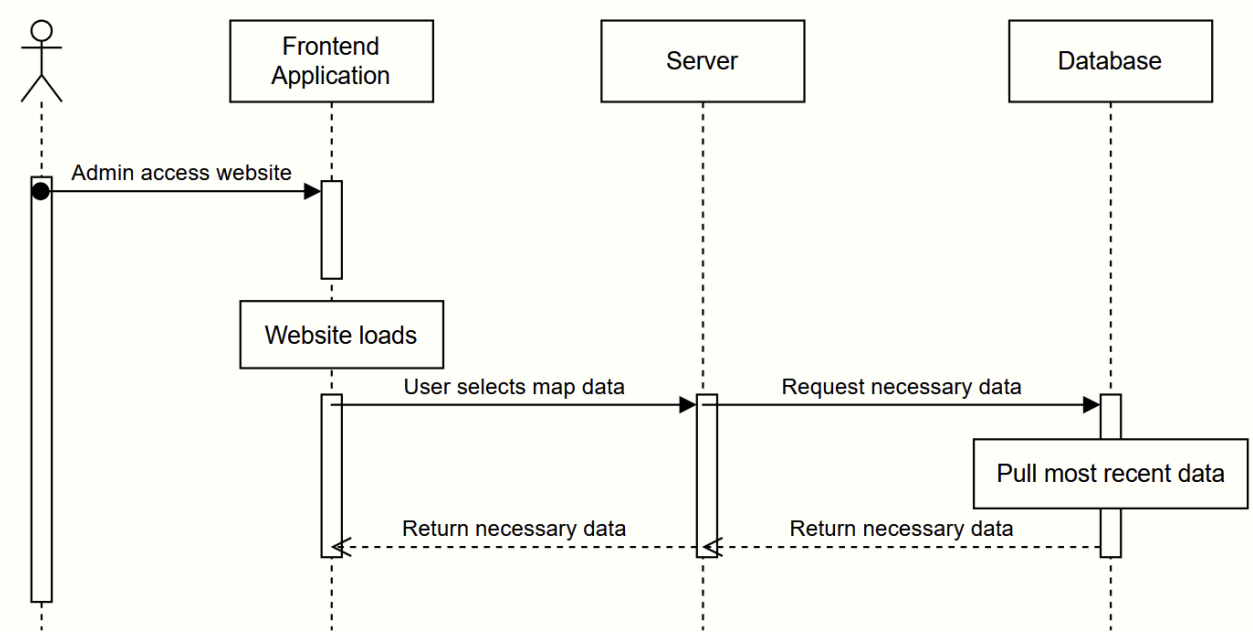

Figure 3: User selecting a map layer

Figure 3 represents a sequence diagram of a user selecting map layers on the website. They make a request from the front end which then triggers a server pull of most recent data, following the given filter. Then that data is returned to the front end where the user can now see the new display selected.

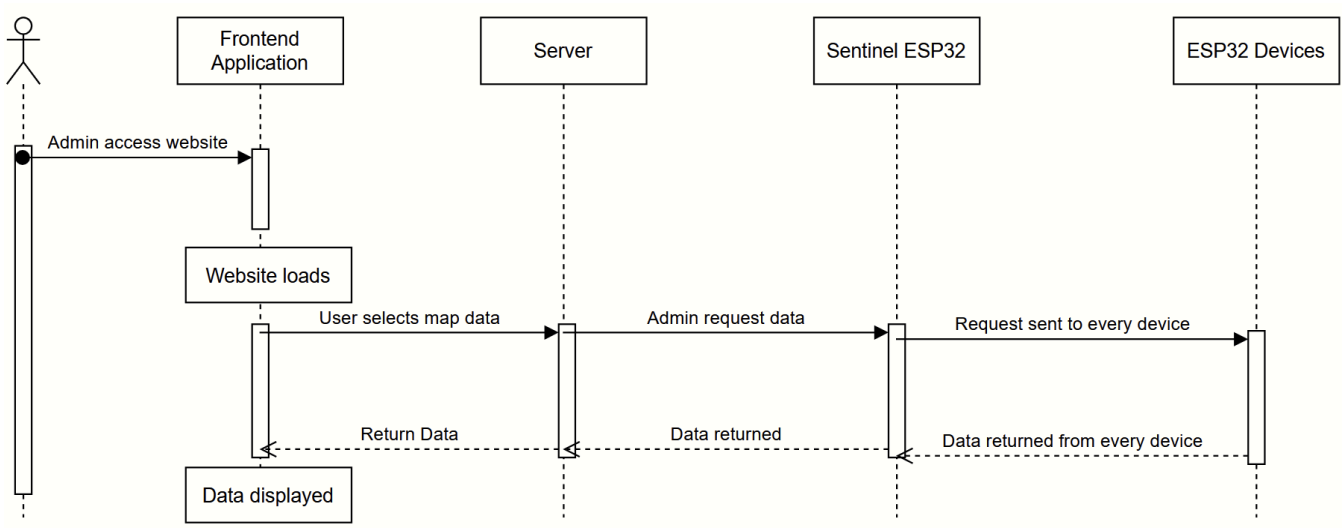

Figure 4: Administrator requesting current IoT data

Figure 4 represents a sequence diagram of an administrator requesting the current IoT data. This is data from the ESP32 devices that will be outside collecting real time data. The admin requests this data pull which makes its way down to the ESP32, which then returns the data which travels back up to a display for the admin.

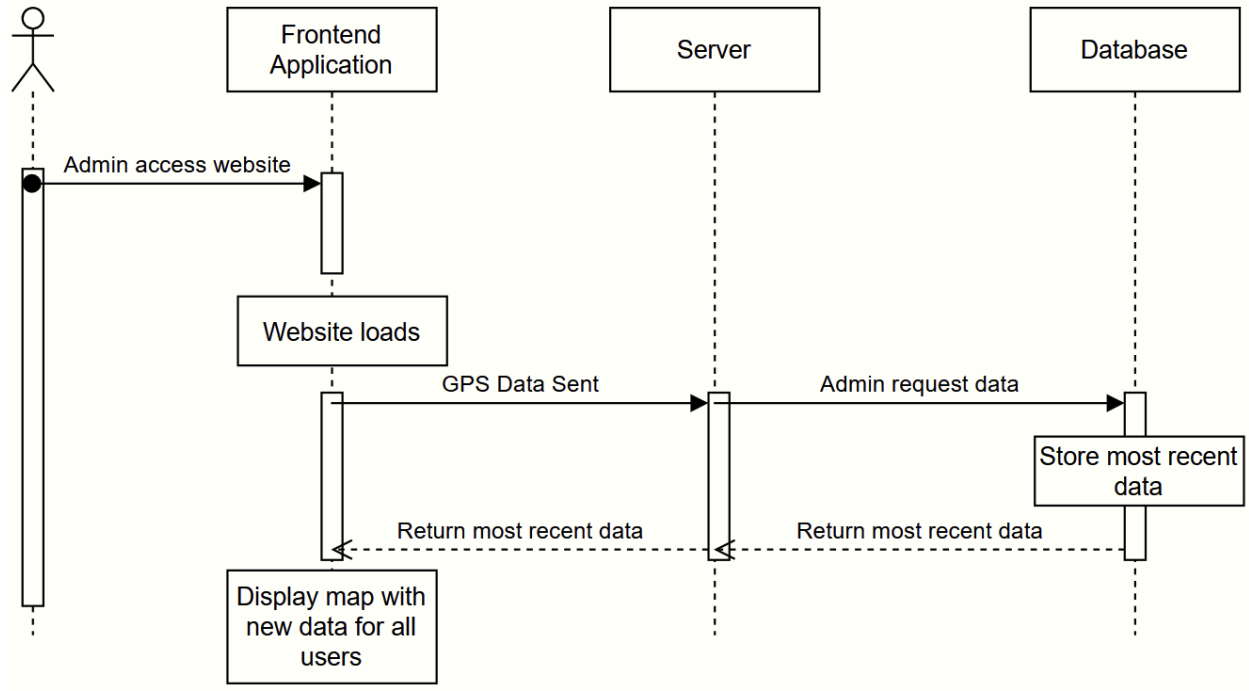

Figure 5: User (snow plow) reporting GPS data

Figure 5 represents a sequence diagram of a situation where a snow plower sends their GPS data to help track plowed roads. The front end sends this GPS data to the server which then updates the database accordingly. Then a return of the live map update happens.

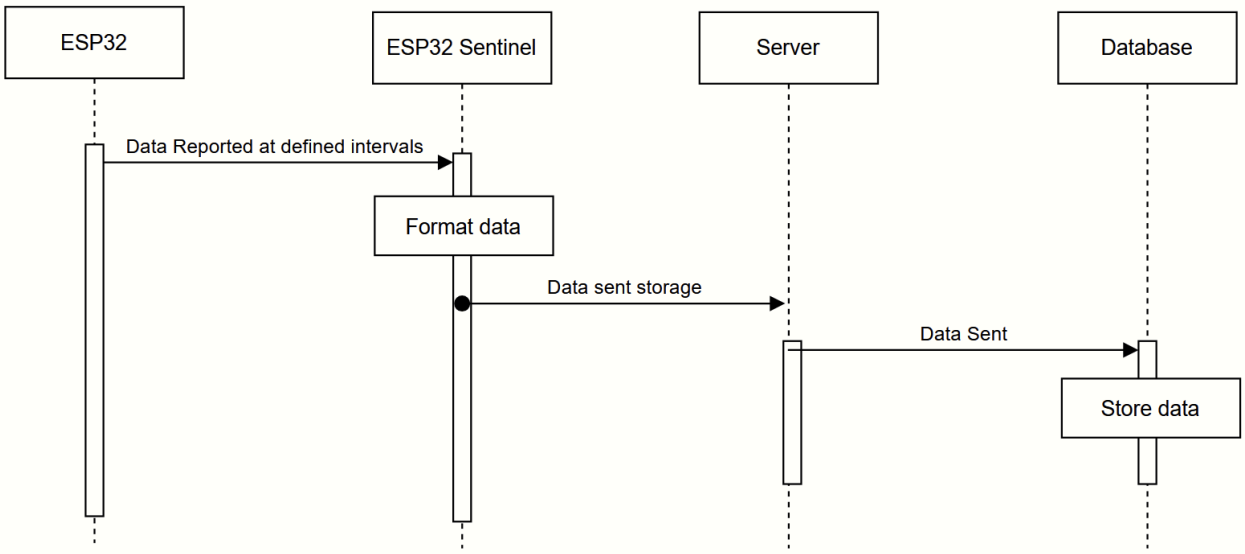

Figure 6: Data reporting (continuous)

Figure 6 is a sequence diagram that shows the concept of how the ESP32 continuous data reporting will work. The device is the central controller. It collects the data and initiates the data being reported to Sentinel. The data is then formatted to then be sent to the server. Once the server has the data, it can then be stored in the database to be extracted whenever needed.

Potential UI designs

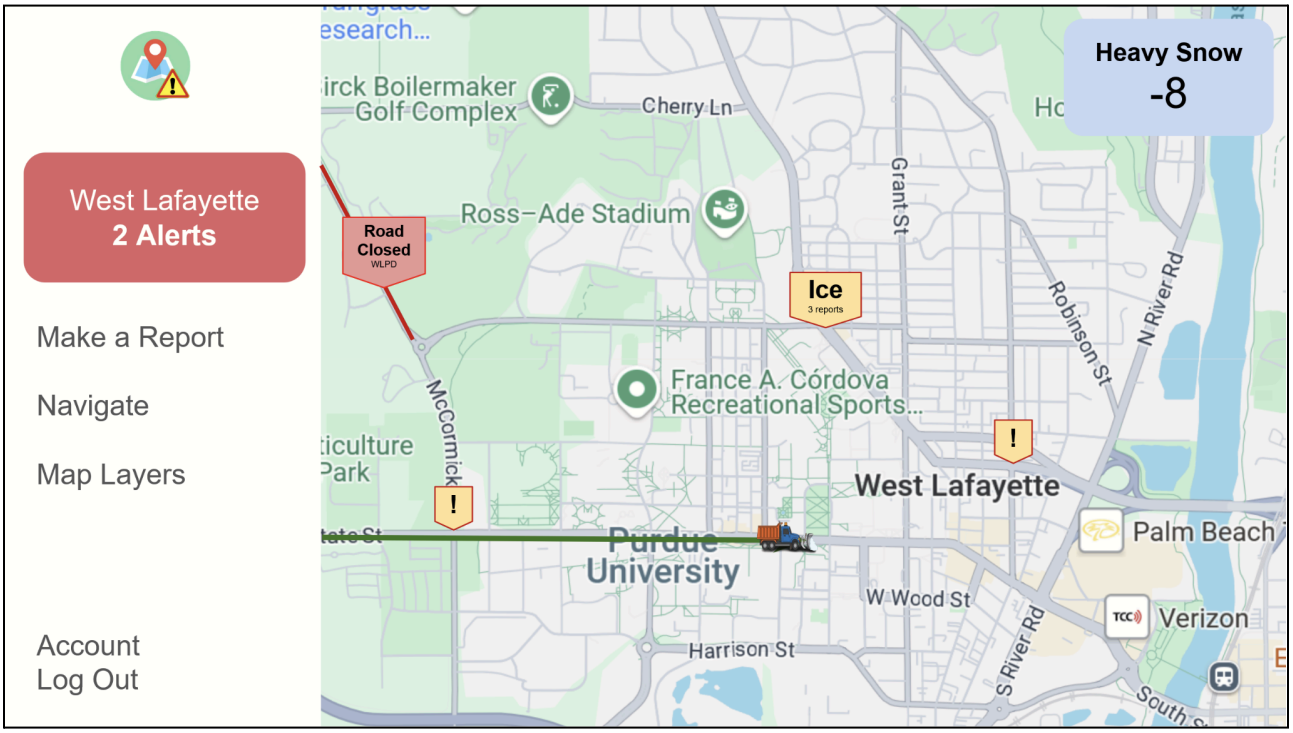

Figure 7: General user view:

Figure 7 is a UI mockup representing the General user home screen when viewing weather events and data. Here the user has the ability to make a report, navigate to a new destination or change the map layers for different views (this will be managed by the Google maps API). The user also has the ability to log out or access their account settings from here. There are also multiple indicators on the map which are generated by manual reports made by other users and ESP32 mesh network sensor data as well as the ability to check current alerts made by government officials.

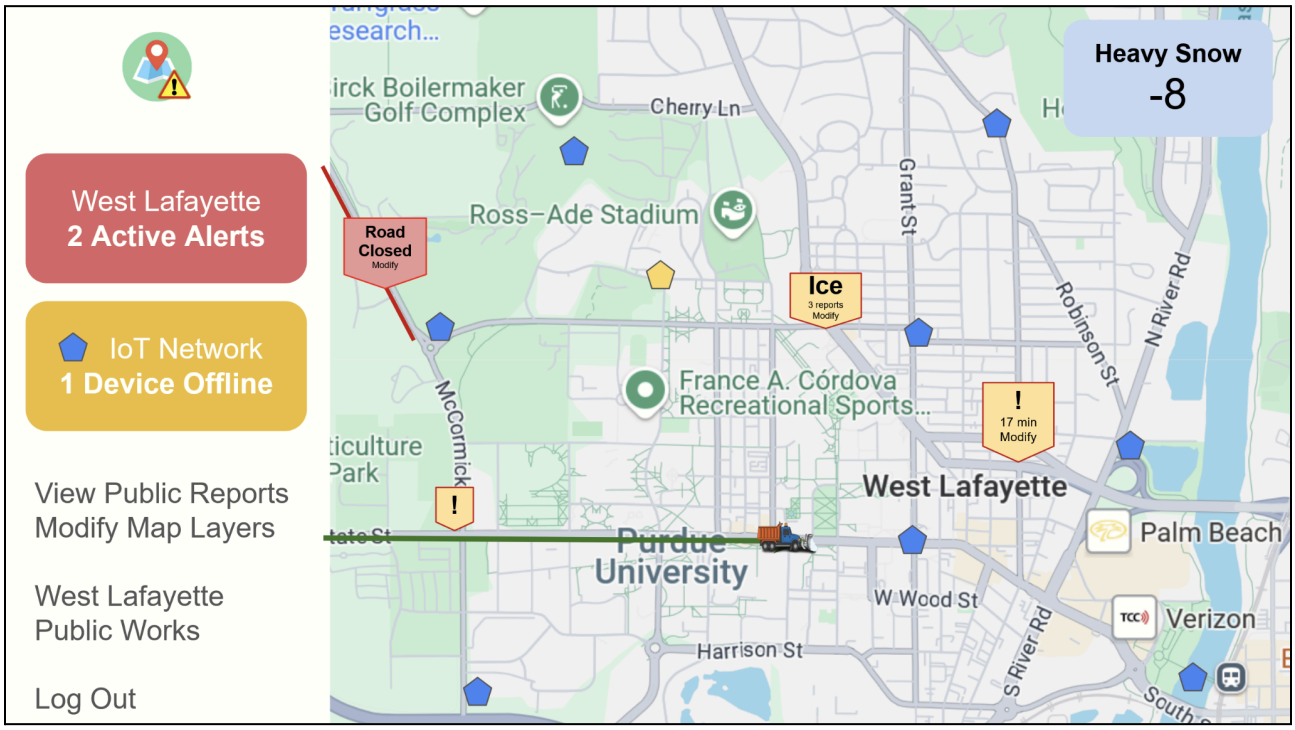

Figure 8: Administrator view

Figure 8 is a UI mockup representing the Administrators homepage view. It shows the available IoT devices and their locations as well as weather events generated on the map. It also gives the administrator the ability to modify the map layers (managed by Google Maps API) as well as log out and view the cities account page. Another important feature is the ability to create public alerts and review reports made by users. Importantly it also shows the offline IoT devices so that they can be checked on.