



WPI

Social Distance and Contact Tracing Dashboard

A Major Qualifying Project Submitted to the Faculty of
WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the
Degree of Bachelor of Science

Written by
Vivek Wong
Xiaoyue Lyu
Yezi Chen

Advised by
Professor Xinming Huang

September 4, 2020

Abstract

During the global COVID-19 pandemic, the Centers for Disease Control and Prevention (CDC) suggested people keep six-foot distancing from each other to prevent the spreading of the virus. Collaborating with Amphenol TCS, the team created a contact tracing dashboard to monitor the close contacts among employees in a manufacturing facility. In addition to providing statistical summaries, the dashboard has the ability to trace the people who have had frequency and lengthy contacts with someone if he or she is tested positive. It also includes an API for receiving data directly from the hubs of the ultrawide-band contact sensors.

Acknowledgements

We would like to thank our sponsor Amphenol TCS for sponsoring the project and providing us with such great opportunities. We would like to thank Richard Schneider, Senior Executive of Amphenol TCS, for trusting us and making the project possible. We would like to express our sincere gratitude to Jack St. Hilaire, Terry Gelbart, David Mutton and Aaron Gough for supporting us. We would also like to thank Professor Xinming Huang for advising the team.

Table of Contents

Abstract	ii
Acknowledgements	iii
Table of Contents	iv
List of Illustrations	vi
1. Introduction	1
2. Background	3
2.1 COVID-19	3
2.1.1 Coronavirus Outbreak	3
2.1.2 Symptoms and How it Spreads	4
2.1.3 Social Distancing	4
2.2 Indoor Positioning and Tracking	5
2.2.1 Bluetooth Beacon Technique	5
2.2.2 UWB Technique	6
2.3 Equipments for Indoor Localization and Tracking	7
2.3.1 DWM1001	8
2.3.2 Woxu UWB Wristband Tag UT-206	8
2.4 Existing Web Apps to Monitor and Analyze Social Distancing	9
2.4.1 Ubudu	9
2.4.2 Sormas	10
2.5 Frontend	14
2.6 Backend	15
2.7 About the Sponsor Amphenol TCS	17
3. Methodology	18
3.1 Simulation Program	18
3.2 Database	21
3.3 Web Application	23
3.3.1 Introduction	23
3.3.2 Objectives and Expectations	23
3.3.3 Initial Architecture and Tech Stack	25
3.3.4 User Authentication	28
3.3.5 Permission Control	32
3.3.6 File Uploading	34

3.3.6.1 Store Files in Local File System	35
3.3.6.2 Store Files in Amazon S3	37
3.3.7 Contacts Page	38
3.3.8 Devices Page	43
4. Results	46
5. Future Work	49
Bibliography	51

List of Illustrations

Figure 2.1 Woxu UWB Wristband Tag UT-206	8
Figure 2.2 The Screenshot of Ubudu Dashboard	9
Figure 2.3 Sormas	10
Figure 2.4 Sormas Dashboard	11
Figure 2.5 Sormas Contact Dashboard	12
Figure 2.6 Sormas Case Directory	13
Figure 2.7 Create New Case	14
Figure 3.1 Simulation Plot	19
Figure 3.2 Simulation Program	20
Figure 3.3 Raw Sample Contact Info	21
Figure 3.4 .cvs Contact Info	22
Figure 3.5 Sample Device IDs and Battery Level	22
Figure 3.6 Initial Architecture of the Web Application	25
Figure 3.7 Sequelize Function to connect to PostgreSQL	27
Figure 3.8 Hash Password	29
Figure 3.9 Global Variables to Contain Messages	30
Figure 3.10 Check User-Filled Fields of the Registration Page	30
Figure 3.11 Format Error Message	31
Figure 3.12 Includes Messages in the Page	31
Figure 3.13 Handle User Login	32
Figure 3.14 Permission Control Page	32
Figure 3.15 Permission Type	34
Figure 3.16 Bulk Add Data to Contact Table	34
Figure 3.17 Upload File with Multer	35
Figure 3.18 Extract Date from Date Sheet and Convert it to JSON	36
Figure 3.19 Delete Uploaded File	36
Figure 3.20 Setup for Amazon S3 Service and Multer	37
Figure 3.21 Create Readable Stream and Convert to JSON	38
Figure 3.22 Render Data to the Contacts Page	39
Figure 3.23 Insert Page-Specific Elements	40
Figure 3.24 Renders Data on Contacts Page	40
Figure 3.25 Setup for DataTables	41
Figure 3.26 Filter Form on Contacts Page	41
Figure 3.27 Autofill Program	42

Figure 3.28 Fetch and Send Filtered Data to the Frontend	43
Figure 4.1 Contacts Page	46
Figure 4.2 Devices Page	47

1. Introduction

Since the first report of respiratory illness due to a novel coronavirus in December 2019, the outbreak of COVID-19 has impacted 188 countries and territories, including the United States¹ (JHU CSSE, September 2020). The National Centers for Disease Control and Prevention (CDC) has activated its Emergency Response System and published relevant announcements to notify people to stay at home and keep a social distance. According to the data from Statista, the number of COVID-19 cases worldwide has reached 22 million and the number of cases in the U.S. has reached 5,656,204 as of August 19, 2020 (Statista, August 2020).

To prepare for lifting the coronavirus lockdown, our project sponsor Amphenol TCS aims to keep the employees' health as the priority when their manufacturing facilities reopen for production. Thus, they would like to track and record the employees' close contacts during working hours. If an employee is tested positive for COVID, the human resource manager can trace a list of people that were in close contact with him/her and only ask them to self-quarantine for health concerns.

To be more specific, Amphenol TCS would like the team to develop a web application to display a contact tracing dashboard that helps managers control the spread of COVID-19 in the factory. In Amphenol TCS, a hardware engineer group has already worked on building wearable devices based on ultra-wideband wireless localization technology. In collaboration with Amphenol TCS hardware team, this project aims to build a web app for illustrating the data received from the devices and analyzing those data in a user-friendly manner.

¹ <https://gisanddata.maps.arcgis.com/apps/opsdashboard/index.html#/bda7594740fd40299423467b48e9ecf6>

The project objectives are:

1. To build a web application that would allow a manager to conduct contact tracing;
2. To allow the user to enter the device ID of an infected person, set parameters such as distance, duration, and timeframe, and return a list of close contact device IDs;
3. To ensure the organization, storage, and security of data from devices;
4. The web application also shows if the wearable sensor devices are working properly.

2. Background

This chapter reviews the coronavirus outbreak and the policies that have been attempted to prevent the spread of the virus. Next, it explains the effectiveness of social distancing, and how it protects people from the global pandemic. After that, several indoor positioning techniques and the hardware devices used in this project are discussed. Other existing dashboards are also reviewed as references and the terminology used in this project is explained. This chapter is concluded by introducing our project sponsor Amphenol TCS.

2.1 COVID-19

2.1.1 Coronavirus Outbreak

Since the first cases of individuals with pneumonia of unknown cause were reported by the World Health Organization (WHO) China National Office in late December 2019 (WHO, 2020), more than 11 million people worldwide have been infected with coronavirus disease (COVID-19) as of this report on July 6, 2020 (worldometer.info, 2020). On January 21, 2020, the first case of COVID-19 in the United States was identified in Washington State. On March 26, the U.S. became the country with the most confirmed coronavirus cases. In response to the rapid spread of the virus, on March 23, the governor of Washington Jay Inslee issued a statewide stay-at-home order to extend at least two weeks. Two weeks later, more stringent prohibitions were imposed and the stay-at-home order was extended to last at least to May 31 (O’Sullivan, 2020). Similar prohibitions have been enacted in other U.S. states as well as in countries in Europe (Schnirring, 2020).

2.1.2 Symptoms and How it Spreads

At first, no one had antidotes for the novel coronavirus. Thus, everyone was at the risk of catching and spreading it. The virus can be transmitted through droplets after coughing or sneezing and can be taken in by people nearby through noses, mouths, and eyes (cdc.gov, 2020). Evidence also proves that the virus can live in the air for 30 minutes and on the surface from hours to days. The average time interval between the first infection and the first appearance of symptoms is 5 days but is also dependent on various factors. Some people may be infected but do not show any symptoms for 3 weeks or longer (WHO, 2020). Some initial symptoms include dry cough, sore throat, shortness of breath, and fever. Symptoms get more severe when the virus enters the lower respiratory system. COVID-19 can be deadly to the elderly people, while 80% of the cases are mild to moderate and people recover in one or two weeks.

2.1.3 Social Distancing

To reduce the spread of coronavirus, the U.S. government implements various types of social distancing measures. Though researchers have proven the effectiveness of restrictive social distancing measures such as isolation and quarantine (Nussbaumer-Streit et al., 2020), these measures are also economically disruptive in a time that most of the businesses and non-essential retail stores have to close due to the restrictions. Thinking against the idea that quarantine and business closures are a “tradeoff” between public health and economic health, many employers reopen their businesses, and employees are also going back to workplaces. However, the pandemic is not over yet. People inevitably need to interact with their co-workers, clients, and many other individuals in their day-to-day life at work, yet these simple interactions violating social distancing principals expose individuals to a higher risk of being infected by the virus. In the wake of this

contagious virus, how to maintain an effective physical distance among people in workplaces, therefore, has become an urgent issue.

2.2 Indoor Positioning and Tracking

To monitor contacts among employees at the workplace, an indoor positioning and tracking system are required. Compared to outdoor navigation techniques such as Global Positioning System (GPS) and cellular localization, indoor positioning is still a frontier area. In an indoor environment, the position of an object can be localized through Bluetooth Beacons, Ultra-Wideband, Wi-Fi, cameras, etc. Generally, cameras are suitable for motion detection and accurate positioning. Under certain circumstances where the position of thousands of employees needs to be detected and recorded, however, calculating the distance between people in every frame of camera view will add unnecessary complexity to the project. Moreover, camera-based techniques may cause privacy concerns. Wi-Fi localization is also not the ideal solution for the project. As the Wi-Fi signals may be absorbed or reflected by the walls and equipment in the factories, the strength of the Wi-Fi signal at the certain spot in the room is unproportional to the distance from that spot to the signal receiver. As a result, the distance between the spot and the receiver cannot be accurately measured by Wi-Fi sensors. Considering that precise distance is required for keeping social distance, large measurement errors in distance are not acceptable in this project.

2.2.1 Bluetooth Beacon Technique

Bluetooth Beacon technique, also known as low-energy-bluetooth Beacon technique, is used for indoor navigation. Similar to a lighthouse, Beacon is a wireless device that sends low-energy-bluetooth signals to the devices within its range. Unlike classic bluetooth, bluetooth low

energy (BLE) remains in sleep mode before a connection is initialized, therefore it consumes less power (Li, 2015). As the beacon sends data only, it will have no access to the information stored in the receivers, which protects the privacy of the receivers.

Since BLE Beacon can work without the Internet connection, it can stand alone without deploying other communication means to transmit or receive signals between two or more devices. Nowadays, some Beacons are connected to other communication techniques for remote update or sending the data package. More importantly, BLE Beacon measures Received Signal Strength Indicator (RSSI), which is required by the line of sight from the devices' antenna, to detect the distance between two objects. However, as the signal sent by Bluetooth Low Energy can be absorbed by the human body and other obstacles. The error margin of Bluetooth is from 2m to 5m (Dahlgren & Mahmood, 2014). To improve the precision, multiple Beacons should be employed with triangulation technique.

Beacon is power-efficient, and its battery life is relatively long. Based on the functions and customer requirements, Beacon sensors are produced in many sizes. If the range of the Beacon is far, it requires more power. If its range is seven meters, its duration is six months. If its range is two meters, it can last for two years. Its frequency of sending the package also influences the power consumption.

2.2.2 UWB Technique

Ultra Wideband (UWB) is a small-range wireless technology that covers a wide operational frequency band. Impulse radio, the traditional UWB, was invented in the war time. In February 2002, the Federal Communication Commission (FCC) released an amendment (IEEE 802.15.4a) and specified the regulations of UWB transmission and reception. Later, Multi-band OFDM (MBOA) and Direct Sequence UWB (DS-UWB) were invented. However, it is still one of the

superior techniques in the wireless communication area that has not been widely used in consumer applications until recent years. The UWB wireless sensor is expected to consume little power and to be relatively energy-efficient.

UWB can be used for accurate indoor positioning. UWB has the operation frequency from 3.4 G to 10.6 GHz. As it has a lot of spectra, no less than 500MHz, it can send fast impulse. Since the UWB communication system can reach several GBps, it has been employed for high data rate local transfer such as wireless USB. Similar to radar, UWB can measure the time taken for a signal to travel through a medium, also known as Time of flight (ToF), and localize the UWB sensor with high accuracy. Thus, it is more accurate than the Bluetooth.

UWB has a low possibility of interception (LPI) and low possibility of detection (LPD). As we mentioned before, UWB has a wide operational frequency band, from 3.4 G to 10.6 GHz. The carrier frequency of cellular is at about 2.1 GHz; the carrier frequency of Bluetooth is at 2.4 GHz; Wi-Fi is at either 2.4 GHz or 5 GHz (Pahlavan, 2005). Therefore, the signals sent by UWB will only be interfered with 5G Wi-Fi. This problem can also be solved by modulation and coding.

2.3 Equipments for Indoor Localization and Tracking

In this project, our sponsor Amphenol TCS decides to apply UWB technique to monitor the close contact among employees. As UWB is more accurate for indoor positioning than BLE, the distance between employees can be measured more precisely. This section introduces the wearable devices we used in the project for indoor localization, their functionalities, and their specifications.

2.3.1 DWM1001

DWM1001 is an IEEE 802.15.4 UWB implementation produced by Decawave based on Decawave's DW1000 Ultra Wideband transceiver IC. It integrates UWB, Bluetooth antenna, RF circuitry, Nordic Semiconductor nRF52832 and a three-axis motion detector. It can localize the object within 5 cm precision. It supports data rates at 110kps, 850kps, 6.8Mbps. It is power-efficient that its optimised current consumption for low power sleep mode is less than 15 μ A. Its supply voltage ranges between 2.8V to 3.6V. It can cover a 100 meter line-of-sight region. Its high immunity to multipath fading also allows it to work in high-fading areas.

2.3.2 Woxu UWB Wristband Tag UT-206

Produced by Nanjing Woxu Wireless Co. Ltd, UT-206 is the wristband tag built on the chip of DWM 1001. It is shown in Figure 2.1. Using 802.15.4a UWB technique, the tag can be used to localize an object or a person to the accuracy within 5 cm. It is also able to be detected in a range up to 100m. It contains a 600mAH lithium battery and is able to vibrate for alert function. Displayed in one of the sample videos, the wristband tag can be set to vibrate when the distance between two people is less than 2m to remind people to keep social distance.



Figure 2.1 *Woxu UWB Wristband Tag UT-206*²

² <https://www.decawave.com/woxu-wireless-company-ltd/>

2.4 Existing Web Apps to Monitor and Analyze Social Distancing

Besides the equipment has been selected for social distance measurement, a web application is required as a dashboard that can fulfill the social distance monitoring task requirements at Amphenol. Currently, there are some existing contact tracing web apps in the market. We will briefly review two apps, Uбудu and sormas, that we can use as references when building our own web app.

2.4.1 Uбудu

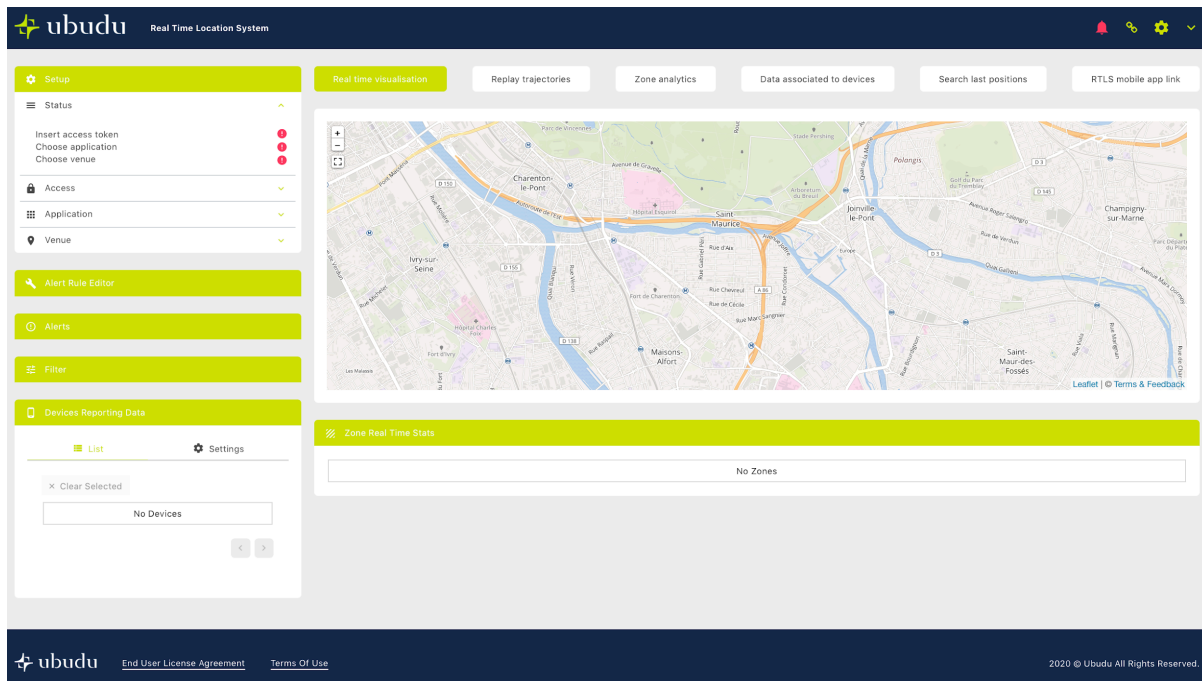


Figure 2.2 The Screenshot of Uбудu Dashboard³

Uбудu is a real-time location system that can display the real-time location of a device and replay its trajectory in the past. To activate the tracking system, users need to enter access tokens

³ <https://traject.ubudu.com/#/>

by name and venue. Then the real-time location of the device will show on the map. If users would like to replay the trajectories, they can click the second icon ‘replay trajectories’ on the top row and enter some values to change the setting. Besides, users can also label the devices with some external information.

2.4.2 Sormas

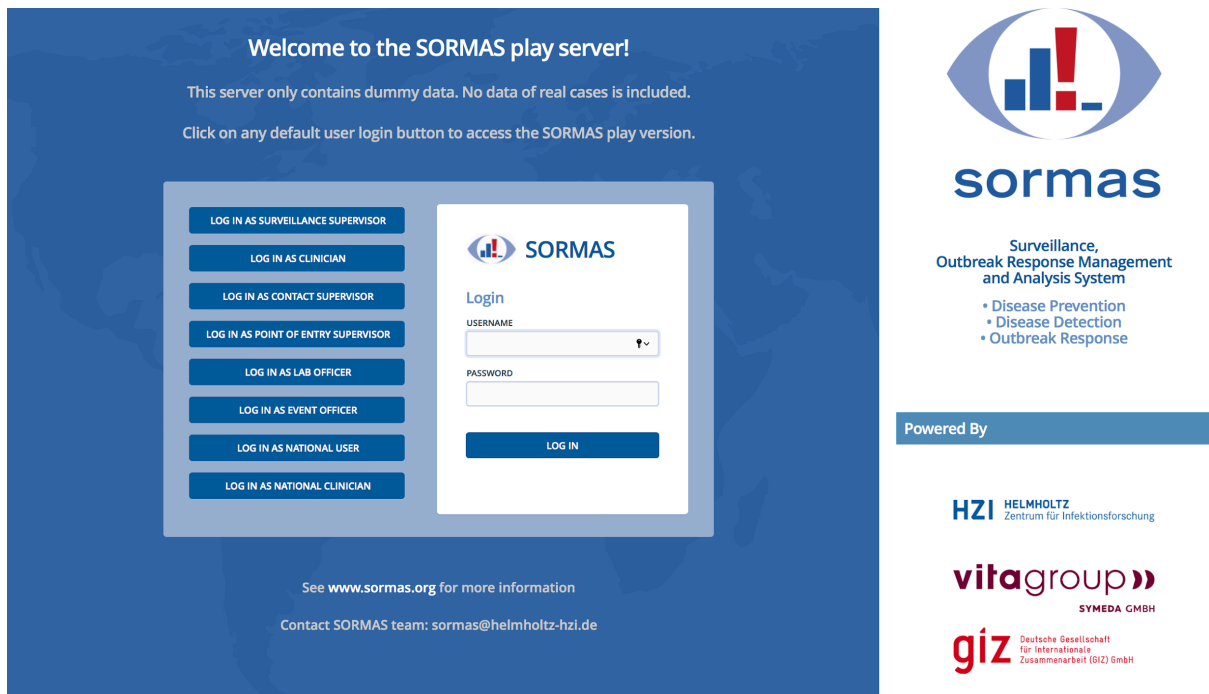


Figure 2.3 Sormas⁴

Sormas is a web application published by Helmholtz Zentrum für Infektionsforschung for disease prevention, disease detection, and outbreak response. Before accessing the data, users can choose the corresponding identification to login in. Those with higher authorities can access more complete data, edit the reported events if something was not recorded properly and generate a report before the public are permitted to read it.

⁴ <https://sormasorg.helmholtz-hzi.de/sormas-demo.html>

After logging in as supervisors, users are able to access several functions, which include tasks, cases, contacts, events, samples, reports and etc. The home page illustrates the surveillance dashboard and contact tracing dashboard, as shown in Figure 2.4. The data are listed in the table and the diagram for the processed data on the right enables users to compare the number of cases of each disease.

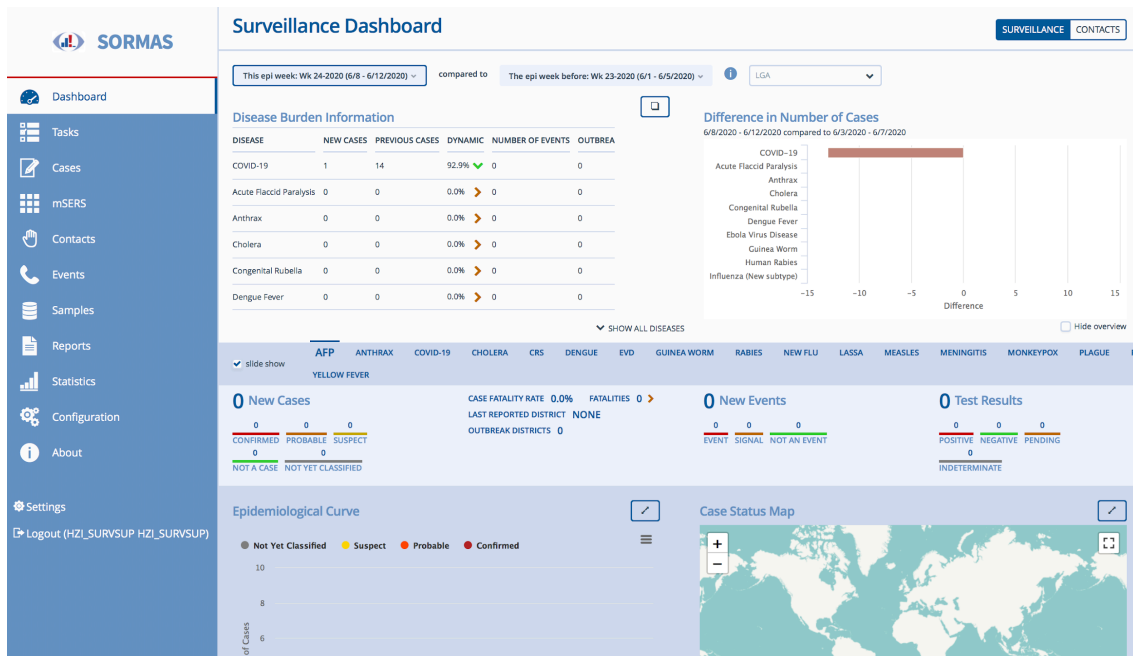


Figure 2.4 Sormas Dashboard

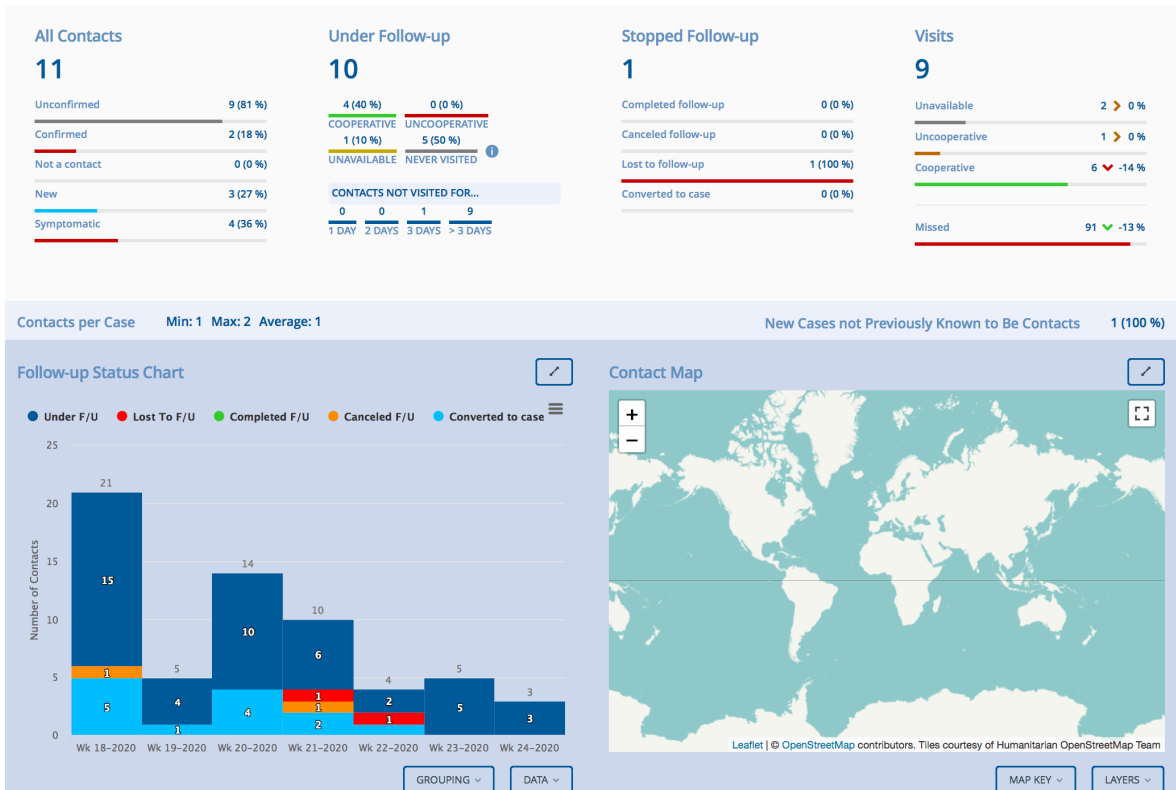


Figure 2.5 Sormas Contact Dashboard

In terms of the contact dashboard, people who are suspected to have contact with the target are grouped as either confirmed or unconfirmed as shown in Figure 2.5. If someone is unconfirmed, more follow-ups are available. The information on the dashboard is updated on a daily basis. It also allows users to see the trend of each disease.

CASE ID	EPID NUMBER	DISEASE	CASE CLASSIFICATION	OUTCOME OF CASE	INVESTIGATION STATUS	FIRST NAME	LAST NAME	DISTRICT	HEALTH FACILITY NAME
V4N7ZW	NIE-OYS-BQJ-20-001	COVID-19	Suspect case	No Outcome Yet	Investigation pending	ham	Ako	Ibadan North	Home or other place - EZM26, Itabale olugbode area.
XXK5FP	NIE-OYS-FHF-20-022	COVID-19	Suspect case	No Outcome Yet	Investigation pending	Mus	Adam	Egbeda	Home or other place - Nasarawa LGA, Kano
QBQFMZ	NIE-OYS-EGB-20-002	COVID-19	Suspect case	No Outcome Yet	Investigation pending	Luku	Ade	Egbeda	Home or other place - No3A, Zone B, Irekari, street Adem
XBGGSL	NIE-OYS-DDA-20-001	COVID-19	Suspect case	No Outcome Yet	Investigation pending	Ori	Yin	Ido	Home or other place - N
RUW6GO	NIE-BO5-ASU-20-001	COVID-19	Not yet classified	No Outcome Yet	Investigation pending	benjamin	elais	Askira/Uba	EYN Dispensary (Askira/Uba)
SUZ5SX	NIE-ADS-GMB-20-003	COVID-19	Not yet classified	No Outcome Yet	Investigation pending	test	test	Gombi	Baure Health Post
VYCF5Z	NIE-ZAS-ANK-20-006	COVID-19	Confirmed case	No Outcome Yet	Investigation done	Christina	Kampf	Anka	Home or other place - Zuhausa
R75556	NIE-KNS-FGE-20-003	COVID-19	Suspect case	No Outcome Yet	Investigation pending	ADO	ADAMU	Fagge	Sheikh M Jidda General Hospital
UT775L	NIE-KNS-NSR-20-005	COVID-19	Suspect case	No Outcome Yet	Investigation done	Abubakar	Mohammed	Nassarawa	Home or other place - eHealth Clinic
U7E0FO	NIE-KNS-NSR-20-006	COVID-19	Suspect case	No Outcome Yet	Investigation done	Abubakar	Mohammed	Nassarawa	Home or other place - EHA Clinics
WPF2UX	NIE-BAS-BAU-20-005	COVID-19	Confirmed case	No Outcome Yet	Investigation done	Hallima	Amina	Bauchi	Tashan Babye Comprehensive Health Centre
QEABBK	NIE-KNS-AJG-20-004	COVID-19	Not yet classified	No Outcome Yet	Investigation pending	maryam	lawan	Ajngi	Home or other place - ajngi primary health care

Figure 2.6 Sormas Case Directory

Next is the case directory, shown in Figure 2.6. Click the tab “cases” in the left column. The case directory records the case ID, the ID of a suspected person, their case classification, outcome of case, and investigation status. Users can filter out certain cases within a specific time period or find the status of a specific person by entering the corresponding information. New cases can be created on the top-right corner. Users can also export all the cases to a .csv file, if they need to read the data or analyze the data offline. New cases can be created on the top-right corner. Contact Directory is similar to case directory. It records the contact between two people and categorizes people by the contact type they had with the target person.

In conclusion, Sormas system is a complete platform to record suspected people who may have disease and record their contact with others in a country. However, it seems that all data have to be entered manually as shown in Figure 2.7

The image shows a 'Create new case' form with the following fields and options:

- CASE ORIGIN ***: Radio buttons for 'IN-COUNTRY' and 'POINT OF ENTRY'.
- DATE OF REPORT ***: Date picker.
- EPID NUMBER**: Text input field.
- DISEASE ***: Dropdown menu.
- REGION ***: Dropdown menu with 'Abia' selected.
- DISTRICT ***: Dropdown menu.
- COMMUNITY**: Dropdown menu.
- HEALTH FACILITY ***: Dropdown menu.
- FIRST NAME ***: Text input field.
- LAST NAME ***: Text input field.
- DATE OF BIRTH (MONTH / DAY / YEAR)**: Three dropdown menus for 'Year', 'Month', and 'Day'.
- SEX**: Dropdown menu.
- PRESENT CONDITION OF PERSON**: Dropdown menu.
- DATE OF SYMPTOM ONSET**: Date picker.

At the bottom right of the form are two buttons: 'DISCARD' and 'SAVE'.

Figure 2.7 Create New Case

2.5 Frontend

In software development, frontend is the part that clients can view and interact with. Basically, it consists of the graphical user interface (GUI) and the command line. The frontend elements include the navigating menus, texts, graphs, videos and the website designs.

Frontend is written by markup and web languages, such as Hyper Text Markup Language (HTML), Cascading Style Sheets (CSS) and JavaScript. HTML is used to define the basic format and contents. Assisting HTML, CSS can control layout, colors and fonts and JavaScript can read input and respond correspondingly.

2.6 Backend

Backend, also known as server-side, is how any application or a set of applications connect to the Internet and to service client requests. Generally, we use the word “backend” to refer to databases, web servers, and the applications. The relevant background knowledge about the backend is reviewed below.

2.6.1 Database

Database is a set of structured data stored in a computer. Commonly, a database management system (DBMS) is used to access databases, manipulate data and organize the representations of data.

There are four major DBMS types, hierarchical DBMS, network DBMS, relational DBMS and object oriented relational DBMS. Hierarchical DBMS is the most traditional type. Its structure is like a tree. The upper layer represents the parent layer, and the lower layer represents the child layer. Each child only has one parent. As its structure is simple and little variety can be provided, it is rarely used to support complex databases. Compared to hierarchical DBMS, Network DBMS offers more flexibility. It allows a child to have multiple parents. The third type, also known as the most widely used type of DBMS, is the relational DBMS (RDBMS). Typically, data is stored in two dimensions, columns and rows. It is loosely linked regardless of where in the hierarchy. Structured Query Language (SQL) is created for this purpose. SQL is the standard programming language to build a relational database that allows the update and retrieval of structured data. Some extension versions of SQL include mySQL, Oracle, Microsoft SQL server, and Sybase. The fourth type is object oriented relational DBMS. Besides the advantage of RDBMS, it bridges the gap between relational databases and the object-oriented modeling techniques used in programming

languages, such as Java and C language. Its flexibility leaves developers more space to extend data models and to custom their own data types and methods with attributes. PostgreSQL is an example of object oriented relational DBMS.

2.6.2 Web Servers

A web server is a hardware or software that satisfies client requests on the World Wide Web (WWW). Briefly, it stores websites, processes data and delivers pages to users. Web servers can be built from scratch, but most often we deploy as a service by the cloud providers, such as Heroku, Microsoft Azure, Google Cloud Platform (GCP) and Amazon Web Services.

Suggested by our sponsor and advisor, Amazon Web Services (AWS) and Heroku are preferred because of its popularity and convenience. Amazon Web Services provides a cloud-computing platform and APIs to individuals, developers, companies and the government. To be more specific, it provides AWS Elastic Beanstalk (Platform as a Service/PaaS), Amazon Simple Storage Service (AWS S3), Relational Database (AWS RDS), AWS Lambda (a serverless computing service), and other services. For developers, the monthly charge starts from \$29; for business, the monthly charge starts from \$100. The price can be higher than the minimum cost and may increase based on the usage.

Heroku is a cloud platform supporting deploying, scaling and delivering the apps. It supports Node.js, Ruby, Java, PHP, Python, Go, Scala and Clojure. It uses AWS, so it is more expensive than AWS. Some functions provided by Heroku can be used to replace AWS Elastic Beanstalk, AWS S3 and AWS RDS. However, it is more user-friendly and suitable for beginners to start with.

2.7 About the Sponsor Amphenol TCS

Amphenol TCS (ATCS), founded in 1968 as a division of Teradyne, Inc. and acquired by Amphenol Corporation in December 2005, is a global leader in high-speed, high-density connection systems. The company designs and manufactures connectors and backplane systems for application in networking, communications, storage, and computer markets. Amphenol TCS has a facility located in Nashua, New Hampshire. Our project is therefore subjected to the relevant policies in the State of New Hampshire.

3. Methodology

The chapter explains the technical methodology in detail. In each section, we briefly describe what the team did, how the team achieved it and what platform, code or packages are used.

3.1 Simulation Program

In order to evaluate the functionality of our program, we would need test data from the UWB contact tracing devices that were still in development by our project sponsor. Thus, before the real data are provided, a simulation program is required to fill in the database for running and testing the web app. We use Java to simulate the random movement of employees within a given area. If a person is within 6 feet of another person, that is defined as a close contact. The relevant data of the contact is collected.

Shown in Figure 3.1, the program scatters 50 workers on a hypothetical 500-by-500 square feet factory floor with random x and y coordinates. For every time step (1 second) in the simulator, each worker will move towards a random angle, 0 to 359 degrees, with a displacement of 0 to 4 feet. During each time step, the simulator also checks for close contacts. Once a worker comes within six feet of another worker, both of them will be added to the list of the active close contact. The simulator then creates a new case of close contact, which contains two device IDs, the current time, date, duration of the contact (in seconds) and battery life. The duration of the contact will be updated, as long as the two people are still in close contact. By the end of the contact, the latest data will be sent to the database. The people will be removed from the list of active close contacts.

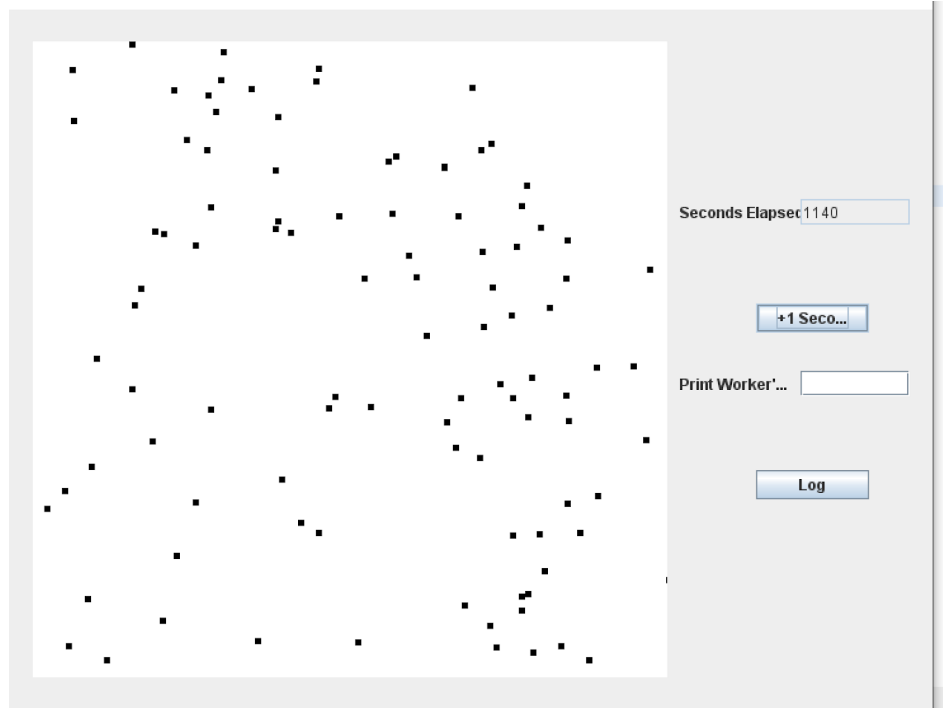


Figure 3.1 Simulation Plot

The simulation utilizes Entity Boundary Control (EBC) design. In the EBC design, actors or participants interact with the given boundary, and then the controllers send the information from the boundaries to the entities. The boundary is the screen. The actors will interact with the buttons on the panel, which will trigger methods in the simulation to run. In this case, we see the buttons act as our boundary, and then our controllers would tell the entities, the employees, to call some sort of function to move a step.

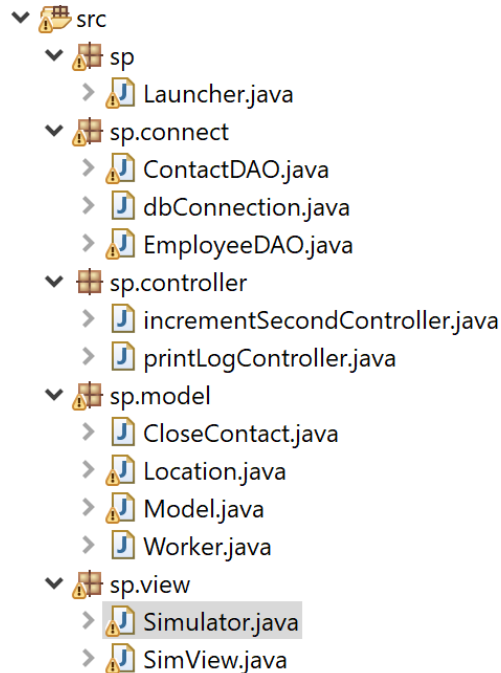


Figure 3.2 Simulation Program

According to the EBC design, the simulation program is split into 4 main parts, the connect, the controller, the model and the view. First of all, the connect package sets up a connection to the database. When new employees are added or a close contact is detected, the relevant data will be sent to the database directly. In addition, the purpose of our controller is to allow the program to take the actions of the user and invoke function calls of the base entities. As we only have two buttons, two controllers is sufficient. Furthermore, the model package holds all the low-level entities. Our lower-level entities consist of *CloseContact.java*, *Location.java* and *Worker.java*. *Location.java* has the function that allows employees to move. Each employee's location is defined by two integers, an x value and a y value. The system picks a random direction and a random speed, and then uses trigonometric functions to calculate the new location based on the given direction and magnitude. *CloseContact.java* is an object that has the *employee IDs*, the time the contact began, the time the contact ended and the *duration* of the contact.

Worker.java has the *employee ID*, a list of all closed and open close contacts and a position. Finally, a model entity ties all of the functions together. It has an integer that counts how many seconds in the simulation it is, list of workers. The model has a function that increments time. Everytime the function is called, it evokes each worker to move. After that, it checks to see which workers are within six feet of each other. If there are any workers within six feet, it either adds a new close contact to that worker’s list of close contacts or extends the duration to an unclosed close contact.

3.2 Database

Regarding the database, we use SQL to create two tables to store the attributes of close contacts and status of the devices. The initial data in the database were produced from the simulators or are assigned with random value manually. After we received an excel sheet from Amphenol, those data are replaced with the simulated data. Later, the managers can upload the .csv file on the web pages to update the database. More information about how the database is connected to the web application is discussed in Chapter 3.2.6.

Number	Date	Reporting Time	Reporting Device ID	Battery%	Contact Time	Contact Device ID	Contact Times	Contact Duration(s)
1	2020/6/24	18:56:43	20F3C	44%	18:51:48	210FF	1	155
2	2020/6/24	18:56:40	20F3C	45%	18:51:02	21186	1	192
3	2020/6/24	18:54:32	20F3C	45%	18:50:44	21181	1	61

Figure 3.3 Raw Sample Contact Info

The first table lists the case ID, date, reporting device ID, battery level, contact time and contact duration. The raw sample data we received from Woxu is shown in Figure 3.3. The raw date format is written in YYYY/MM/DD, and the reporting time is written in HH:MM:SS based

on a 24-hour format.

The raw data can be converted from .xlsx to .csv, if necessary. The date format will be converted to US standard date format MM/DD/YYYY and the “second” in *contact time* will be ignored.

```
id,reportingtime,reportingtag,battery,startat,contacttag,contacttime,duration
1,6/24/2020 18:56,20F3C,44%,6/24/2020 18:51,210FF,1,155
2,6/24/2020 18:56,20F3C,45%,6/24/2020 18:51,21186,1,192
3,6/24/2020 18:54,20F3C,45%,6/24/2020 18:50,21181,1,61
```

Figure 3.4 .csv Contact Info

The other table lists the device IDs and corresponding battery level. A fully charged device will have 100% battery level, and a device that is completely out of power will have 0% battery level. While testing, the device ID and battery level are inserted manually, shown in Figure 3.5. Some of the devices have high battery levels, while the others have relatively low battery levels. As 20% is the boundary to distinguish whether the device needed to be charged. We have both the values slightly higher than it and the values slightly less than 20%.

Device ID	Battery
213C4	18%
2101A	15%
2637E	20%
207BD	21%
22A01	95%
210FF	70%
2118C	80%
2116A	100%

Figure 3.5 Sample Device IDs and Battery Level

3.3 Web Application

3.3.1 Introduction

This section is dedicated to introduce the Dashboard we designed for Amphenol. First, we share the reasons that made us decide to design this web application; we discuss the company's requirements and expectations, and how we settled on the initial structure of the app. Then we introduce main features of the application by showing how they work as well as explaining how they are implemented behind the scene. To make technical content more straightforward and tangible, we use plenty of code snippets borrowed directly from the source code to show exactly how we wrote the code to perform certain tasks. By doing so, we ensure we have covered all the features in the web application. And this also lets anyone who needs to work with this program in the future have a better understanding of our work. We conclude by comparing the company's expectations, our initial structure, and the final product we delivered. Though the basic structure is set up and we enjoy the overall process of designing and implementing the application, some features have not been implemented successfully by the end of this project. Partly this is due to the time limit of a seven-week project, and there are also occasions that we are not able to find competent solutions. For each of these unfinished parts, we share our recommendation on possible implementation and why we believe this specific feature is important for the application.

3.3.2 Objectives and Expectations

This project aims to produce a contact tracing dashboard for human resource managers in a manufacturing facility. During the project, we communicated with Amphenol and read the

documents from them. Based on their requirements, the web application should: (1) allow a manager to conduct contact tracing; (2) provide analytics for case tracking and other services; and (3) assure data security and monitor the wearable devices' working status.

First of all, the application should be able to display the close contacts of people within a day. Due to safety and privacy concerns, devices will not record the location of each employee. Instead, it records the event of close contacts that are closer than 6 feet between two employees. The information obtained from the devices include the time when two individuals meet, the duration of each contact, and how many times they meet in total within the day; any information regarding the distance or location is unattainable. Therefore, the primary objective of our web application is to display all the close contact events captured by the devices. Each record is generated with a unique case ID, the device ID of the reporting device, the device ID of the contact device, the date and time, as well as the duration.

In addition to generating and displaying data, the dashboard should support analytical purposes as well. According to the discussion within our sponsor, analytics features range from the basic filtering and sorting to diagrams and more advanced data visualizations. The total contact duration and times within a day can be summarized and calculated from the raw contact case data. For example, if person A and person B meet at 9:30 am and the contact lasts for 5 minutes, meet at 12:00 pm and the contact lasts for 10 minutes, and meet at 4:00 pm and the contact lasts for 3 minutes, the dashboard should display all three cases and conclude that they have met three times and the overall duration is 18 minutes for that day.

Furthermore, the application should take security into account. One key consideration of Amphenol is their employees' privacy. Thus, we decide not to include any data involved in the actual contact distance, specific location, or any other employees' personal information when

displaying data on the web application. Instead, their names are replaced by their device IDs. In addition to privacy, accuracy of data is also critical. In order to assure all the data on the application is accurate, we need to assure the devices are working properly.

Besides the explicit requirements from the company, some implicit functionalities should be fulfilled. For instance, the web application should be deployed to a cloud server that allows users to access it on the Internet; all the data should be stored in a database; before the hardware team provides us with actual data, a simulator is required to generate test data for the web application to test its features.

3.3.3 Initial Architecture and Tech Stack

After lining out all the requirements of the web application, we come up with the initial outline of the application and tech stacks we plan to use.

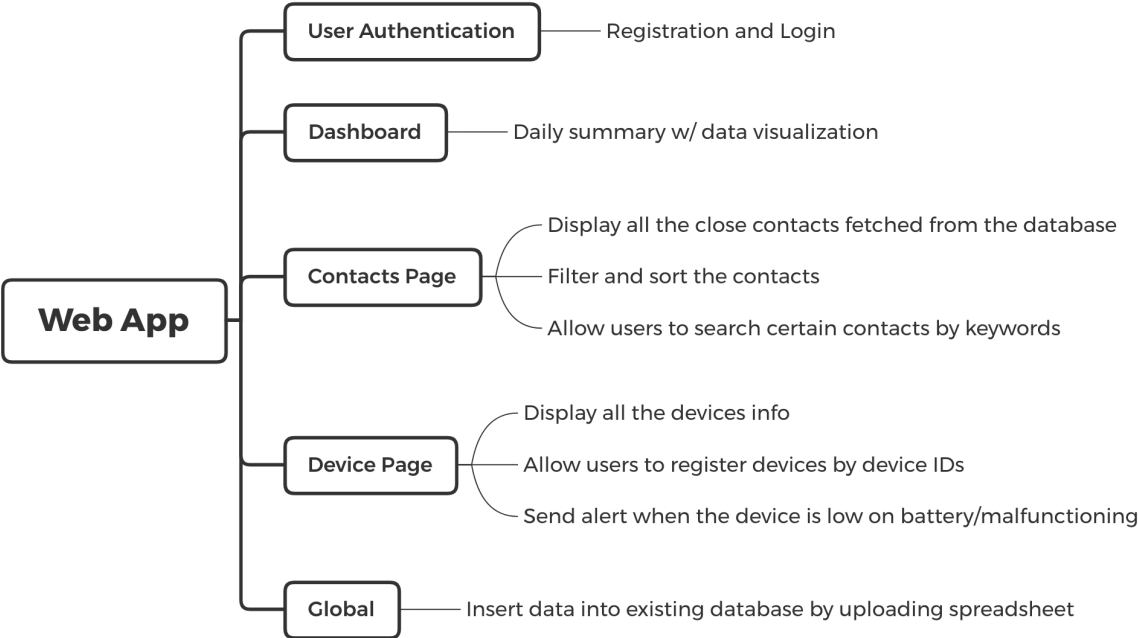


Figure 3.6 Initial Architecture of the Web Application

As shown in Figure 3.6, we group all the requirements into small chunks and assign each chunk to a dedicated web page. By doing so, the general structure becomes clear to us, and it also makes tracing the progress of development and debugging easy.

In terms of tech stack, we use HTML/CSS and JavaScript when developing the frontend of our application. To make the design process easier, we also make use of Bootstrap, one of the most popular CSS frameworks, and EJS, an template engine used to render HTML content. In addition to the framework, we also write our own CSS file to customize our web application to cater to needs specific to our project. For the backend, we use Node.js. Although Python is also a handy tool when it comes to web development and is gaining a lot of popularity over the past few years, Node.js, as a JavaScript runtime engine, allows us to avoid dealing with different semantics and syntax of different languages as we have already been working on the frontend using JavaScript. Node.js also has a huge variety of packages that assist programmers to perform certain tasks without writing code from scratch and a very reliable package manager, npm. These powerful tools prevent us from reinventing the wheel and focus on designing and implementing exciting features for our web application instead. During the process of development, we utilize several handy JavaScript packages to accelerate the progress and make our program more robust. The details will be covered when we introduce the specific lines of code.

To build a web application, simply knowing how to generate a website to display content is not enough, we also need a database to store the data. At first, we chose SQLite, an open source database that implements most of the SQL standards. However, in the middle of development, we decided to abandon SQLite and migrate all the existing work to a similar database, PostgreSQL. Though doing so means to drop all the previous effort we have put into working with SQLite, we had to make this decision since SQLite does not support cloud service

as all of the data in it is stored in files, similar to Microsoft Excel Worksheet. It was acceptable when we were only accessing the web application from our local machines. However, since we want to deploy the web to a cloud platform to let people use it on the internet, we need to populate our database to a cloud service. Though migrating the whole database to a completely new database service sounds daunting, the actual process is quite simple, which is partially thanks to another tool we use, Sequelize. Sequelize is an Object-Relational Mapping (ORM) programming technique that is promise-based and specific to Node.js. This technique is used for converting data between incompatible type systems using object-oriented programming languages, and Sequelize supports both SQLite and PostgreSQL. Therefore, in our case, at first we set up data models using Sequelize and used Sequelize to connect to SQLite local server, then we migrate to PostgreSQL, as shown in Figure 3.7. We only need to change the dialect option to PostgreSQL and pass username and password used to access the PostgreSQL to the function, and any other work such as transforming data tables to adjust the new database will be taken care by Sequelize.

```
const Sequelize = require('sequelize')

module.exports = new Sequelize(DATABASE_NAME, USERNAME, PASSWORD, {
  host: HOST_ADDRESS,
  port: 5432,
  logging: console.log,
  maxConcurrentQueries: 100,
  dialect: 'postgres',
  dialectOptions: {
    ssl: {
      require: true,
      rejectUnauthorized: false
    }
  },
  pool: {
    maxConnections: 5,
    maxIdleTime: 30
  },
  language: 'en'
})
```

Figure 3.7 Sequelize Function to connect to PostgreSQL

In addition to SQLite and PostgreSQL, Sequelize also supports other mainstream database services such as MySQL, MariaDB, etc. This makes it easy for our sponsor if they decide to change database service in the future. As the basic setup is similar to what is included in the code snippet above, any other language-specific instructions can be found in its official documentation (sequelize.org, 2020).

Above is the basic tech stack we choose to use to build our web application. There are more frameworks and packages used, and we will introduce them when we cover specific features of the application with the corresponding code snippets.

3.3.4 User Authentication

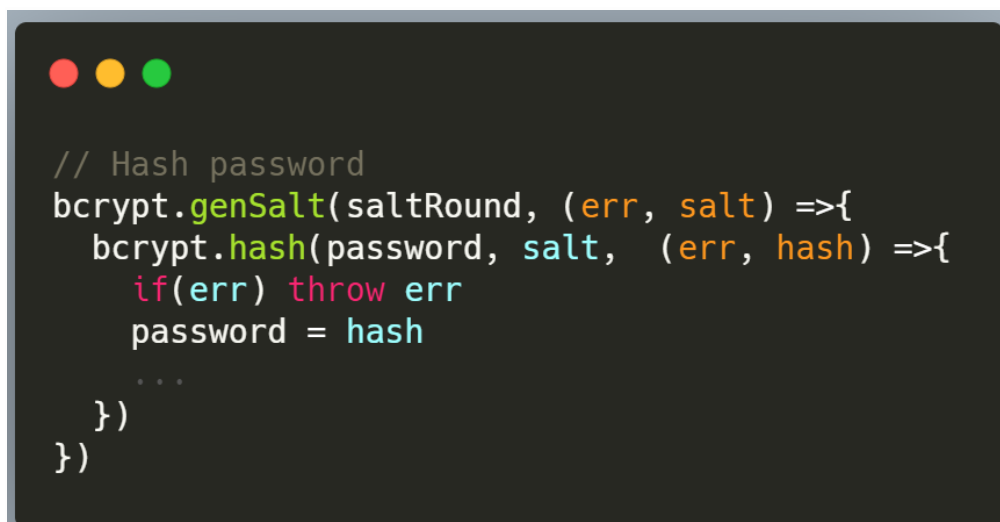
User authentication pages, including the user registration page and login page, are the first feature we implemented. According to the information from the company, the web application is mostly used by only internal employees, yet we still decide to ask users to go through identity verification every time when they access the content of the application. We believe it is wise to separate the public from the content of the application and to protect employees' privacy. In addition to this, even for internal employees, we still want the company to be able to control who are allowed to access certain content of the website, such as the information related to the employees and devices, and who are trusted to perform certain operations, such as adding and modifying data stored in the database.

In terms of techniques, we use Passport.js, authentication middleware for Node.js, to handle authentication requests. Passport.js is known for its modularity and supporting a comprehensive set of strategies, which are different methods of authentication packaged into independent modules. With the help of Passport.js, handling users' login requests can be

achieved with only a couple lines of code. But before heading to the login page, we need to set up the environment for Passport.js in our program.

For the sake of simplicity, we choose the local strategy that is the most widely-used option for websites. The majority of code used in our program is directly borrowed from the official documentation of Passport.js and is organized in a single file called *Passport.js* under the folder *config* in source code.

During the registration process, we always want to avoid storing plain text passwords, thus after the user fills out their email address and password, we use *bcrypt.js* to hash the user's password before adding the user's information into the database.

A screenshot of a code editor window with a dark background and light-colored text. The code is as follows:

```
// Hash password
bcrypt.genSalt(saltRound, (err, salt) =>{
  bcrypt.hash(password, salt, (err, hash) =>{
    if(err) throw err
    password = hash
    ...
  })
})
```

Figure 3.8 Hash Password

In addition to hashing passwords, the user authentication program is also responsible for checking if the user is using proper and unique email addresses when registering and the length of password is longer than 6 characters. If the user fails to meet any of these requirements, an alert will pop up. To do this, we first set up JavaScript Global Variables to make sure the messages we want to send are accessible anywhere in the application. Then, the program checks

each of the user-filled fields on the registration page and pushes error messages to the global variable when it detects any violations.

```
    // middleware for flash
    app.use(flash( ))

    // Global variables
    app.use((req, res, next) =>{
      res.locals.success_msg = req.flash('success_msg'),
      res.locals.error_msg = req.flash('error_msg'),
      res.locals.error = req.flash('error')
      next()
    })
```

Figure 3.9 Global Variables to Contain Messages

```
    //Check required fields
    if(!email || !name || !password || !password2){
      errors.push({ text : 'Please fill in all fields.' })
    }
    // Passwords match
    if(password != password2){
      errors.push({ text: 'Passwords do not match.' })
    }
    // Check pass length
    if (password.length < 6) {
      errors.push({ text: 'Password must be at least 6 characters' });
    }
  }
```

Figure 3.10 Check User-Filled Fields of the Registration Page

```
<% if(typeof errors != 'undefined'){ %>
  <% errors.forEach(function(error) { %>
    <div class="alert alert-warning alert-dismissible fade show" role="alert">
      <%= error.text %>
      <button type="button" class="close" data-dismiss="alert" aria-label="Close">
        <span aria-hidden="true">&times;</span>
      </button>
    </div>
  <% }); %>
<% } %>
```

Figure 3.11 Format Error Message

To display the error message, we use the template engine EJS to render the error message to the page and format it in consistent with other content on that page. To do this, first we need to format the error message. As shown in Figure 3.11, this piece of code represents how to use a template engine to display various messages. The appearance of the message is defined by the HTML code, and the content of the messages is only determined as the program runs, in our case, when the program checks what is inside in the global variable named *errors* on line 2. As the program goes through each message in *errors*, every message is independent from each other, which allows us to display multiple messages at the same time and each has different content yet in a consistent format.

```
<%= include('./partials/messages.ejs'); %>
```

Figure 3.12 Includes Messages in the Page

Finally, we can insert this one line of code (shown in Figure 3.12) to anywhere we want to see the error messages. The same process also applies to any future usage of errors and success messages in our web application.

Much simpler than registering a new user, the login program omits all the validation checks except for making sure the email and password entered by the user matches the user's registered email address and password. As passwords are hashed before storing, we need to dehash before we can compare. The whole process is basically doing the opposite of hashing a password, and in our case, Password.js provides that function.

```
// Handle login
router.post('/login', (req, res, next) => {
  passport.authenticate('local', {
    successRedirect: '/dashboard',
    failureRedirect: '/users/login',
    failureFlash: true
  })(req, res, next)
})
```

Figure 3.13 Handle User Login

3.3.5 Permission Control

Permission Control						
Name	Email	Edit Permission	Bulk Create Contacts	Bulk Create Devices	Register New Device	Update Existing Device
USER2	test@test.com	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Admin	test@dev.com	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
USER3	TEST@GMAIL.COM	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Dev	dev@test.com	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
User1	user@test.com	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Figure 3.14 Permission Control Page

In addition to basic user authentication, we also need to control users' authority to perform certain tasks within the application, especially when it comes to adding or modifying data in the database (See Chapter 3.3.6 File Uploading). Therefore, we decide to implement another layer of authentication to let the administrator control other users' access permissions. In order to achieve this goal, we add one extra column in the database where we store the *USER* table to keep track of each user's access rights. As shown in Figure 3.14, there are five access permissions, each of which is independent from others and corresponds with one function of the application. The *permission* is attached to each user once they register successfully and their record is stored into the database. By default, all permissions are set to be false, which means the user does not have access to the corresponding functionality, with only one exception. The user who registers with the email address 'test@dev.com' is set to be the administrator and is able to edit other users' permissions in the permission control page. With this ability, the administrator is able to edit all the access permissions for other users. In addition to *permissioncontrol*, there are four other types of permissions. The *addcontacts* controls the ability to add contacts data to the database by uploading data file; *adddevices*, similar to *addcontacts*, controls the ability to add devices data to the database; *updatedevices* controls the ability to edit the status of existing devices; *registerdevices* controls the ability to register new devices and add it to the database. All five permissions are independent from others.


```
// set up users' permission type
let permission = {
  "permissioncontrol": false,
  "addcontacts": false,
  "adddevices": false,
  "updatedevices": false,
  "registerdevices": false
}
```

Figure 3.15 Permission Type

3.3.6 File Uploading

At the beginning of our web application design process, the company mentions that the data captured by devices are generally stored in the format of data sheets. To avoid wasting time manually entering data row one by one, they want a way to bulk upload all the data from the sheet to the database. Therefore, we implement the features that take user-uploaded files, extract data and add them to the corresponding database. To do this, first we need to allow users to upload files in the application and make sure the application is able to accept the file. Then, the application converts the file into the desired format. Finally, we add the data into the database. Fortunately, Sequelize can add data in JSON (JavaScript Object Notation) format into the database directly, thus we only need to focus on the first two steps, taking the files as input and converting them into JSON.

```
CONTACT.bulkCreate(JSONdata)
  .then(() => res.redirect('/contacts'))
  .catch(err => res.send(err))
```

Figure 3.16 Bulk Add Data to Contact Table

During the early development when our application is solely running on the local machines, we store user-uploaded files in a local folder, and then the program fetches the files and converts the data into JSON. However, when we decide to deploy the application to Heroku, we notice that all files uploaded to Heroku server are deleted due to its ephemeral file system. To resolve this problem, we follow the official suggestion (Heroku, 2020) and use Amazon S3 (Amazon Simple Storage Service) instead. In the following two sections, we introduce separately how we implement file uploading features with both local file systems and Amazon S3.

3.3.6.1 Store Files in Local File System

For this method, we first accept files uploaded by users. We accept both .xlsx or .xls files and .csv files. No matter what format users use, they only need to upload it on the Upload Page. To get the file, we use a JavaScript third-party package named *multer* using the following lines of code. It sets *multer* up to take the file as well as tells it where we want to store the file. Then, we include *multer* as middleware in where we want to use the file, and the file can be found in *req.file*. Note that in this case we set the storage destination as *./uploads/*, which indicates it is somewhere in our local file system.

```
const storage = multer.diskStorage({
  destination: function (req, file, cb) {
    cb(null, './uploads/')
  },
  filename: function (req, file, cb) {
    cb(null, Date.now() + path.extname(file.originalname)) //Appending extension
  }
})

const upload = multer({ storage: storage })

router.post('/sheettojson/*', upload.single('test'), function(req, res, next) {
  ...
  // uploaded file can be found in req.file
})
```

Figure 3.17 Upload File with *Multer*

Then inside the router, the program reads the data from the recently uploaded files, and converts them into JSON. In this step, we handle files of .csv format and .xlsx or .xls format separately. In both cases, JSON data will be stored in the variable named *JSONdata*, and will be ready to be added to the database using Sequelize.

```
// extract data from a csv file
if(req.file.originalname.split('.')[req.file.originalname.split('.').length-1] === 'csv'){
    JSONdata = csvToJson.fieldDelimiter(',').getJSONFromCsv(req.file.path)

// extrate data from Excel sheet
if(req.file.originalname.split('.')[req.file.originalname.split('.').length-1] === 'xlsx'){
    exceltojson = xlsxtojson;
} else if (req.file.originalname.split('.')[req.file.originalname.split('.').length-1] === 'xls') {
    exceltojson = xlstojson;
}

try {
    exceltojson({
        input: `${req.file.path}`, //the same path where we uploaded our file
        output: '',
        lowerCaseHeaders: true
    }, function(err,result){
        if(err) {
            return res.json({error_code:1,err_desc:err, data: null})
        }
        JSONdata = result;
    });
}
```

Figure 3.18 Extract Date from Date Sheet and Convert it to JSON

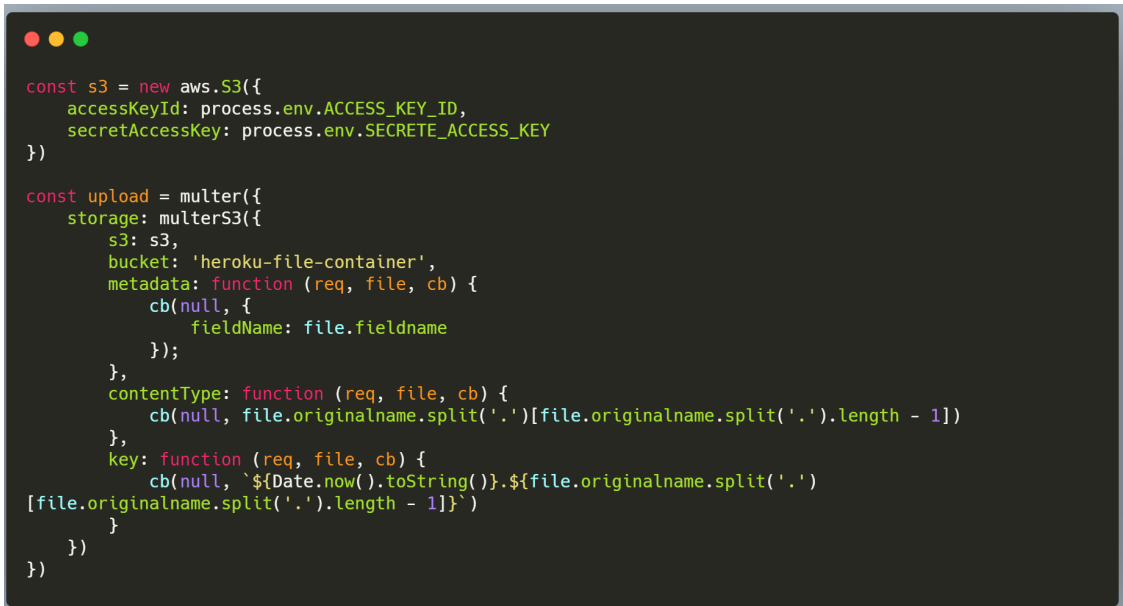
At this stage, data has been extracted from the data sheets and added to the database, yet the files uploaded by users are still in the local storage. To clean it up, this line of code deletes the files from the file system since we don't need them anymore.

```
fs.unlinkSync(req.file.path)
```

Figure 3.19 Delete Uploaded File

3.3.6.2 Store Files in Amazon S3

Similar to uploading to the local file system, we use *multer* to accept user-uploaded files. After that, instead of saving files in local storage, we upload the file to a S3 bucket. As shown in Figure 3.20, we set up Amazon S3, so that our program has access to its service, and then we set the storage destination to S3 service in *multer* upload method.



```
const s3 = new aws.S3({
  accessKeyId: process.env.ACCESS_KEY_ID,
  secretAccessKey: process.env.SECRETE_ACCESS_KEY
})

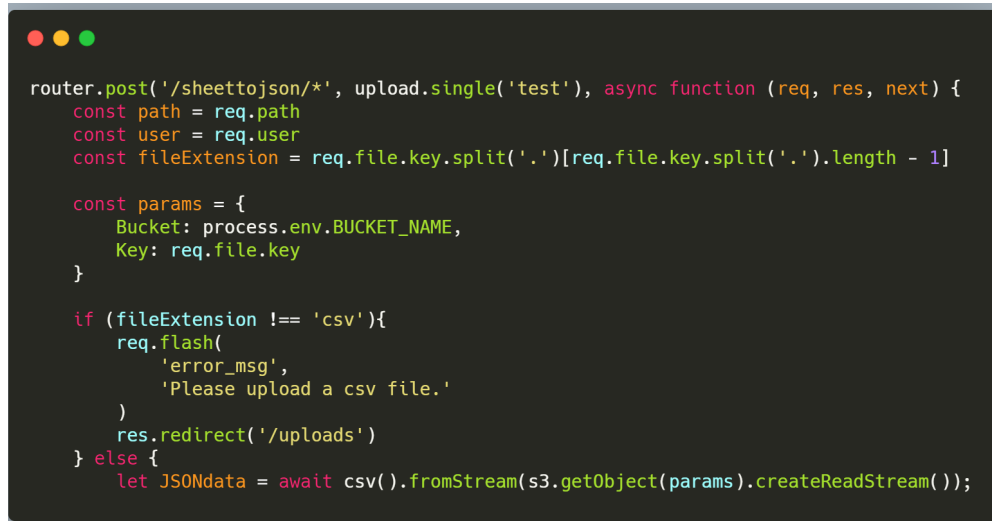
const upload = multer({
  storage: multerS3({
    s3: s3,
    bucket: 'heroku-file-container',
    metadata: function (req, file, cb) {
      cb(null, {
        fieldName: file.fieldname
      });
    },
    contentType: function (req, file, cb) {
      cb(null, file.originalname.split('.')[file.originalname.split('.').length - 1])
    },
    key: function (req, file, cb) {
      cb(null, `${Date.now().toString()}.${file.originalname.split('.')
[file.originalname.split('.').length - 1]}`)
    }
  })
})
```

Figure 3.20 Setup for Amazon S3 Service and Multer

Note that when files are uploaded to S3 bucket, their extension names will be omitted by default. By setting *contentType* and *key* in the *multer* storage method, we make sure S3 knows the type of uploaded files by extracting extension name from the user uploaded files and appending it to the file name when uploading it to S3.

Though saving user-uploaded files is straightforward, it becomes a bit tricky when we try to get the file from S3 and convert it into JSON. Since we cannot use local file systems, we have to work with streaming data instead of extracting data from the file. After several failed attempts, we realize streaming Excel data is much harder to handle than .csv data. Therefore, in our

application, we only implement how to convert streaming .csv data to JSON. For data stored in Excel format, we write an Amazon Lambda function that converts any uploaded Excel spreadsheet to .csv format before it is stored into our S3 bucket.



```
router.post('/sheettojson/*', upload.single('test'), async function (req, res, next) {
  const path = req.path
  const user = req.user
  const fileExtension = req.file.key.split('.')[req.file.key.split('.').length - 1]

  const params = {
    Bucket: process.env.BUCKET_NAME,
    Key: req.file.key
  }

  if (fileExtension !== 'csv'){
    req.flash(
      'error_msg',
      'Please upload a csv file.'
    )
    res.redirect('/uploads')
  } else {
    let JSONdata = await csv().fromStream(s3.getObject(params).createReadStream());
```

Figure 3.21 Create Readable Stream and Convert to JSON

To convert the .csv file in S3 bucket to JSON data, as shown in Figure 3.21, first we need to check if the file is in .csv format. If not, an error message pops up (refer to Chapter 3.3.4 *User Authentication* for how *req.flash* works). Next, we generate a readable stream from data in the .csv file and convert it to JSON data. Note that this time we use *async/await* in the router, that is because conversion takes time and we don't want the program to proceed before the conversion is finished. Similar to the previous section, *JSONdata* is where we store JSON data which is ready to be added to the database.

3.3.7 *Contacts Page*

On the Contacts Page, we display all the close contacts information. All data are fetched from the database, so first we need to figure out how to request data from the database and how

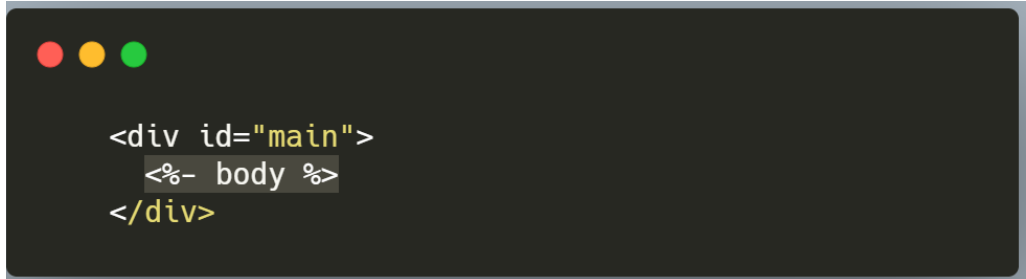
to render the data on the web page. In other words, how to let the backend program communicate with the frontend web page.

With the help of Node.js and Sequelize, the first task is straightforward. As shown in Figure 3.22, we ask Sequelize to find all the instances in *CONTACT* table, and then we let Node.js render the Contacts Page for us and send all the contacts data as an array to that page so that these data are accessible when the page is ready. Note that we also specify the layout for the Contacts Page.

```
// Contacts page
router.get('/', ensureAuthenticated, (req, res) => {
  CONTACT.findAll().then((contacttime) => {
    res.render('contacts', {
      layout: 'layoutB',
      contacttime
    })
  })
})
```

Figure 3.22 Render Data to the Contacts Page

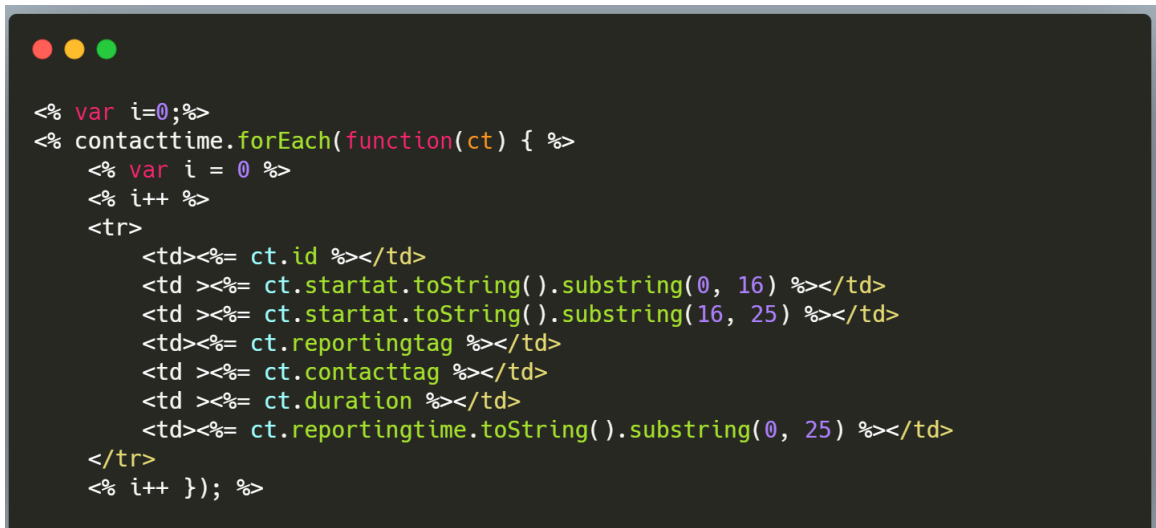
For the sake of uniformity, we use the same layout for all the pages of our web application. In *layoutB.ejs*, we define the elements that we want each web page to have in common; after that, we use the code in Figure 3.23 to tell the program where to place elements in each page. For example, in terms of Contacts Page, `<%- body %>` will be replaced by elements in *contacts.ejs*. This is a feature provided by EJS, and with the help of it, we can maintain uniformity across the website while avoiding repetitive code.



```
<div id="main">
  <%- body %>
</div>
```

Figure 3.23 Insert Page-Specific Elements

The next task is to develop the frontend program that renders data from the backend to the web page. This task is also made easy by EJS. Since it allows us to write JavaScript code inside of HTML, as shown in Figure 3.24, we go through each element in the data array sent from the backend using JavaScript method, then set how we want to display these data using HTML elements, and finally EJS renders them on the web page.



```
<% var i=0;%>
<% contacttime.forEach(function(ct) { %>
  <% var i = 0 %>
  <% i++ %>
  <tr>
    <td><%= ct.id %></td>
    <td ><%= ct.startat.toString().substring(0, 16) %></td>
    <td ><%= ct.startat.toString().substring(16, 25) %></td>
    <td><%= ct.reportingtag %></td>
    <td ><%= ct.contacttag %></td>
    <td ><%= ct.duration %></td>
    <td><%= ct.reportingtime.toString().substring(0, 25) %></td>
  </tr>
  <% i++ }); %>
```

Figure 3.24 Renders Data on Contacts Page

In addition to simply displaying all the data in the database to the web page, we also want our application to have basic filters and sorting functions. In terms of sorting, there are already many third-party packages that have added it to HTML tables. In our case, we use an open-source JavaScript library called *Data Tables* that sorting functions come with it by default. Setup of it is very straightforward by including files and several lines of code provided by its

documentation, it will automatically turn the plain HTML tables to more professional-looking data tables. Other than sorting, several filters are implemented to allow users to display certain kinds of close contacts. For example, users can choose to see close contacts of certain devices, the contacts that happened within a certain time period, as well as the contacts that last for a certain period of time.

```
<script src="https://code.jquery.com/jquery-3.5.1.js"></script>
<script type="text/javascript" src="https://cdn.datatables.net/v/bs4/dt-1.10.21/b-1.6.2/fh-3.1.7/kt-2.5.2/r-2.2.5/sc-2.0.2/sp-1.1.1/sl-1.3.1/datatables.min.js"></script>
<script type="text/javascript">
  $(document).ready(function() {
    $('#example').DataTable({
      fixedHeader: true
    });
  });
</script>
```

Figure 3.25 Setup for DataTables

In these cases, we do not send all the data from the database to the frontend. Instead, we filter out unnecessary data in the backend program and only send data that meets the user’s requirements. Thus, the frontend program remains the same since all the filtering has been taken care of by the backend.

On the Contacts Page, we add several input fields as filters. Users can fill in the form to specify how they want the data to be filtered. In addition to manually filling all the fields, blank fields will be filled in with default values, if users press Enter before entering any values.

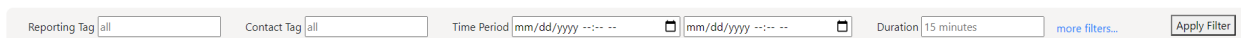


Figure 3.26 Filter Form on Contacts Page

For *Reporting Tag* and *Contact Tag*, the default value is set to be *all*, meaning the returned data includes close contacts captured by all the devices. The default value for *Duration* is 15 minutes since according to the U.S. CDC, a close contact is defined as a contact within 6 feet and lasts for at least 15 minutes (US CDC, 2020). The following is the JavaScript code that implements the autofill in the frontend program.

```
const rtag = document.getElementById('reportingtag')
const ctag = document.getElementById('contacttag')
const duration = document.getElementById('duration')
rtag.onkeypress = autofill
ctag.onkeypress = autofill
duration.onkeypress = autofill
function autofill(e) {
  event.preventDefault()
  const ele = event.path[0]
  if(event.key == 'Enter' && event.value == undefined){
    if(ele.id == 'reportingtag' || ele.id == 'contacttag'){
      ele.value = ele.placeholder
    } else if(ele.id == 'duration'){
      ele.value = 15 * 60
    }
  } else{
    ele.value += event.key
  }
}
```

Figure 3.27 Autofill Program

After users fill out the form and click *Apply Filter*, data will be sent to the backend as a data array. Data will be double checked to make sure all fields have been filled in. Then we use Sequelize to fetch the data that meets the user's requirement. Similar to displaying all the data, Contacts Page will be re-rendered, and filtered data will be sent to the page.

```
let rtag = req.body.rtag
let ctag = req.body.ctag

let start = new Date(req.body.start).toString()
let end = new Date(req.body.end).toString()

let duration = req.body.duration

rtag == 'all'? rtag = '' : rtag = rtag
ctag == 'all'? ctag = '' : ctag = ctag
duration == 'undefined'? duration = 15 * 60 : duration = duration

CONTACT.findAll({
  where: {
    reportingtag: {
      [Op.like]: `%${rtag}`
    },
    contacttag: {
      [Op.like]: `%${ctag}`
    },
    duration: {
      [Op.gte]: Number(duration)
    },
    startat: {
      [Op.between]: [start, end]
    }
  }
}).then(contacttime => {
  res.render('contacts', {
    layout: 'layoutB',
    contacttime
  })
})
)
```

Figure 3.28 Fetch and Send Filtered Data to the Frontend

3.3.8 Devices Page

Similar to the Contacts Page, Devices Page displays all the data stored in the database and supports sorting functions that come with the *Data Tables* library, but it does not have filters

as the size and complexity of devices data is much smaller than that of close contacts data. In addition to displaying data, Devices Page allows users to register new devices manually, which is similar to uploading .csv files of device data on Upload Page but only adding one device at a time. To register new devices, first click on the ADD button on Devices Page. After users fill out the form and click *Submit*, the data will be sent to the backend program and will be stored into the database if the registering *Device ID* has not been registered previously.

On the Devices Page, users can also click any *Device ID* to see detailed information of that device. The information includes status (active, malfunctioning, low batter, etc.), battery level, etc.

When a device is added to the database, default values for its status and battery level are set to be '*active*' and '*100%*', respectively. When users refresh the page, the backend program will look for the device's ID in the *CONTACT* table and update each device's status by. If the corresponding device has one more more new close contacts uploaded, the device's status will be updated based on the most recent close contact. By doing so, device information is ensured to be up-to-date whenever the page is open.

```
// update devices' battery levels based on newly added contacts info
DEVICE.findAll().then((device) =>{

  device.forEach((d) => {

    CONTACT.findAll({
      limit: 1,
      where: {
        reportingtag: d.deviceid
      },
      order: [[ 'startat', 'ASC' ]]
    }).then((contacts) => {
      if(contacts[0]){
        d.update({
          battery: contacts[0].battery
        })
      }
    })
    const batteryNumber = d.battery.substring(0, d.battery.indexOf('%'))
    console.log(batteryNumber)
    if(batteryNumber <= 20){
      d.update({
        status: 'Low Battery'
      })
    }
  })
})
```

Figure 3.29 Update Device Battery Level Based On Close Contacts Data

4. Results

The team successfully developed the web app that fulfills most of the objectives proposed by our sponsor. It allows managers to conduct contact traces and assure wearable devices are working properly. The web app also provides data analytics for asset tracking and other services.

Regarding the contact tracing function, users can access all the close contact information on Contacts Page, including the *device IDs*, *reporting time* and *duration*. Shown in Figure 4.1, all results can be sorted and filtered based on users' requirements. All the filter functions have default values. Thus, users can enter the least information. Users can also enter the relevant information in the search textbox to find the specific cases. At the bottom of the page, the number of cases is presented that may help users roughly evaluate the overall situation.

Index	Date	Contact Time	Reporting Tag	Contact Tag	Duration(s)	Reporting Time
1	Wed Jun 24 2020	18:51:48	20F3C	210FF	155	Wed Jun 24 2020 18:56:43
2	Wed Jun 24 2020	18:51:02	20F3C	21186	192	Wed Jun 24 2020 18:56:40
3	Wed Jun 24 2020	18:50:44	20F3C	21181	61	Wed Jun 24 2020 18:54:32
4	Wed Jun 24 2020	18:46:44	20F3C	2115E	295	Wed Jun 24 2020 18:54:28
5	Wed Jun 24 2020	18:50:41	20F3C	210FF	5	Wed Jun 24 2020 18:54:13
6	Wed Jun 24 2020	18:49:47	20F3C	210FF	40	Wed Jun 24 2020 18:53:38
7	Wed Jun 24 2020	18:44:21	20F3C	210C3	259	Wed Jun 24 2020 18:51:46
8	Wed Jun 24 2020	18:48:28	20F3C	210FF	8	Wed Jun 24 2020 18:51:46
9	Wed Jun 24 2020	18:45:53	20F3C	20FA4	130	Wed Jun 24 2020 18:50:39
10	Wed Jun 24 2020	23:02:48	210FF	0	0	Wed Jun 24 2020 18:50:30

Figure 4.1 Contacts Page

Similar to the Contact Page, the Devices Page shows all the device information, including the *Device ID*, *battery percentage* and *status*, shown in Figure 4.2. The battery level is automatically updated with the database. Users can check what devices need to be charged by

entering *Low Battery* in the search bar. When the battery percentage of the devices drops to 20% or lower, its status will shift from active to low battery. The *malfunctioning* status is prepared in advance. As the team can only access the battery level to check whether the devices are working properly, more details about how to define the malfunctioning status can be answered by the hardware team. In addition, if the owner of the devices changes, the admin can edit its ownership manually.

Index	Device ID	Battery	Status
0	0E764	100%	active
1	k	100%	active
2	A	18%	Low Battery
3	X	20%	Low Battery
4	K	18%	Low Battery
5	20F3C	45%	active

Figure 4.2 Devices Page

There are some changes of the specifications during this MQP project. Initially, the contact tracing dashboard should illustrate the specific routines of the users and the distance between users when they meet. However, location information is removed in order to protect employees' privacy. Without the location information, the hotspots are unknown, so the hotspots figure is omitted. All the contacts recorded are the close contacts within six feet. Moreover, we are not completely sure about how often the contacts data will be uploaded. Information from the sponsor indicates that the devices will be collected for charging at the end of each day and the

contacts data will be uploaded by then. This decision may be changed during their actual deployment. Without the real-time data, it is harder to tell whether a device is offline.

Employees' privacy and information security are also one of the highest priorities. First of all, there are several levels of permission for different people. Admins have the highest permission who can edit the permission of other users, upload csv files to the database and edit on the ownership of the devices. Only users who register and get the permission from admins can access the web application. Otherwise, they will be blocked out. Meanwhile, users will be logged out automatically if they close the window or do not respond for a long time. Second of all, all the data are stored on the reliable cloud services, AWS. The ownership of the accounts will be transferred to the developers who will continue the development of the Dashboard web application. Finally, employees' names are not present on the website. Managers will have a local excel file to store the employees' name and their corresponding device ID.

5. Future Work

This chapter provides some recommendations for further development of the web application. We compare the company's expectations, our initial structure, and the final product we delivered. Though the basic structure is set up and we enjoy the overall process of designing and implementing the application, some features have not been implemented successfully by the end of this project. Partly this is due to the time limit of a seven-week project, and there are also situations where we are not able to find a good solution to meet certain design specifications. For each of these unfinished parts, we share our thinking on possible implementation and why we believe this specific feature is important for the application.

First, the data visualization implemented in our web application is very basic. For now, on the Statistics Page, we only include one chart that represents the number of close contacts for each day within a week. To provide a more comprehensive view of what is going on in the building in terms of close contacts and social distancing for users, more charts and advanced data visualization should be implemented in the future.

In addition to more diagrams, according to what we learned from the company, they also want the web application to be able to push notifications and send emails to users when one close contact has lasted for more than 15 minutes. This feature allows people to be aware of all the violations of social distancing that occur in the building. In terms of pushing notification, one possible solution is to utilize the *flash session* that we used on User Authentication Page as well as when users attempt to perform certain tasks that they are not allowed to on Upload Page and Devices Page. In this case, when new contacts data is being added to the database, the application can go through all the newly-added contacts data and push the notification when it finds the duration of any of the contacts is longer than 15 minutes. However, since close contact

information is not being added to the database and displayed on the website simultaneously when the contact occurs, users only receive the notifications when the data is uploaded to the website; the delay of sending the notification caused by the delay of updating contacts data may undermine individuals' abilities to protect themselves from contacting people who have had prolonged close contacts.

Besides, what we learned from Amphenol is that the wearable devices will be collected by the end of the day and get disinfected and charged. According to what we heard from Woxu, the wearable devices can work for several months without charging. In case of the malfunctioning and low battery of the devices, the list of low-battery devices can be displayed on the homepage and updated each morning before the devices are distributed to employees.

Furthermore, a “delete” function may be useful. Currently, the website is mainly used to display the contacts and the information of devices, which can be enough to assist managers to make some decisions. If managers would like to edit the history manually or would like to delete the history before certain dates, a “delete” function will provide more flexibility.

Last but not least, as the web application we developed as a prototype, the structure of the web applications has been set up successfully. There are still many aspects that can be improved. For instance, the security can be improved to better protect the information collected by devices.

Bibliography

- Dahlgren, E.; Mahmood, H. Evaluation of Indoor Positioning Based on Bluetooth Smart Technology. Master's Thesis, Department of Computer Science and Engineering, Chalmers University of Technology, Department of Computer Science and Engineering, Göteborg, Sweden, June 2014.
- Jianyong, Z., Haiyong, L., Zili, C., & Zhaohui, L. (2014). RSSI based Bluetooth low energy indoor positioning. *2014 International Conference on Indoor Positioning and Indoor Navigation (IPIN)*. doi:10.1109/ipin.2014.7275525
- Li, Xin, et al. "A Bluetooth/PDR Integration Algorithm for an Indoor Positioning System." *Sensors*, vol. 15, no. 10, 2015, pp. 24862–24885., doi:10.3390/s151024862.
- Nussbaumer-Streit, B., Mayr, V., Dobrescu, L. A., Chapman, A., Persad, E., Klerings, I., O'Sullivan, Joseph. "Gov. Inslee Extends Washington State's Coronavirus Stay-Home Order through May 4." *The Seattle Times*, The Seattle Times Company, 2 Apr. 2020, www.seattletimes.com/seattle-news/politics/gov-inslee-extends-washington-states-coronavirus-stay-home-order-through-end-of-may-4/.
- Pahlavan, Kaveh, and Allen H. Levesque. *Wireless Information Networks*. John Wiley, 2005.
- Pahlavan, K., et al. "Indoor Geolocation Science and Technology." *IEEE Communications Magazine*, vol. 40, no. 2, 2002, pp. 112–118., doi:10.1109/35.983917.

Qureshi, Umair Mujtaba, et al. "Evaluating the Implications of Varying Bluetooth Low Energy (BLE) Transmission Power Levels on Wireless Indoor Localization Accuracy and Precision." *Sensors*, vol. 19, no. 15, 2019, p. 3282., doi:10.3390/s19153282.

Schnirring L. (2020, March 6). *France orders lockdown to slow COVID-19 spread*. CIDRAP News. Center for Infectious Disease Research and Policy. Retrieved from <http://www.cidrap.umn.edu/news-perspective/2020/03/france-orders-lockdown-slow-covid-19-spread>

"Symptoms of Coronavirus." *Centers for Disease Control and Prevention*, Centers for Disease Control and Prevention, Retrieved from www.cdc.gov/coronavirus/2019-ncov/symptoms-testing/symptoms.html.

Systems Science and Engineering (CSSE) at Johns Hopkins University. (2020, September). *COVID-19 Dashboard by the Center for Systems Science and Engineering (CSSE) at Johns Hopkins University (JHU)*. Retrieved from <https://gisanddata.maps.arcgis.com/apps/opsdashboard/index.html#/bda7594740fd40299423467b48e9ecf6>

Using AWS S3 to Store Static Assets and File Uploads: Heroku Dev Center. Retrieved from <https://devcenter.heroku.com/articles/s3>.

Wagner, G., Christof, C., Zachariah, C. & Gartlehner, G. (2020). Quarantine alone or in combination with other public health measures to control COVID-19: a rapid review. Retrieved from

<https://www.cochranelibrary.com/cdsr/doi/10.1002/14651858.CD013574/information#C>
D013574-sec-0063

Zhou, Yuan, et al. "Ultra Low-Power UWB-RFID System for Precise Location-Aware Applications." *2012 IEEE Wireless Communications and Networking Conference Workshops (WCNCW)*, 2012, doi:10.1109/wcncw.2012.6215480.