NUS
National University
of Singapore

SafeTogether

Technical Handover Document
DSA3101-05-Security-Frontend
Tiffanie Leong
A0239929J

1. Project Framework and Structure

1.1 Project Overview, Goals and Scope

Our project, titled “SafeTogether”, revolves around the development of a comprehensive system for actively monitoring criminal incidents within the National University of Singapore (NUS) campus. It seeks to provide end-users, the security personnel, with a working and value-adding tool for enhancing safety and security within the campus.

1.2 Purpose of the Report

This technical report aims to document the front-end development work conducted as part of Project SafeTogether. It aims to provide an overarching understanding of the front-end team's contributions, including the features, wireframes, codebase, as well as the influence of user interviews on the project's evolution. Here is a rough outline of the report:

- Section 2 : Features of our web app
- Section 3: Questions and findings of 2 user interviews conducted throughout the project
- Section 4: Wireframe and evolution of our design
- Section 5: Codebase documentation
- Section 6: Bugs and Issues Faced

2. Features

The majority of the features developed by our team are functional as of the writing of this document. Images of the interface are available in section 5. For our web application, we have 3 main features.

2.1 Map and Heat map

For the Map tab, users can view a Google Map with precise crime incident locations. Clicking on markers reveals detailed information about each incident. Two filters allow users to sort incidents by crime category and contact number. There would be another Heat Map tab included, which provides an overview of common crime hotspots on campus, aiding efficient resource allocation.

2.2 Dashboard

The dashboard summarises crime rates and security trends within the campus, utilising various visualisation charts such as bar graphs, line graphs and pie charts. Filters are included to allow users to focus on specific date ranges for analysis. Through these visualisations, our dashboard aims to help the security team carry out risk assessment, maximise allocation of resources and plan preventive measures to anticipate the increase or decrease of prevalent crimes happening on campus.

2.3 Incident reporting

The Incident Reporting feature streamlines the process of data collection. It includes an incident reporting form with fields for capturing various crucial information. The data submitted through this form is systematically stored in a centralised database, ensuring easy accessibility for subsequent analysis and reporting. Additionally, users are provided with the option to further enrich their reports by uploading audio files, accessible through dedicated tabs, offering flexible means of incident documentation.

3. User Interview

We had the pleasure of interviewing Mr Ian Tan, the Campus Security/UCI point of contact. He will act as a representative of the other members of the campus security team. He would have extensive understanding of the existing structure regarding the state of crime monitoring within NUS and will have sufficient expertise to articulate the most pressing requirements for our web application.

3.1 Interview 1

Our initial interview, held on September 26, 2023, served as a foundational step in gaining a comprehensive understanding of our end users, their specific requirements, and the features they desire to see implemented. It provided essential insights that functioned as a launching point for shaping the features and user interface of our web application.

3.1.1 Interview 1 Questions

Our approach to formulating questions and conducting interviews is grounded in three core principles shared by Mr Jaffry: Understanding frustrations, delving deeper with follow up questions as well as observing behaviours and patterns. Listed below are some of the questions asked during the interview:

- Can you describe the current measures adopted for monitoring crimes on campus?
- What are the main incidents/types of incidents that occur in NUS?
- What are some potential features a security team would potentially look out for and what sorts of tools would be the most helpful for the team?
- What are the challenges you face with your current systems?
- What are your preferences for the user interface?
- What kind of data presentation works best for you?
- How do you envision transforming data into actionable insights?

Granted, our questions are not limited to these few. Nevertheless, the next subsection will detail the most important findings of the first user interview.

3.1.2 Interview 1 Findings

Current methods and measures adopted

- Basic data collection and analysis of incident information, including only basic information such as nature of incidents
- Presentation of basic data insights to the security team through manual collation
- Data collection through forms on the NUS website and phone calls.
- Current means of monitoring crime in NUS includes the use of centralised command and call centres as well as CCTVs.
- The main incidents in NUS were mostly thefts, peeping toms, lost and found and accidents. However it was also noted that incidents were generally rare

Challenges with current system

- Manual collation of data insights can be time-consuming and prone to errors. This process may hinder the security team's ability to respond quickly to emerging and time-sensitive incidents.
- The current system lacks detailed data visualisation tools, making it challenging to identify trends and patterns in incident data.
- Allocating resources such as manpower efficiently based on incident patterns is a complex task, and the current system may not provide the necessary tools for this
- Challenges in handling vast data and analysing locations with high human traffic.
- Challenging to handle location based data

Potential features and tools most useful for a security team

- Interested in features that enhance proactive monitoring and incident prevention.
- Want a way to streamline data collection and to make collected data easily accessible
- Interested in ways transform data on hand into actionable insights such as to better optimise the resources they have on hand
- Interested in customisable features so that each member of the team is able to gather information based on individual needs, preferences or daily job requirements

Preferences for user interface

- Visibility should be the top priority and nothing should be too complicated
- User-friendly, straightforward, and visually accessible with no micro details

Upon obtaining an understanding of the team's needs and preferences, we prioritised 3 main features. Based on those, we then constructed our first wireframe, which will be elaborated in Section 4.1.

3.2 Interview 2

We conducted our second interview on 20 October 2023. This time, the focus revolved around obtaining feedback for our initial wireframes and to see how to improve our proposed features. The interview format for this interview was not centred around a group of pre-fixed questions but rather discussion was carried based on the drawn prototype. The following subsection details the potential issues, feedback and possible changes that the security raised as per the initial prototype.

3.2.1 Interview 2 Findings

On the issue of location logging

- Security team expressed concerns about the challenges in logging locations, emphasising the need for a realistic/actionable approach without overwhelming the guards with too many fields to fill in
- Worried about inconsistency should they have to fill in the locations themselves.
- Raised questions regarding complaint submission, interested to know how the next person analyses/collects the data?
- Emphasised the importance of data privacy and raised concerns about how sensitive information such as details of the crime is stored and accessed.

Regarding the map

- Generally satisfied with the UI of the map
- Wishes to have a clicker function (bring cursor to desired location and information will pop up)

Regarding the dashboard

- Interested in trends based on NUS calendar (longitudinal studies)
- Charts present should compare past and present and graphs should establish a reference point, giving a general sense of the types of crimes that occur during and off semester.
- Dashboard as a whole should provide good summaries and information can be easily understood with a single glance.
- Would like to see more than 3 charts/graphs and noted that around 5 would be an ideal amount.

4. Wireframes and Design Evolution

4.1 Low Fidelity Wireframe

We constructed our first wireframe after we conducted the very first interview, details of which are provided in section 3.1 prior to this section. Below are our initial low-fidelity wireframe.

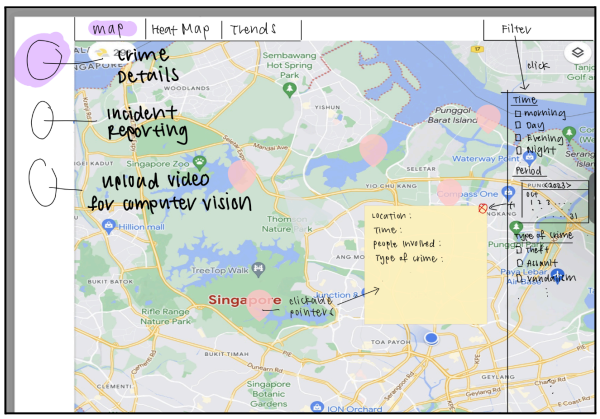


Figure 4.1.1

Figure 4.1.2

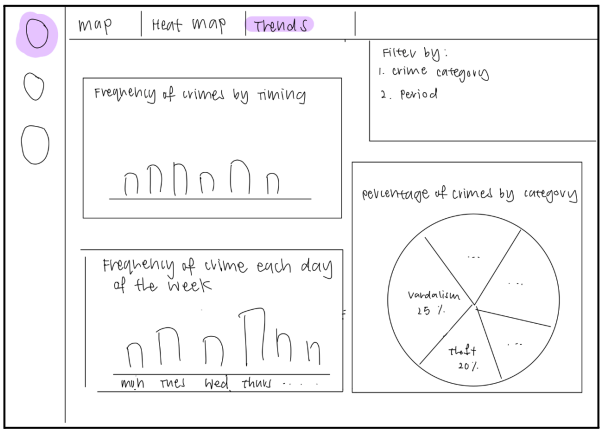


Figure 4.1.3

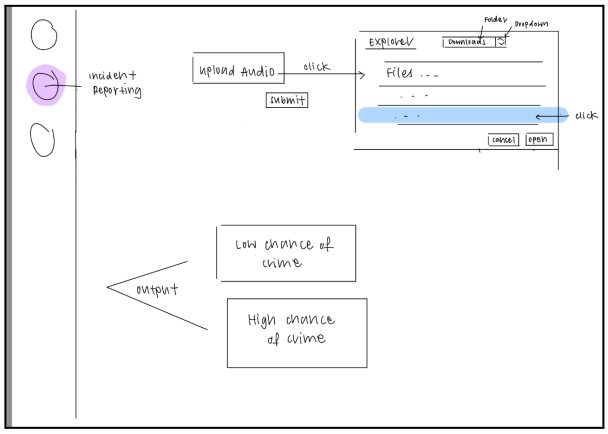
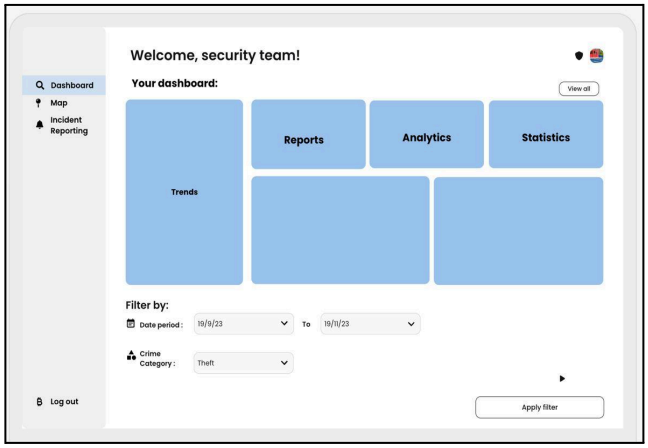


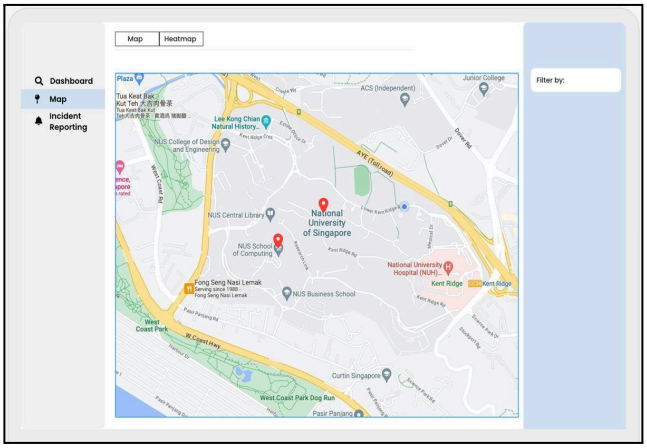
Figure 4.1.4

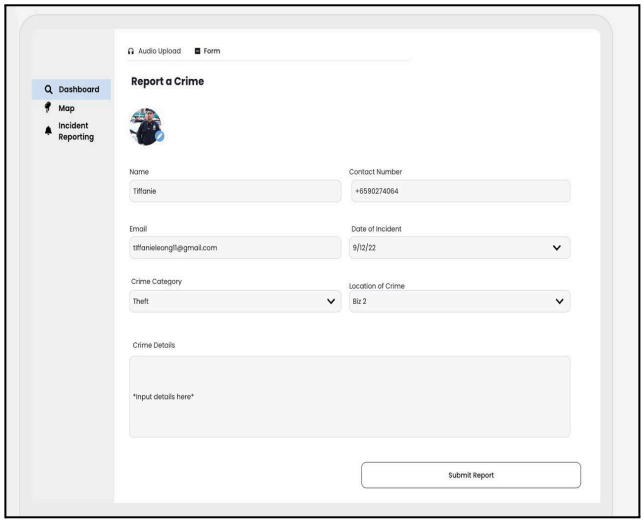
Since there was a clear challenge stated by the security team regarding the handling of location based data, a map feature as shown in figure 4.1.1 was devised to tackle this challenge. It would allow the team to pinpoint incident locations effectively, including specific areas or buildings, hence getting a better sense of which area has the most incidents. Another feature to help with this would be the heatmap (figure 4.1.2), which offers a colour-coded representation of incident density. Areas with a higher concentration of incidents are emphasised visually with this tool, allowing easy identification of crime hotspots. Next, the Dashboard (figure 4.1.3) is designed to mitigate the current system's lack of comprehensive data visualisation tools. It offers various data visualisation options such as bar graphs, line graphs, and pie charts for ease of viewing. As such, the team can identify specific time patterns, prevalent crimes, and other critical insights (eg. which day of the week has the most crime occuring) This would allow them to allocate resources efficiently by dispatching manpower accordingly. Filters for time period/ crime category will be put in place so that individual security team members would be able to gather their desired information. Lastly, the Incident Reporting feature (figure 4.1.4) addresses the issue of manual data collection. This feature streamlines data entry and simplifies the reporting process, saving time and reducing mistakes. A centralised database will be used and information collected via the form will be used as data for the above two features, allowing for up to date information that will aid the team in their tasks.

4.2 High Fidelity Wireframe

From our initial wireframe, we have opted to preserve the 3 proposed features. Subsequently, in response to insights gathered during the 2nd interview, we developed a 2nd wireframe. This iterative process has led to specific refinements and enhancements aimed at addressing the feedback and requirements identified through our ongoing engagement with the security team. It is with close reference to this high fidelity wireframe that our team will begin to model our actual application after.







1. Red markers on the map are set to allow for clicking, which will cause a pop up text box with the relevant details. We have also confirmed that the filters used here will be crime category as well as contact number (a unique identifier of any person)

2. Changed the layout of the dashboard where 5 boxes indicates 5 different types of visualisations

3. Used a drop-down box for location of crime on the incident reporting form. Hence, only pre-fixed locations can be chosen. We will ensure that the list provided is comprehensive enough to cover all locations in NUS. This is also to ensure that information sent to the database is in a consistent format

4.3 Future developments

We acknowledge that certain features/enhancements may not have been fully realised. However, to ensure the continued improvement of the app, future developers can consider implementing the following:

Potential Features (To be implemented)
User authentication and authorisation page for security team members to log in
Allow for users to export and download incident data and visualisations (generation of reports in different formats eg. PDF, CSV for further analysis or sharing with stakeholders)
Consider the use of CV (computer vision) to analyse live camera feed across the campus

5. Code Base

5.1. System and Code Architecture

5.1.1 Frontend

The primary framework chosen to create an engaging and interactive user interface for incoming users is **ReactJS**. This framework is pivotal in achieving a responsive design that minimises compatibility issues across various devices. To streamline the development process, pages are initially prototyped in **UIZard** before being transformed into production-ready code. For efficient development and deployment, we've incorporated **Vite** as both a build tool and a means to deploy the user interface. In our repository, we dealt mainly with the “frontend” directory, which acts as the storage location for our JavaScript and CSS files, responsible for rendering the application’s UI. Despite segregating our frontend (ReactJS) and backend (Flask Server), it is essential to establish a connection to ensure their collaborative functioning. This can be achieved by routing all API calls from the frontend to the backend.

5.1.2 Version Control

Git serves as the chosen version control system for tracking and preserving the history of changes and commits as the project evolves over time. The project's source code is securely hosted on **GitHub**, organised within a dedicated GitHub Organisation. This organisation encompasses distinct repositories, effectively separating the front-end and back-end architecture components. You can access these repositories by following this link: https://github.com/yell0wbear/DSA3101-2310-05-sec

5.2 Set up

In order to set the project up from scratch, we run ```*npm create vite@latest*``` in our terminal. This command gives us a default website since Vite is used as a build tool here. In the following sections, the files added will serve as the customisation to the generated website. Once the project has been created, a frontend folder will be generated. Inside the folder will contain the usual elements found in any React app structure such as the **package.json** file, a **node_module** folder, a **public** folder and an **src** folder.

Next, a flask server would need to be constructed. In the **/backend** folder in our repository, we created a python file called *app.py*. The same directory should also include the incident management module (**incident_creation**). The purpose of this is to create the API that enables connection between the frontend and backend of our project. The flask application is initialised with Flask(__name__). In the python file, we also define various endpoints to handle different actions related to incident management. We will use these endpoints to retrieve, add, filter and export incident data.

Next, we need to navigate into **/frontend** and run ``npm install`` to install the project dependencies specified in the **package.json** file. A detailed list of all the libraries and dependencies for installation can be found in the *README.md* file as well. As the project has been dockerised, information about how to get the server up and running will be documented in detail in section 5.4 of this document.

5.3 Front-end Components

For the front-end, we have three main pages: the map, the dashboard and the incident reporting. All codes for the relevant pages are stored in **/src** in our frontend folder.

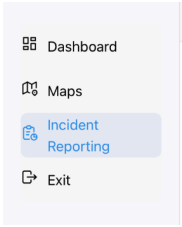
5.3.1 App

The entire webpage is only functional with *App.jsx* which serves as a linkage for all the files of our application. It is the core of the whole application, orchestrating its structure and navigation, rendering and displaying the different components based on the selected tabs and headers. In order for *App.jsx* to be functional, it is important to remember to import all the necessary components and styles used for the other pages of the app.
Here, we make use of **State** hooks to manage the active tab and active header tab in the application. Basically, the hooks allow us to track state in a function component. We have chosen to use the value, 0/ "Dashboard" to start with as we would like the website's default landing page to be the dashboard. Here, activeTab and activeHeaderTab are the current state values that can be used in this component and setActiveTab and setActiveHeaderTab are the functions used to update the state, triggering a re-rendering of my desired component (allowing pages to render when switching tabs). The code for rendering different header tabs based on the activeTab is modular and easy to follow and should there be future changes or additions to the webpage, one can simply follow the existing format of the code.

Next, we will utilise a **useEffect** hook which allows the user to synchronise a component with an external system. This hook manages an array that contains the state variable which responds to any change, triggering a callback function. The first useEffect allows the current tab to be refreshed, loading the active tab from local storage during component mount. Here, the callback function is only called once the page renders in this case. If the user reloads the page, the last active tab and header tab are restored from the previously stored values in the local storage. The second useEffect is used to update the local storage when the active tab changes.

The rest of the code is used for content and tab rendering and can be regarded as self-explanatory. In the *App.jsx* file itself, comments written from line 53 to 57 should provide sufficient information of which component and header tabs are rendered based on the combination of active tabs selected. Should there be any additional tabs or navbar options to be added, it can be added to line 37 of the code and onwards.

5.3.2 NavBar



Navbar.jsx is the code for our navigation bar as seen in the left side figure. It provides links to different sections, such as the Dashboard, Maps, and Incident Reporting. The main NavbarMinimal component defines an array of labels, each associated with an icon (from @tabler/icons-react library) , representing different pages of the app. It maps over these labels to create NavbarLink components for each page. The code also uses the **Stack** component to vertically align the navigation links. What makes the bar functional is the **UnstyledButton** component which renders the clickable areas in the navigation bar. Lastly, the component also uses CSS Modules for styling. Styles are imported from the *Navbar.module.css* file. More details regarding the styling can be read from the actual css file itself as it is rather self explanatory.

There are no hooks used directly in this component (hooks are already utilised in *App.jsx*). Nevertheless, it receives setActiveTab and activeTab as props. These props are used to manage the active tab in the app, indicating which section is currently selected.

Lastly, the **onClick** event handler allows the user to call a function and trigger an action when they click an element (eg.button) in the app. In this case, the **onClick** handlers are provided to each NavbarLink to manage the activation of tabs. When a link is clicked, it calls the **setActiveTab** function with the index of the clicked link, allowing the page that is linked to the clicked navbar option to be rendered.

5.3.3 Tabs

Tabs.jsx tracks which tabs are active and displayed as seen on the left. In the code, **HeaderTabs** and **SecondHeaderTabs** differentiates the "Map/Heat Map" tab from the "Audio Upload/Form" tab. Both functions work in the

same way and have the same structure. According to Jakob's Law, users would prefer to have a website that works the same way as the others that they are familiar with. Hence, we choose to use tabs at the top of the webpage as many website interfaces largely feature clickable tabs to switch pages at the top of web pages too.

In addition, the **tabs** component is used for generating the tab interface while the **onChange** prop is set to the **setActiveHeaderTab** function, indicating that when a tab is clicked, it will update the active tab in *App.jsx*, rendering the new page. Next, the **tabs.map** function iterates over each element in an array. Here, it is used to loop through the tabs array, which contains the names of the tabs ('Map'/'Heat Map' or Audio Upload/Form). The value prop is set to the current tab name, which indicates which tab is currently active.

5.3.4 Map and Heatmap

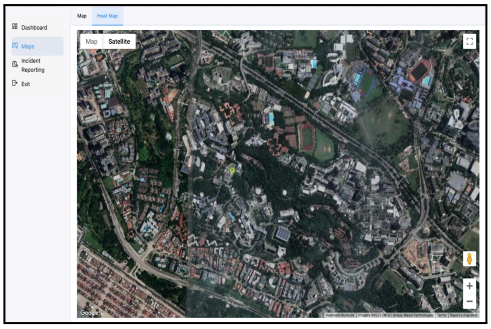


Figure 5.3.4.1

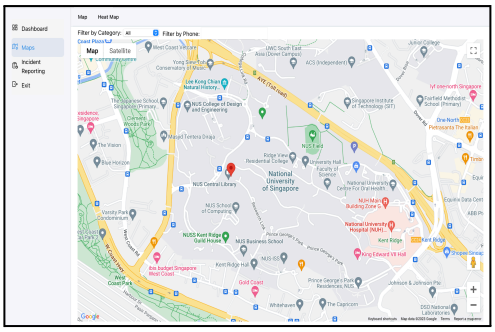


Figure 5.3.4.2

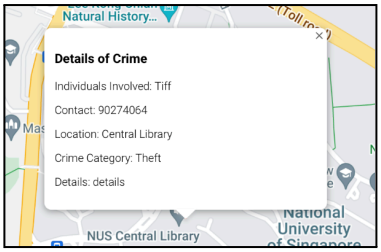


Figure 5.3.4.3

The *Map.jsx* component is a React-based module designed for displaying an interactive map with clickable markers. It integrates with Google Maps API to provide a visual representation of where incidents are located on campus. Furthermore, information is easily accessible through a simple click which allows a pop up window detailing the details of the crime. In addition to a regular map, *Heatmap.jsx* allows the rendering of a heatmap of incidents using the @react-google-maps/api as well. This allows for easy identification of areas with a high concentration of crime incidents as hotspots can be easily seen at first glance. These hotspots indicate locations where crimes are more prevalent hence allowing the team to allocate more manpower at crime heavy locations. Figures 5.3.4.1 and 5.3.4.2 show the UI of the heatmap and map respectively.

In *Map.jsx,* the code begins with an initial initialisation of `map`, `clickerMarker`, `filterCategory`, `filterPhone`, and `position`, which are all initialised to empty/null elements. This indicates that on first initialization, there are no elements selected/loaded. Since **useState** is a hook, calling the function in the array will allow the user to update the state accordingly. Example: By calling **setMap(newValue)**, React will re-render the component with the updated value for map. Here, the **isLoaded** state indicates whether the Google Maps API has been successfully loaded. The **map** State acts as a storage for the reference to the Google Map instance, allowing interaction with the map. The **clickedMarker** state is used to track the currently clicked marker and also used to display relevant information when clicked. The filter states, **filterCategory/filterPhone**, are for managing the selected filter options for crime category/phone number respectively. This is the most crucial element to note as should there be any additional filters to be added, this is where they should be added. Lastly, the **positions** state stores an array of incident positions, demarking the latitude and longitude of each incident.

To retrieve actual data, we use **axios.get('/incidents/incidents'**) to make a GET request to the specified API endpoint to fetch incident data that was keyed into the incident reporting form. Next, data collected in the database in a JSON format is being parsed into a Javascript object to be further processed. This maps the raw incident data to the desired format, extracting crucial information such as latitude, longitude and other details such as contact number. In line 43, **setPosition(formattedData)** then proceeds to update the **positions** state with the formatted data, re-rendering the map and allowing the markers with the relevant data to appear.

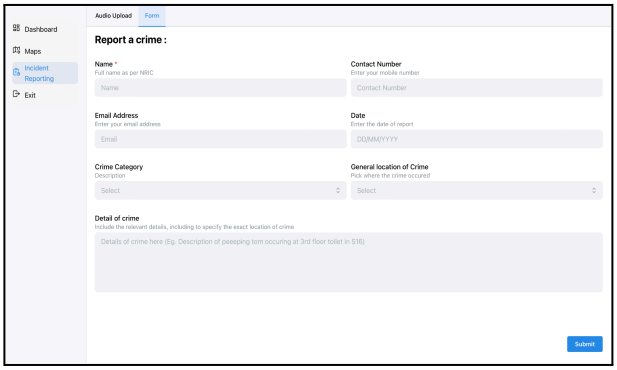
In *map.jsx*, line 77 of the code and onwards will be regarding the Google Map component. Lines 81-96 of the code is responsible for rendering the filters for crime category and phone number, in which crime category has a dropdown clicking option while the phone number filter is a typed-in text box. Lines 110-116 is responsible for rendering a **Marker** component for each position by mapping the **filteredPositions** array. The **onClick** event is then set to call the **handleMarkerClick** function such that clicking with a user's mouse or touchpad is what causes the information window to pop up. Lastly, lines 118 to 131 renders the information window (Figure 5.3.4.3) titled "Details of Crime" which contains information such as contact and location.

Now I will seek to explain Heatmap.jsx. I will note that the code shares many similarities to *Map.jsx* due to their similar nature, hence I will omit a repeated explanation. Here are 2 other aspects of the code worth mentioning. Firstly in *Heatmap.jsx*, the main difference as compared to *Map.jsx* is that the **useEffect** block only fetches the latitude and longitude details and does not require all the additional information like contact number etc. Apart from this, the function acts the same way as described above. In *Heatmap.jsx,* we have utilised the **HeatmapLayerF** component, from the react-google-maps/api library. The point here is retrieved from the database and each coordinate point has its own distinct latitude and longitude. Here, the code maps over each point in **heatmapData** and creates a new instance of **google.maps.LatLng** for each distinct point on the Earth's surface. The resulting array is a set of LatLng objects. By combining all the points, we obtain a layer which is reflected on the satellite map as the heat map visualisation.

Apart from *Map.jsx* and *Heatmap.jsx*, a *GoogleMapsProvider.jsx* file is required as it serves as a provider for the Google Maps JavaScript API within our app. It uses the **useJsApiLoader** hook to manage the loading of the Google Maps API. This is required as the API cannot be called twice in two separate files (Map and Heatmap). A custom hook, **useGoogleMaps**, is defined using **useContext** to conveniently access the Google Maps context state (e.g isLoaded/ loadError) in other components. Essentially, GoogleMapsProvider establishes a context that helps manage the global state of Google Maps API loading across various parts of the app. This is crucial for future development as should there be additional pages that require the use of a map visualisation, this custom hook can be reused. The use of **useContext** for managing the Google Maps API loading state centralises the state management process which means that any updates or modifications to the state are reflected universally, minimising the risk of inconsistencies when carrying out more edits in the future.

5.3.5 Incident Reporting

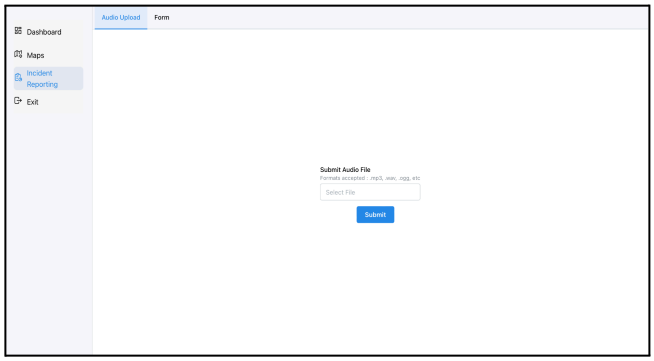
Incident.jsx serves as an incident reporting form, allowing security to fill in the details of any victims. We have chosen to continue incorporating components from the Mantine library for our UI design to maintain consistency. As shown, the form captures crucial details such as name, contact information, date of the incident, email address, category of crime faced (eg. theft, assault, vandalism etc), a general location, and additional information. In addition, *Incident.jsx* includes a long predefined set of coordinates corresponding to different areas on the campus (these are listed as options under the general location of crime). These coordinates are used when submitting the form to provide the latitude and longitude of the reported incident.



The **handleInputChange** function serves as an event handler for the input fields. It allows for components to be dynamically updated based on user interaction. In response to an input, the function updates the component's state, ensuring accurate reflection of values keyed in the input boxes. For example, should a user type in "tiffanie " into the name input field, the function updates the state as follows: name: "tiffanie"

Next, we take a look at the API to link the page to the database. Before we send the request to the server, we must prepare the form data in a JSON format. The form details sent to the database are the information filled in in the

incident reporting form. The **Axios** library is then used to make an PUT request to the server's /incidents/add_incident endpoint. The prepared form data is sent as the request payload and collated in a local database. Lastly, the submit button triggers the **handleSubmit** function when clicked. It is styled and positioned using inline styles for a clean and accessible design. Fitts's law states that the time to acquire a target is a function of the distance to and size of the target. Hence the submit button is designed to be larger in size. In addition a striking colour of blue is used to capture the user's attention.



In addition, we also implemented an audio submission function which will enable the uploading of phone calls that provide the details of the incidents. Upon upload, the information will be sent to the database in the same format as should one manually key an input into the incident reporting form. This function enhances the versatility of the incident reporting system, providing users with a flexible and accessible way to report crimes. Regrettably, at present, this functionality is non-operational as integration with the backend API has not been established.

5.3.6 Dashboard

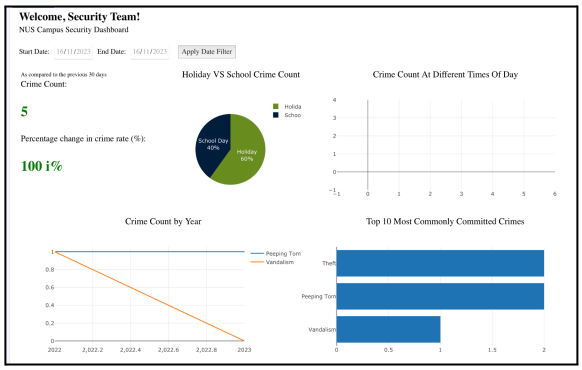
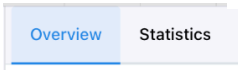


Figure 5.3.6.1

Figure 5.3.6.2



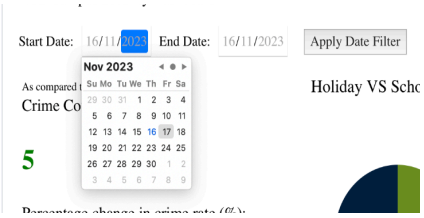
For our Dashboard page, we have split it into 2 separate tabs as shown on the left figure: overview based on API data as well as the statistics. Firstly, **Overview(API Data)** presents insights related to crime incidents within the campus. The API based plots will serve as our application's launch page (the first thing a user sees when opening our application). This is such that users are immediately presented with critical insights related to the campus upon opening the application. On this page, we chose to display the five following plots: "crime count as compared to the previous 30 days", "holiday vs school crime count", "crime count at different times of the day", "crime count by year" and "top 10 most commonly committed crimes". We chose these 5 charts as they collectively provide a holistic understanding of the security environment in NUS. The plots would allow the security team to address short-term patterns, anticipate variations based on academic seasons, optimise daily security measures and manpower deployment, plan for the long term, and prioritise efforts based on the most common types of crimes.

Here, *Dashboard.jsx* is the main code chunk for Overview(API Data). In additional to this file, we have created 5 separate files, namely *CrimeLast30Days.jsx, CrimeCount_HolvsSch.jsx, CrimeCount_TimeofDay.jsx, CrimeCount_Time.jsx* and *CrimeCount_CommonlyCommited.jsx* for rendering the individual visualisations as mentioned above. All 5 visualisation plots are then rendered together as a whole with *Dashboard.jsx* as seen in Figure 5.3.6.1 We chose to have separate files for separate visualisations to allow for a more modular and organised code structure. Each visualisation can be developed, tested, and maintained independently, making the codebase more manageable and allowing multiple developers to update or modify a particular chart without affecting the rest. Henceforth, should any future developers intend to add any additional plots to the dashboard,

they can simply do so by creating a brand new *.jsx* file, then proceeding to utilise the library "react-plotly.js" and subsequently code out their desired visualisation chart. The new file can then be added into the Dashboard.jsx file and the new plot will be rendered on the main dashboard page of the application.

In each of the individual files for the visualisation, the **type** variable will be used to denote which sort of chart will render on the dashboard. For example, type: “bar” will produce a bar chart while type: “pie” will produce a pie chart.

In *Dashboard.jsx*, we will first import the necessary dependencies such as the React hooks (useEffect, useState, and useCallback) as well as components from the Mantine UI library for styling and the Axios library for making asynchronous HTTP requests. Mostly important, the various crime-related visualisations from separate files should be imported as well. We then define the **LeadGrid** function, which uses **State** hooks to hold the fetched data, the filtered data, and the date filter values. Next, we utilise the **useEffec**t hook to fetch incident data when the component mounts, making an asynchronous request to the "/incidents/incidents" endpoint using Axios. By doing so, it is pulling the data from the database which contains the details of crimes made through the incident reporting form. The data fetched would then be displayed accordingly on our dashboard.



In the API data dashboard, we have included a date filter which enables the filtering of data based on a certain date range. If no range is selected, the dashboard should display all the data collected across time. To render this, we utilised the **filterData** function which filters the incident data based on the selected start and end dates, ensuring that only data within the specified date range is included in the **filteredData** state. Hence when users select their desired data range, only data which falls within the specific range is displayed. Finally in both *Dashboard.jsx* and *DashboardSF.jsx*, the **return** statement triggers the overall rendering of the dashboard. Here, “**Grid**” is used to format the layout into 5 different sections, a button is included to apply the date filter, activating the **filterData** function, and inline styles are applied for font size, padding and background colour.

Lastly, the **Statistic** tab is present to showcase crucial statistical information and visualisations related to crime occurrences, with a specific focus on Larceny Theft during weekdays and weekends. Here, the backend team has implemented a Flask API, allowing communication with our frontend to retrieve Plotly plots. These plots specifically showcase crimes near to universities around San Francisco and users can dynamically filter data based on parameters such as crime category and the number of k-means clusters, all of which are valuable insights for security members. Figure 5.3.6.2 exemplifies the potential insights that can be derived when setting the appropriate filters. Here, we use static plots as this function mainly serves as a proof of concept, providing the security team with a basis to experiment with other parameters using their own dataset, allowing them to retrieve any sorts of insights suited to their specific needs in the future.

5.4 Dockering the code

We will next dockerize our code due to the many benefits that arise from the usage of containers such as continuation of integration and delivery, increasing propensity for collaboration as well as making our project fast, lightweight and scalable. It also prevents compatibility issues, ensuring that the app runs consistently for all environments. For our project, the following 4 files will be necessary in helping us dockerize our code.

We start this process by creating a docker file (*flask_dockerfile*) for our backend, the Flask server. This defines how the docker image for the Flask backend should be built. To begin, we will be using python 3.10.6-slim-buster as a base image. We will set /app as the working directory. Then, we copy over the models and files built by our backend. Next, we upgraded pip and installed all the packages specified in the *Requirements.txt* file. Those details would be elaborated more in the technical document written by the backend team members. The last line specifies the default command to run when the container starts, proxying the container’s regular entrypoint and

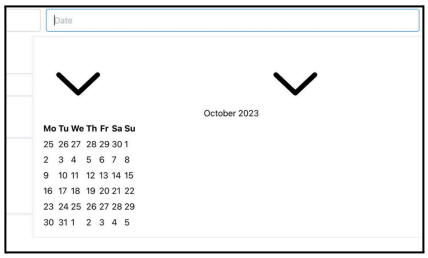
hence running the main app module. We should note that *Requirements.txt* lists down all the Python dependencies required for the Flask, including MySQL connector, Flask, requests, and pandas. This is crucial for the backend component of our project.

Next, we have to create a docker file in the **/frontend** folder (*Dockerfile*). This file defines how the docker image for the frontend should be built. We have to set our working directory to **/app** and copy all relevant files into the container which includes package.json as well as our src and public folder. We then install using npm and this would install all the npm dependencies. The last instruction (``npm run dev```) is what needs to be executed when the Docker container is starting, setting the default command to run the server for our app.

Lastly we have *docker-compose.yml* which acts as a setup to link the backend, frontend, and database services, allowing them to work together in a unified application after composing. We defined the build option for each container by specifying the folder + name of the dockerfile (`app` builds from ./backend with *flask_dockerfile* while `front-end` builds from ./frontend with *Dockerfile*). In this file, we map port 5000 from the container to port 9001 on the host for the backend. The db uses MySQL 8.0.34, which maps to port 3306, and initialises the database with a SQL file. Lastly, the frontend maps port 5173 from the container to port 5173 on the host. We also set the environment to a specified API token such that this key can only be accessible by the applications in our container, enhancing security. Typically, docker would create a common network for all containers, linking the frontend, db and the backend. To get the container up and running, simply type “make run” in the terminal.

6. Bugs and Issues Faced

In this short section I will provide a table which details the unresolved bugs or issues we have encountered during our project as well as some remarks.

Bug/Issue	Development
Drop down calendar in the incident reporting form cannot be formatted correctly 	Possibly an issue with the Mantine Library itself and requires further investigation. Instead, we used a typed box that only takes in the date in the DD-MM-YYYY format. Data routed to the database is still in the same form.
Incident reporting form does not automatically clear after pressing the submit button. Data is sent to the database but the input fields of the form remain filled in rather than being cleared to an empty box after submission.	Requires further adjustment to the code. Instead we used a pop-up message “successfully submitted” to indicate that the data has been ported to the database
When the tab for the entire website is minimised beyond a certain size, the website fails to render.	Possibly an issue with scaling however due to time constraint, this has yet to be addressed.

7. Conclusion

It is hoped that through this report, the reader has gained an overarching understanding of all the frontend components as well as the other important elements of our project and that the information conveyed is sufficiently comprehensive to allow for future continuation of our project by new developers.