Ripple

Design and Planning Document
Version 1.0, 2021-03-05

Authors: Eric Li, Junyu Wang, Kerui Wang, Penghai Wei, Sunjun Gu, Titus Smith

---

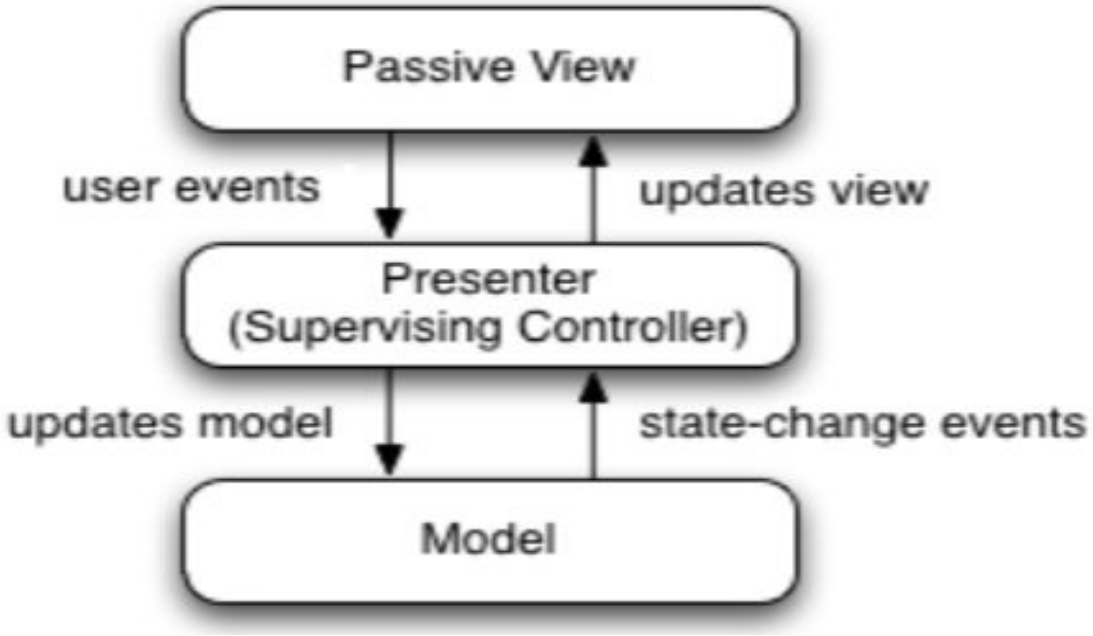
Document Revision History

Rev. 1.0 <*2020-03-03*>: initial version

System Architecture

Overview:

Our application will use the Model-View-Presenter (MVP) architecture. Our front-end framework, React (Native), wasn't developed to fit in perfectly to any one architecture, and upon further research, there is quite a lot of debate about what architecture React (Native) is a part of. That being said, we chose MVP for two reasons. 1) Unlike MVC, where the Model directly interacts with the view and the view interacts with the model, the presenter is the middle man. This allows us to both make passive views (functional components in react) and also map onto an HTTP request more accurately. When the presenter communicates with the model, it can immediately receive a response with the current or updated state. 2) MVP allows us to completely separate the model from the user interface. We are able to reuse the model with a new UI if we decided in the future that we wanted to add a web app in addition to the planned mobile application. Here is a diagram from class diagramming the MVP view.



View

The View takes in state/data from the Presenter and displays it to the User. The View is what the user interacts with (Maps, Reporting Modals, etc) and upon certain user events, like button clicking, the View will call on the Presenter to let it know a user event has happened. In the case of the creation of a new incident report, the view informs the presenter of this event through some onClick function.

Presenter

The presenter is the middleman between the View and Model. It will take in user events from the View, and depending on the user event, it can send and receive an HTTP request from the Model and update the View Screen.

Model

Since the client side of the application doesn't need to maintain client state, since we don't have user authentication or host user data, we can easily separate the model to be purely on the server side of the app. The data itself will be stored in a MongoDB backend, including incident records info and other necessary d. The Model will serve as the interface between our MongoDB backend and hold the necessary logic to search over the database. This will include the ability to add/delete with accounts, update/add/delete crime reports and so on. Because the Model abstracts away the MongoDB database, any later changes in the database's design can be done without disruption to the rest of the application.

Design Details

Risks

The big risk in a design for an app like this is data privacy. Our solution to this is to completely anonymize the app and not have any user authentication with the data. This will allow us to still serve the mission of the app and the client's needs while meeting privacy concerns. This will diminish the complexity of the app, by reducing user authentication, but we plan on making up for the complexity with adding a support system for discrimination incidents and adding to the complexity of the Incident Map.

Platform

We opted for a mobile platform for Ripple because it allows a lot more accessibility. Someone is unlikely to experience an incident in their own house and having a desktop application wouldn't allow such easy accessibility. Additionally, desktops do not have GPS, so having a mobile application takes advantage of using current location if a user wants to either report or view incidents for a specific location.

Mobile Platforms and Deployment

With regards to platforms, instead of using iOS or Android specific languages like XCode, we opted for a framework that works on both platforms, React Native. Eric and Titus knew some React Native coming in so that was a determining factor. Also it is similar to React, so the learning curve is not as steep. React Native has platform specific components, both for iOS and Android, but we are making sure to use components that work on both platforms and not just one. For instance, if you take a look at this modal document, some props like onDismiss only work on iOS and some propos like hardwareAccelerated, only work on Android. If we used props like these, we would have to have two different sets of code, which defeats the purpose of us only needing to have one platform.

With regards to deployment, the app runs on the Expo Go app. Both Android and iOS run this app. For the purposes of this assignment, we don't have plans to deploy the app to any app stores, as there is a cost to deploy, especially for iOS, but this is a possibility for the future. With that being said, any device (iPad, iPhone, Samsung Galaxy, Google Pixel) that downloads Expo can run our app.

Algorithms

Searching

Searching will be done via database queries. In a select query, for instance, we will build up an index using the B+tree algorithm. In a join query, we will use an inner-loop join or hash join algorithm based on our data to achieve the best performance.

Input Validation

Input validation will be satisfied largely by dropdowns and the Google Maps API. There is no case where the user will be able to provide input that isn't through either a dropdown or the Google Maps API.

a. Incident Type Dropdown - No need for *Input Validation* check

Use the dropdown menu to provide comprehensive case types and avoid random or fake/prank input; Also convenient for statistics

b. Incident Details Dropdown - No need for *Input Validation* check

Use the dropdown menu to provide comprehensive case types and avoid random or fake/prank input; Also convenient for statistics

c. Locations - Handled through the Google Maps API

- Only correct location name and real location allowed. We will defer to Google Maps to validate this input as we will be using their API. Refer to the limitations section.

External Interfaces

Google Maps (via react-native-maps library and react-native-heat-map)

In order to provide the user a map of our data, we will obviously use a map. There are platform specific maps, like Apple Maps, but following the discussion on platforms above, we will use Google Maps because it is compatible across platforms. We will use the React Native Maps Library maintained by Airbnb that will make use of Google Maps with an API key. The library comes with built in components like <MapView /> or <Marker />.

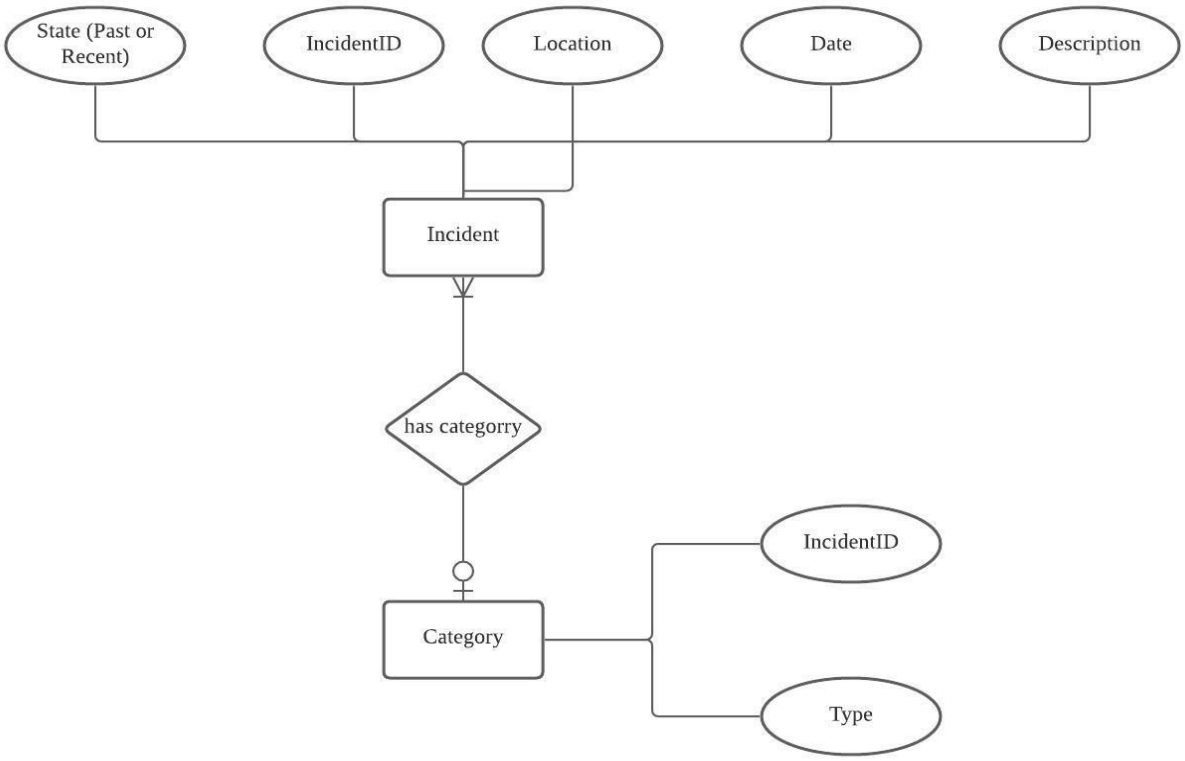
MongoDB (via pymongo library)

In the Flask application which is based on python as the programming language, it is sufficient to use the pymongo library to maintain and manage a database on cloud using the pymongo

interface. The tutorials and documentation are easy to find and understand: https://pymongo.readthedocs.io/en/stable/. The MongoDB database is stored on cloud with a visualized, interactable interface that can be accessed both on browser and MongoDB Compass application. The collections of the database can be both created in the application and in the code. Without a strict format on the data stored, the database is easy to work with and bug free.

Data

ER Diagram :



Design:

The design to satisfy the requirements is possible to be deployed with two collections of data. The Incident collection will store the data of each incident with its unique ID, state, location, type, date, and description. And the category collection will record the type of each incident. The major requirements of the database includes the rapid generation of a heatmap based on the number of incidents in a certain region, the cancellation of the visualization of incidents in the past, and the accessibility to details of the incident including its type and description. As a result,

a database with a single collection should be the most efficient to satisfy all the requirements with the least amount of data and processing time.

Data Collection:

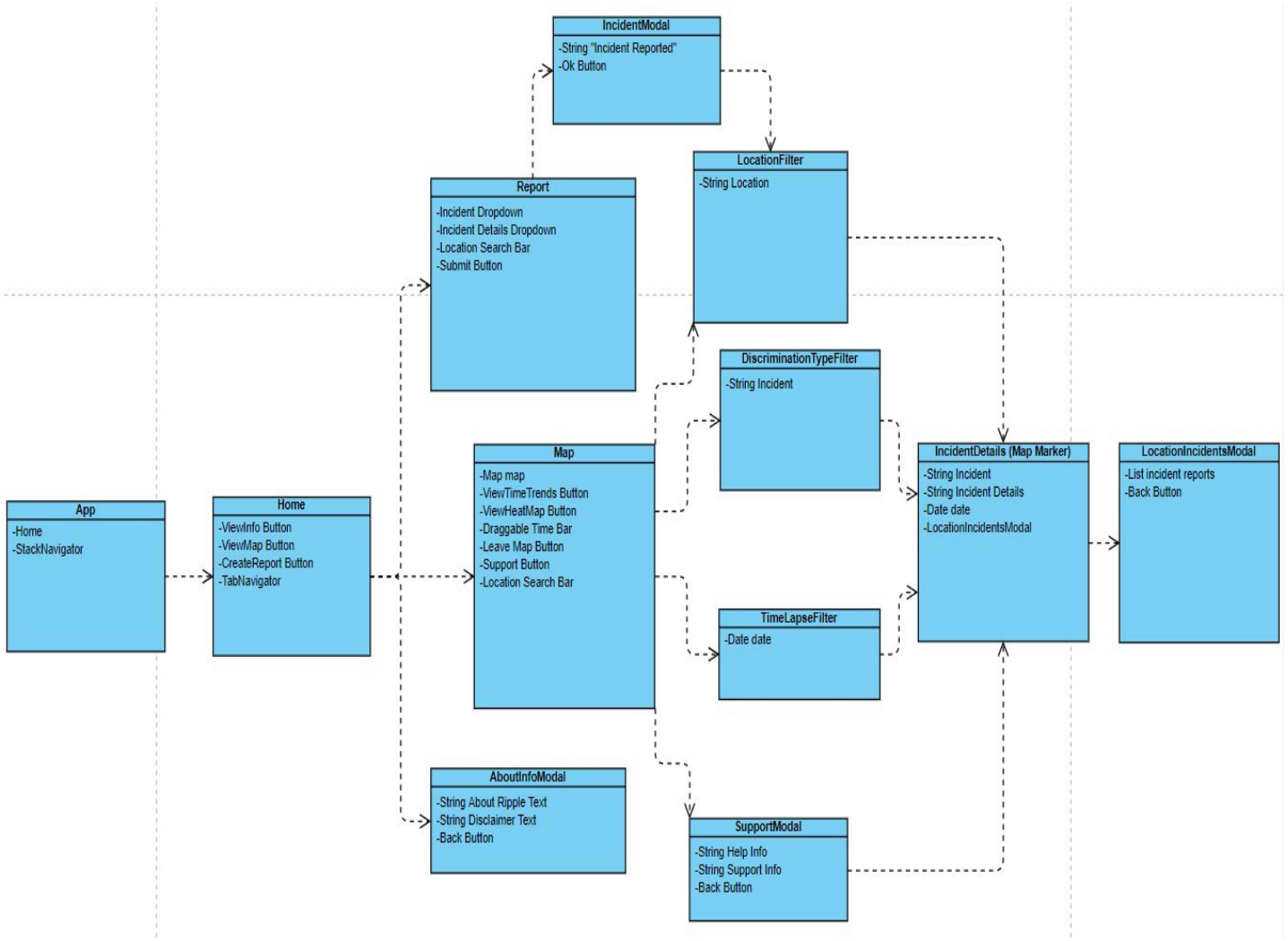
Our data will be collected via our online application across the country. Each user can access our application and fill in an incident report. The main difficulty in the collection of data can be the early stage data collection and the division into categories. It is not clear how many and what kind of categories that the incidents will have and if there is repetition in different naming of the categories. In general, we will get the data of the map via the Google Map API and the early stage data via use of the users in the late testing procedure.

Google Map API is already introduced in the external tools. We can use the data to generate the map for the users to choose from and fill in the reports. Then, the heat map generated from the data based on our database will be visualized via the Google Map API. This remains to be a core for our application and it should be reliable because of the reputation of the Google company.

Early stage report data will be collected via a small testing user group. They will use the application to test its functionality and therefore generate some early stage data that can be continued to be used in its publication.

Data Testing:
The database itself will be tested via written automated functions in Flask to run specific queries in the mongoDB database. We will add testing data and check if the database is returning the correct result as specified. In each iteration of our development, new testing data should be generated based on the new functionality added and the automated test should be run to check if the database remains solid. To check the boundary conditions of our application, we will also consider multiple use case scenarios to check if some possible input and request can be handled correctly.

Class Diagram



View Details

Name: Map Container

Content	Component	Description
Filter by Time button	Button	On click, all balloons not in the last three months are removed.
Location Search Bar	Text input	Text box for user to type in a location/city/store/etc to report an incident

Location Submit Button	Button	On click, a call to the API is made for that specific location
Incident Type Search Bar	Text input	Text box for user to type in an incident type (disability discrimination)
Incident Type Submit Button	Button	On click, all balloons not pertaining that incident type are removed
Heatmap Button	Button	If pressed, map will change to heatmap instead of the balloon based map
Discrimination Concentration Heatmap	Concentration Google Map	If the Heatmap button is clicked, this will become the new map that the user sees.
Discrimination Map for Specific Incidents (Balloons) Map	Specific Incident Google Map	The default map unless “heatmap” is clicked. Markers will show a list of other incidents at this same marker.

Name: Discrimination BalloonMap (Specific Incidents) Component

Content	**Component**	**Description**
Balloon Map	Google Map	An interactive map (move and zoom) given through the Google Maps API that shows various locations
Incident Markers	Google Map Balloon	A balloon that appears on the google map for each reported incident. If clicked, any relevant incident details are magnified in the balloon.

Name: Discrimination HeatMap (Concentration) Component

Content	**Component**	**Description**
Heatmap	Google (Heat) Map	An interactive map (move and zoom) given through the Google Maps API that shows various locations and their incident concentrations
Concentration Datapoint	WeightedLatLong Marker	A marker that shows the concentration of incidents at a specific location

Name: Landing Screen Component

Content	**Component**	**Description**
Report Discrimination Button	Button	If pressed, Landing Screen will change to Reporting Screen, where the user could report the location, type, and details of the discrimination incidents
View Discrimination Incident Map Button	Button	If pressed, Landing Screen will change to the Incident Map Screen, where the user could view time trends, heatmap and searching.
About Ripple Disclaimer	Button	If pressed, Landing Screen will change to the Info Screen Component, where the user could view the Ripple Disclaimer

Name: Info Screen Component

Content	**Component**	**Description**
About Ripple Message	Text	Shows information about the application, mission, purpose
Ripple Disclaimer	Text	Shows details of the Ripple Disclaimer
Back Button	Button	If pressed this button, it will control returning back to the Home screen.

Name: Reporting Screen

Content	**Component**	**Description**
Location Search Bar	Text input	Text box for user to type in a location/city/store/etc to report an incident
Incident Type	Dropdown	The dropdown menu provides a comprehensive variety of discrimination

		incident types, convenient for users to choose
Incident Details	Dropdown	The dropdown menu provides a comprehensive variety of discrimination incident details, convenient for users to choose
Confirm Report Button	Button	On click, the report of a discrimination incident (location, type, details) is recorded and the backend data is updated; A confirmation page will appear after a successful upload
Confirmation Page	Confirmation Page	Remind that the incident has been successfully recorded

Name: Confirmation Screen

Content	**Component**	**Description**
Report Confirmation	Text	Use words to remind that the incident has been successfully recorded
‘OK’ Button	Button	Click the 'OK' button to exit the Confirmation Page

Class Descriptions

Name: App

1. Home()
    a. This component is housed within App.js which will be the entry screen when using the app
2. StackNavigator()
    a. This layout for the app will put Home and the other tab navigators as part of the stack (so 2 entities on the stack). The stack will house the inner app tab navigators and Home. A back arrow will allow for movement from these views allowing for logout after the user types correct info from Home screen for login.

Name: Home Screen

1. ViewInfoButton()
    a. This button controls viewing the AboutInfo modal for about and disclaimer info.
2. ViewMapButton()
    a. This button controls viewing the world incident map.
3. CreateReportButton()
    a. This button controls for getting to the create report screen where users can create an incident report to the map.
4. Tab Navigator()
    a. This layout for the app will have Report tab and Map tab

Name: Report Screen

1. IncidentDropdown()
    a. This dropdown element will provide the user with incident options to pick from.
2. IncidentDetailsDropdown()
    a. This dropdown element will provide the user with incident response detail options to pick from.
3. LocationSearchBar()
    a. This location search bar which will be utilizing Google API location search, the user will select this location and when submitting the report save this location query string to post on the map
4. ReportSubmitButton()
    a. This button controls submission of the report form.

Name: Incident Modal

1. String IncidentReported
    a. This will be the selected Incident type reported for view on this confirmation modal “x incident reported!”
2. Ok Button
    a. This button will just serve to give update of status to the user that their report has been submitted and posted to the map view

Name: About Info Modal

1. String AboutRippleText
    a. This will be information about the application, mission, purpose
2. String DisclaimerText
    a. This will be disclaimer information
3. Back Button
    a. This button will control returning back to the Home screen

Name: Support Info Modal

1. String HelpInfo
    a. This will be help information regarding all the types of incidents that users can learn more about
2. String SupportInfo
    a. This will be support information regarding resources users can reach out to for assistance
3. Back Button
    a. This button will control returning back to the Incident report map view

Name: Location Incidents Modal

1. ListView IncidentReports
    a. This will be a list view of all the reports at the same location expanded from clicking the balloon marker of a particular location.
2. Back Button
    a. This button will control returning back to the Incident report map view

Name: Map Container

1. filterByTime()
    a. This method is involved when the filter by time button is clicked. All markers not in the last three months are removed from the map.
2. userIsEnteringIncidentLocation()
    a. This method will be be bound to the Google Maps API's Filter Search function and it will suggest possible addresses or building names as the user types in places
3. filterByLocation()
    a. This method is invoked when the filter by location button is clicked and the respective user input is not empty. This will both readjust the map location and make an API call to the database for data in a certain map window region.
4. userIsEnteringIncidentType()
    a. This method is invoked on press of the Incident Type Search Bar. This method will reduce the number of the dropdown so the user can choose from valid incident types.
5. filterByIncidentType()
    a. This method is invoked when the filter by incident type button is clicked and the respective user input is not empty. This will not make a call to the API, but instead, it will filter out the amount of data received from the API, depending on whether or not it's the respective incident type
6. toggleHeatmap
    a. This method is invoked on press of the View Heatmap button. It will change the map displayed to the Concentration/Heat map instead of the Specific

Incident/Balloon map. If there is already a heatmap, it will toggle the map back to the incident concentration map

Name: Discrimination BalloonMap (Specific Incidents) Component

This function has many built in functions that the component can subscribe to. Functions that we will use include:

1. onRegionChangeComplete
    a. This function will be used to give the controller the coordinates needed for an API request to get incident reports for a location
2. onMarkerPress
    a. This function is called when an individual marker is clicked. Upon clicking, a modal displaying more details about incidents.

Name: Discrimination HeatMap (Concentration) Component

1. onRegionChangeComplete( )
    b. This function will be used to give the controller the coordinates needed for an API request to get incident reports for a location

Name: DatabaseHandle

1. reportCreate()
    a. This method is involved when a user uploads a report, and will insert a new data record in the database
2. delateReportByID()
    a. This method is involved when the admin wants to delete a report. This method will find that report by its ID and delete it. Note: Since we don't plan on having admins on the user interface for now, this will simply be called on the back end.
3. getReportByCategory()
    a. This method will search the database and return the reports of a specific category.
4. getReportByDate()
    a. This method will search the database and return the reports created during a specific time period
5. getReportByLocation()
    a. This method will search the database and return the reports around a specific location

Limitations

Here we include some limitations of our design that did not seem relevant elsewhere.

Google Maps

For Map, we would install *react-native-maps* components for iOS + Android. And we would enable the Google Maps API in Google developer console. With this API, we can automatically handle access to the Google Maps server, download data, display map, and response to map gestures. In addition, we can use this API to add markers to the map as well as allow users to interact with the map. However, we might run into development limitations that will be beyond our control, like restrictions on what functions we can have our map subscribe to. For example, on our heat map, it's not possible to click a unique point, as this isn't a feature of the heat map. All of our maps will be limited in scope by the props and functions the map can subscribe to.

Platform Differences

One of the limitations is the difference between Android and iOS. Even though we are using React Native and it works across platforms which is a benefit, things will be displayed slightly differently, depending on how the react bridge works on that screen. For example, the SafeArea renders differently. Additionally, some functions which would be convenient to use we might have to bypass for the sake of cross-platform availability. See the comments on "Platforms and Deployment" above.

Implementation Plan

Disclaimer: There are many technologies we are new with so some estimates might be a little higher or lower than actuality. None of us have any experience with Jest, for instance. While the Spike Project is designed to get us experience with our technology stack, it didn't teach us how to use Google Maps API as well. All time units are estimates

Key:

- **People:** K (Kerui), E (Eric), J (Junyu), P (Penghai), S (Sunjun), T (Titus)
- **Task Type:** Blue = Logistics, Green= Development, Red = Testing

Iteration 1

	Task Name	Hours Estimated	People	Dependencies
1	Create Git repo (have a client folder and a server folder)	0.5	E	-
2	Construct Basic Reporting Dialog	3	E, S	1
3	Get API key for Google Maps and learn API	0.5	T	-
4	Construct Discrimination Balloon Map with Dummy Data	6	T	3
5	Create a Landing screen	3	S	1
6	Create App Navigation Skeleton	3	E	1
7	Create Map Container Screen	2	T	1
8	Create Incident Details Screen	2	T	1
9	Create Database on MongoDB Cloud	2	P	1
10	Create Incident class on Flask	2	P	1, 9
11	Create Category class on Flask	2	P	1, 9
12	Create Automated Test for Database	4	P	1, 10, 11
13	Create Incident related handler	6	K	1
14	Create Category related handler	6	J	1
15	Deploy the project on Heroku	0.5	J	1
16	Create Unit test of Incident handler	4	K	1, 13
17	Create Unit test of Category handler	4	J	1, 14
	Create Unit tests for Code Skeleton	6	E	1
	Create Unit tests for Map Screen and Container	4	T	1, 7

Iteration 2

#	Task Name	Hours Estimated	People	Dependencies
1	Implement Discrimination Map with API data	5	T	
2	Investigate Resources for Discrimination and Harrassment	2	E, S	
3	Implement Resources into Info Screen	2	E, S	2
4	Implement Map Filter: Concentration/Heat Map	3	T	
5	Implement Map Filter: Time Trends	1	T	1
6	Implement Map Filter: Discrimination Type	1	T	1
7	Implement Map Search: Location/Incident Type	2	T	1
8	Create Informational Screen Landing Page	2	E, S	
9	Create Query Methods according to Frontend Request	5	P	
10	Create Test Data for Query Testing	3	P	
11	Cover and test edge cases of Category handler	3	J	
12	Add access control and request validation for Category handler	3	J	
13	Add some security strategies for Category handler	3	J	
14	Design and implement searching algorithm	8	K	
15	Determine valid input	4	K	
16	Integrate Modals with API data		S	
17	Create Query Testing Plan on Postman	3	P	

Iteration 3

#	Task Name	Hours Estimated	People	Dependencies
1	Recruit end users for usability testing	4	Everyone	
2	Conduct Usability Testing with various use cases	6	E, T	
3	Brainstorm front end edge cases and testing	4	S	
4	Save Early Stage Data to Database	3	P	
5	Add more Flow Control strategies for Category handler	3	J	
6	Conduct Frontend Performance Testing for Internet Speeds	2	S	
7	Brianstorm Backend edge cases	4	K	
8	Implement fixes for backend edge cases	4	K	

Testing Plan

Unit Testing

Using unit tests is to make sure that individual sections of code are correct. Everyone will be responsible for writing unit tests and unit testing any code they write. We chose to do unit testing from several aspects.

Back End Unit Tests

Our server is developed by Python Flask. The test plan to cover the server side will consist of two parts: database and interface. It should be reasonable for us to test our own individual parts of the interfaces that we write for the frontend request. And the database management system is going to be tested as a whole.

MongoDB Cloud Testing: It must first be checked that the database performs according to the ACID properties: atomicity, consistency, isolation, and durability. Then we must check if the data integrity is maintained across different online users. The main operations of the database should be insertion, deletion, and searching queries. We will implement different testing cases for the three parts to make sure each is working properly.

Reference: https://www.softwaretestinghelp.com/database-testing-process/

Backend Interface Testing: We will test our backend interfaces with both Postman Cloud and simple pytest. Each interface will be implemented together with a unit test to certify its correctness. The tests should be written before the actual code. Also, the test should be reusable to ensure the solidity of previous interfaces after certain changes are made to the database or other helper functions. If necessary, unit tests should also be written for individual helper functions inside the classes.

Reference: https://learning.postman.com/docs/writing-scripts/test-scripts/

Front End Unit Tests

Because of our decision to use React Native, we don't need to have differing unit tests for Android and iOS. Jest, developed by Facebook, is an industry standard testing framework for React Native. This should be very easy to set up as the jest-testing library is simply an npm package. We also acknowledge that as we are inexperienced as testers so this is subject to change. We are using this article as our guide.

The three main functions for unit testing in Jest are use describe(), it() and assert(). Describe() creates a test case grouping, or a test suite,and it() creates a single test case. Test cases are successful if their assertions, via assert() and a *matcher*, are true. Some examples of matchers are toEqual(), toBeTruthy(), and toContain().

Integration Testing

Frontend Integration Testing

Integration testing we will combine individual software modules and test them as a group. So in the frontend we are using Jest for tests. How this will work is we will create tests that encompass the scope of all the individual components. For example, in our project once we have developed the report and map interface, we will create an integration test of if a user makes a report does that report get reported correctly on the map interface and does all the reports from all users as a whole get reported correctly on the map. In addition, we can test integration with all our map modes (heatmap, location search, time incidence analysis) as these will incorporate the map components and report components.

Backend Integration Testing:

As the objects related to the table element of the database will be hidden in the inner level of the backend Flask application, it is important to make sure the inaccessible parts are working to support the interface used by the frontend team. The integration testing is responsible for the grouping of test cases used to test the use of certain classes and functions combining together in the interfaces. Postman Cloud is also suitable for the integration testing as it can test the HTTP

request methods targeting specific interfaces. Also, testing scripts implemented in the Postman can be run in collections. As a result, the test can be automated to be run after every update to the application itself to ensure the integrity of the program.

System Testing

Frontend Integration Testing

We will test the software application as a whole, focusing on system elements such as computer hardware, external equipment, third-party software, data and personnel, and environmental factors, etc. We will have different team members simulate users to perform operations in the client to test all possible user interface components that can be performed on various operations. For our user interface interaction testing, we will test all the APP functions through touching the screen and all the useful buttons on EXPO by instrumentations to send requests and events directly.

Backend System Testing

System testing is a more comprehensive and holistic test, and here we are going to test that my entire back-end logic code is working correctly. We're going to do this in an automated fashion. The first step is to run it on the local machine and access APIs one by one by the local ip through python scripts (selenium) carrying parameters to ensure that each API is executing the logic properly and interacting with the database properly. The second step is to deploy the backend to the cloud server. We will use a python script to simulate the request behavior of the front segment, access the public ip and port, and observe the logic execution and the database.

Regression Testing

Unit and Integration tests will be run after each user task is completed, front-end or back-end. While this will increase the hours invested, this should be minimal as we can script having all the tests run together. This will likely be subject to change as we learn more about testing with the Testing Module in Lecture still weeks away. System testing can be scripted apart from Unit and Integration testing.

Performance Testing

Backend Performance Testing

For the backend, there are two aspects of performance testing. The first one is the average response time test. We should request and calculate the average response time, and give a detailed analysis for each API. The second one is the pressure test. We need to use some tools to mock high Query Per Second(QPS) requests and record the highest QPS the backend can stand for, and what behavior the backend logic will have if the QPS exceeds the system's limit for the backend.

Frontend Performance Testing

Internet connection is the only real thing to test from the frontend perspective. Hardware also comes to mind, but since Google Maps is optimized for phones across the decade, we will assume our app will not be used on a phone older than a decade.

From an internet perspective, we can test the qualities of various maps at different internet speeds. This will be possible, as we have team members living all over the world, and we can also test on WiFi and data. We can also test to ensure that even if a user doesn't have internet, that their incident report will send on connection to internet.

Usability Testing

Usability testing, or compatibility testing, will naturally need to be done as with any user interface. There is no true automated way to do this, so we will plan on doing testing with some sample end-users during iteration 3. Eric and Titus have experience with UI design through a previous course, so there shouldn't be any giant design flaws. Nevertheless, we will plan on testing with at least 5 end-users to gather feedback for the design and possibly change the design.