POLITECNICO
MILANO 1863

Computer Science and Engineering

A.A. 2019/2020

Software Engineering 2 Project:

"SAFE-STREET"

Desgin **D**ocument

December 14, 2019

Prof.Rossi Matteo Giovanni

Amirsalar Molaei

karim Zakaria Saloma

Erfan Rahnemoon

Contents

Figures

Tables

1 Introduction

The Software Design Document is a documentation of the intended system design used to convey the expected output of the development phase. In this section, an overview of the content and intended use of the document is discussed.

1.1 Purpose

The Software Design Document is built to describe a detailed description of the *Software To Be* from the architectural and technical aspects. This document specifies the manner in which the software shall be built through the use of narrative and graphical tools to aid in the communication of the necessary information to the concerned audience.

This document is intended to provide a clear and complete description of the system to the persons who shall be developing the system. In order to, assist in the understanding of how the system should be built and how the end product should function in accordance with the previously decided upon requirements and specification of the system.

1.2 Scope

In this section, the scope of the system previously described in the *Requirements and Specification Document* shall be revisited. As well as, consider some more in-depth aspects of the system.

As previously stated in the *Requirements Document*, the *SafeStreets* system shall be providing four main functions to various users; in this section, the system boundaries and scope used to define the limitations and different responsibilities of the S2B.

The first of the main functionalities is the enabling of users to report traffic violations. Regarding this, some phenomena are regarded as world phenomena not viewed by the system due to its limitations such as the fact that the system does not directly detect a violation. However, it can be accounted for by the system through a traffic report made by the users. Moreover, another functionality that has to do with the users is the publishing of collected data to be viewed by the users in a refined representation to help them consider the safety of various areas based on traffic violations. The data is also communicated to the authorities but with different levels of details.

The other two main functions have to do with the *SafeStreets* system providing services to government authorities. The domain limitations of the system affecting this interaction are also discussed in this section. Such as, the fact that the system is only able to make suggestions for preventive measures to the authorities based on the accident data that have been communicated. Meaning, that the system does not have any knowledge of accidents unless they are reported by the authorities and that the system can only suggest interventions and neither put them into place nor can detect them being applied. Moreover, a second function to the authorities would be the communication of traffic reports received from users to be later used by government officials to give out traffic tickets, the system responsibilities to support this process is to prevent the users from tampering with images *digitally* and to provide the collected reports to the authorities proactively. In other words, physical tampering with license plates to mislead authorities and the actual process of giving out tickets is not part of the application domain.

Moreover, in this document, some more technical issues regarding the functioning of the system need to be discussed. Primarily, the security aspect of the system; more specifically, data security. Since the system will be dealing with the collection and communication of sensitive data while performing more than one of the main functions; the system must, at all times ensure the safe transmission and storage of data and the application of measures to prevent any means of data tampering. The data being discussed includes but is not limited to, user personal data and detailed data of traffic reports.

1.3 Definitions,Acronyms,Abbreviations

1.3.1 Acronyms

S2B *Software To Be*

GPS *Global Positioning System*

UI *User Interface*

GDPR *General Data Protection Regulation*

UML *Unified Modeling Language*

RASD *Requirements Analysis and Specification Document*

OS *Operating System*

SMS *Short Message Service*

MVP *Model View Presenter*

REST *Representational State Transfer*

API *Application Program Interface*

VPS *Virtual Private Server*

ORM *Object-relational mapping*

SDK *Software development kit*

1.4 Revision history

Revision	Date	Author(s)	Description
0.0	24/11/2019	karim Zakaria Saloma, Amirsalar Molaei, Erfan Rahnemoon	First document issue

1.5 Reference Documents

References

[1] *Standard for Information Technology–Systems Design–Software Design Descriptions IEEE 1016.* 2009.

[2] UML,
`https:omg.org/spec/UML`

[3] Specification Document. *SafeStreets Mandatory Project Assignment.*

[4] Microservices Architecture,
`https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices`

1.6 Document Structure

The Software Design Document(**DD**) is comprised of six main chapters. Which shall be described in this section of the document:

Chapter 1 (Introduction): provides an overview of the document as a whole; describing, the various sections constituting this document, as well as, the intended use of this document.

Chapter 2 (Architectural Design): a detailed description of the architecture to be developed during the implementation phase of the system; spanning from a high-level component view to a detailed

run-time description of the different modules of the system. This is used as a guideline for the development team in order to have a clear idea of how the system should be built.

Chapter 3 (User Interface Design): an overview of the design of the different interfaces that the users of the system shall be interacting with the system through; in order to utilize the functionalities of the system according to their needs. This overview is concerned with the visual aspects of the user interfaces.

Chapter 4 (Requirements Traceability): provides a link between the design decisions in this document and the requirements of the system described in the *Requirements and Specification Document*. This is done by providing an explanation of how the system design described in this document fully satisfies the requirements the system must abide by.

Chapter 5 (Implementation, Integration and Test Plan): describes the approach to be followed during the development and testing phase of the system. This is also provided as a clear guideline for the development team to follow.

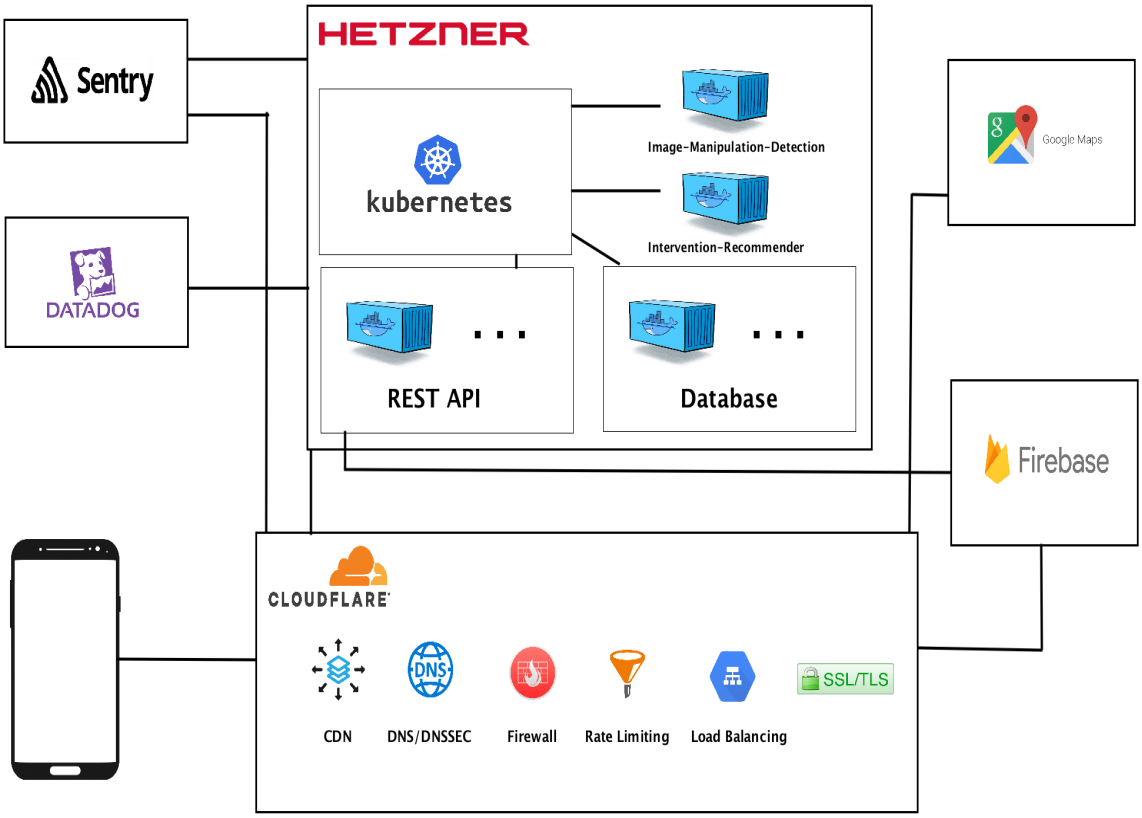
Chapter 6 (Effort Spent): summarizes the efforts of the team members in developing this document in terms of time spent on each of the sections of the document.

2 Architectural Design

2.1 Overview

The high-level architecture of the system to be developed is illustrated below, which gives abstract information about different parts of the system. For the backend part, the system uses a VPS which is hosted on Hetzner company, renting VPS provides the possibility of avoiding problems related to managing server's hardware and configuration. Furthermore, in order to guarantee the flexibility and scalability of the system for the future, we use *Kubernetes* which is an open-source container-orchestration system, it allows us to maintain and manage containers in real-time. Moreover, for development and deployment environment, we use *Docker* containers for implementing microservice architectures that will be elaborated in detail in the following sections. In this case, all services can operate independently.
What is more, for logging and error tracking both in frontend and backend, SafeStreet takes advantage of *Sentry* which is a monitoring service application that allows us to track chaotic and inevitable errors in the SafeStreet system. Apart from this, *Datadog* which is a monitoring service for applications is used for VPS monitoring, aiming to display relevant information of the VPS in real-time through a user-friendly and professional dashboard.
As a reliable geo-location service, the system uses the *GoogleMap API* that provides rich functionality for our system. When it comes to security, the system utilizes *Cloudflare's* CDN, DNS/DNSSEC, Firewall, Rate limiting, Load balancing and SSL/TLS to ensure data confidentiality and integration. Additionally, for the aim of authentication and storage *Firebase* is used, because of its simplicity that helps developers to implement their application.

Figure 1: High Level Architecture



2.2 Component View

In the component diagram, four parts of the system are the API from other companies that provide some functionalities as service. The Google map API provides the map and Geo-location services for getting the map in the mobile phone app and injecting the data comes from the server. Firebase provides easy authentication for mobile users by push notification which uses the cloud messaging REST API of this service, this will help to reduce the app expenses because we pay for what we used unlike any SMS API and it is way more easy to integrate into the android app and another side pros is avoid email or SMS spamming. Besides the storage REST API of the Firebase is used to store violation images. Storing the amount of image that the program is going to work with it needs too much space and handling a database specialized for multimedia usage, but as said previously, in Firebase we pay for what we used and for sure this method has more advantages to buy the storage beforehand.
Sentry system helps to log the system in certain parts and also makes a bug tracking mechanism for the problem happing in the android app and also server-side. Moreover, the database component is the representer of the database for each component of the system to simultaneously follow the microservice best practice and hold the scalability of the system. The components of the system will work with the database through the ORM used in the system, which is the SQLAlchemy. The authentication, storage, and map API are connected to the android app and server, which as shown in the diagram each of these interfaces has their own usage and the functionality of them is not the same. Furthermore, two authority API is assumed in the system because in one of them the manipulability sends the data for the system, and in another they the system will send the data related to violation reports and interventions by their request. The authorities usually work with a lot of APIs, and usually, they are not integrated into one system, therefore to make the system as compatible as possible with existing systems this architecture is chosen.
On the server-side, for more isolation, the system is broken into seven-part which help to easy deployment and further development in the system. There is no direct communication between the component and for the accessing the data related to one of these components from another one they will communicate like two separate systems, for example, the API provided by the "RegistrationAuthentication and login" part which the ViewMe and ReportViolation part are using this API. In this case, the reasons to provide internal API instead of directed access by the functions to this subsystem is first to make the system more scalable in future development; second, for security which this part has sensitive information about the user is better to be isolated from the whole system and has more strict rules to access to it. The reset of subsystems do their job without any knowledge about the other parts of the system, and there is no direct connection between the outside of the system and database. Even for the data which comes from outside first, the data will be stored in the database by responsible component for that request or response and then it can be used by other components by request. The component of the system will be described in the following:

The ReportViolation will handle the functionalities related to reporting a violation or editing or removing a report, after the request received by this component based on the request it will perform the required functionality. For submitting a report all data will be stored in the database of the after the component checks the user-ID by the request from registration and login component and will store the URLs provide ed for images by the firebase. For editing a request the component only allows the user to edit the type of violation and comment part. Finally, for removing a report will be removed from the database.

ReportSubmission is responsible for the request from authorities which after getting the request based on the type of the report will send a request to the report violation part or intervention recommender then when the report was ready it will store the data related to the report in its databse.
ViewMe is part that allows users to change the data of their profile and see their previous reports. This part of the system after getting the request will ask from report violation or register and login components based on the requests to get the data related to each part and for other type requests like edit or remove will do the same. The reason for this component in the backend is the one-to-one mapping of the part from client-side to server-side to provide an easy path for future development.
ViewSafety will manage the request for the data, the system needs to provide for showing safe areas

on the map on the client-side. This component after getting the request will ask for the violation report from the report violation component and at the same time will send the request to the authority database to get the accidents, after all, will make a proper response to the request by collected data. The "RegistrationAuthentication and login" is in the head of user registration and credential of the users to work with the system. This component after getting the registration request will use the Firebase API to verify the use of authentication and also get the specific user-ID assigned to the user to store it in the database for furher functionalities. Also, other components will send requests to this component to verify a user before sending data for the client-side.
There is two part of the system which they do not provide any service for the normal end-users one is the interventions recommender which will analyze the data and recommend the necessary intervention to authorities to decrease the number of violation and accidents for its functionalities this component will send requests for the report violation component. Next is the mechanism to check the images of the violation report from users and detect those who are manipulated and they are not original which if the image is manipulated will send a request to the report violation to flag the report.
In the android-side of the system, there is one important component, which is the plate number recognition which detects the plate number by images is taken by users and shows the plate number in pre-report. The last component in the android app is the local database, which is necessary to store some data about the user and its reports. Nevertheless, the android app has some other part which they just representer and can not assume them as a component.

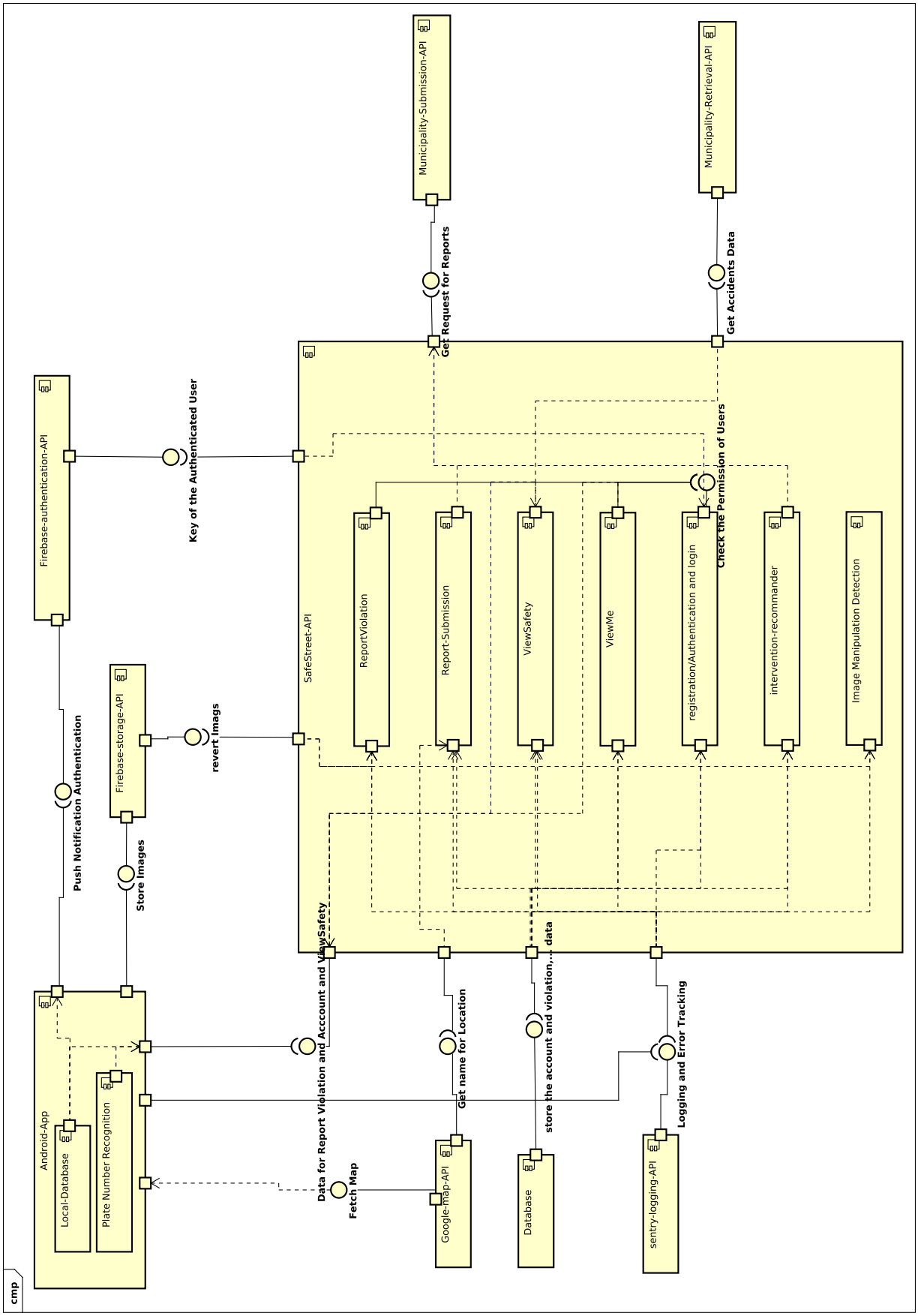


Figure 2: Component Diagram of whole system

2.3 Deployment View

For the deployment of the project, the automated approached is chosen, which can be seen in the Deployment diagram. First of all by each merge in the master branch of the server-side of the repository the Travis CICD will run unit-test and integration test on the system and if all of them was successful then will check the style of the code, and if everything goes right it will make the docker image from the changed service. Then Travis CICD will connect to the Kubernetes through the ssh and deploys the docker image made in the previous section. Each component will be deployed in a separated container for better isolation and a further improvement of the system in the future, also each component has it owns database in a separated container for better scalability. Further, because we used the SQLAlchemy as the ORM of the system then we need a bytecode of each model (represented as an artifact) in the related REST API container, so the deployed component in the container could communicate with the associated database container.
To avoid a complex diagram we just showed one container Database, which is the representer of all the containers. Except for IMD container and recommender container which include the image manipulation detection and intervention recommender respectively; the reset can be scaled based on the upcoming load to more than one container but these two, because of batch execution always will be one instance of them. Reset of the devices in the diagram is the infrastructure of other companies which provide a different kind of functionalities as service. In the reset each of the devices and their entity will be described briefly:

- *Github infrastructure:* The repository of the code in one Github infrastructure which will help to have complete and distributed control one the code and also make a concrete Gitflow, to avoid conflict between different parallel feature or version of the system which are developed or are under development. Also, provide a hook for the other service like CICDs to access the code.

- *Travis-CI:* The continuous integration and delivery will use the Travis infrastructure this service will test the code in different approaches and check the style of the code. To have the same code schema and test the code to avoid crashes of the system for predictable inputs.

- *Kubernetes:* This part of the main server of the system will help to could manage container and their connection with each other to could scale them as easy as possible.

- *REST API:* each REST API container is one the subcomponent of the REST API component in the Figure (X) which the functionality of each one is described in the component view of the system. But, each of these components in the container is developed in the way that it has responsible for requests and responses related to itself, in other words, each container will handle an event end-to-end.

- *Database:* the database container is the representer of the private database for each component in the REST-API. This means each container in the REST API has its database container to have isolation between services and also could scale each part separately.

- *IMD Container:* IMD component will run a Cron job to detect the manipulated images in this component and will store the ID of the image in its database. This component, because its intrinsic property does not need scaling so always, will be one but its database could be more than one based on the speed and storage the system needed.

- *Recommender container:* This container like the IMD contain the intervention recommender which runs a job based on reports and accident to suggest some mediations for authorities.

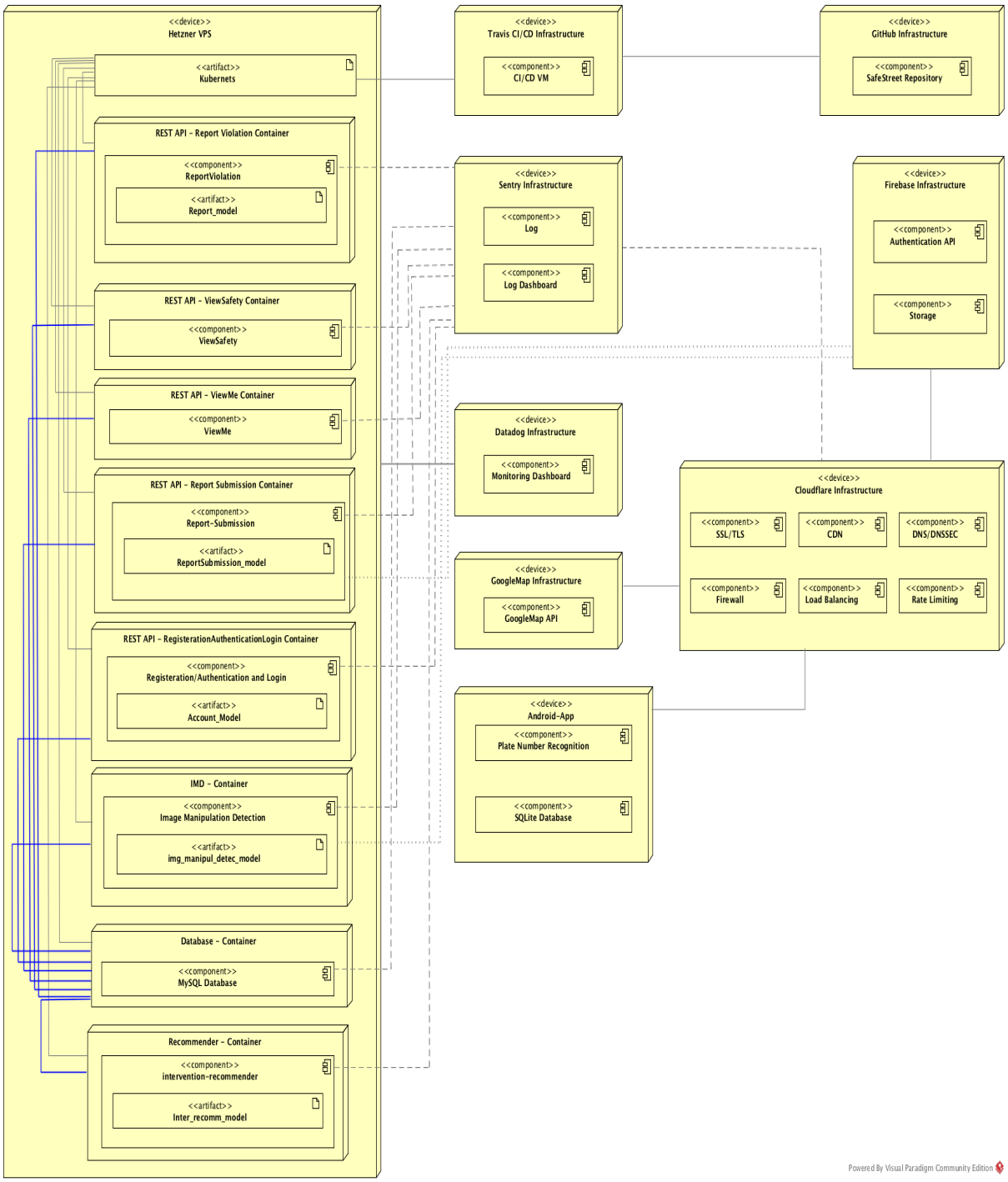
- *DataDog:* In the monitoring of the infrastructure of the system the service of DataDog is used which allows for common monitoring tasks with simple configuration, and the Dashboard for this task is in the DataDog infrastructure and using the system just need to install a package in the host OS.

- *Sentry:* for the logging and error tracking of the system itself the service provided by the Sentry is used. The reason for this is an easy setup and configuration which sentry provides for logging

and error tracking. Also, in the error tracking of the android app, this service is used again. Using this service in the system just needs to use their SDK in our system.

- *GoogleMap API:* All the functionalities related to the geo-coordination will be handled by this service because it is the most comprehensive and reasonably reliable option in the mark with proper price.
- *Firebase Infrastructure:* Firebase has different API for the developer which in the system the authentication and storage services are used. The reason behind this decision is the authentication is the reliability of this service which implementation of JWT is too much risk because of the escaped bug and it is the invention of the wheel again. In the storage part because the system needs to store the images of the violations there is a requirement for powerful multimedia storage and database. This service will solve the problem for storage without any hassle to configure and manage the storage space for this kind of file and has a fast response time.
- *Cloudflare infrastructure:* The SSLTLS service will help the system to guarantee the security of the massages communicate between the client app and the server-side. Both the load balancing and CDN services will help to API work faster in heavy load and load the binary files faster. Firewall and rate-limiting services avoid misuse of the API by anyone who wants to use the API functionalities in abnormal numbers like bot which sends to thousands of requests in a second and also these two services will avoid DDOS attack. The DNSDNSSEC service will map a URL to the IP of the server and also avoids the DNS attacks.
- *Android App:* unlike the rolling update of the server-side the android side has a manul deployment in which the release version of the app will be uploaded in the google play manually. The app includes two main components which one is the SQLite database for all the needs to store data on the client-side. The plate number recognition component will be upgraded by each release of the Android app. In the final, it is necessary to describe all the communication between the service in the server that will happen by RabbitMQ and each component in the server implemented in the structure of the REST.

Figure 3: Deployment Diagram



2.4 Runtime View

This section of the document is concerned with the dynamic interactions between the various components of the system during runtime to achieve the desired functions of the system. The section is constituted of four major subsections grouping together various runtime views of different parts of the system. These subsections are as follows *general functions, violation reports, view safety* and *municipality interaction*. In the following diagrams some simplifications were implemented in order to remove redundant details; however, they should still be mentioned, such as, that each and every server module interaction should correspond to a *log entry*, and all exceptions that may occur in the system should be error logged. Note that in the diagrams, some examples are used for logging and error logging interactions. Moreover, it is noteworthy that all external incoming and outgoing data passes through the *Security module* encryption and decryption and the same process occurs on the mobile app.

2.4.1 General Functions

This first subsection considers the runtime behavior of system components for the delivery of basic general functions. Specifically, *registration and verification, login* and *profile viewing and info editing*. The first two runtime diagrams below describe the user registration and verification process. As can be seen in the diagram a new user fills in the registration form with his information the submits the form. The data is relayed to the *Safestreets-API* which saves the user data and prompts the *Firebase-Authentication* server to verify the new user. Which in turn, sends the verification message to the user and returns feedback of the user confirmation.

Figure 4: Registration Runtime View

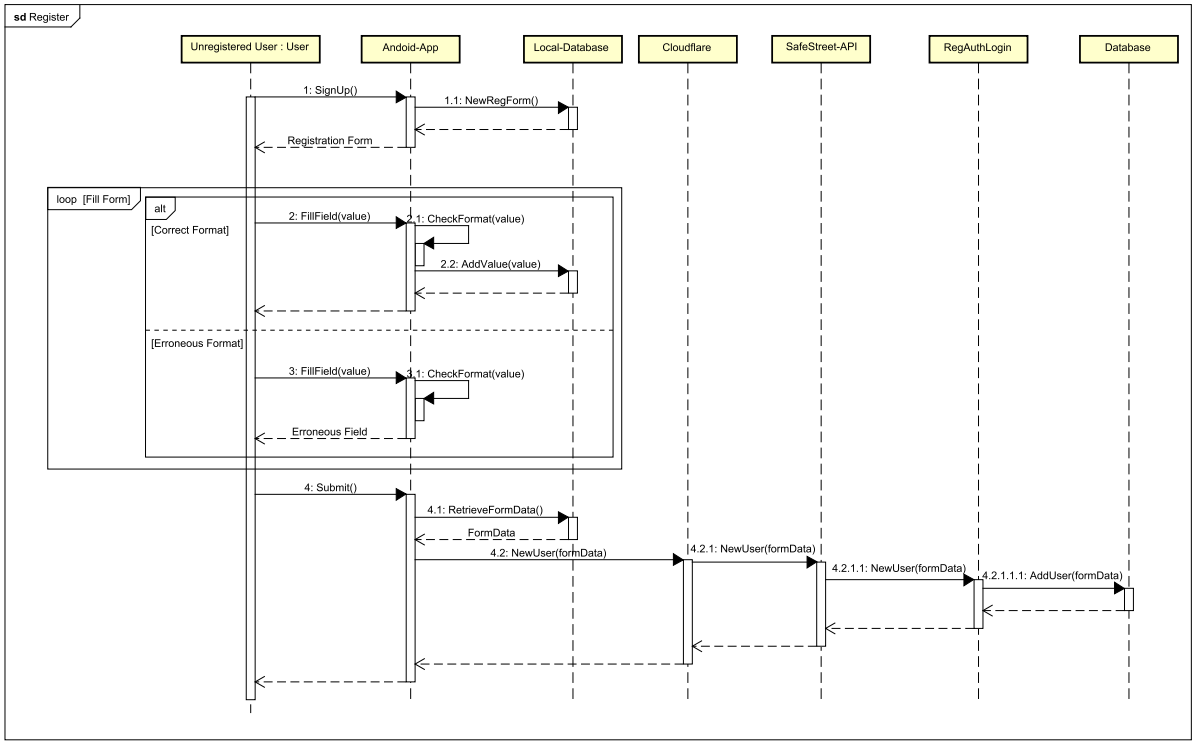
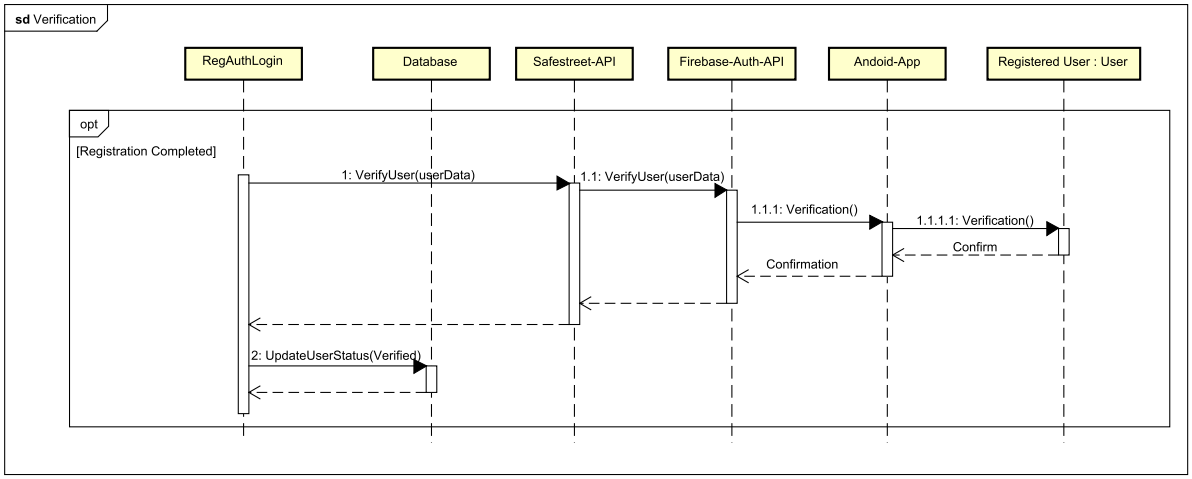
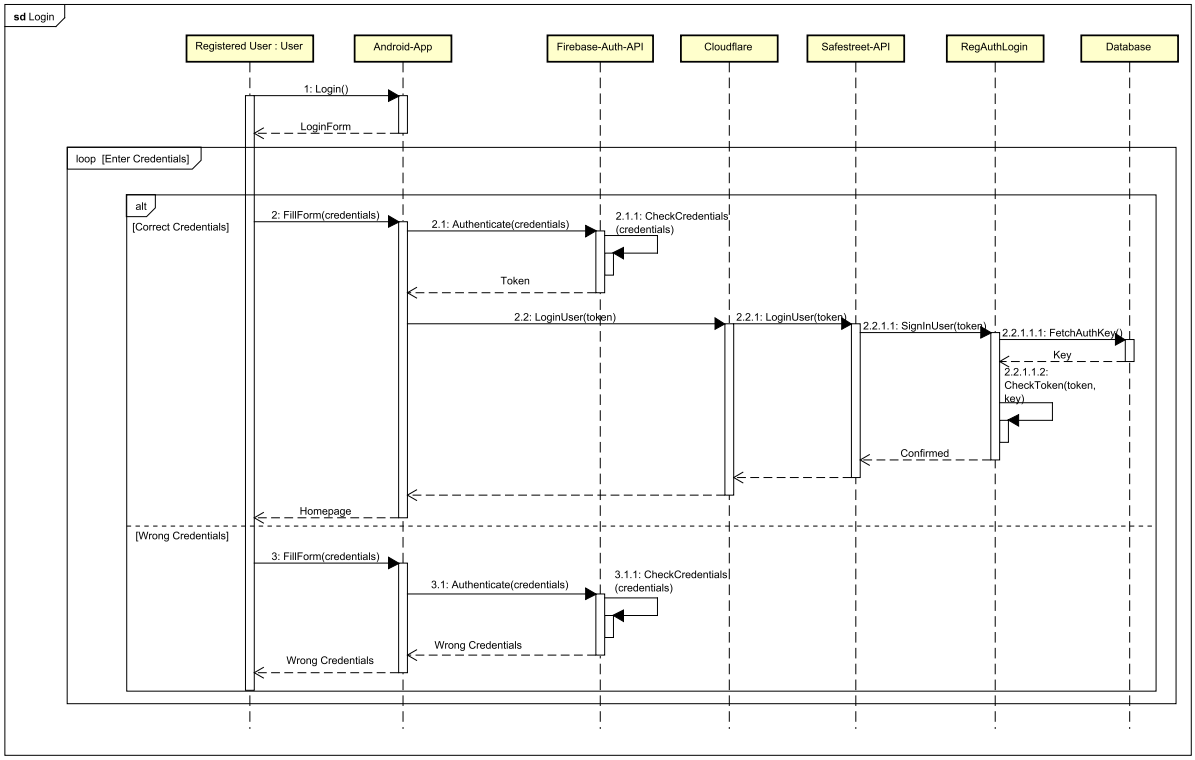


Figure 5: Verification Runtime View



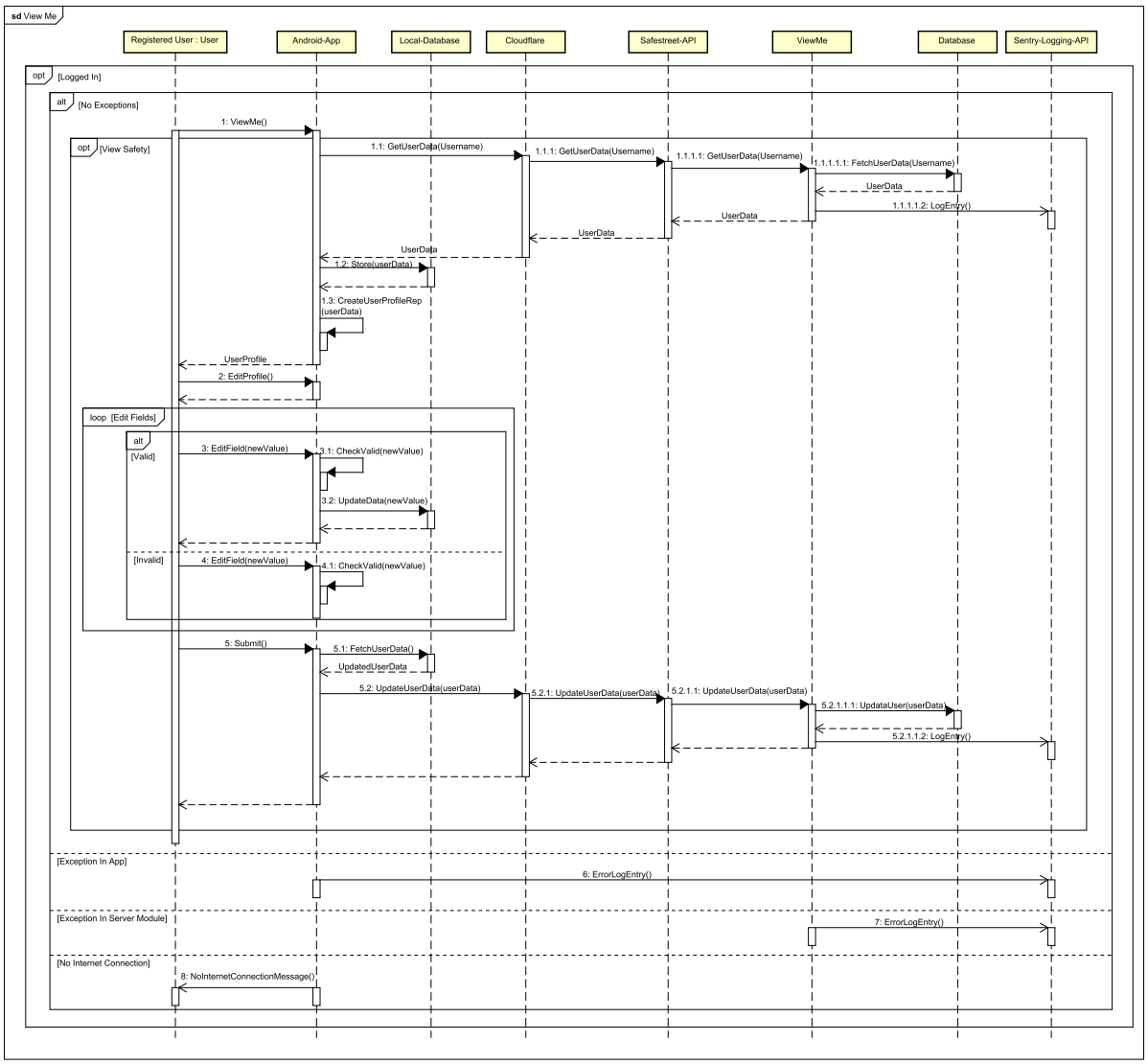
The below figure describes the user login process. Which again, relies on the *Firebase-Authentication* server to authenticate the user credentials and issuing a token to be sent to the app server to confirm the authentication process.

Figure 6: Login Runtime View



The last of the general functions in the figure below is the *ViewMe* functionality. Where the user can view their profile and edit their personal information.

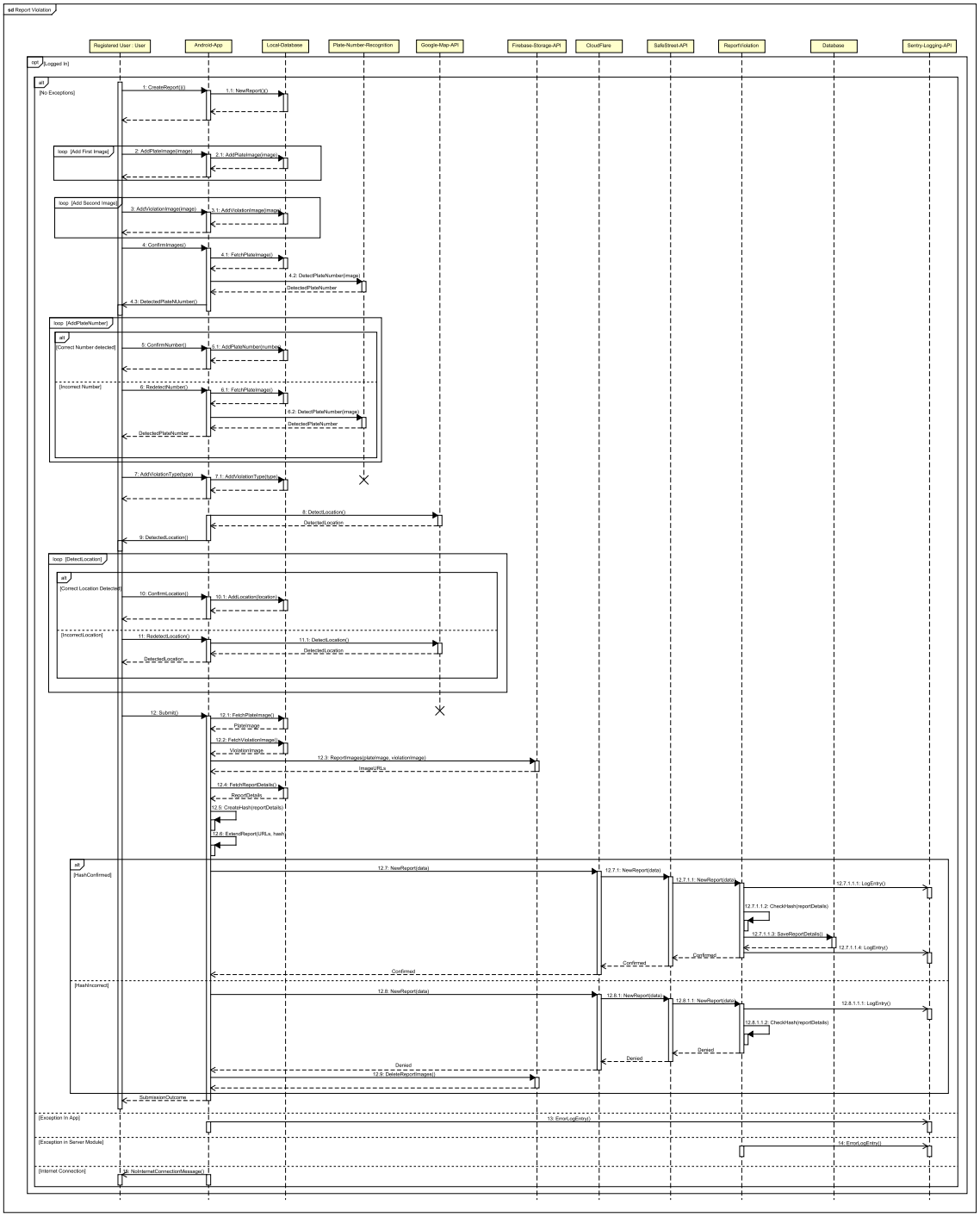
Figure 7: ViewMe Runtime View



2.4.2 Violation Reports

The process of *Violation Report* submission performed by the users is constituted of the user report formulation including images showing the violation, the location, and other useful information. The images are saved on the *Firebase-Storage* and their respective URLs are used to access them for any future use. Finally, the report details, image URLs and a computed hash which is verified to ensure data integrity are sent to the server. A detailed breakdown of the step-by-step process of violation report submission is described in the following figure.

Figure 8: Report Violation Runtime View



The report history function described in the following figure is a function offered to the user to search through reports that were previously submitted by them according to criteria that they define. Such as, reports that were submitted in a certain time period or a certain type of violation reports.

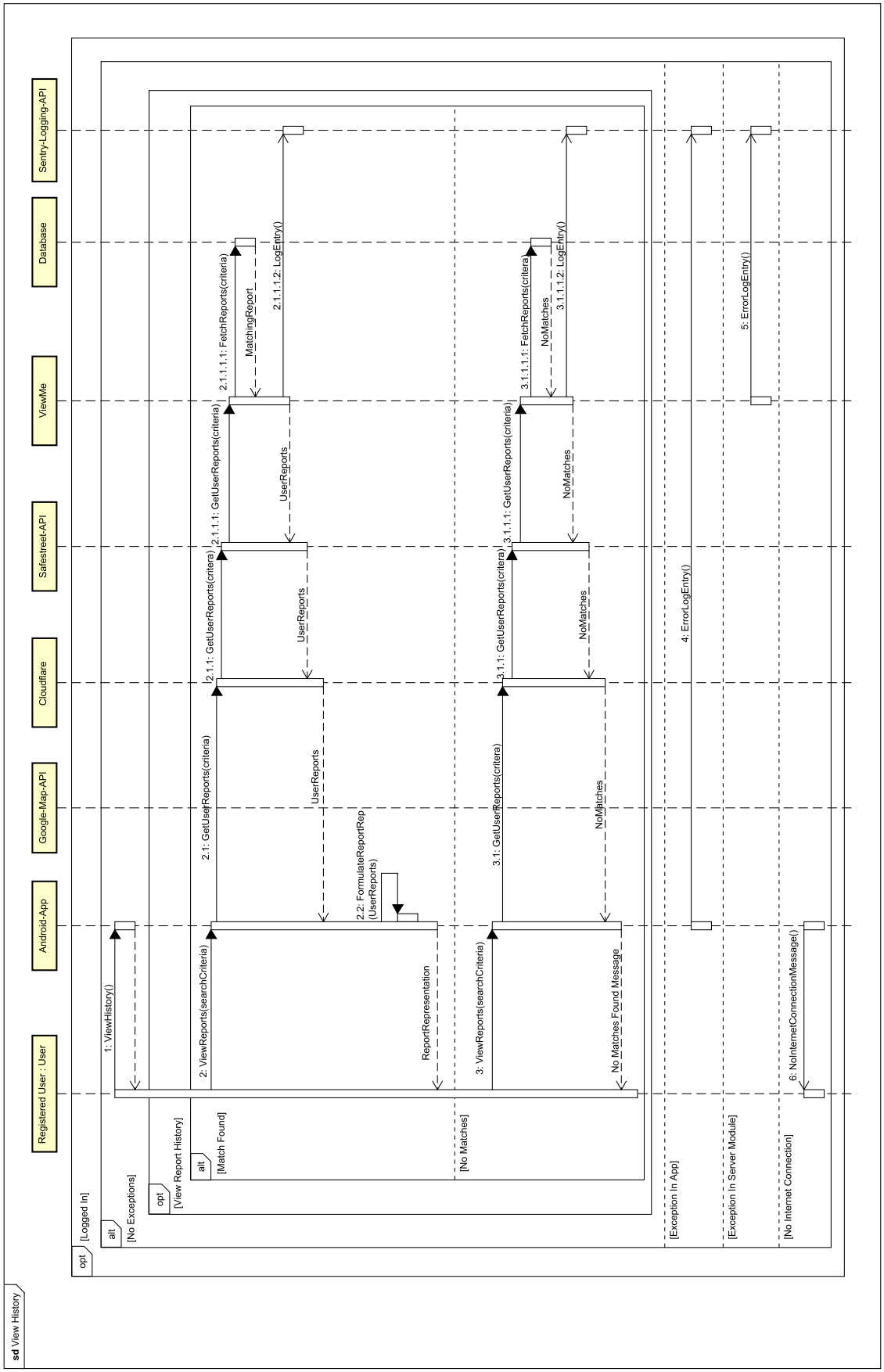


Figure 9: View History Runtime View

2.4.3 View Safety

This subsection discusses one main feature of the *Safestreets* that is the *View Safety*. This feature entails the graphical representation of the safety of the various areas according to the user-specified search criteria. Where the user would search for a certain area and the app would retrieve report data of both violation reports filed by the users and accident reports retrieved from the municipality and formulate a geographical representation using the *Google Map API* as can be seen in the figure below.

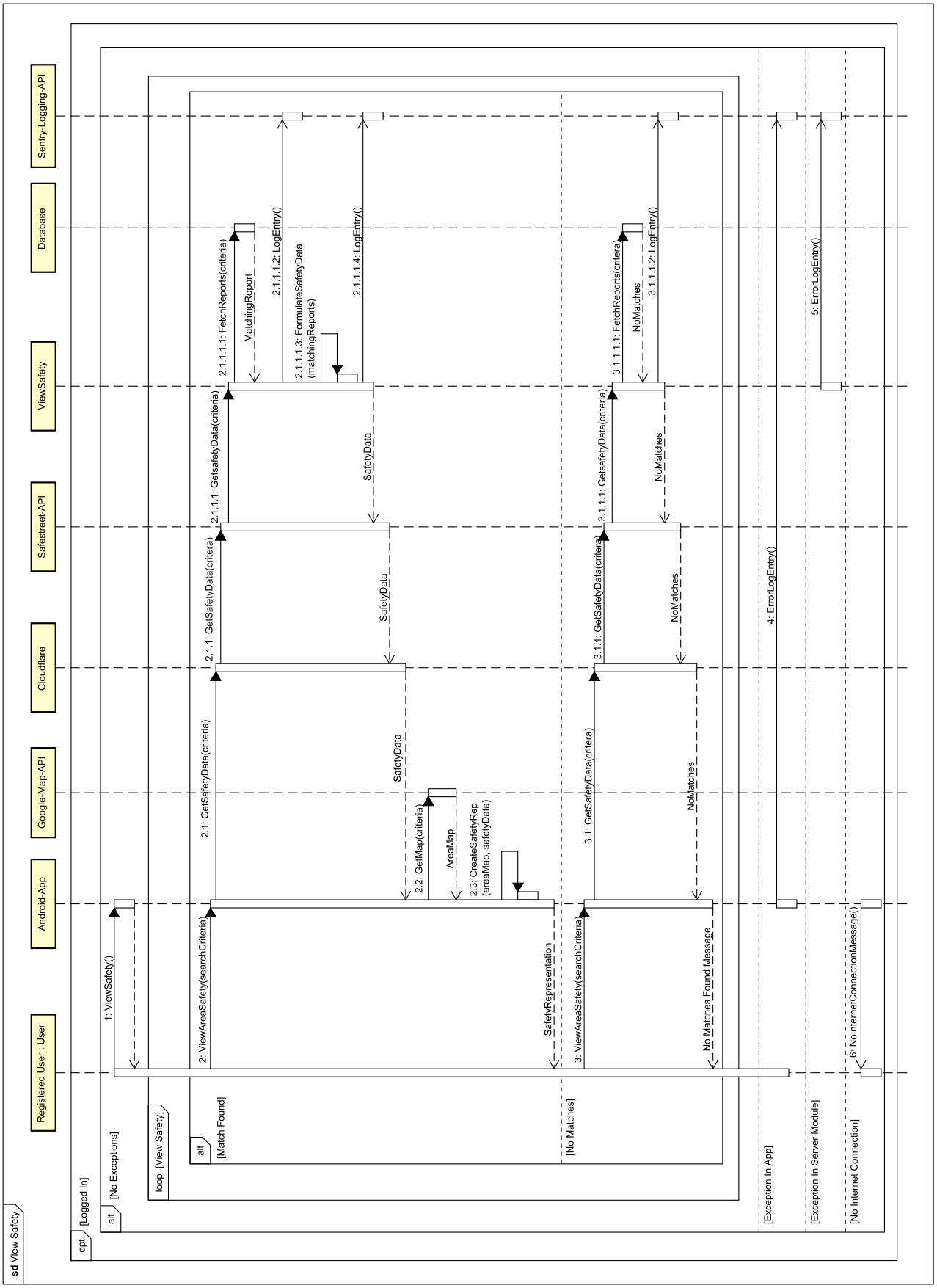


Figure 10: View Safety Runtime View

2.4.4 Municipality Interaction

This last subsection is concerned with the various interaction between the *Safestreet* system and the municipality. To be precise, there are kinds of interactions between the system and the municipality; which are as follows, firstly, the communication of user-generated violation reports to the authorities through periodic submissions to the municipality submission API. Second of the three features, the retrieval of accident reports from the municipality retrieval API to be used in the *View Safety* feature and in the intervention suggestion which is the last feature described in this subsection. In the intervention suggestion process, the user-generated reports and the retrieved accident reports are processed together to gain insight into the traffic violations which are causing a high accident rate in various areas and suggest appropriate measures to resolve these issues. The following figures provide a more detailed description of the aforementioned processes.

Figure 11: Municipality Report Submission Runtime View

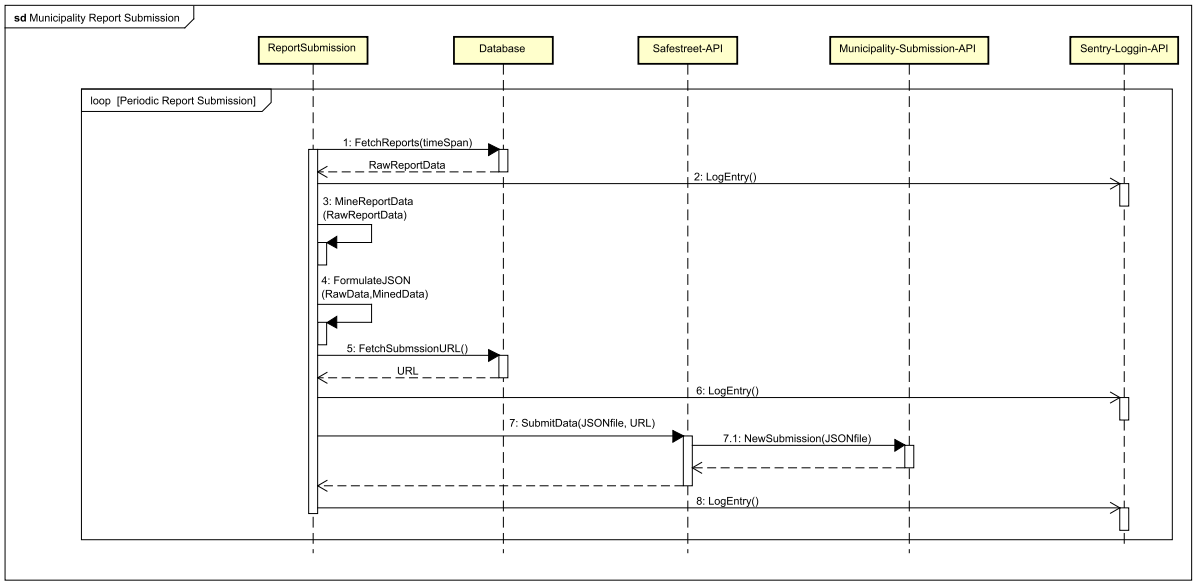


Figure 12: Accident Report Retrieval Runtime View

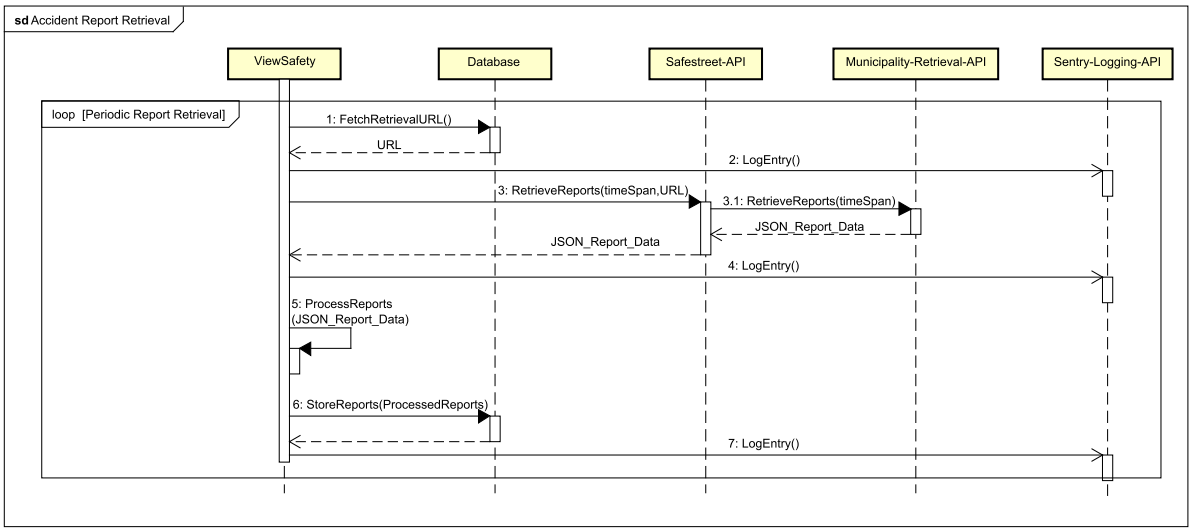
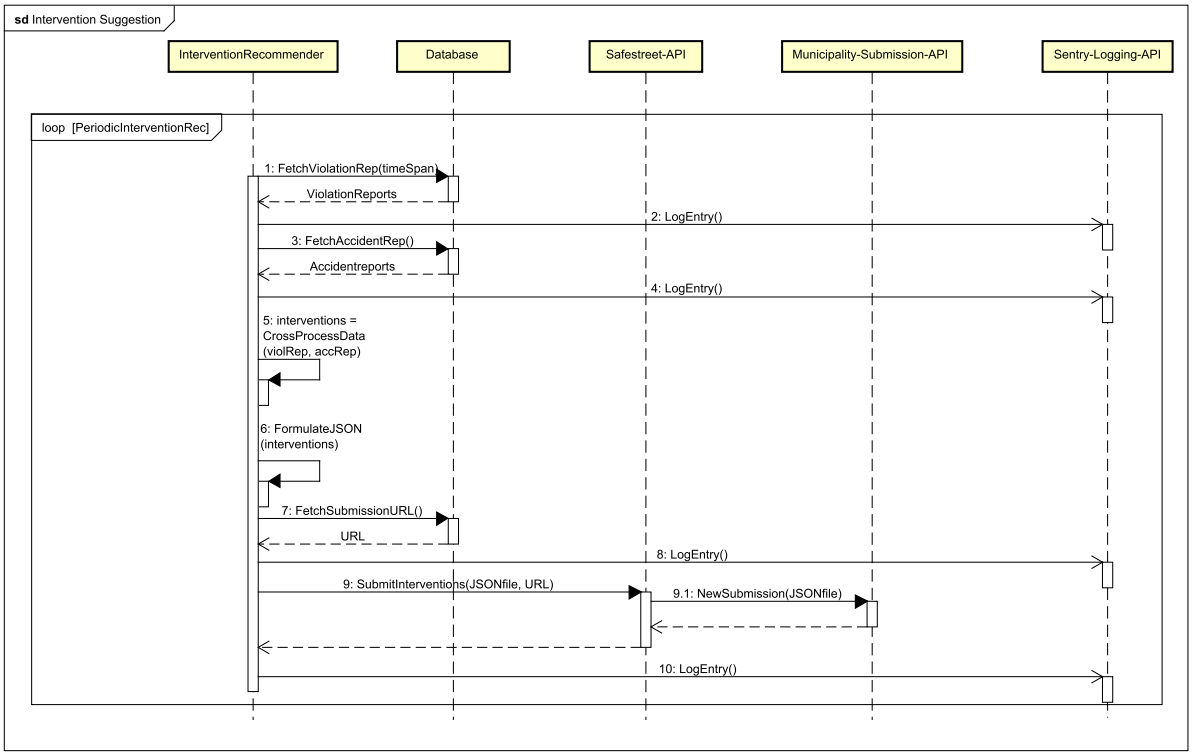


Figure 13: Intervention Suggestion Runtime View



2.5 Component Interfaces:

The endpoint diagram of the API shows the interfaces of each component in the server-side and how can use the API. The API is divided into five main sections in which four of them will handle functionalities for the end-user and one is for authorities to get the report from the system. The *users* path will handle the *sign-in*, *sign-up*, *login* and *edit* or *remove* the account by the *HTTP* command. The *Get* command is used for the login because of the security reasons which is not secure to use *POST* command for the password. The *violation* path will be in charge of the *submit*, *edit*, and *remove* each violation report. The username in this path is hashed string from Firebase after authentication to avoid username leak to avoid any 'Man-In-The-Middle' attacks which with plain username there is this possibility to submit an unoriginal report with other user's username. The *ViewSafety* path works without a credential to let third-party developers to use it in other apps. This path will get the location and type of data are needed and return the information. Moreover, the *ViewMe* part of the android app has some functionalities which will use the *users* path of the API but the *ViewHistory* which is the part of this section in android app will use this path to return the history of the reports by one user which this subpath also allows user to edit some parts of the report or remove a report, and obviously, this part also needs the hashed username. Finally, the authority path will handle the request from the different office of the municipality which by this path can get the aggregated report between two dates and also could get the recommended intervention between two dates.

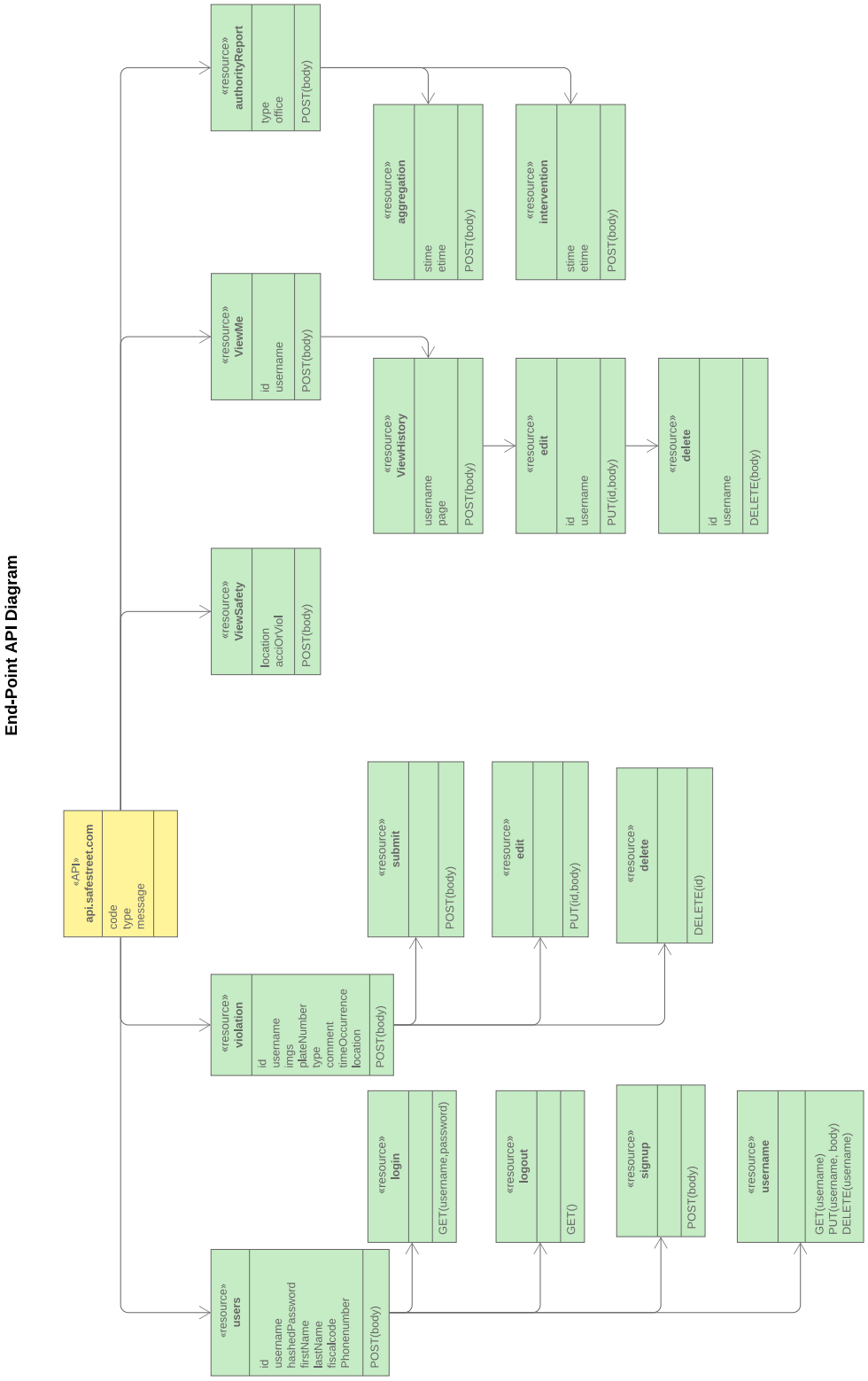


Figure 14: End-Point API Diagram

The class diagram follows the separation of the system in the component diagram, hence there are seven subsystems with one auxiliary class. In the five subsystems, there is a class in which their name is finished with the *"model"*, these classes are used with ORM (*SQLAlchemy*) which makes the database API for the app. The *SQLAlchemy* will use these classes to maps these models to the database and make tables for them in *MySQL*. The reason for choosing the ORM to communicate with the database is to increase the speed of development and make the system more portable for the feature development in which the database could be changed or a *NoSQL* can be used besides the *MySQL*, therefore the ORM will make a layer of abstraction. However, the cost of easy adaptation of the future updates is slower query run which by optimization the *SQLAlchemy* the difference with the bare-metal approach is ignorable.
Based on both class diagram the input data will be checked in three stages one in the android app which the view of the app then in the REST API which the presenter layer and the last time in the model class before committing into the database and all the function started with *"isValid"* are for checking the fields of request in body of JSON and the JSON itself to avoid the *SQL injection* and *NoSQL injection* attacks or any sabotage of the API. The database model of the intervention recommender will store all the recommends to use them in the future as input and also send them for authorities whenever they want. Moreover, the database model of image manipulation detection will store all the data which are necessary for the next run of this model because this model is not realtime to avoid too much overhead on system and database in *violation report* request so this model runs time to time and need the last time run's details to start from right place in database and if the is problem in the images of a report the originality filed of the report will be set to false by the functions in *img_ manip_ dete* class, therefore, there is no new record for the manipulated images. Also, in the *report_ submission_ model* the details of the request which comes from authorities will be stored and then the system will respond to the request. Those classes which do not have the *"model"* at the end of their name are in charge of the functionalities of each subsystem. The *ViewSafety* and *ViewMe* classes are just the consumer of the data and they will not make any new record so they do not have the database model class.
As shown in the diagram the *account_ model* is the main database of the system because of the main functionality of the system is required this model after this model the *report_ model* is another main model with the same reason which many parts of the system working by the data stored by this model. The *GeoCoordanite* class is the dependency class for the *reportViolatoin*, *ViewSafety*, and *Intervention-Recommender* which these three classes have some location related data this class will handle all the functionalities which are necessary for the system and is not part on any subsystem.
The system is implemented in the *Thin-client Thick-server* so the android part of the system is lightweight and generally is just the view classes that will get the information from the API or it will send data for it. However, the android app has the *plate recognition* subsystem which will detect and show the plate number in the pre-report to the user by the images are taken by himher. In this subsystem, the *PR_ record_ model* will store all the palate numbers by the model in the local database to allow offline submit of the violation besides all the data of the violation report and also to collect them for the future improvement of the detection model. As same as the server-side, the *Geo-coordinate* will handle all the functionalities related to location and *ReportViolation* and *ViewSafety* have a dependency to this class. Also, each class will handle its communication to the server and there is no message router to avoid a breakpoint or bottleneck in the system in which the same approach is being followed by the server.

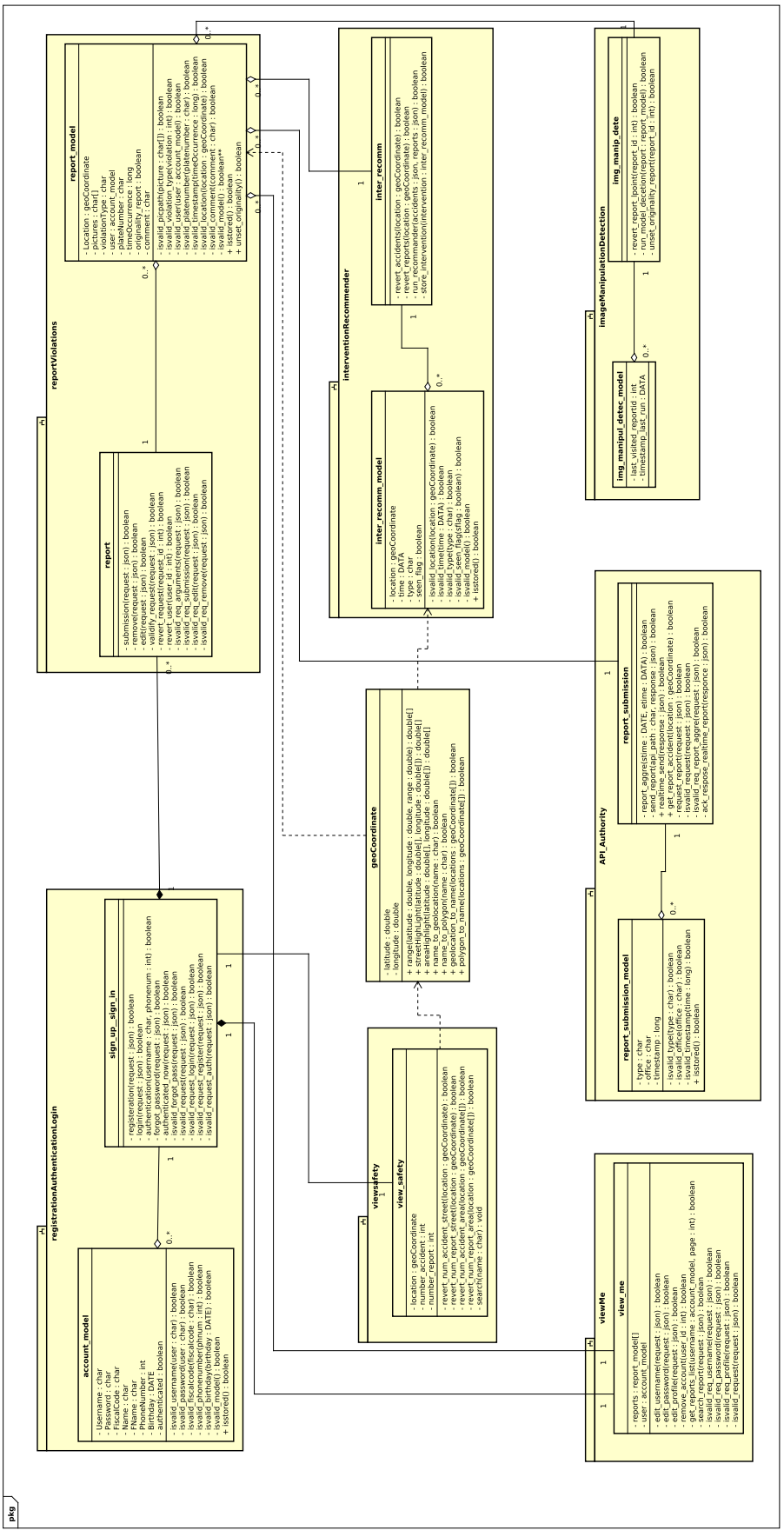


Figure 15: Class Diagram of the server-side

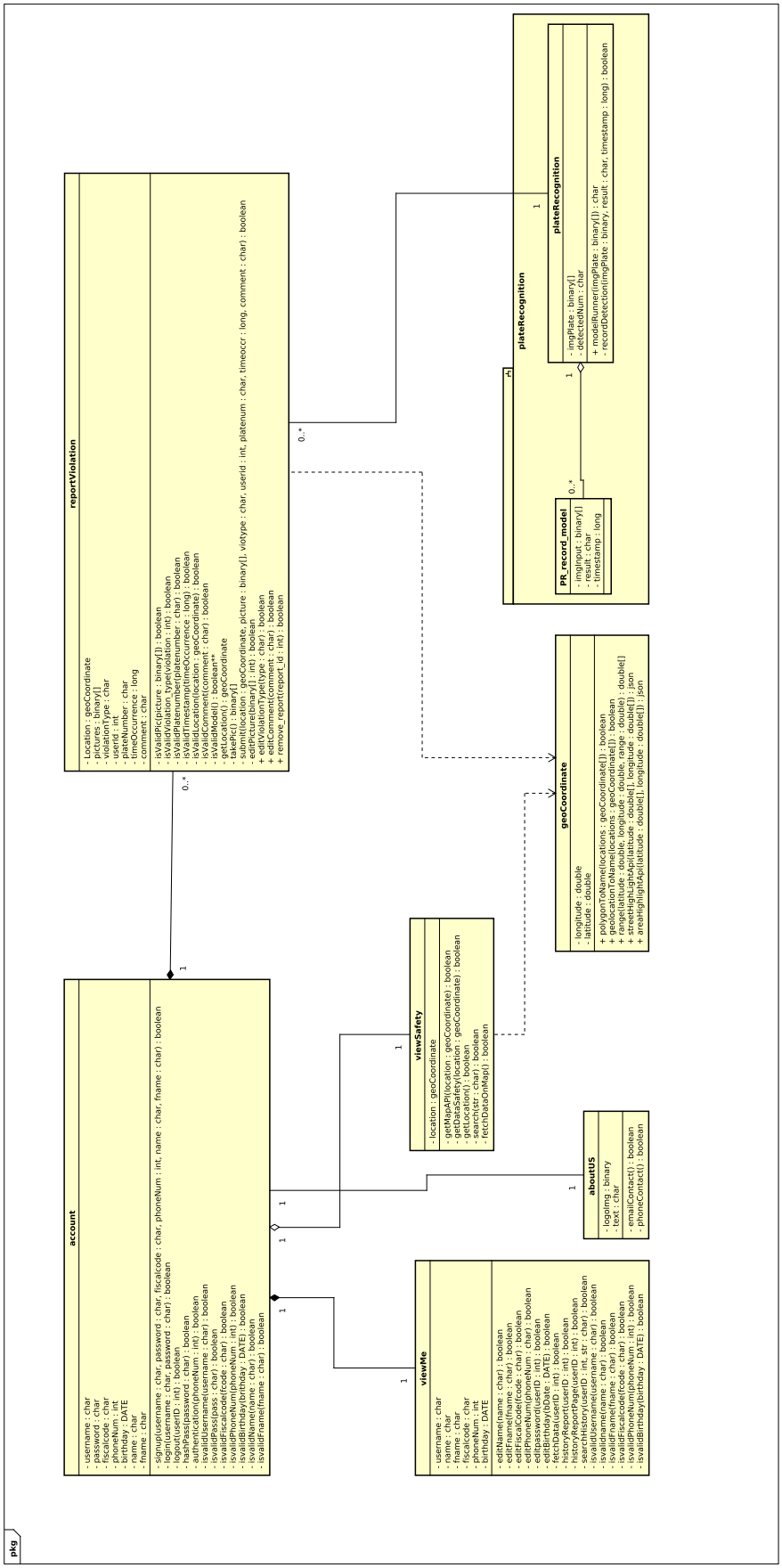


Figure 16: Class Diagram of the client-side

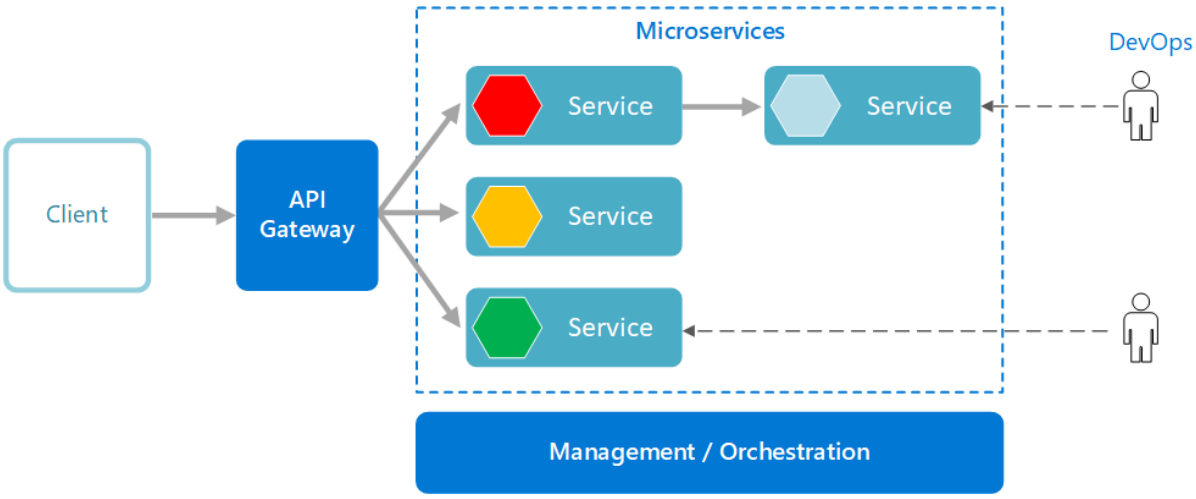
2.6 Selected Architectural Styles And Patterns

In the following section, the selected architectures and design patterns for the implementation phase of the system shall be discussed. Due to nature of the system as a whole and of the services expected to be provided to the users of the system; particularly, the fact that the system provides multiple services which are only connected through the data being produced and consumed by each of them, the system design decisions provided in this section are mostly due to the clear need for decoupling of the various services among each other and their decoupling from the client-side of the system. To be more precise, this system shall adopt the *Microservices Architecture* and the *MVP* design pattern, with the use of *RESTful APIs*. In the hereinafter subsections these design decisions shall be explored.

2.6.1 Microservices Architecture

The first of the design decisions to be implemented is the *Microservices* architectural style which subdivides the system into smaller services each of which has limited interaction with other services and provides a unique service to the clients. It is evident that this architectural style perfectly fits the previously described nature of the system. This architectural style not only provides decoupling of the services from the user but also among themselves. Moreover, the architecture at hand is rather scalable with the respect to the number of users since multiple instances of the various microservices may be instantiated to serve the users which is rather simple due to their stateless nature. Below is a figure representing the *Microservices* architectural style, in which the previously discussed aspects of the architecture can be seen.

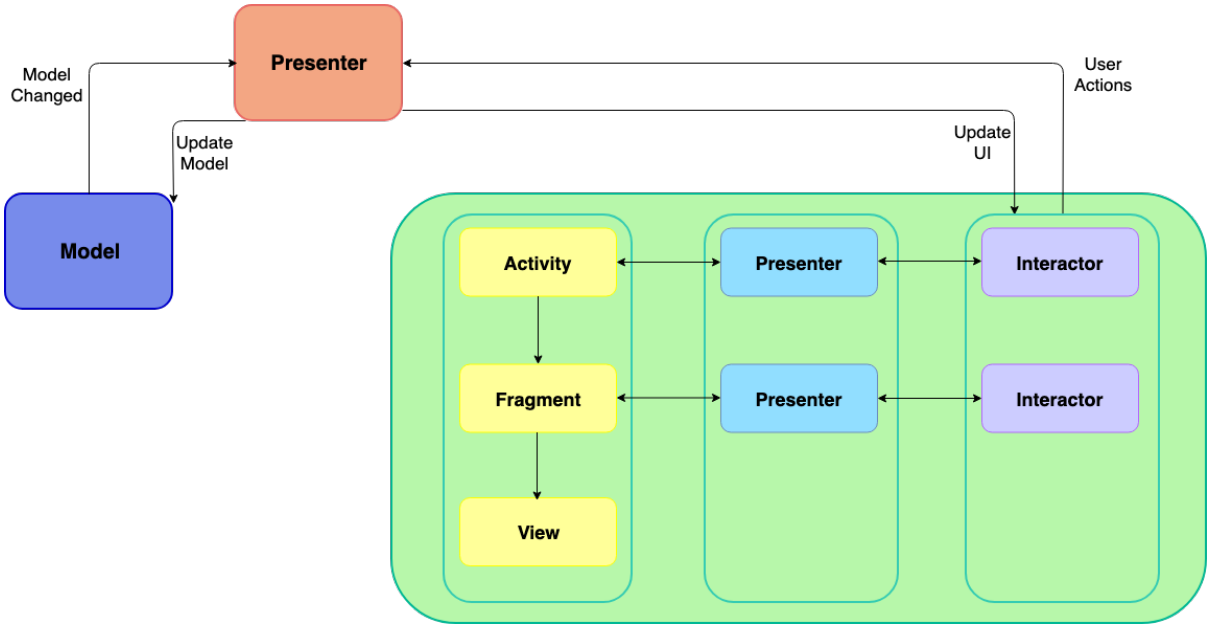
Figure 17: Microservices Architectural Style (Microsoft Azure 2019)



2.6.2 Model View Presenter

In this subsection the design pattern to be used in the system implementation shall be discussed; in particular, the ***M**odel **V**iew **P**resenter (MVP)* design pattern. This design pattern subdivides the system into three general segments; the *Model* representing the business logic and the data, the *View* which interacts directly with user, and the *Presenter* which has a two way interaction with both the view and model; presenting data from the model to the View and manipulating the model based on the user interaction with view. By definition, the *MVP* completely decouples the View and the Presenter and their communication is completely through interfaces; which brings us to the third of the decisions to be discussed in this section later on. The following figure shows the interaction between the system sub-parts as defined by the *MVP* pattern.

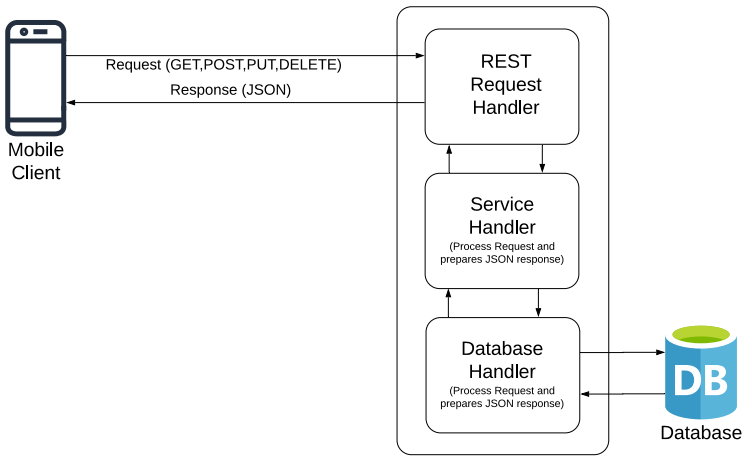
Figure 18: MVP Design Pattern



2.6.3 RESTful APIs

This third and last subsection is concerned with the use of *RESTful APIs* for the interaction between the client and server sides. These APIs can be viewed as the final piece of the puzzle that is the system architecture as a whole. Since the nature of the *RESTful APIs,* falls so perfectly into the system architecture described up to this point. Some of the main properties of these APIs which prove to be rather relevant to this architecture is the fact that they are stateless and flexible in the sense that the data is tied to neither resources nor methods enabling simple scaling with concurrent calls. Therefore, these aspects enable *RESTful APIs* to be a perfect fit for the role of interfacing the View and the Presenter described in the previous subsection.

Figure 19: Representational state transfer (REST) API Diagram



2.7 Other Design Decisions

2.7.1 Technologies and Platforms

To begin with selected programming languages for the implementation, the backend part of the system is going to be implemented using the *Flask web application framework*. There are a set of technical reasons for this decision however important ones are mentioned in the following. Flask allows us to develop our backend as fast as possible because of its simplicity and rich documentation, besides that, it's flexible for providing *REST APIs* based on our desired functionalities. Additionally, the flask is a lightweight framework compared to Django in many terms, in particular enabling developers to create a service-driven system which means to meet users' needs thoroughly and completely. Apart from this, flask comes with the flask-SQLAlchemy which is a bundle of SQLAlchemy for this framework. SQLAlchemy is an ORM for the python programming language, by using ORM we get several advantages namely, shielding application against SQL injection through filtering data, having an abstract concept of the database that makes database migration easier and enabling easy updating, maintaining and data manipulation. Last but not least, considering flask for backend gives us the possibility of having invaluable numbers of python modules. Furthermore, for the client-side implementation we opted to target Android platform, the foremost reason is based on the statistics which indicates that Android platform accounts for more than 60 percent of the market share of mobile operating system among users. In order to develop our mobile application, we will use Java programming language which is the native programming language for Android platform. This choice gives us the feasibility of developing high performance application by taking advantage of rich libraries. Moreover, we are going to use Android SDK version 21 which is for Android 5.0(Lollipop) because Google has stopped supports for lower version than that.

3 User Interface Design

In the following page, the UX diagram is presented in the form of a Wireframe. The Wireframe is a specific type of UX representation which demonstrates significant abstract outline about the application and indicates all the flows that user can experience. Taking a closer look at the diagram, each user flow corresponding to a specific macro functionality of the system is represented with a unique color. The various flows represented in the diagram are to now be discussed. The light green flow has to do with the user violation report main functionality. Functionalities such as login and sign up, and sign out are represented by the pink and turqoise flows respectively. The view history, view profile and app info are depicted by the light blue flow which flows from the homepage (report violation) through the menu to each of the app sections. Lastly, as is illustrated by the lilac and orange flows the user may return to the report page from each of the app's main sections. Finally, it should be brought to your attention that the specific mockups of the user interfaces are included in the *Requirement Analysis and Specification Document (section 3.1.1).*

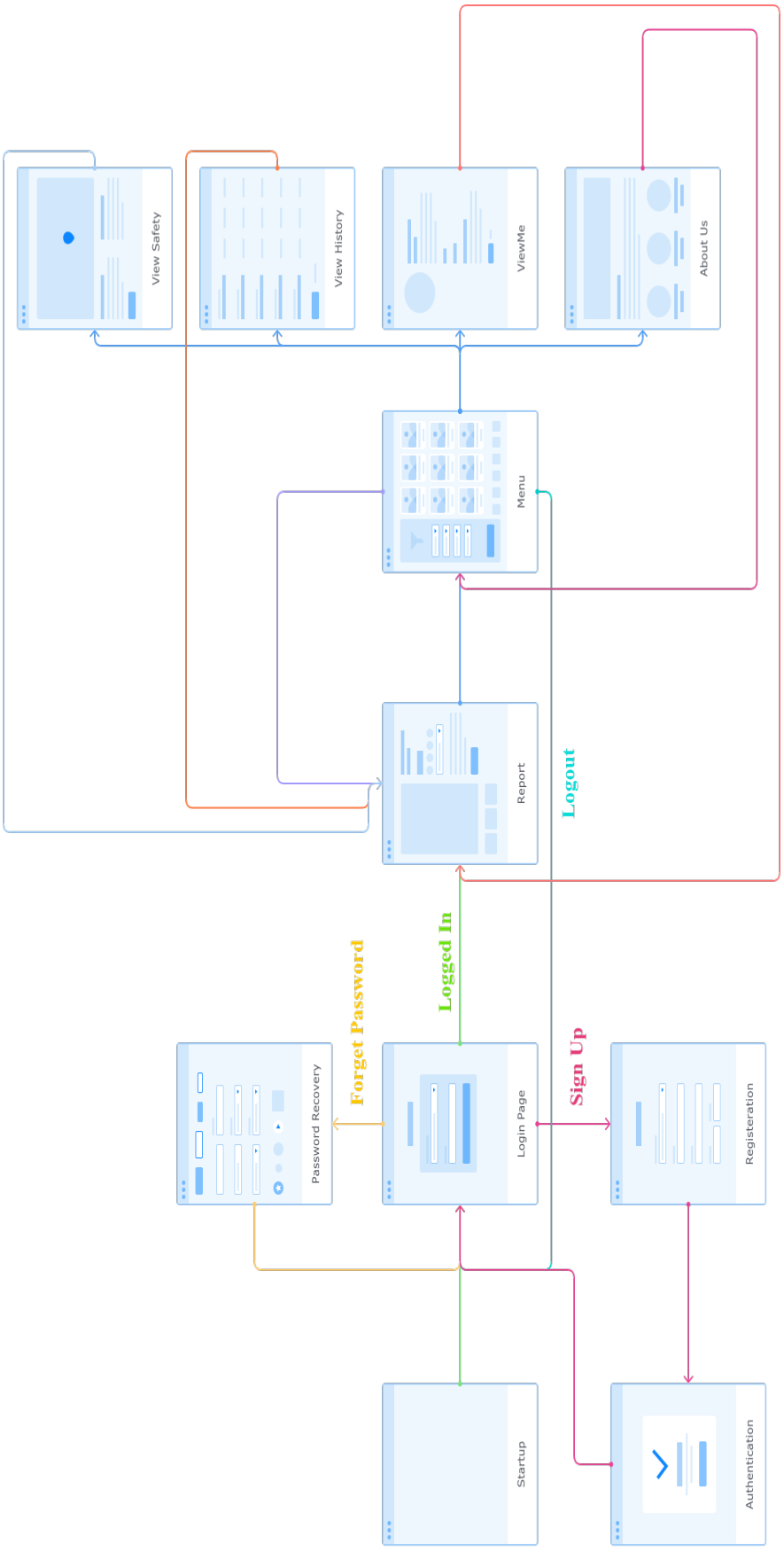


Figure 20: User Experience (UX) Diagram

4 Requirements Traceability

This section shall present a traceability matrix to ensure the satisfaction of all functional requirements and therefore the satisfaction of the underlying goals which are expected of the system. The *Requirements traceability matrix(RTM)* presented below shows a correlation between all the functional requirements previously defined in the *RASD* document and the system modules that satisfy and perform these functionalities. Apart from the components mention in the *RTM* some other components were defined in previous sections; though these modules are not direct contributors to the realization of the requirements they do however perform some roles ensuring the smooth and secure operating of the system as a whole.

REQ ID	**REQUIREMENT**	**COMPONENT**
[R-1]	Users should be allowed to register to services provided by the system	*Registration, Authentication and Login-Firebase Authentication API*
[R-2]	Users should provide unique identification to the such as fiscal code during registration	*Registration,Authentication and Login-Firebase Authentication API*
[R-3]	Registered users should be allowed to login	*Registration,Authentication and Login-Firebase Authentication API*
[R-4]	Each registered user should have a unique username used for logging in chosen at registration time	*Registration,Authentication and Login-Firebase Authentication API*
[R-5]	System should enable registered users to report traffic violations	*ReportViolation*
[R-6]	When reporting a violation, users should be able to take an image of the violating vehicle's license plate	*Android App- Firebase Storage API*
[R-7]	When reporting a violation, users should be able to fill in the details of the reported violation such as the type of the violation	*Android App*
[R-8]	The system should be able to detect the current user location when reporting a violation	*Android App-Google Map API*
[R-9]	The system should extract the plate numbers from the image taken by the user	*Plate Number Recognition*
[R-10]	The received reports must be stored by the system to be used by other services	*Report Violation-Database*
[R-11]	Registered users should be allowed to view a representation of the safety of selected areas possibly with the help of a map API	*Android App-Google Map API-View Safety*
[R-12]	The system should implement a means to measure the safety of various areas based on reported violations in said areas	*View Safety*
[R-13]	Incoming reports should be integrated and used to update the safety of areas	*View Safety-Report Violation*

[R-14]	If accident reports are provided by authorities the system should take that data into account when calculating the safety of a certain area	*View Safety*
[R-15]	The system should present on-demand to the users a record of all the reports previously submitted by them	*Report Historyy*
[R-16]	The system should keep records regarding the submission dates of violations to the municipality	*Report Submission-Database*
[R-17]	The system should aggregate the data regarding reported traffic violations since the last submission to the municipality	*Report Submission-Database*
[R-18]	The aggregated traffic violation data should be converted to a form acceptable by the municipality interface	*Report Submission*
[R-19]	The system should periodically submit the new traffic violation data to the municipality interface	*Report Submission-Database*
[R-20]	The system should be able to store submitted reports coming from users with proper metadata	*Report Violation-Database*
[R-21]	The system should be able to export reported violations in form of specific file	*Report Submission*
[R-22]	The system should be able to filter data based on desired requests from authorities	*Report Submission*
[R-23]	The logging functionality has to be implemented in the system	*Sentry Logging API*
[R-24]	The system should extract insights from the users' traffic violation reports such as the most frequent types of violations in certain areas	*Report Submission-View Safety*
[R-25]	The system should be able to decide on appropriate interventions to minimize frequent traffic violations in the various areas	*Intervention Recommender*
[R-26]	The system should formalize the interventions to be suggested in a form acceptable by the municipality interface	*Intervention Recommender*
[R-27]	The system should submit the interventions regarding the areas with a high frequency of violations to the municipality interface	*Report Submission*
[R-28]	The system has to be able to mine reports to find insights based on types of violation and different areas	*View Safety*
[R-29]	The system should analyze the data from the reports to produce statistics such as the most frequent plate numbers that commit violations	*Report Submission*
[R-30]	The system should formalize the refined data and submit them to the municipality interface periodically	*Report Submission*

Table 2: Traceability matrix

5 Implementation Integration And TestPlan

In this section, the plan for the development phase of the system shall be discussed. The subsections shall specify the implementation order of the different system modules, how they shall be integrated into the system as a whole and the testing plan to verify and validate the system as a whole.

5.1 Implementation

In this section the sequence by which the modules of the system are to be implemented. Note that the modules here are referred to by the functions which they perform; moreover, the vertical alignment of more than one activity signifies that these two or more activities may be implemented and developed in parallel with no codependency. The following two figures represent the implementation order to be followed for the App Server and the Android App.

Figure 21: Server Implementation

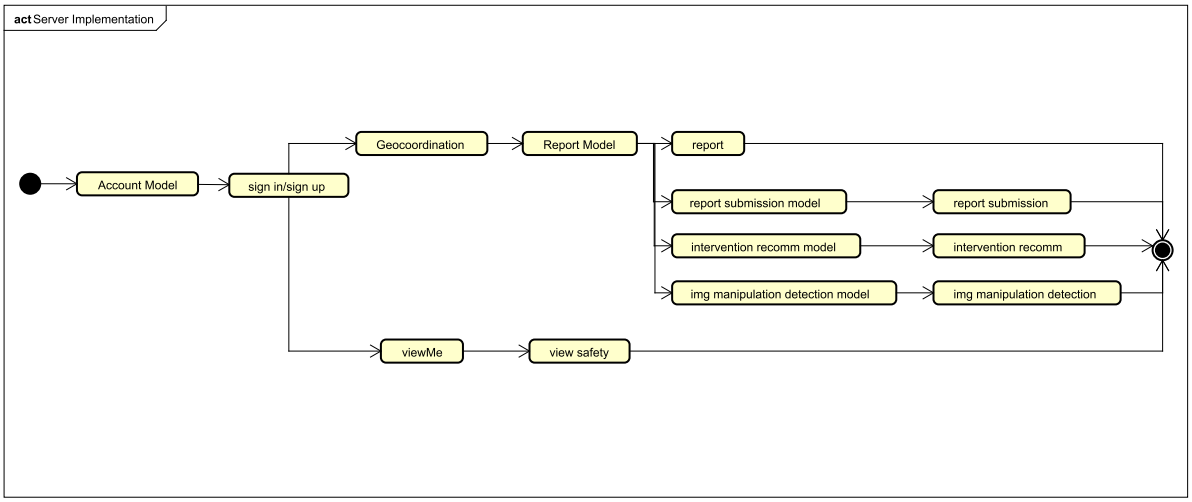
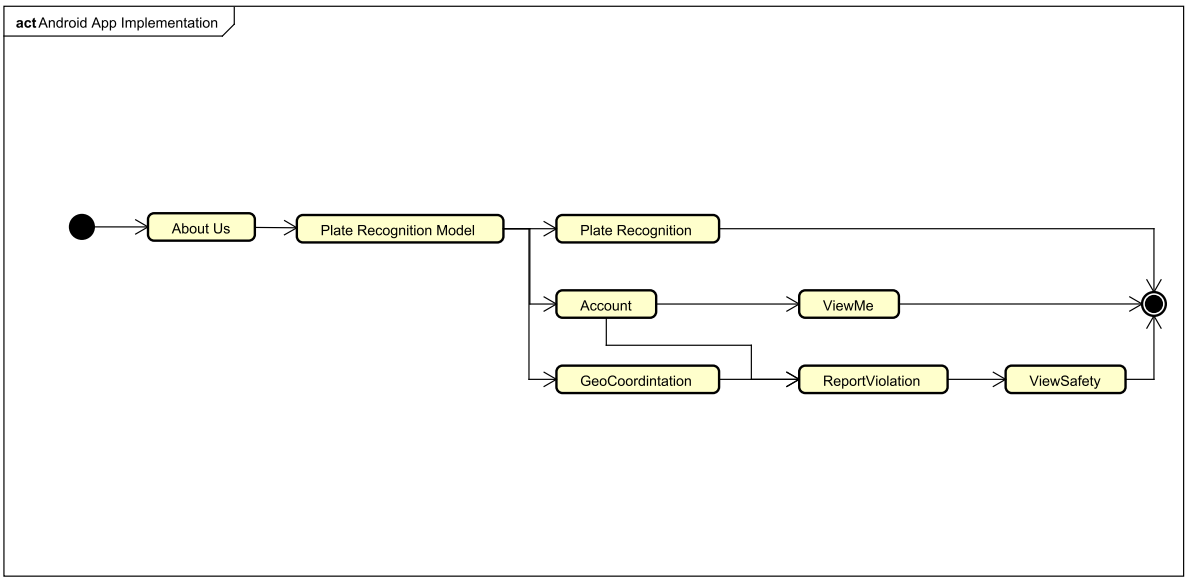


Figure 22: Android App Implementation



5.2 Integration

As for the integration of the system modules, we opted for the use of continuous integration and continuous delivery(CI/CD). To be more clear, the system modules shall be iteratively built-in sequence and for each one that is completed, that module shall be automatically built and integrated into the development branch of the repository. This decision on the integration plan eases the deployment at any point since after each build the codebase is in a deployable state, moreover, it is very convenient given the nature of the system and the decided upon architecture discussed in the previous section of the document; since, the system shall be implemented in the Microservices architecture, each of the unique services shall be integrated into the system as soon as they are developed which is possible due to the minimum interaction between the modules and whatever interaction there is can be handled by following the order of implementation discussed in the previous subsection.

5.3 Test Plan

The test plan to be used follows the implementation; i.e., the bottom-up approach. Meaning, that each component of the system shall be unit tested at each build ensuring the internal functionality of that module then it shall be integrated into the system as discussed in the preceding subsection. After the integration of more multiple modules that function in unison an integration test shall be performed; this way the system verification is incrementally ensuring the early detection of any issues. Finally, after the completion of the system as a whole a system test is performed then the system is handed over for validation testing of the system.

6 Effort Spent

Discription of the Task	**Hours**
karim Zakaria Saloma	
Introduction	3
Architectural Design	23
User Interface Design	0.5
Requirements Traceability	2
Implementation, Integration and Test Plan	6
Amirsalar Molaei	
Introduction	0
Architectural Design	18
User Interface Design	5
Requirements Traceability	0
Implementation, Integration and Test Plan	0
Erfan Rahnemoon	
Introduction	0
Architectural Design	21
User Interface Design	1
Requirements Traceability	1
Implementation, Integration and Test Plan	0.5

Table 3: Effort Spent by Each Team Member.