

Software Engineering 2



POLITECNICO
MILANO 1863

SafeStreets

Design Document

Authors:	Federico Cazzola	mat 945835
	Francesco Dotti	mat 945232

Professor: Di Nitto Elisabetta

Academic Year 2019-2020

Version 1.0

Contents

1	Introduction	4
1.1	Purpose	4
1.2	Scope	4
1.3	Definitions, Acronyms, Abbreviations	4
1.3.1	Definitions	4
1.3.2	Acronyms	5
1.3.3	Abbreviations	5
1.4	Revision history	5
1.5	Reference Documents	5
1.6	Document Structure	5
2	Architectural design	7
2.1	Overview: High-level components and their interaction	7
2.2	Component view	8
2.3	Deployment view	10
2.4	Runtime view	11
2.4.1	Report Violation	12
2.4.2	Traffic warden receives notification	13
2.4.3	Traffic warden takes on report	14
2.4.4	Login	15
2.4.5	Cross data to suggest possible interventions	16
2.4.6	Visualize critical areas on the map	17
2.4.7	Visualize the ranking of vehicles that committed more violations	18
2.4.8	Visualize suggestions for possible interventions	19
2.5	Database View	20
2.6	Component interfaces	21
2.7	Selected architectural styles and patterns	26
2.8	Other design decisions	26
3	User interface design	28
3.1	User interface mockups	28
3.1.1	User	29
3.1.2	Traffic Warden	31
3.1.3	Municipality's clerk	33
3.2	UX Diagrams	34
4	Requirements traceability	36
5	Implementation, integration and test plan	38
5.1	Integration Process	38
5.1.1	Integration Order	39
5.2	Testing	40
5.2.1	Unit testing	40

5.2.2	Integration testing	40
5.2.3	Other types of testing	40
6	Effort spent	41
7	References	42
7.1	Software used	42

1 Introduction

1.1 Purpose

The purpose of this document is to describe how the system will be built, giving specific and technical details about architectural and design decisions. Also implementation, integration and testing plans will be discussed. In particular the document presents:

- The SafeStreet System architecture (its parts and how they interact)
- The runtime behavior
- The design patterns
- User Interfaces
- Implementation, integration and testing plans

1.2 Scope

The scope of the Safestreets project, as already specified in the RASD, is to give the user the possibility to report traffic violation, in particular, parking violation (eg. parking in a spot reserved for disabled people, parking in an helicopter pitch, parking on the sidewalk).

The system will allow the users to select the type of violation leading to different amount of ticket value.

The system will help the authorities to identify more infractions and therefore to issue more tickets which should increase the attention and respect of the citizens regarding the traffic rules.

Furthermore, thanks to the increased number of tickets, the municipality will have more money to invest in the community. This extra money could be used to do some interventions following the suggestions provided by SafeStreets.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

- **Mine:** to process data for obtaining new data
- **Violation:** an infringement of the rules
- **Unsafe:** area area where often happen violation and accident
- **Report Open:** a report with no traffic warden assigned

1.3.2 Acronyms

- **RASD**: Requirements Analysis and Specifications Document
- **DD**: Design Document
- **FOSS**: Free and Open Source Software
- **S2B**: Software to Be
- **API**: Application Programming Interface
- **ALPR**: Automated License Plate Recognition
- **JS**: JavaScript
- **JWT**: JSON Web Tokens
- **FCM**: Firebase Cloud Messaging
- **REST**: REpresentational State Transfer
- **DBMS**: Database Management System
- **SDK**: Software Development Kit
- **UX**: User eXperience

1.3.3 Abbreviations

1.4 Revision history

- Version 1.0 — First Release

1.5 Reference Documents

- Specification document: “SafeStreets Mandatory Project Assignment”
- UML diagrams
- IEEE Standard for Information Technology—Systems Design—Software Design Descriptions (IEEE Std 1016TM-2009)

1.6 Document Structure

- Chapter 1 is the introduction.
- Chapter 2 provides details about the system architecture, its components and how they interact.
- Chapter 3 describes the UX and shows the user interface providing mock-ups.

- Chapter 4 describes how the requirements (defined in the RASD) are mapped to the design elements defined in this document.
- Chapter 5 presents the implementation, integration and test plan. It shows how the different components of the application are integrated with each other and how they react. Also the testing strategies are described and the risks in the application are analyzed.
- Chapter 6 shows the effort spent by each group member

2 Architectural design

2.1 Overview: High-level components and their interaction

The application will be developed with three logic software layers:

- Presentation (P) manages the user interaction with the system
- Application (A) handles the business logic of the application and its functionalities
- Data access (D) manages the information with access to the database

These layers are thought to be divided on three different hardware tiers (a machine or a group of machines), so that any logic layer has, in principle, its own dedicated hardware (three-tier architecture).

This architecture should give the system more scalability and flexibility as the server side is split into two nodes.

The second tier must contain only the business logic to physically separate clients and data to guarantee more safety in accessing to data since the system deals with sensitive data.

The following image shows an high-level view of the architecture of the system using the ArchiMate modeling language.

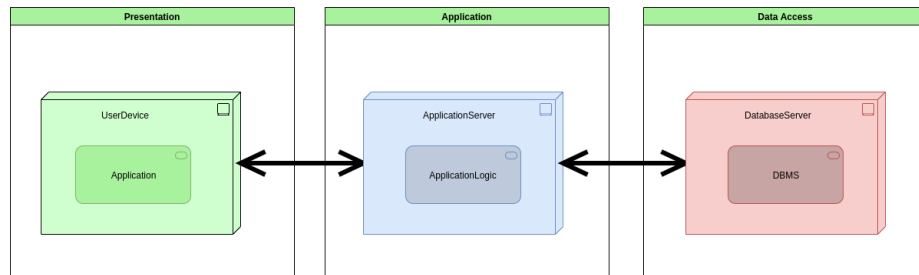


Figure 1: SafeStreets 3-tier architecture

2.2 Component view

In the following diagram only the application server is analyzed in detail

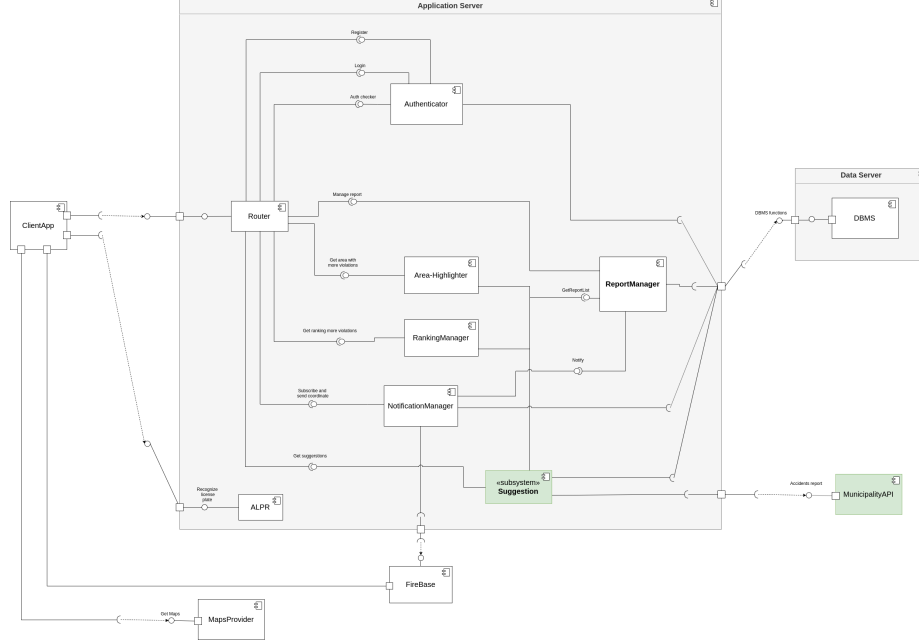


Figure 2: Component diagram

Here are described more in detail all the components and interfaces that the application server uses to offer its functionalities.

- **Router:** Manages all the request coming from the clients, forwarding them in the appropriate component for the specific request. It also interacts with the *Authenticator component* to check if a request comes from an authenticated user and then denies or forwards the request according to the user permissions.
- **Authenticator:** Manages the sign up and login requests and checks if a request comes from an authenticated user.
- **ReportManager:** Manages all the requests inerent to reports (e.g. get list of report, send new report) and is responsible to check if the new reports are consistent. Also collects reports form the database to pass them to other components.
- **Area-Highlighter:** Elaborates the reports to find in which area more violations are committed.
- **RankingManager:** Elaborates the valid reports to create a ranking of the vehicles that committed more violations.

- **NotificationManager:** Manages all the subscriptions for notification made by the client's app when a traffic warden log in and send the notification to them when a new report is added.
- **ALPR:** Elaborates the picture of the violation to extract the license plate. This component is designed to be separate by the others so that if after deployment it will became overload it could be deployed on a different server.
- **SuggestionSubsystem:** More in detail:

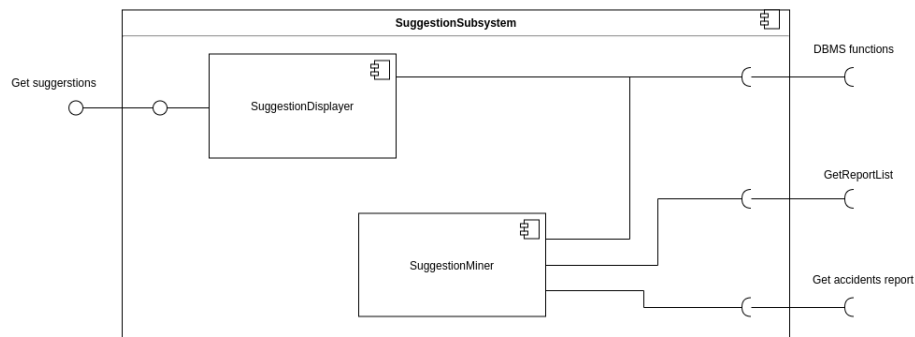


Figure 3: Component diagram suggestion subsystem

- **SuggestionDisplayer:** Sends the list of the suggestions
- **SuggestionMiner:** Gets data from Municipality's servers and from Safestreets' database and elaborates the suggestions for possible interventions

2.3 Deployment view

The following diagram models the physical deployment of artifacts (software components) on nodes (hardware components). External services like the Map Provider are not included.

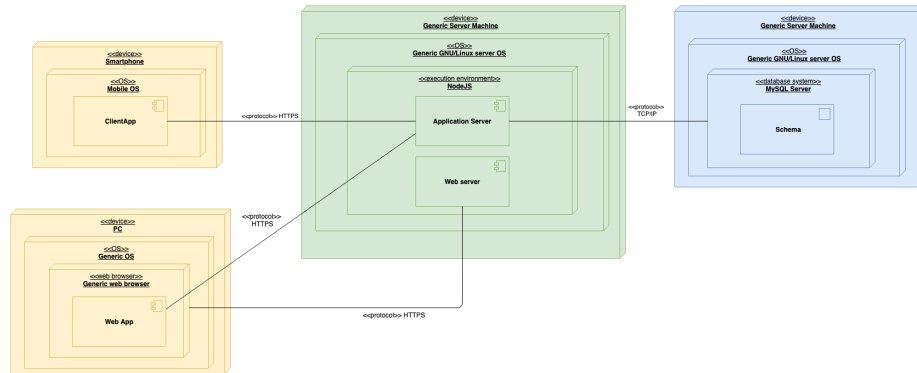


Figure 4: Deployment diagram

- **Tier 1**

Users can access the SafeStreets service using the mobile application available for Android and iOS devices or using the web app (compatible with any modern browser). The web app will communicate directly with the application server using RESTful API, exactly like the apps for iOS and Android.

- **Tier 2**

The same server machine (for budget reasons) will contain all the components handling the business and the web logic. The application server handles all the requests coming from the users applications. The web server is just there to make possible to get the web app from a website.

- **Tier 3**

The database machine will run MySQL server (a relational database management system). On this machine will be stored all the persistent data.

Note1: The webserver will not be mentioned in the next sections. It would be like including the AppStore or GooglePlay. However it has to be deployed.

Note2: The ClientAPP will work in same way on every platform.

2.4 Runtime view

In this section are shown the sequence diagrams of some features. They are useful to clarify the runtime behaviour of the components involved for each functionality.

In order to keep the diagrams more readable the parameters of the method invoked are not shown. The numbers present in some responses refer to HTTP response status codes:

- **201 Created:** The request has been fulfilled, resulting in the creation of a new resource.
- **400 Bad Request:** The server cannot or will not process the request due to an apparent client error (e.g., malformed request syntax, size too large, invalid request message framing, or deceptive request routing).
- **401 Unauthorized:** The user does not have valid authentication credentials for the target resource.
- **409 Conflict:** Indicates that the request could not be processed because of conflict in the current state of the resource.

2.4.1 Report Violation

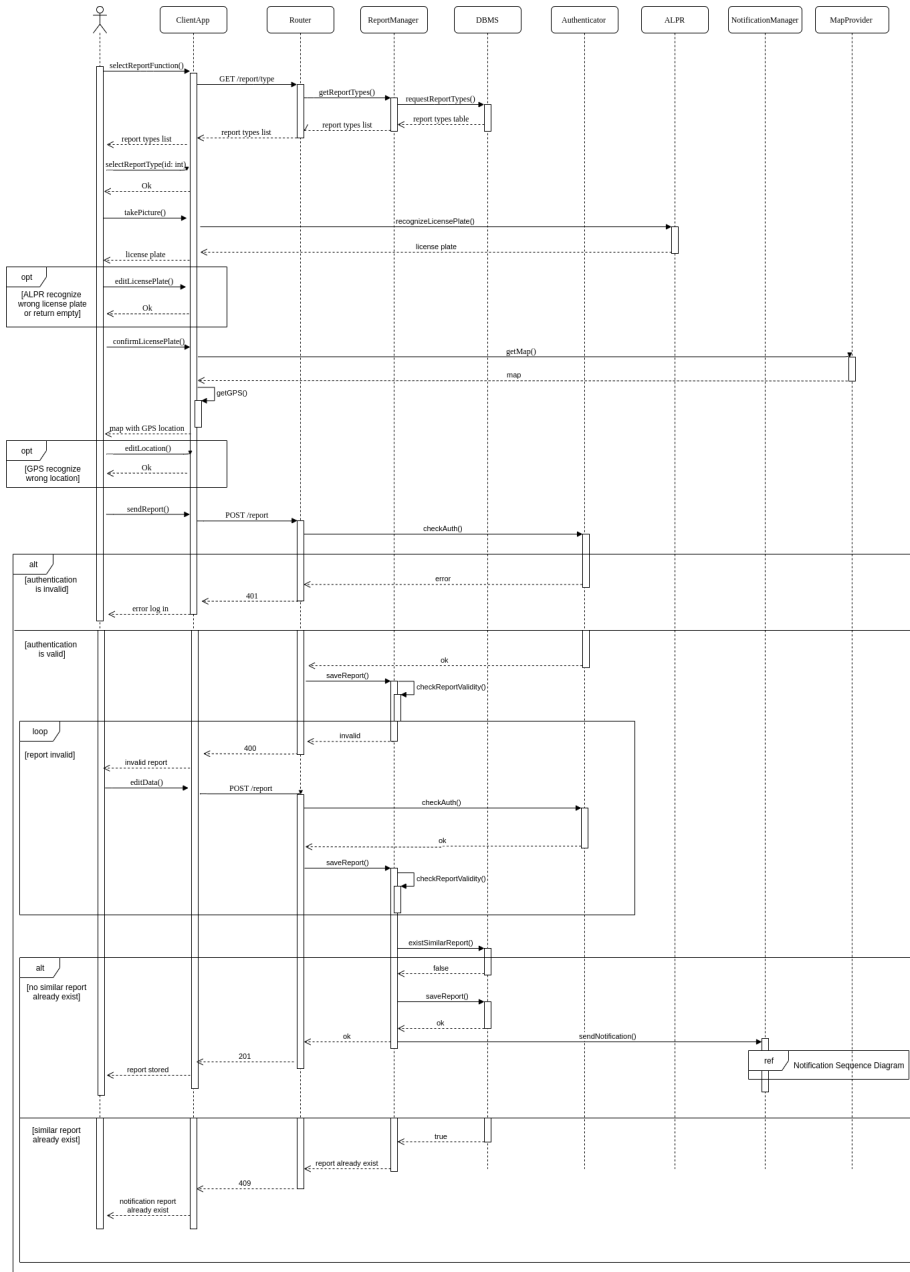


Figure 5: Report violation sequence diagram

2.4.2 Traffic warden receives notification

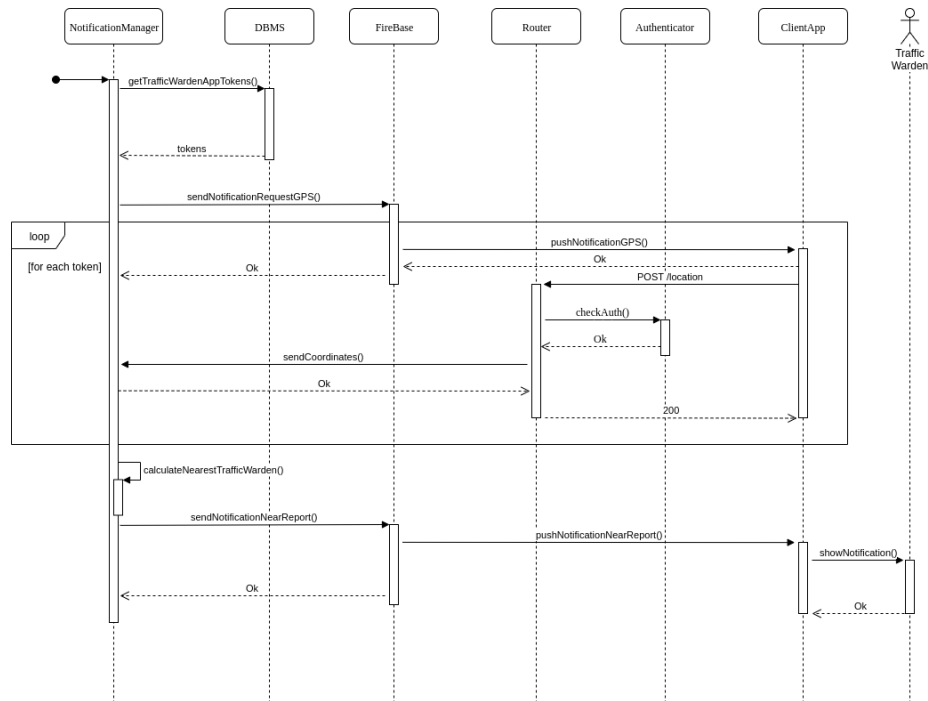


Figure 6: Traffic warden gets notification sequence diagram

Firebase Cloud Messaging (FCM) is a cross-platform messaging solution that lets you reliably deliver messages at no cost. Using FCM, we can notify a traffic warden client app that a new report to take on is available.

2.4.3 Traffic warden takes on report

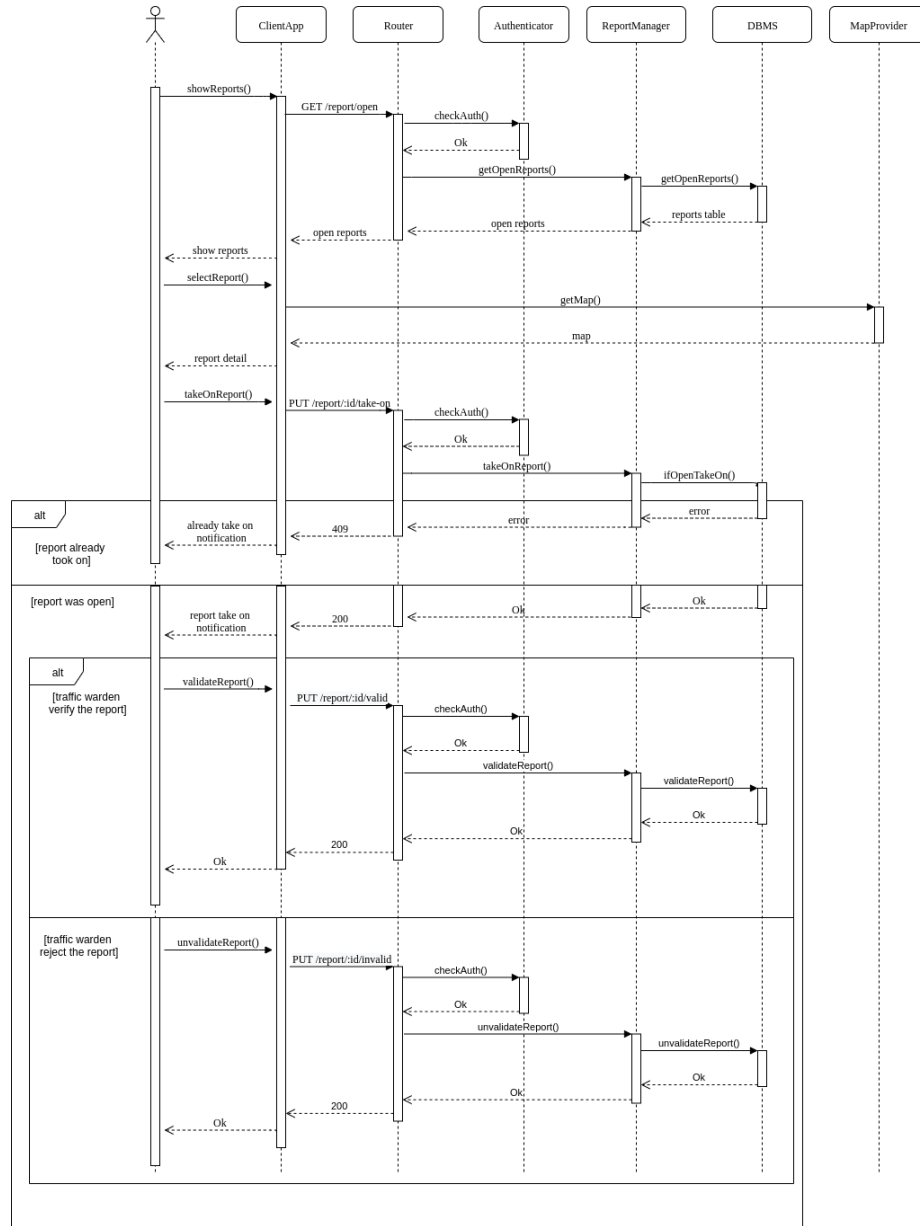


Figure 7: Traffic warden takes on report sequence diagram

2.4.4 Login

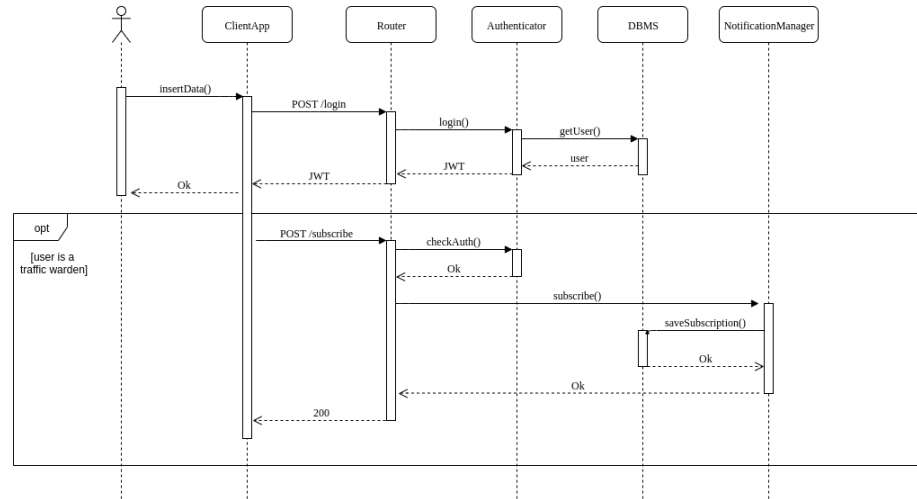


Figure 8: Login sequence diagram

Standard RESTful login process exploiting JWTs.
This diagram also shows the differences in the login process between a standard user and a traffic warden. The latter must also receive notifications so the Notification-Manager is required.

2.4.5 Cross data to suggest possible interventions

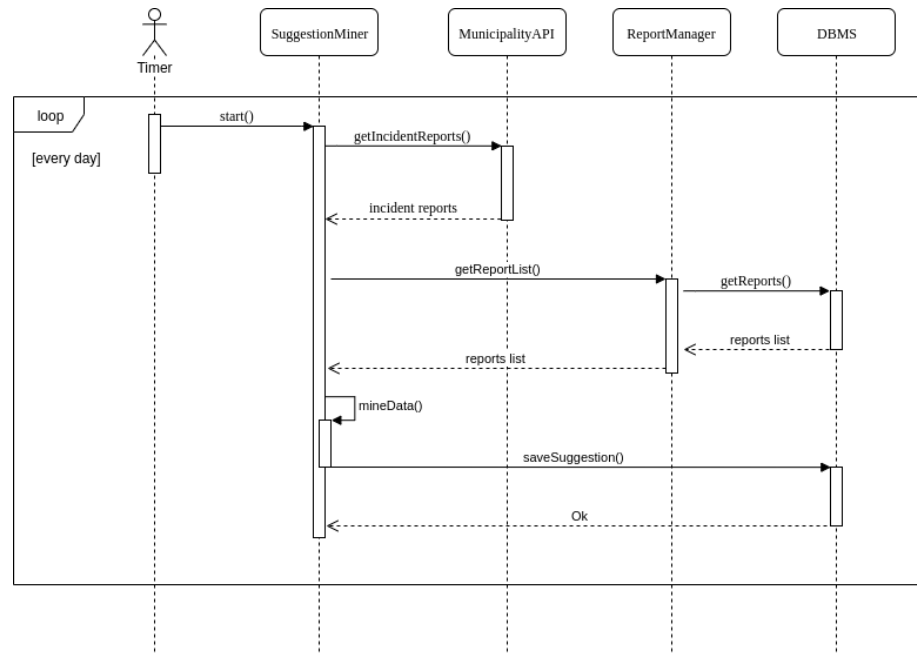


Figure 9: Suggestions sequence diagram

The diagram shows how there are no human actors involved. This is an automated process triggered by a timer (runs once every day). The data needed by the SuggestionMiner are fetched from the Municipality Server and from the SafeStreets DBMS. The updated suggestions list is then stored on the database.

2.4.6 Visualize critical areas on the map

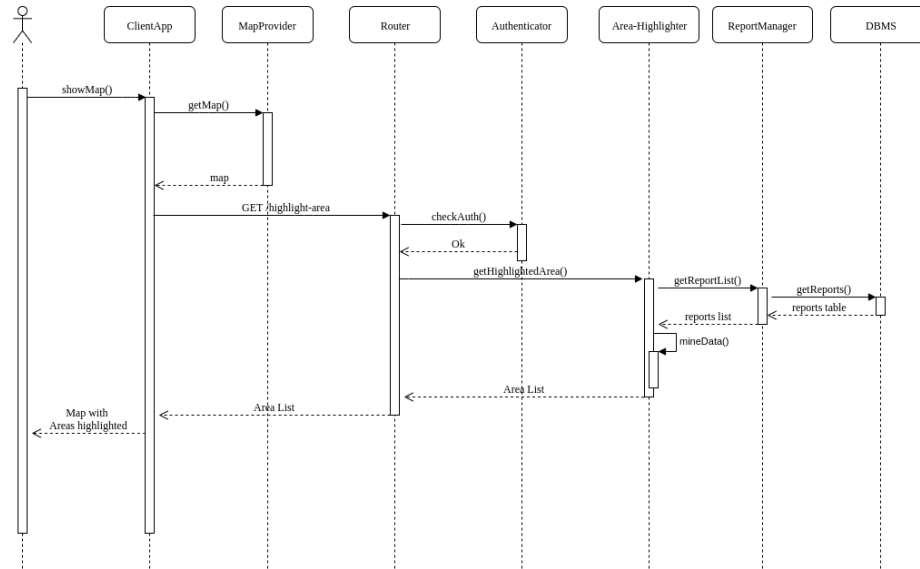


Figure 10: Areas sequence diagram

2.4.7 Visualize the ranking of vehicles that committed more violations

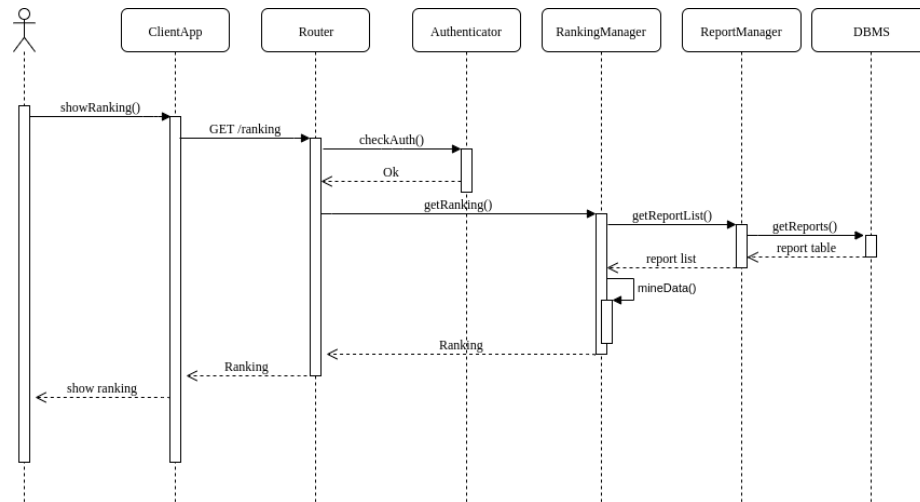


Figure 11: Vehicles ranking sequence diagram

2.4.8 Visualize suggestions for possible interventions

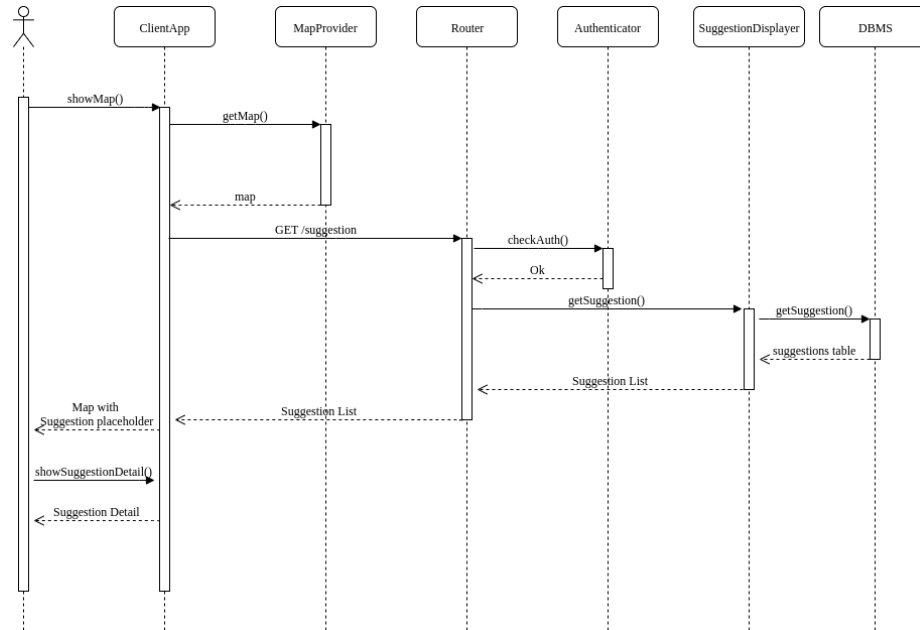


Figure 12: Suggestions sequence diagram

2.5 Database View

It is also important to describe how data are stored. Here we report a simple Entity-Relationship diagram. This model defines the structure which will be implemented in the MySQL database.

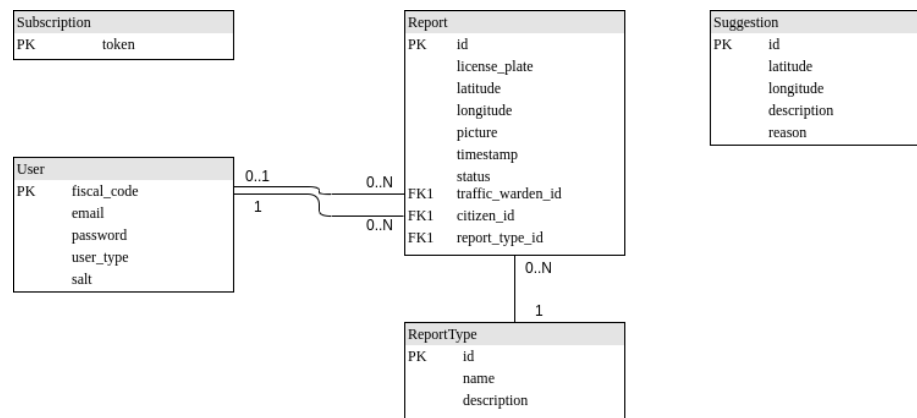


Figure 13: Database ER diagram

2.6 Component interfaces

Here are provided details on methods on interfaces previously indentified. Labels in messages used in the runtime view sequence diagrams must be consistent with the interfaces of components.

This viewpoint consists of a set of interface specifications for each entity.

- **Router:**
 - **GET /report/type:** Get all the reports' types
 - * **Parameters:**
 - none
 - * **Return:**
 - reportType: \langle ReportType \rangle
 - **POST /report:** Post a new report
 - * **Parameters:**
 - JWT
 - report: Report
 - * **Return:**
 - 201: created
 - 401: report invalid
 - 409: report already exist
 - **POST /login:** Log in
 - * **Parameters:**
 - email: String
 - password: String
 - * **Return:**
 - JWT
 - **POST /subscribe:** Subscribe to recive notification for new report (only user defined as traffic warden can access)
 - * **Parameters:**
 - JWT
 - token: String *token used for firebase*
 - * **Return:**
 - 200
 - **GET /highlight-area:** Get the area with more violation
 - * **Parameters:**
 - JWT
 - filter: List \langle ReportType \rangle
 - from: Date *analyze data only starting from this date*
 - to: Date *analyze data only until this date*
 - * **Return:**
 - areas: List \langle Area \rangle

- **GET /ranking:** Get the ranking of the license plate that committed more violation
 - * **Parameters:**
 - JWT
 - filter: List<ReportType>
 - from: Date *analyze data only starting from this date*
 - to: Date *analyze data only until this date*
 - * **Return:**
 - ranking: List<String> *ordered list of license plate*
- **GET /suggestion:** Get the suggestions (only user defined as municipality can access)
 - * **Parameters:**
 - JWT
 - * **Return:**
 - suggestions: List<Suggestion>
- **GET /report/open:** Get all the open reports (only user defined as traffic warden can access)
 - * **Parameters:**
 - JWT
 - * **Return:**
 - reports: List<Report>
- **PUT /report/:id/take-on:** Assign traffic warden calling this function to the report (only user defined as traffic warden can access)
 - * **Parameters:**
 - JWT
 - id: int *this is the id of the report*
 - * **Return:**
 - 200 Ok
 - 409 Already took on
- **PUT /report/:id/valid:** Set the report as valid (only user defined as traffic warden can access)
 - * **Parameters:**
 - JWT
 - id: int *this is the id of the report*
 - * **Return:**
 - 200 Ok
 - 409 Already validate/invalidate
- **PUT /report/:id/invalid:** Set the report as invalid (only user defined as traffic warden can access)
 - * **Parameters:**
 - JWT
 - id: in *this is the id of the report*

- * **Return:**
 - 200 Ok
 - 409 Already validate/invalidate
 - **POST /location:** Send your location for notification process (only user defined as traffic warden can access)
 - * **Parameters:**
 - JWT
 - longitude: double
 - latitude: double
 - * **Return:**
 - 200
- **Authenticator:**
 - **checkAuth():** Check if the JWT is correct
 - * **Parameters:**
 - JWT
 - * **Return:**
 - true *if the JWT is valid*
 - false *if the JWT is invalid*
 - **login():** Return the JWT for the user
 - * **Parameters:**
 - email: String
 - password: String
 - * **Return:**
 - true *if the JWT is valid*
 - false *if the JWT is invalid*
- **ReportManager:**
 - **sendNotification():** Send a notification to the nearest traffic warden
 - * **Parameters:**
 - longitude: double
 - latitude: double
 - **getReportTypes():** Get all the reports' types
 - * **Parameters:**
 - none
 - * **Return:**
 - reportType: <ReportType>
 - **saveReport():** Save the report
 - * **Parameters:**
 - report: Report
 - * **Return:**
 - Ok *if report is saved*

- invalid if report is invalid
 - report already exist if similar report already exists
- **getReportList()**: Get the list of report
 - * **Parameters:**
 - filter: List<ReportType>
 - from: Date *analyze data only starting from this date*
 - to: Date *analyze data only until this date*
 - * **Return:**
 - areas: List<Report>
- **getOpenReports()**: Get all the open report
 - * **Parameters:**
 - none
 - * **Return:**
 - areas: List<Report>
- **validateReport()**: Set the report to Valid
 - * **Parameters:**
 - report: Report
- **unvalidateReport()**: Set the report to Inalid
 - * **Parameters:**
 - report: Report
- **Area-Highlighter:**
 - **getHighlightedArea()**: Get the area with more violation
 - * **Parameters:**
 - filter: List<ReportType>
 - from: Date *analyze data only starting from this date*
 - to: Date *analyze data only until this date*
 - * **Return:**
 - areas: List<Area>
- **RankingManager:**
 - **getRanking()**: Get the ranking of the license plate that committed more violation
 - * **Parameters:**
 - filter: List<ReportType>
 - from: Date *analyze data only starting from this date*
 - to: Date *analyze data only until this date*
 - * **Return:**
 - ranking: List<String> *ordered list of license plate*
- **NotificationManager:**
 - **sendNotification()**: Send a notification to the nearest traffic warden

- * **Parameters:**
 - longitude: double
 - latitude: double
 - **subscribe():** Subscribe to receive notification for new report (only user defined as traffic warden can access)
 - * **Parameters:**
 - token: String *token used for firebase*
- **ALPR:**
 - **recognizeLicensePlate():** Return the license plate recognized in the picture
 - * **Parameters:**
 - image: `ByteStream`
 - * **Return:**
 - licensePlate: `String`
- **SuggestionSubsystem:** More in detail:
- **SuggestionDisplayer:**
 - **getSuggestion():** Get the suggestions (only user defined as municipality can access)
 - * **Parameters:**
 - none
 - * **Return:**
 - suggestions: `List<Suggestion>`
- **SuggestionMiner:**
 - **start():** Start mine data to create suggestion
 - * **Parameters:**
 - none

2.7 Selected architectural styles and patterns

- **RESTful architecture:** REST is a software architectural style.

The communication between SafeStreets ApplicationServer and the user application uses HTTP(S) requests which follows REST principles. One of the most important REST principles is that the interaction between the client and server is stateless between requests. Each request from the client to the server must contain all of the information necessary to understand the request. For instance the client wouldn't notice if the server were to be restarted at any point between the requests. This is achieved using a uniform and predefined set of stateless operations based on strict use of HTTP request types.

The usage of RESTful requests is visible in the sequence diagrams included in the Runtime View section 2.4

- **Three-tier architecture:**

The Three-tier architecture is a client-server software architecture pattern (the most widespread used approach in multitier architecture). The usage of this pattern is already specified in the high-level view section 2.1 and in the deployment view section 2.3.

- **Presentation Tier**

Layer which users can access directly: using the mobile application available for Android and iOS devices or using the web app (compatible with any modern browser).

- **Logic Tier**

Controls application's functionality by performing detailed processing. The application server uses Node.js and handles all the requests coming from the users applications.

- **Data Tier**

The data tier includes the data persistence mechanisms (database servers, etc.) and the data access layer that encapsulates the persistence mechanisms and exposes the data. The database machine will run MySQL server (a relational database management system). On this machine will be stored all the persistent data.

The main advantage of using this architecture is that any of the three tiers can be upgraded or replaced independently, without impacting other areas of the application. This gives the possibility to scale the application up and out (e.g. by adding multiple servers).

2.8 Other design decisions

- **Relational Database:** The advantages of relational databases (Expressive query language, secondary indexes, strong consistency, etc.) make them a preferred choice for this application as it will handle well-structured data.
We've chosen MySQL as a DBMS because it's open source, easy to use, and there is a lot of material available. Also the development team is already familiar with it.
- **Application Server:** Node.js is a FOSS runtime environment for server-side and networking applications. Node.js applications are developed in JavaScript

and the apps run inside the Node.js runtime environment. We've chosen Node.js because there are a lot of packages, tools and material available (power of one of the largest communities) and the development team is already familiar with it. It's fast and easy to set up a Node.js infrastructure. Also Node.js apps are known for performing really well in high volume traffic and easily scaled up.

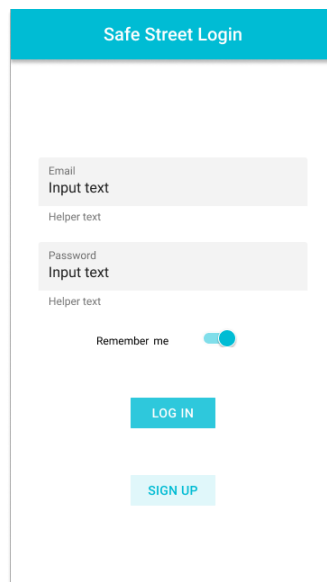
- **Router:** Routing refers to how an application's endpoints (URIs) respond to client requests. We will use Express.js, which is one of the popular framework for Node.js, to provide the router functionalities as it make the development process fast and easy.
- **OpenALPR:** We've decided to use this software beacause it's an open source library with bindings available also for Node.js, so the OpenALPR Software can integrate easily with our application and it will be one of the core components. This software has proved to be precise and efficient. (It has become the standard for these kind of applications).
- **Firebase:** As already said in section 2.4.2. Firebase Cloud Messaging (FCM) is a cross-platform messaging solution that lets you reliably deliver messages at no cost. Using FCM, we can easily notify a traffic warden client app that a new report is available.
- **ClientAPP:** The client applications will be developed with Flutter. Flutter is an app SDK for building applications for iOS, Android, and web from a single codebase. The main advantages is the possibility to provide the same application with the same UX on multiple platform writing the code only once. (Do more write less).

3 User interface design

The idea is to build a simple, intuitive and material UI. The user interface and the UX should be the same on every device (android, iOS, webapp).

3.1 User interface mockups

Here we provide some mockups of the user interface. The purpose of these images is to give an approximate idea of the look of the application. The actual implementation may differ a bit.



The mockup shows a mobile application interface for 'Safe Street Login'. It features a teal header bar with the title 'Safe Street Login'. Below the header, there are two input fields: 'Email' and 'Password'. Each field has a label, an 'Input text' placeholder, and a 'Helper text' label. The 'Remember me' checkbox is checked, indicated by a teal toggle switch. Below the input fields, there are two buttons: a teal 'LOG IN' button and a light blue 'SIGN UP' button.

Figure 14: Sign-in page

3.1.1 User

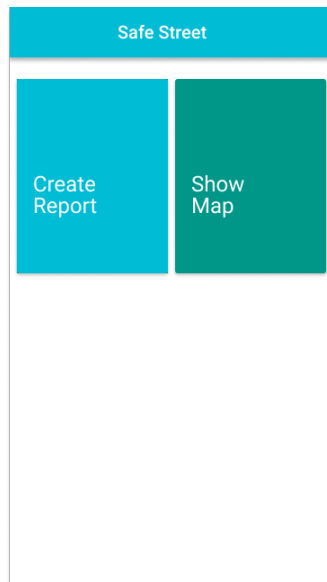


Figure 15: User HomePage

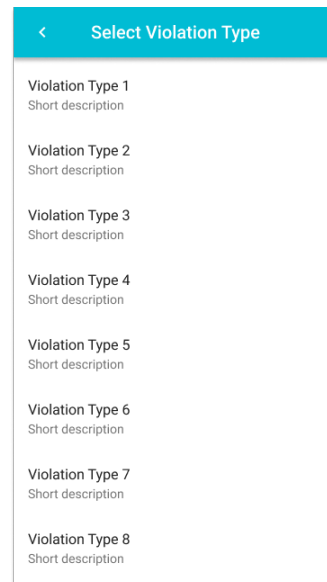


Figure 16: Select violation type

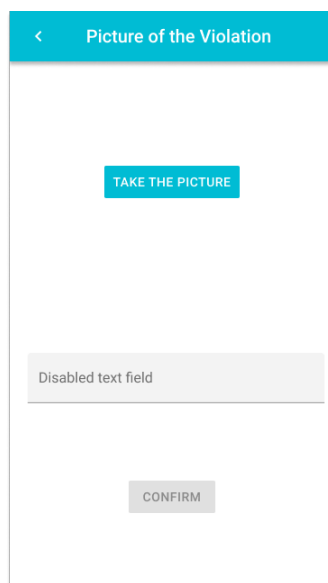


Figure 17: Take Picture

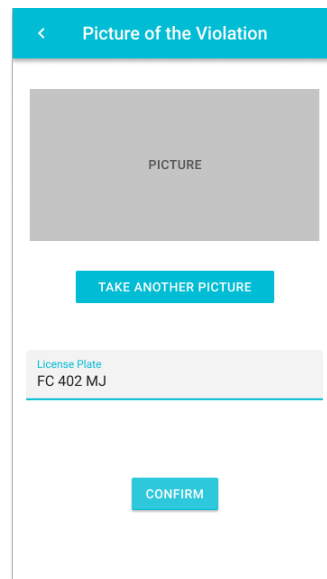


Figure 18: Check License Plate Number

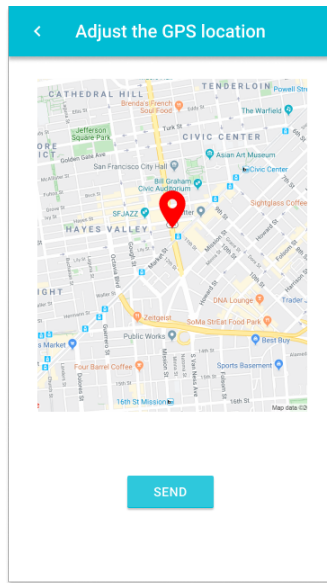


Figure 19: Check Location and submit report

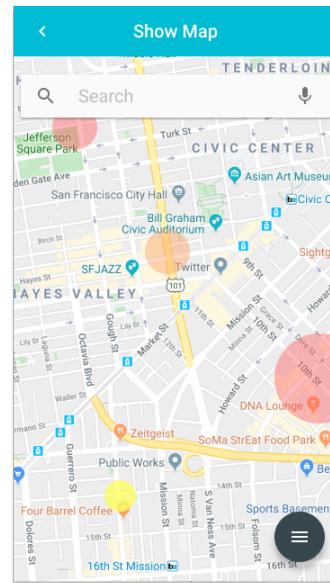


Figure 20: Show Map with highlighted area

Show Map

Select the type of violations you want to filter

Type violation 1

☐

Type violation 2

☒

Select the date range

From:

2019/12/03

Helper text

To:

2019/12/10

Helper text

✓

Figure 21: Select data to show on the map

3.1.2 Traffic Warden

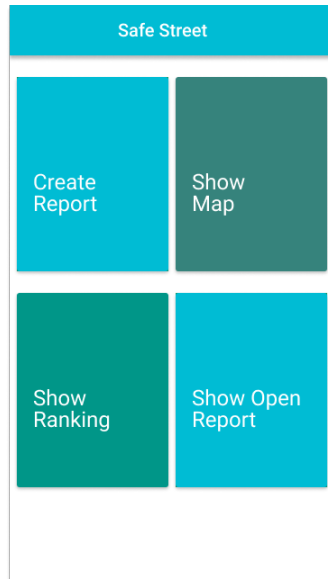


Figure 22: Home

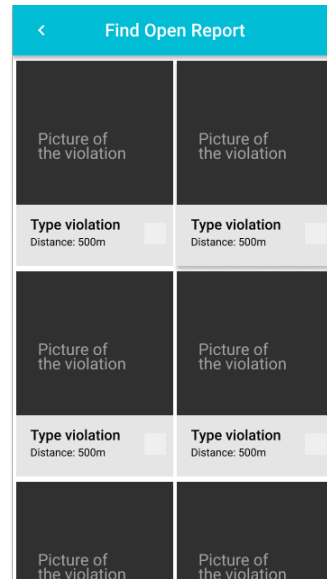


Figure 23: Find open report

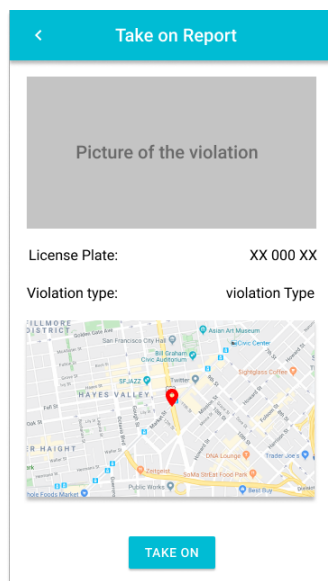


Figure 24: Take on report

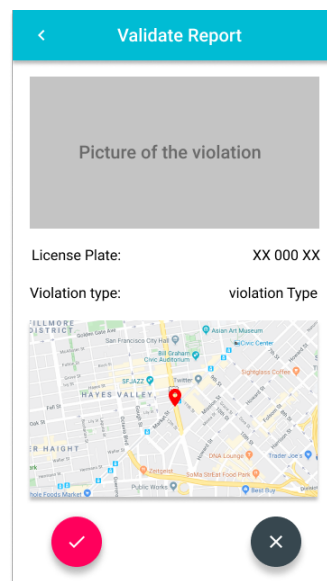


Figure 25: Validate report

<

Ranking Filter

Select the type of violations you want to filter

Type violation 1

☐

Type violation 2

☒

Type violation 3

☐

Select the date range

From:

2019/12/03

Helper text

To:

2019/12/10

Helper text

✓

<

Ranking

1	XX 000 XX
2	XX 001 XX
3	XX 002 XX
4	XX 003 XX
5	XX 004 XX
6	XX 005 XX
7	XX 006 XX
8	XX 007 XX
9	XX 008 XX
10	XX 009 XX
11	XX 010 XX

Figure 26: Set filter for the Ranking of vehicles

Figure 27: Check Ranking of vehicles

3.1.3 Municipality's clerk

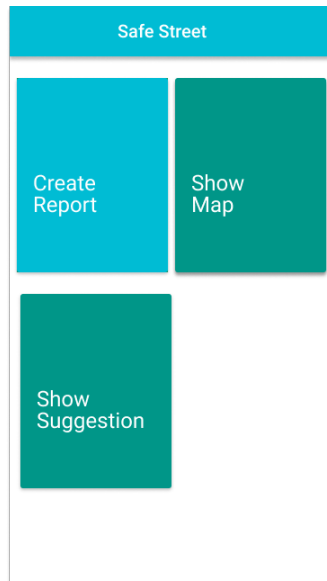


Figure 28: Home

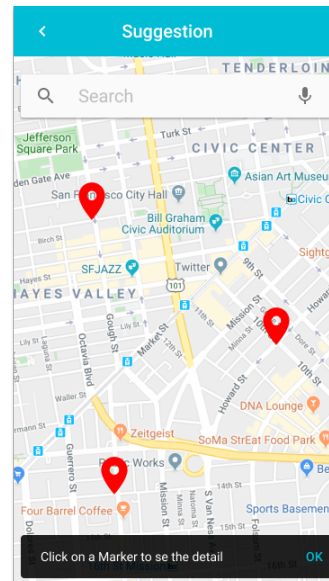


Figure 29: Check suggestions

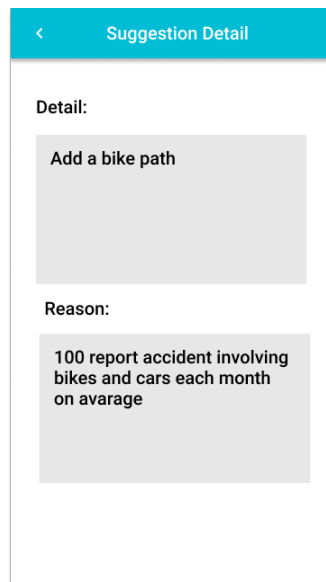


Figure 30: Check suggestion details

3.2 UX Diagrams

In all the diagrams the actions to go back are omitted to simplify the diagrams.

- **User**

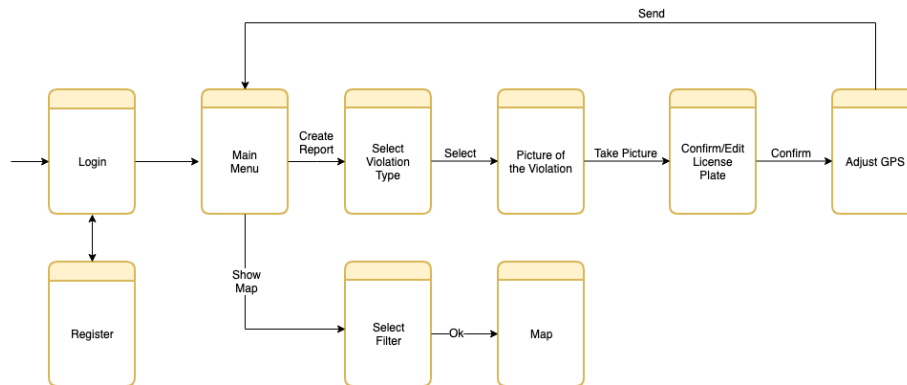


Figure 31: User UX diagram

- **Traffic warden**

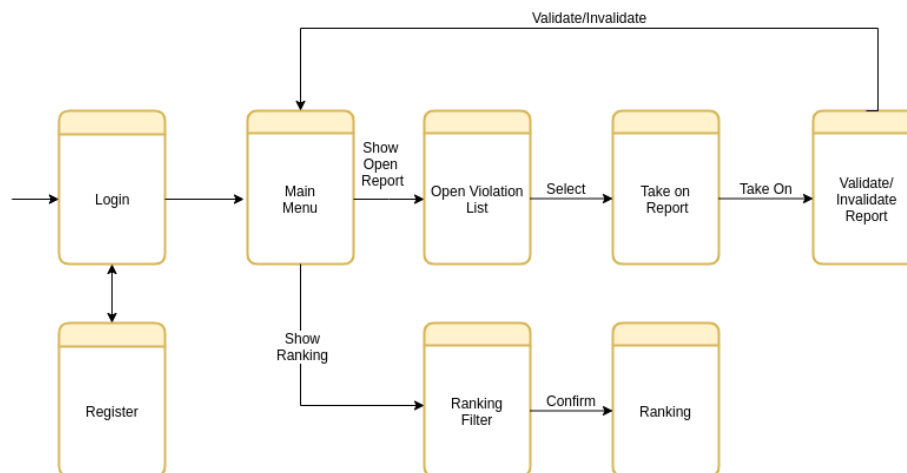


Figure 32: Traffic warden UX diagram

- Municipality's clerk

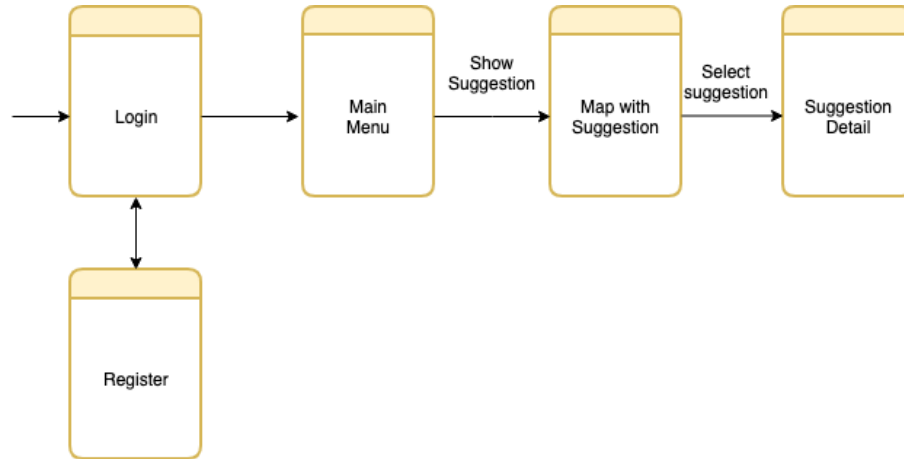


Figure 33: Municipality's clerk UX diagram

4 Requirements traceability

The following requirements were described in the RASD:

- **R1** The system has to allow sign up of new users.
- **R2** The system has to allow users to login.
- **R3** The system must allow users to send pictures of violations, including their date, time, and position to authorities.
- **R4** The system, when it receives a picture, must run an algorithm to read the license plate.
- **R5** The system must allow the user to fix the license plate if the algorithm read it wrong.
- **R6** The system must allow the user to fix the location if the GPS acquired the wrong location.
- **R7** The system must store the information of the report sent by the users.
- **R8** The system must send a notification to the traffic warden.
- **R9** The system has to allow one traffic warden to "take on" the report.
- **R10** The system has to allow the traffic warden to validate or to invalidate the report.
- **R11** The system must mine the reports to highlight the streets (or the areas) with the highest frequency of violations.
- **R12** The system has to allow search for a specific location and select from a menu which kind of data he wants to see.
- **R13** The system must distinguish privilege of account (normal, traffic warden, municipality clerk).
- **R14** The system must mine the reports to create a ranking of the vehicles that commit more violation.
- **R15** The system must gather data about accidents from the municipality.
- **R16** The system must cross the information provided by the municipality about the accidents with its own data to identify potentially unsafe areas.
- **R17** The system must show possible interventions to improve unsafe area.

Here it's explained how the requirements defined in the RASD map to the components defined in the previous sections.

Requirement	Components
R1	Authenticator DBMS ClientAPP Router
R1	Authenticator DBMS ClientAPP Router
R3	ReportManager ClientAPP Router
R4	ReportManager ALPR ClientAPP
R5	ClientAPP
R6	ClientAPP
R7	ReportManager DBMS
R8	Firebase NotificationManager DBMS
R9	ReportManager DBMS
R10	ReportManager DBMS
R11	AreaHighlighter ReportManager DBMS
R12	ClientApp Map Provider
R13	Authenticator DBMS
R14	RankingManager ReportManager DBMS
R15	MunicipalityAPI SuggestionMiner
R16	MunicipalityAPI SuggestionMiner ReportManager DBMS
R17	SuggestionDisplayer

5 Implementation, integration and test plan

An accurate analysis of the RASD and DD must be done before starting the implementation of the system.

The system is divided in the following subsystems which can be implemented independently one from each other:

- Client Applications (Android, iOS, webapp)
- Application Server
- Web Server
- DBMS
- External components:
 - Map Provider
 - ALPR
 - Firebase
 - MunicipalityAPI

The idea is to adopt a bottom-up approach for the implementation strategy, but also, in order to provide first at least the key features of the system, the parts needed by those features will have an high priority. The table below lists the importance and implementation difficulty of the main features of the system which it is helpful to better understand the implementation and integration plans.

Feature	Importance for customers	Implementation Difficulty
Sign up and Login	Low	Easy
Report violations	High	Medium
Get notifications	Medium	Easy
Take on reports	High	Medium
Cross data to suggest interventions	Medium	Hard
Show ranking of vehicles	Low	Easy
Show critical areas	Medium	Medium
Show suggestions	Medium	Medium

The table shows how the priority is to have users sending reports and traffic wardens to take on them, which is the main goal of the system. The functionalities that allow users to visualize data should pretty easy to provide as they consist in getting data from the database with simple queries and visualize it (as a plain list or on a map). A more challenging task is to provide an algorithm that can elaborate proper suggestions for possible interventions, however this is an advanced function that does not affect normal users.

5.1 Integration Process

As already stated, the integration will be performed from following a bottom-up approach. The modules which are being integrated should pass the unit tests in order to be sure that the components are working fine on their own and that if an integration test fails, the problem is regarding the integration itself.

5.1.1 Integration Order

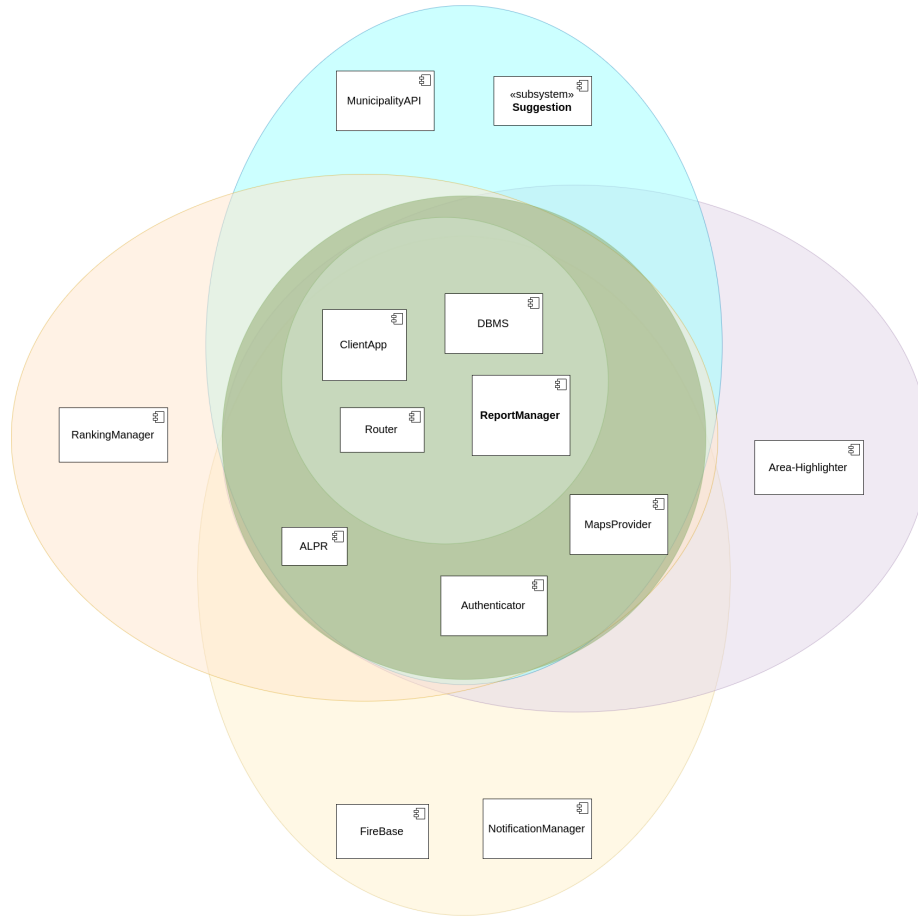


Figure 34: Integration Order Diagram

1. The priority is to have the main functionality working ASAP, so the integration between the key components (ClientAPP, Router, ReportManager and DBMS) must be done first. This will be helpful to test the core of the main functionality.
2. However to actually make it usable in a real use case for an user also the ALPR, MapProvider and Authenticator must be integrated.
3. The NotificationManager and FireBase will then be integrated.
4. The component needed to visualize the critical areas (Area-Highlighter) will be integrated.
5. The component needed to elaborate the ranking of vehicles (RankingManager) will be integrated.
6. Eventually the components needed to provide the suggestions (MunicipalityAPI and the Suggestion Subsystem) will be integrated.

This is the best strategy from the point of view of the standard users. Which could be a reasonable strategy to adopt because the project is crow-funded. However, as clearly visible in the diagram, the phases 3,4,5,6 can be switched.

5.2 Testing

A Continuous integration (CI) service like Travis will be used. This service runs tests automatically when pushing code changes providing instant feedback on the code changes. Integrated with proper settings on github it also avoids the merge of code that doesn't pass tests to the master/production branch.

5.2.1 Unit testing

Significant unit tests should be written for each component. We don't suggest to adopt a test driven development approach, however appropriate unit tests give several benefits. (For instance: make it easier to identify bugs in code earlier, simplify debugging process, provide documentation, reduce the time needed to fix bugs.)

Mocha (a simple and easy to use JS test framework running on Node.js) will be used for unit testing the application server components.

Unit test for the ClientAPP (developed with Flutter) will be written using the packages suggested by the official documentation: the test package, which provides the core framework for writing unit tests and the fluttertest package, which provides additional utilities.

It's important to remind that unit tests only test individual software components and methods. They don't test databases, network resources, render to screen, or the receive of user actions from outside the process running the test. In fact also other type of testing are needed.

5.2.2 Integration testing

The main goal of integration process is to avoid as much errors as possible at each step of the process. The system will incrementally integrate components as soon as they are completely developed and released. The Bottom-up strategy will be adopted for most of the integration process. In this way we can obtain feedbacks about system functionalities as soon as components are released and it's possible to parallelize integration of different parts of the system.

The router component (which will be an express.js router) will be tested using Mocha and Supertest. Supertest is a library written to test HTTP calls in Node.js, which is perfect to write tests for RESTful API.

5.2.3 Other types of testing

Once the system is completely integrated, also non-functional requirements should be tested. For example with security, load, performance and stress testing.

The security of the system may be tested, for instance, by doing some penetration tests.

6 Effort spent

Cazzola Federico @f-cazzola

Date	Hours	Description
23/11/19	2	Architectural design
24/11/19	3	Architectural design
29/11/19	2	Architectural design
30/11/19	6	Architectural design
01/12/19	8	Architectural design
04/12/19	4	UI design
05/12/19	6	UI design and UX
06/12/19	4.3	UI design and UX
07/12/19	8	Interfaces, General improvements
total	43.3	

Dotti Francesco @dottif

Date	Hours	Description
15/11/19	1	Initial Structure
22/11/19	1.5	Introduction
23/11/19	1.5	Architectural design
29/11/19	0.5	Component view
30/11/19	6.5	Component Deployment Runtime view
01/12/19	8	Architectural design
03/12/19	0.5	Architectural design
04/12/19	2.5	Architectural design
05/12/19	4	Req. traceability and Implementation
06/12/19	3	Implementation and Testing
07/12/19	8	IT,UI, General improvements
total	37	

7 References

- Flutter
- Firebase
- ALPR
- MySQL
- Express.js
- Node.js
- Supertest
- Mocha

7.1 Software used

- Figma: UI mockups
- Draw.io: Diagrams
- LaTeX: Document preparation system
- VIM: The best Text Editor
- Latex plugin for VIM
- vs code: Text Editor with user-friendly live share plugin