POLITECNICO
MILANO 1863

SafeStreets
Design Document
version 1.1

Frangi Alberto, Fucci Tiziano

A.Y. 2019/2020

Table of contents

Chapter 1

Introduction

1.1 Purpose

The purpose of this document consists of giving more technical details than the RASD concerning the SafeStreets system.

The RASD completely describes the system in terms of functional and nonfunctional requirements and serves as a contractual basis between the customer and the developer and it must be written in the language of the customer's domain of business/expertise, the DD's purpose instead is to provide a description for how the new system will be constructed, it provides a description of the system architecture, software, hardware, database design, and security.

In particular this document will explain the following topics:

- high level architecture and its components' requirements;
- run-time behavior;
- design patterns;
- additional user interfaces information;
- implementation, integration and testing plan.

1.2 Scope

Here a review of which is the scope of the application is made referring to what has been stated in the RASD.

SafeStreets is an application to be used both from civilians (users) and authorities, in order to help the latter and reduce traffic violations. Registered authorities can automatically receive reports made by users, so the service acts as an intermediary.

1.3 Definitions, acronyms, abbreviations

1.3.1 Definitions

- **User**: a civilian customer that can use the application to:
  - notify authorities of some violation;
  - check which are the most dangerous (i.e. with the most violations) streets.

  In this document, "user", "citizen" and "civilian" are completely equivalent, where not specified.
- **Authority**: a member of the local police who has access to reports made by users. The authorities evaluate the reports sent by the user to determine if the violation stands.
- **Report**: a message consisting of:
  - a picture showing the car in order to show the occurring violation;
  - date and time of the picture;
  - GPS position of the place where the violation occurred;
  - the street where the violation occurred (automatically retrieved from the geographical position);
  - the type of the violation (input by the user)
- **Available**: a report is available for an authority if its position is within the municipality assigned to the authority.
- **Violation**: a situation that, according to the user who sent the report, is a violation of the traffic laws.
- **Intervention**: a brief text suggesting a possible solution in order to improve safety and discourage future violations.

1.3.2 Acronyms

- **AES**: *Advanced Encryption Standard.*
- **API**: *Application Programming Interface.*
- **DB**: *Data Base.*
- **DBMS**: *Data Base Management System.*
- **DD**: *Design Document.*
- **DMZ**: *DeMilitarized Zone.*
- **GPS**: *Global Positioning System.*
- **HTML**: *HyperText Markup Language.*
- **ICM**: *Item-Content Matrix.*
- **IEEE**: *Institute of Electrical and Electronics Engineers.*
- **MVC**: *Model-View-Controller.*
- **OCR**: *Optical Character Recognition.*
- **PKC**: *Public Key Cryptography.*
- **RASD**: *Requirements Analysis and Specifications Document.*
- **RDBMS**: *Relational Data Base Management System.*
- **REST**: *REpresentational State Transfer.*
- **SHA-3**: *Secure Hash Algorithm 3.*
- **SS**: *SafeStreets.*
- **UI**: *User Interface.*
- **UML**: *Unified Modeling Language.*
- **UX**: *User eXperience.*
- **XML**: *eXtensible Markup Language.*

1.4 Revision history

- **Version 1.0**:
  First release.
- **Version 1.1**:
  - Improve class diagram;
  - correct some typing errors;
  - correct some errors in 2.4;
  - other minor fixes.

1.5 Reference documents

The main reference documents are the "SafeStreets Mandatory Project Assignment" specification document and the SafeStreets RASD. The complete list of references is provided in chapter 6.

1.6 Document structure

- **Chapter 1** provides a brief explanation on the DD purpose and a quick introduction SafeStreets.
- **Chapter 2** aims to provide a description of the system's architecture.
- **Chapter 3** specifies the design of user interfaces and describes the user-application interaction.
- **Chapter 4** contains requirements traceability.
- **Chapter 5** describes the implementation plan.
- **Chapter 6** shows the effort of each group member.
- **Chapter 7** contains all the references used to make this document.

Chapter 2

Architectural design

2.1 Overview: high-level components and their interaction

The application is built following the principles of the three-tier architecture: the three logic layers of presentation, application and data access rely on three corresponding hardware layers. This architecture is preferred to one-tier and two-tier architectures due to some important characteristics, some of which are:

- *Flexibility*: each tier can be manged or scaled independently at any time, without affecting the others;
- *Scalability*: following a scale-out approach, performances can be improved through node replication, without affecting the other tiers. Load balancing systems distribute the working load among the nodes;
- *Maintainability*: because each tier is independent from the others, updates or changes can be released without affecting the whole system;
- *Availability*: with this architecture, it is less likely to have failures that compromise the whole application. Load balancing and node replication minimize the performance loss when a failure occurs.

These aspects will be even better explained in section 2.6. A general view of the system architecture is provided in the following diagram, which provides an overview of the system architecture:

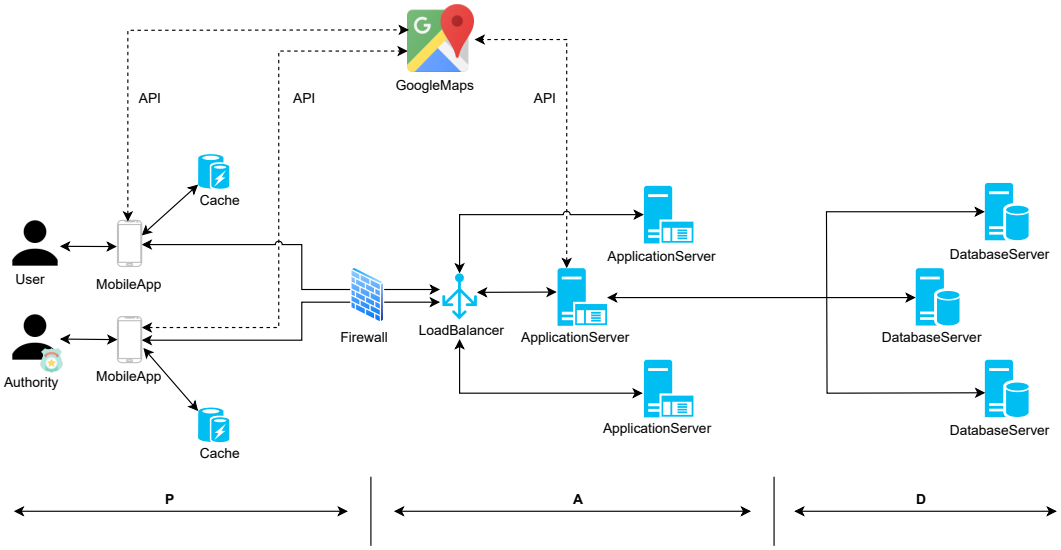


Figure 2.1: System architecture

Users and authorities are provided with mobile devices and access the service through the SafeStreets mobile app. The mobile app communicates with the application layer, which is made by one or more application servers, linked to the database servers. The scale-out approach allows to adjust the number of hardware resources at any time. For what concerns the data access layer, database servers contain sensitive information, such as password hashes, license plate numbers, identification numbers and so on, so it is important to protect all the back-end of the application. In order to do this, the application and data access levels are protected by a firewall that performs traffic control at the level of the single packet, creating a DMZ in which the communication is safe. Another solution to reduce the computational load, as well as the messages, is the use of caches in the presentation tier: the mobile app stores on the mobile device memory part of the user's data, such as the sent reports and the results of reports evaluation, in order to make them available even when the users (or the authorities) have no access to Internet. This avoids many repeated requests for the same data, with heavy impact on the application and data access layers' performance. The system also includes a recommender system: exploiting data mining techniques, such as association rules, it can suggest to the authorities of one municipality possible interventions to improve security on the streets.

2.2 Component view

This section shows the internal composition of the application server, which is the core component of the system and contains the business logic, and its interaction with the external interfaces. In order to obtain a clear and understandable view, the system is described with three different diagrams, which contain the various components that communicate with a certain external component. The diagrams show the interaction of the application with the Google Maps APIs, the mobile application and the DBMS. After each diagram, a brief explanation of each component's role is provided, exception made for the router, which is described before. Note that some managers can appear in more than on diagram, since they interact with more than one external interface.

- **Router**: its role is to manage all the messages incoming from the other subsistems, forwording them to the right component and calling the appropriate method on it.

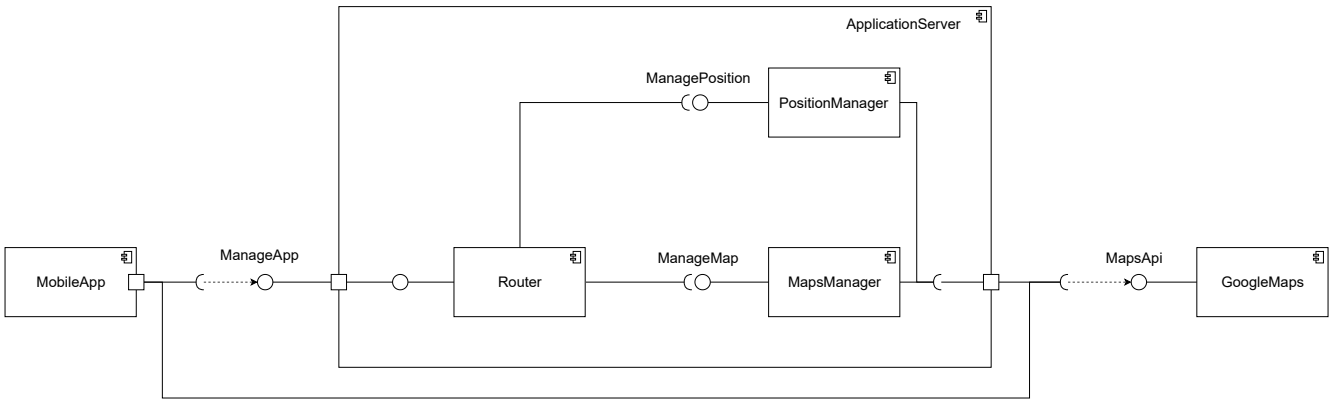


Figure 2.2: Component diagram - Google Maps

- **PositionManager**: interacts with the client to retrieve the position of the user, then returns it to the caller.
- **MapsManager**: contains all the necessary methods to display the map of the violations and interact with it. Given a set of streets, interacts with Google Maps APIs to obtain a map.

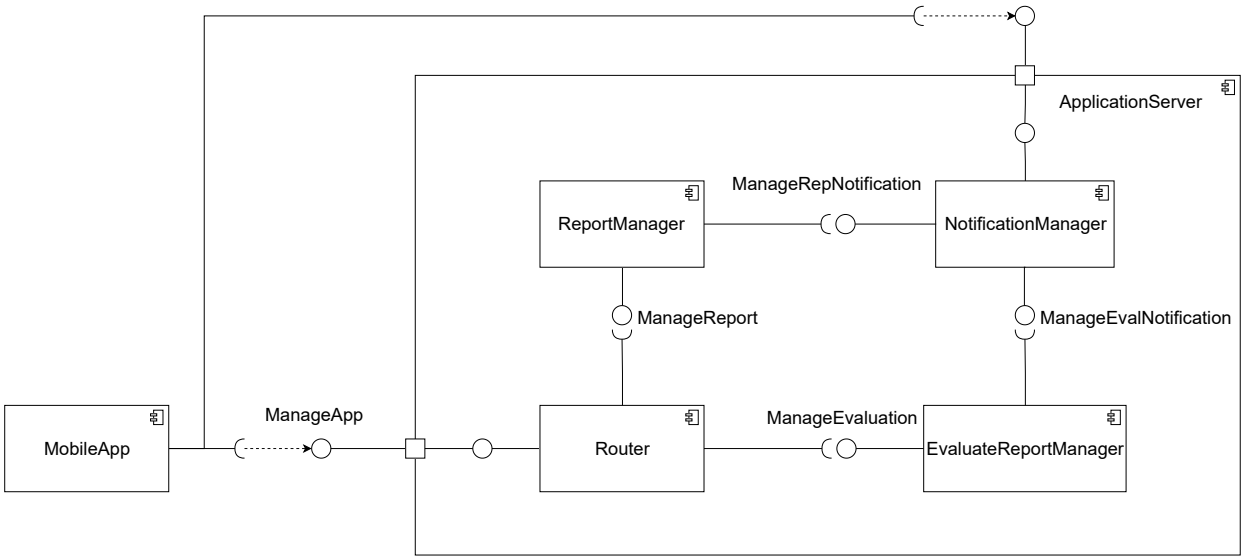


Figure 2.3: Component diagram - Response

- **ReportManager**: manages the logic inherent the notification of the authorities, when a new report is available.
- **EvaluateReportManager**: as the ReportManager, but to notify users when one of their reports is evaluated.
- **NotificationManager**: manages the communication with the mobile application, to make sure that the messages are received correctly.

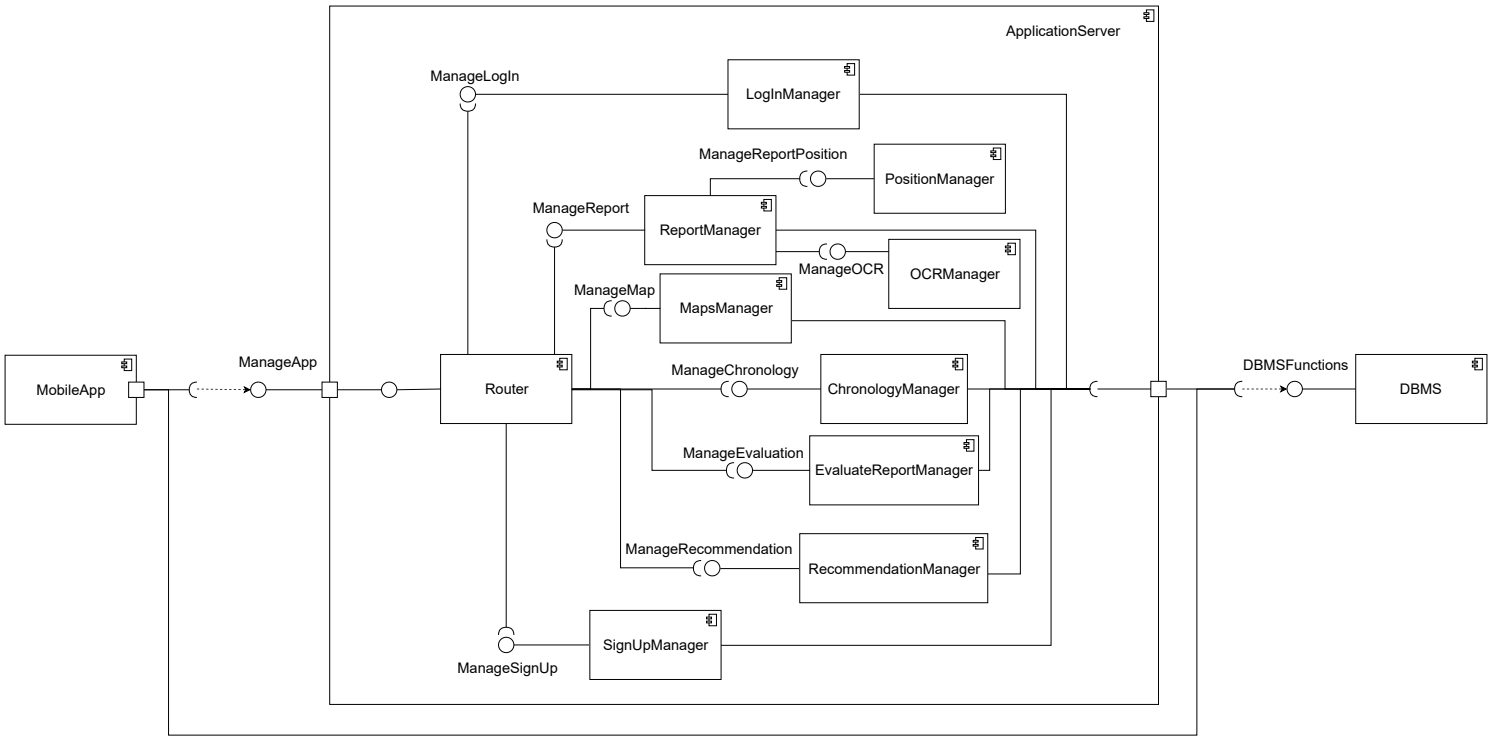


Figure 2.4: Component diagram - DBMS

- **SignUpManager**: handles the subscription process, both for users and authorities. Interacts with the DBMS to check if the provided credentials are already associated to other accounts.
- **LogInManager**: receives two strings, an e-mail address and a hash. Checks if the e-mail address is in the DBMS and if the password hashes match.
- **ReportManager**: contains all the logic necessary to complete a report: given all the information input by the user, completes it with the missing ones, like the user position and the license plate number.
- **PositionManager**: retrieves the device position to be attached to the report.
- **OCRManager**: given a picture, runs the OCR algorithm and returns a string. Notifies the PositionManager if, for any reason, the license plate number cannot be read.

- **MapsManager**: interacts with the DBMS, in order to retrieve the number and the type of reports for a given set of streets.
- **ChronologyManager**: when requested, obtains from the DBMS all the reports made by a given user and their status.
- **EvaluateReportManager**: stores in the DB the result of the evaluation of a report.
- **RecommendationManager**: implements the recommender system that periodically interacts with the DBMS, to check if some streets are involved in particular types of violation. In that case, it will suggest an intervention to the authorities of that municipality.

2.3 Deployment view

This section shows how the core functionalities of the system are developed on different devices, in other words, how software artifacts are distributed on the deployment targets.

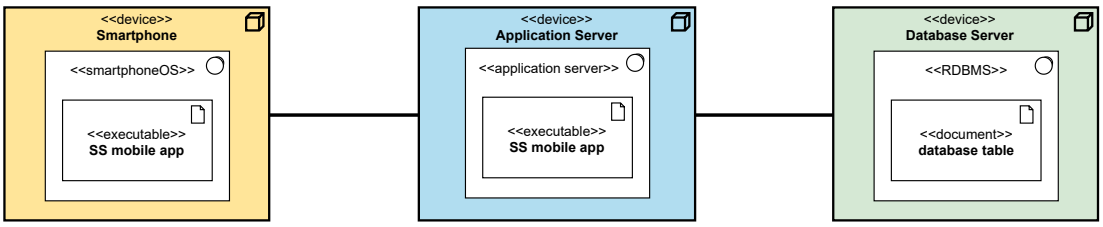


Figure 2.5: Deployment Diagram

The deployment diagram follows UML standard notation and does not represent external systems, like the load balancer and the firewall, in order to focus only on what must effectively be deployed on the three tiers:

- **Tier 1**: hosts all the presentation logic. The mobile application must be developed for both iOS and Android, to make it available for most of the devices.
- **Tier 2**: here the application logic must be deployed. The application server communicates with both the other tiers, handling all the requests.

- **Tier 3**: hosts the data access logic. The database server hosts a relational DBMS. The RDBMS is preferred due to some reasons, like the possibility to use foreign keys in the tables or easily implement some rules (like the one of requirement R13). With a future growth of data amount, a hybrid approach can be taken into account, for instance with the use of structured storage.

2.4 Runtime view

In this section the sequence diagrams of the various functionality will be presented. It is up to the application to control and guarantee that each user can access only to the right functionality.

2.4.1 Making report

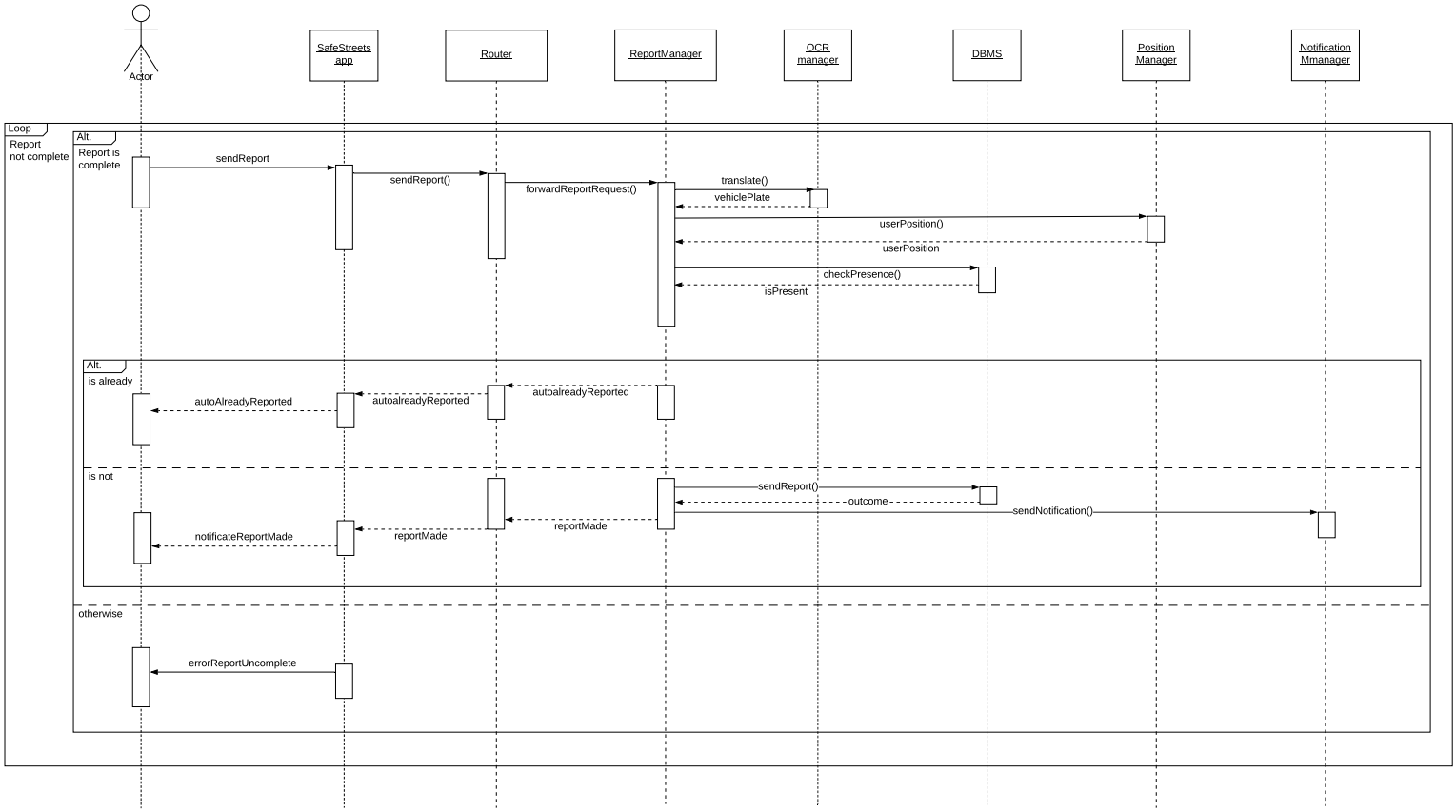


Figure 2.6: How the report works

This sequence diagrams explains the process through which a user can report a traffic violation.

Once he has opened the application and logged in/signed up he can choose the "Report violation" option, when he does that the application sends the request to a router that forwards it to the right component (in this case the "Report Manager").

Once the report has been received, the Report Manager sends the position (intended as geographical coordinates) to the Position Manager, who will take care of transforming it into information on the road on which the user is located.

The user has to fill the form to report the traffic violations with the violation type and a picture of the violation. The request cannot be sent until all the form is completed. It is up to the application to ensure that the position of the user has been correctly recorded, in case it is not possible to obtain it, it must prevent the sending of the report.

Once the form has been sent, the manager takes care of the rest. First, the manager has to convert the picture of the plate into a string, so it sends the plate's picture to the OCR manager, in this way, using some picture manipulation techniques the system is able to obtain a string with the license plate.

Once obtained that, the report manager uses a query to look up the database trying to find if there is already a pending (or resolved) report involving that car in that street in the last 24 hours, with the same violation type reported. In this way a car won't be reported twice for the same violation. If the vehicle has been already reported, the application will display a message like "This vehicle has been already reported", to notify the user that the request failed, otherwise the report is stored into the database, and a notification is sent to the authorities (using the notification manager).

2.4.2 Visualize Maps

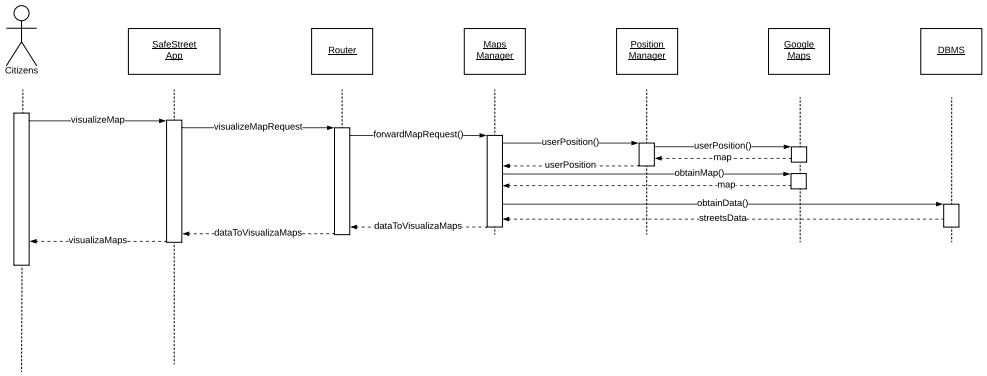


Figure 2.7: How map visualization works

This sequence diagrams explain the process through which a user can visualize the map to identify which streets have the highest number of violations.

Once he has opened the application and logged in/signed up he can chose the "Visualize map" option, when he does that the application send the request to a router that forward the request to the right component (in this case the "Maps Manager").

The manager take care of everything, it obtain the user position in order to load the correct map from google maps, then it contact the db to retrieve information about the streets in that map to apply the correct colors and in the end it sends all to the application.

2.4.3 Visualize reports' history

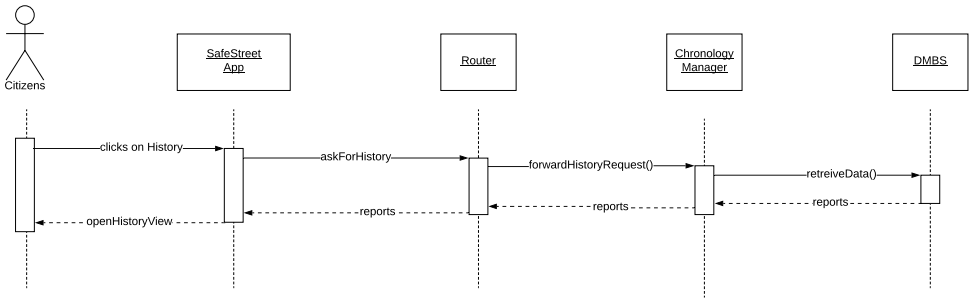


Figure 2.8: How reports' history works

This sequence diagrams explains the process through which a user can visualize the history of his previous reports.

Once he has opened the application and logged in/signed up he can choose the "My reports" option, when he does that the application sends the request to a router that forwards it to the right component (in this case the "Chronology Manager").

The manager simply obtains all the previous reports by querying the DB, if there aren't reports the manager sends a message to the app and displays something like "It seems that you didn't make any report up to now". The user will see date&time of each report, the status (pending, approved or rejected), the street and the type of violation.

The plate won't be displayed to protect the privacy of the other citizens.

2.4.4 Evaluate report

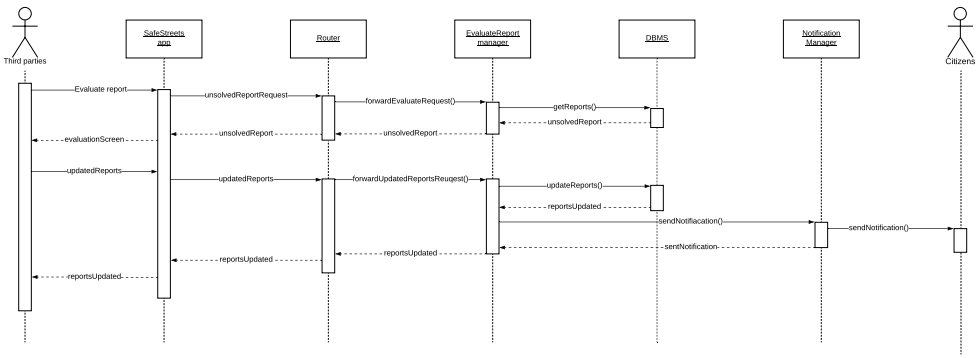


Figure 2.9: How reports are evaluated

This sequence diagrams explains the process through which an authority can evaluate pending reports.

Once he has opened the application and logged in/signed up he can choose the "Evaluate" option, when he does that the application sends the request to a router that forwards it to the right component (in this case the "Evaluation Manager").

The manager retrieves from the DB all the unsolved reports and sends them to the application module, each time that an authority evaluates a report, the manager updates the DB and, using the notification manager, sends a notification to the user who made it to notify him the updated status.

2.4.5 Asking for recommendation

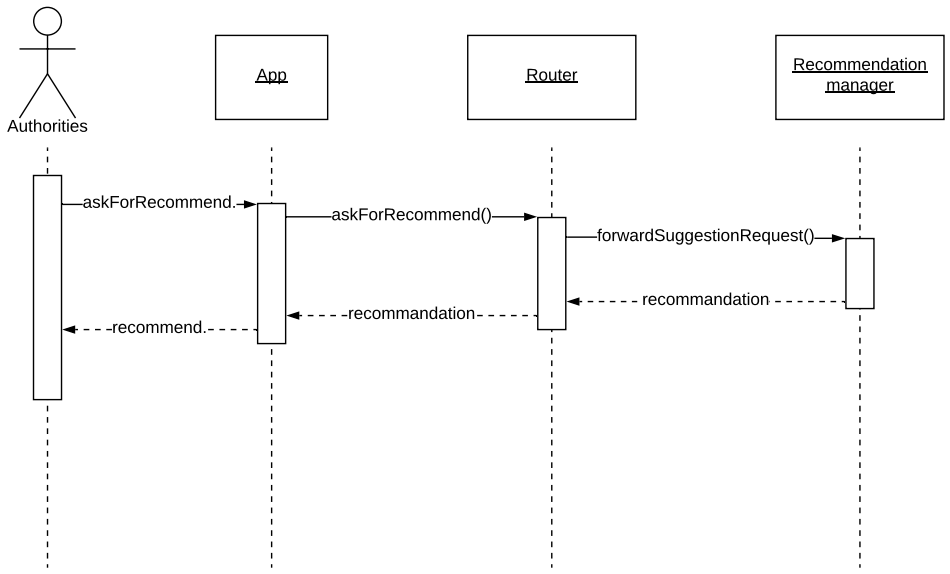


Figure 2.10: How the recommendation is provided

This sequence diagrams explains the process through which an authority can receive suggestions to improve safety on the streets.

Once he has opened the application and logged in/signed up he can choose the "Recommend" option, when he does that the application sends the request to a router that forwards it to the right component (in this case the "Recommendation Manager").

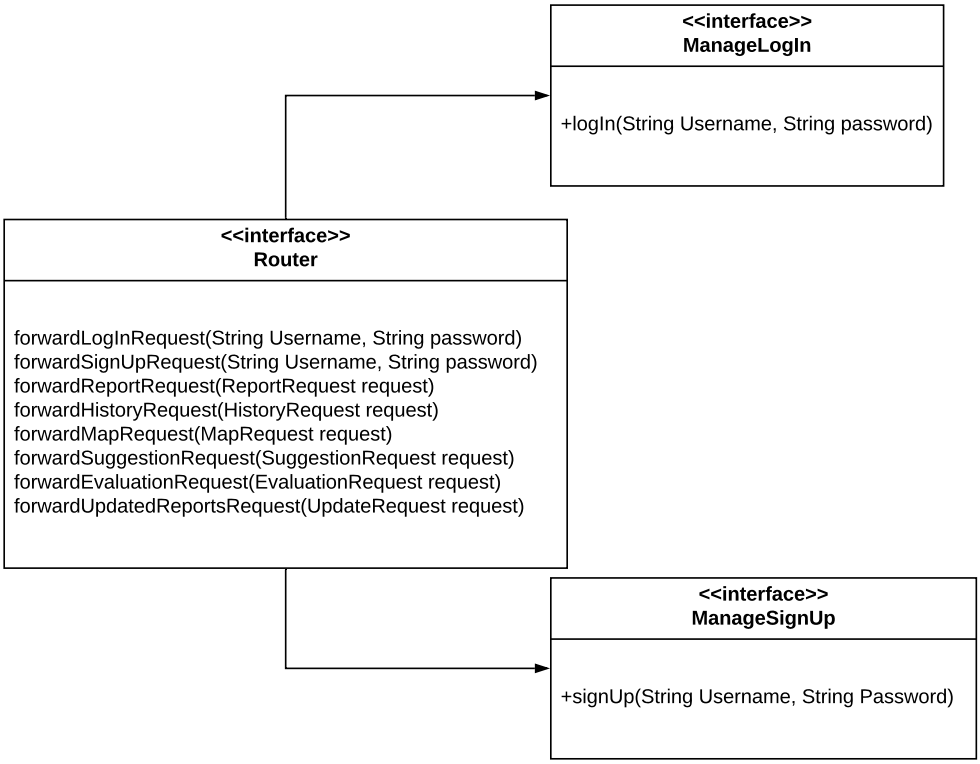
To receive recommendations on how to improve the safety on the streets, the manager will use a recommender system based on content base approach, based on item content matrix. To receive suggestions, the manager will consult the recommender system and obtain the best possible solution, which will be forwarded to the application.

2.5 Component interfaces

2.5.1 Overall summary

In this section the interfaces necessary to each function will be analyzed, explaining the purpose of the main functions.

2.5.2 Main page



In the figure above, the interface to manage the main page of the application is represented . It is important to underline that it is the task of the application to ensure that each user can access the functions dedicated to him and not to others (for example a citizen can't see the evaluate option).

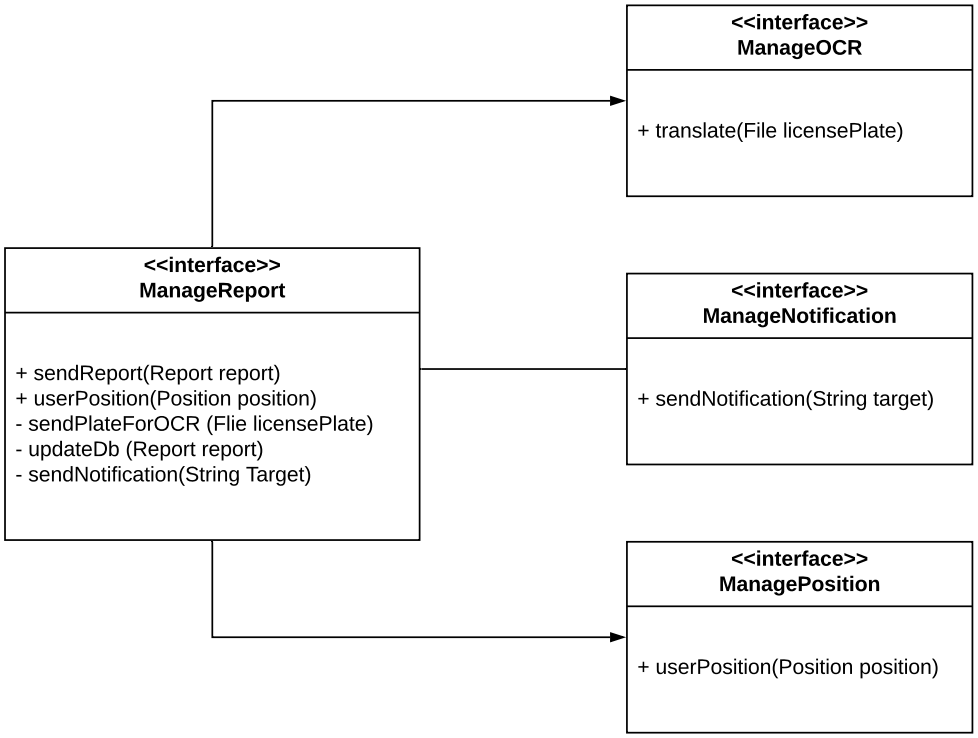
- **Router**
  - **forwardLogInRquest**: this method obtain the signUp request and forward it to the sign up manager, the request is an aggregate of an unique id to identify it, the username and the password.
  - **forwardSignUpRequest**: this method obtain the signUp request and forward it to the sign up manager, the request is an aggregate of an unique id to identify it, the desired username and password.

- **forwardMakeReportRequest**: this method receive the reportRequest, then it extracts the data of the reports and forward it to the report manager. (will be explained in detail to follow). The request consist into an id to identify it, and the complete form of the report.
- **fowardEvaluateRequest**: this method it receives the EvaulationRequest, extracts it and forward it to the "evaluateManager" (will be explained in detail to follow). The request consists in an unique id to identify it and an authority's id that allows the system to retrieve all the pending reports in the zone under the authority's control.
- **forwardHistoryrequest**: it will call all the methods of "historyManager" after obtaining information about the user who requested it (will be explained in detail to follow).
- **forwardMapRequest**: it receive the request to visualize the map, then it extract the data and forward the request to the "Mao Manager". The request consists in an unique id, and the user position.
- **forwardRecommendationRequest**: it receive the request to obtain suggestion about some street. The request is a series of streets name that will be sent to the manager to obtain the suggestion about them.
- **forwardUpdatedReportsRequest:** this method receives a request to update the status of some reports and forwards it to the correct manager. The request consists of a list of updated reports.

- **signUpManager**
  - **signUp**: this method will communicate in a safe way to the DB using an asymmetric key, it will memorize into the DB the Username and the hash of the password.

- **logInManager**
  - **logIn**: this method will communicate in a safe way to the DB using symmetric encryption (established at the begging of the communication using asymmetric encryption), it will obtain the password's hash from the DB and it will confront it to the hash of the one the user inserted.

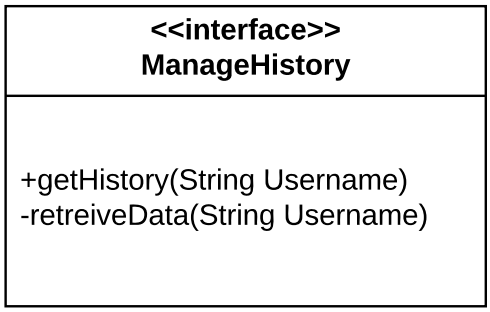
2.5.3 Report



- **reportManager**
  - **sendReport**: this method take the created reports and save it onto the DB adding the status (pending), and the username who created it.
  - **userPosition**: this method contact the position manager to retrieve the user position and send it to the client to include it in the report.
  - **sendPlateForOCR**: this method is called by the client, is used to translate the plate's photo into a String to include it into the DB and facilitate further operation of query
  - **updateDB**: this is the method that take care of every communication with the DB.
  - **sendNotification**: this method is used to notify the authorities of new reports.

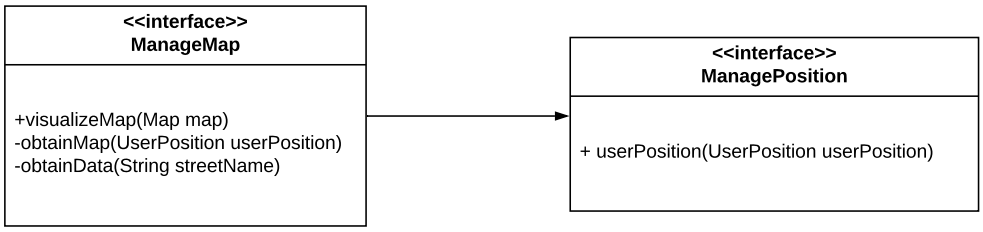
- **ocrManager**
  - **translate**: this is the method that using OCR techniques obtain the license plate from the photo.
- **notificationManager**
  - **sendNotification**: this method contacts the required users (both citizens and authorities) to notify them of new reports (or updated status of previous ones).
- **positionManager**
  - **userPosition**: this method has the task of obtaining information about the position of the user (for example the road on which it is located) from the geographical coordinates.

2.5.4 History



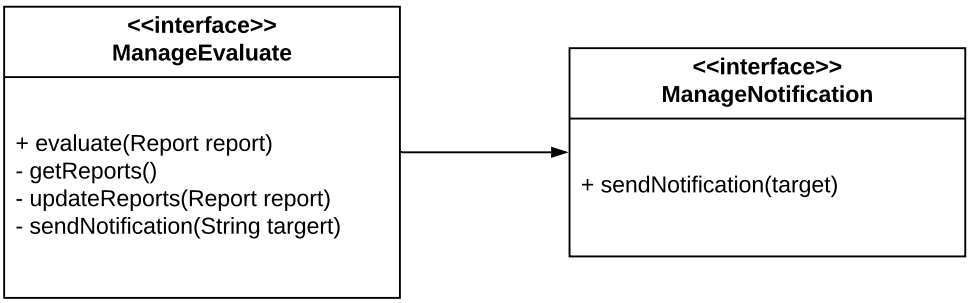
- **getHistory**: this method has to communicate with the application, receiving requests and sending back the data.
- **retreiveData**: this method is specific to contact and query the DB to obtain the history of user's reports.

2.5.5 Map



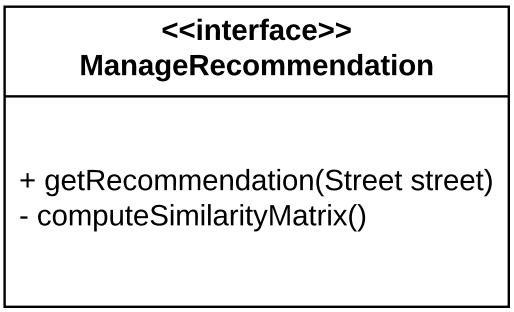
- **mapManager**
  - **visualizeMap**: the method retrieves the data about reports and the streets and then combine it assigning each street the right color.
  - **obtainMap**: it's called by visualizeMap and contact the gpsManager to retrieve the user position and then google Map to obtain the map around him.
  - **obtainData**: it's called by visualizeMap and this method has to contact the DB and it has to obtain the number of reports in each street visualized on the map in order to make able visualizeMap to coloring it correctly.
- **Manage Position**
  - **userPosition(Position position)**: as said above.

2.5.6 Evaluate



- **evaluateManager**
  - **evaluate**: communicate with the client sending the pending reports and receiving all updated reports.
  - **getReports**: this method is called by evaluate and obtain all pending reports in the area under the jurisdiction of the authority who made the request.
  - **updateDB**: this method is called by evaluate and updates the DB with the new status of the evaluated reports
  - **sendNotification**: alerts the user which report has been evaluated
- **notificationManager**
  - **sendNotification**: as said above

2.5.7 Suggestion



- **getRecommendation**: this method receive the request and, with the matrix already compiled it take the 3 most similar recommendation made by the system and forward them to the authority.
- **computeSimilarityMatrix**: this method is used each time some data are added, it compute the similarity matrix between the most frequent problem (each problem is divided in small aspect and each of them is an attribute) and a ICM where each item is a possible solution (the attributes are some aspect of the problems that each solution can resolve)

2.5.8 Class diagram

As in the RASD, the following UML diagram describes the organization of the model.

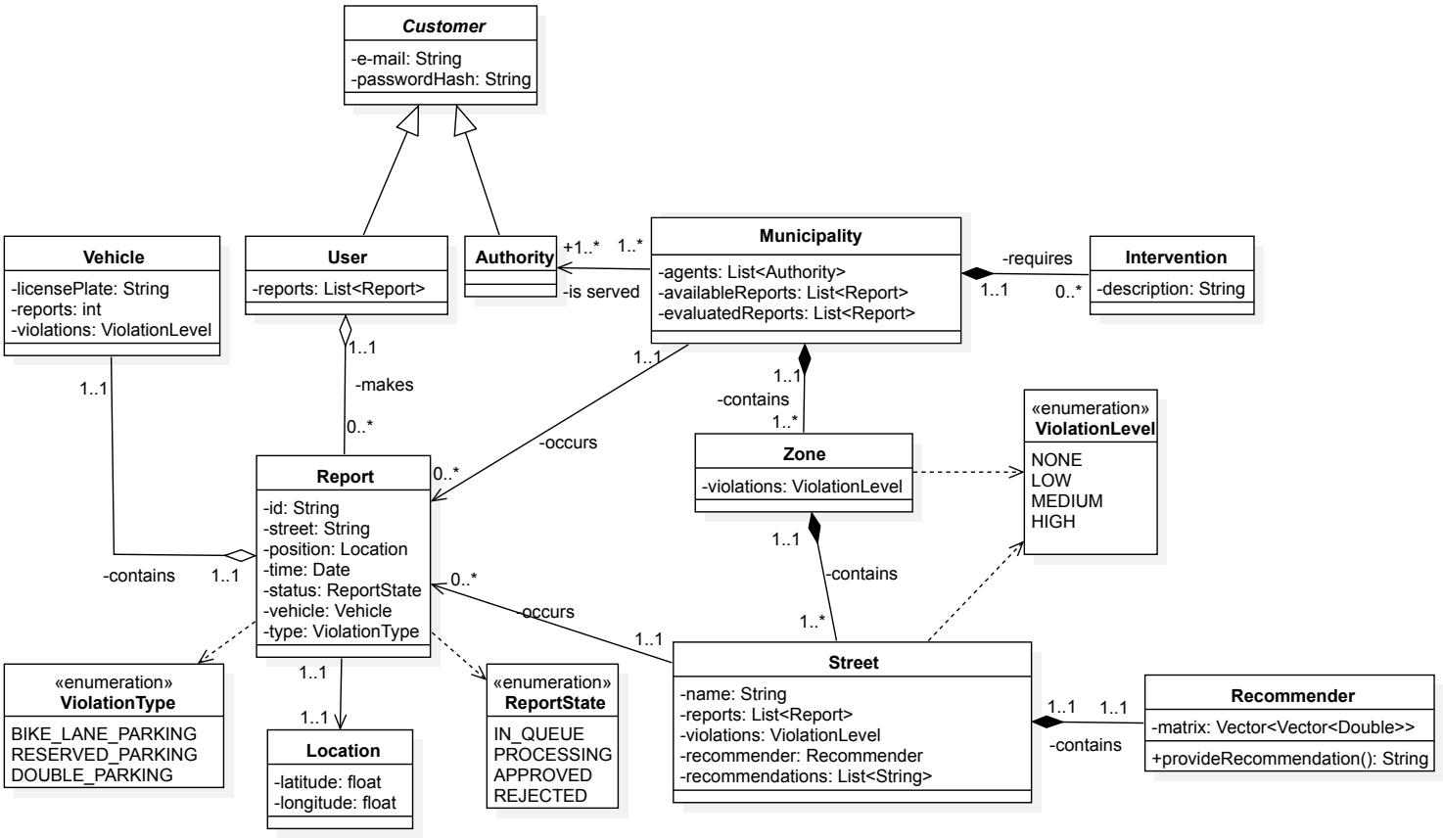


Figure 2.11: UML diagram of the model

2.6 Selected architectural styles and patterns

2.6.1 RESTful architecture

REST is an architectural style that matches the needs for developing this application: in particular, it positively affects some properties of the system, such as performance, scalability, modifiability of components, portability and reliability. To obtain this properties, the REST's architectural main constraints must be followed:

- Client-server: this principle is natural for the objectives of the application: the user interface must be separated (logically and physically) from the back-end of the application.

- Stateless: the client-server communication is constrained by no client context being stored on the server between requests. Each request from any client contains all the information necessary to accomplish the request, and the session state is held in the client. The client begins sending requests when it is ready to make the transition to a new state.

- Cacheable: clients responses from the server, that has to define them as cacheable or not to prevent clients storing the wrong information. Well-managed caching partially or completely eliminates some client-server interactions, further improving scalability and performance.

- Layered: the client doesn't know if it's talking with an intermediate or the actual server. So if a proxy or load balancer is placed between the client and server, it won't affect their communications.. Intermediary servers can improve system scalability by enabling load balancing and by providing shared caches. Also, security can be added as a layer on top of the web services, and then clearly separate business logic from security logic.

- Uniform interface: It simplifies and decouples the architecture, which enables each part to evolve independently. The four constraints for this uniform interface are:

  - Resource identification in requests: the resources themselves are conceptually separate from the representations that are returned to the client. For example, the server could send data from its database as HTML or XML, none of which are the server's internal representation.
  - Resource manipulation through representations: when a client holds a representation of a resource, including any metadata attached, it has enough information to modify or delete the resource.
  - Self-descriptive messages: each message includes enough information to describe how to process the message.
  - Hypermedia as the engine of application state: a REST client should be able to use server-provided links dynamically to discover all the available actions and resources it needs. As access proceeds,

the server responds with text that includes hyperlinks to other actions that are currently available. There is no need for the client to be hard-coded with information regarding the structure or dynamics of the application.

2.6.2 Three-tier architecture

The user interface (presentation tier), functional process logic (application tier) and data access tiers are developed and maintained as independent modules on separate platforms. Apart from the usual advantages of modular software with well-defined interfaces, the three-tier architecture is intended to allow any of the three tiers to be upgraded or replaced independently in response to changes in requirements or technology. For example, a change of operating system in the presentation tier would only affect the user interface code. The other main benefits of the three-tier architecture have been already explained in chapter 2.

2.6.3 Model-view-controller pattern

To split the internal representation of information from how it is presented to the user, this pattern divides the application into multiple layers of functionalities:

- Model: it is the application's dynamic data structure, independent of the user interface. It directly manages the data, logic and rules of the application;
- View: represents the visualization of the data the model contains;
- Controller: controls the data flow into model objects and updates the view whenever data changes. It keeps view and model separate.

The main benefits of this pattern are cohesion, low coupling, ease of modification and simultaneous development of the application.

2.7 Other design decisions

2.7.1 Client

A thin client is characterized by the fact that it is primarily designed to communicate with a server: its features are produced by servers. This design

was chosen to maintain the line of thought underlined till now, and also to make easier the usage of a MVC design pattern.

Indeed, this allows to have an architecture in which the real business logic is implemented on the server. The centralization of data management methods allows to have a completely thin client, because in case the device is offline, the app cannot be used. This choice involves a lighter executive application on the users' mobile devices giving them a better experience, furthermore thin clients are strictly dependent on a network connection, and in our case this is not an issue, since the application was conceived to operate most of the time online.

2.7.2 DBMS

Relational databases are a good choice when there is the need to deal with several transactions and when the data are linked by some relationships (users, reports, authorities etc.). Since there is no need for data analysis with different granularity, it was decided not to use non-relational databases, so as to maintain high performance with a reasonable saving of memory.

Chapter 3

User interfaces design

In chapter 3 of the RASD some screenshots from the user interface were shown. This chapter describes the navigation in the user interface, both for the users and the authorities. The following UX diagrams refer to the mock-ups shown in the RASD, which are only the most relevant of the following.

3.1 User

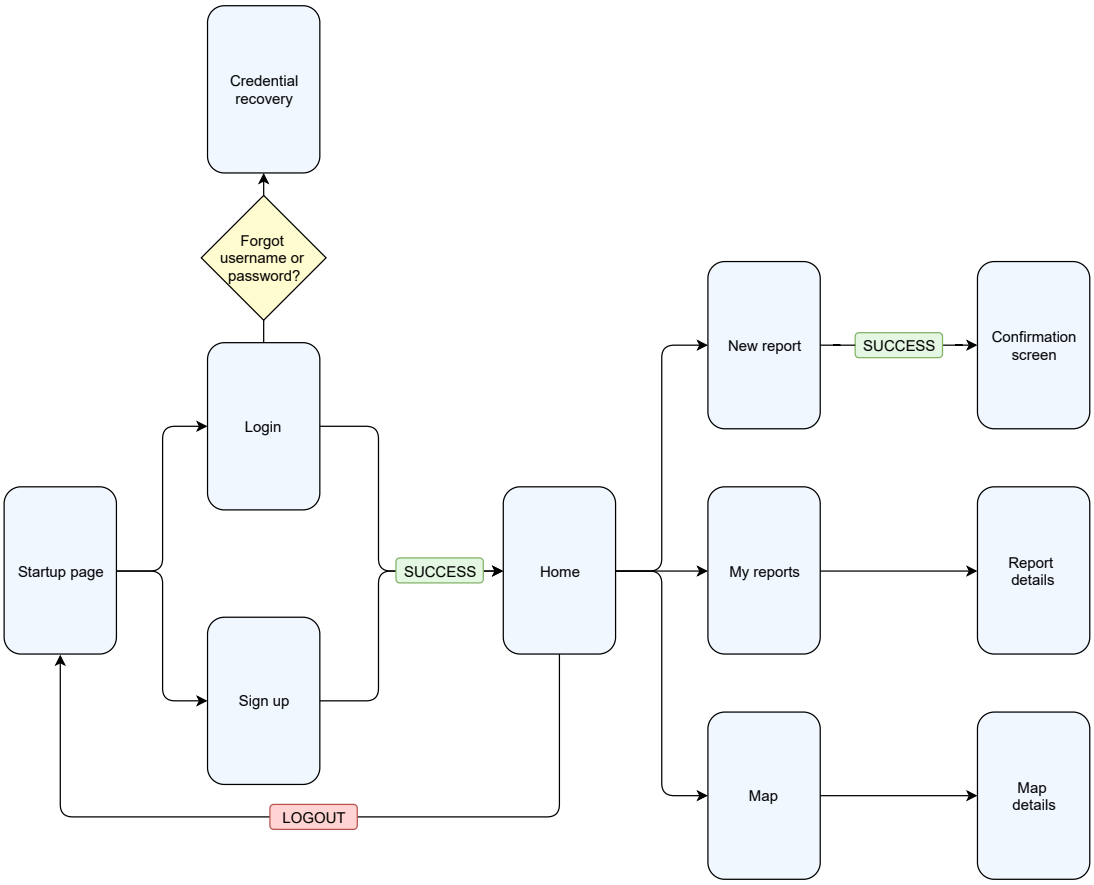


Figure 3.1: UX diagram - user

3.2 Authority

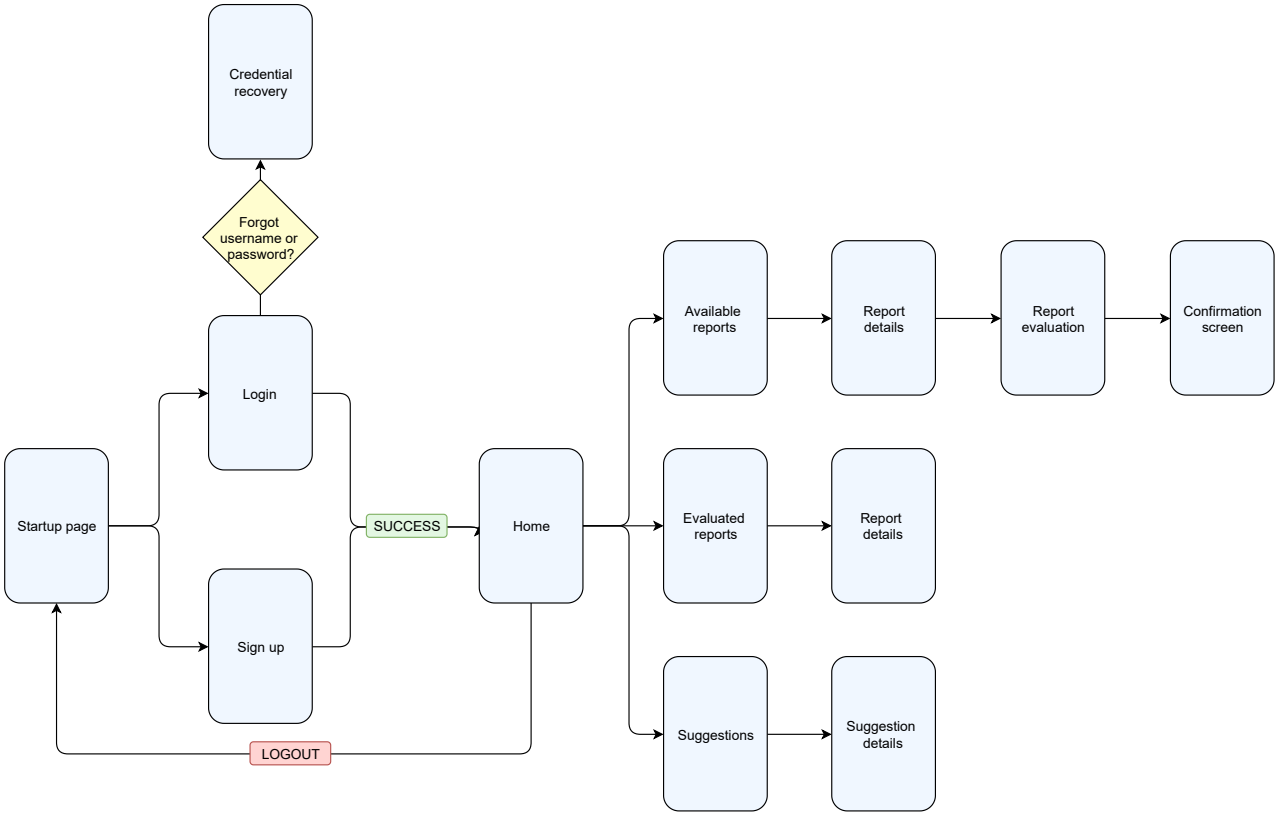


Figure 3.2: UX diagram - authority

Chapter 4

Requirements traceability

Chapter 3 of the RASD presented the functional requirements necessary to guarantee the goals of the application: this chapter shows how every component of the system contributes to satisfy them. For each of the requirements, all the involved components in the application server are listed and their function is explained.

- **R1**: Registration must be allowed only entering a valid e-mail address that is not already associated to an existing SafeStreets account.
  - **SignUpManager**: Checks if the e-mail address is already in the DB. In case of sign up, if the address already exists, the request is refused. In case of login, checks if the password hashes correspond.
- **R2**: Registration must be allowed only entering a password that satisfies the safety conditions.

  This check is made by the client, since the application server can never receive a password as plaintext. Moreover, it is impossible to determine whether a string satisfies the safety conditions by reading its hash.
- **R3**: The system must store the hash of every password, using a safe cryptographic hash function.

  As said above, the encryption must be made by the mobile application, which computes the hash of the password using a safe cryptographic hash function, like one of the SHA-3 family. For what concerns the application server components:
  - **SignUpManager**: Forwards the received hash (computed by the mobile app) to the DB.

- **R4**: The system must check if the location of the report belongs to some municipality exploiting SafeStreets.
  - **ReportManager**: Retrieves the position from the report
  - **PositionManager**: Retrieves the street given the position, exploiting Google Maps APIs
  - **ReportManager**: Checks in the DB if the street belongs to some registered municipality.
- **R5**: The system must add the right date, time and street of the violation to the data provided by the user.

  All this information is added by the mobile app, which exploits Google Maps APIs as well. Date and time are provided by the OS of the device.
- **R6**: The system must correctly read the license plate given a picture attached to a report.
  - **ReportManager**: Retrieves the picture attached to the report
  - **OCRManager**: Given the picture of the violation, recognizes the license plate by running an OCS algorithm. This is made by the server to respect the thin client architecture.
- **R7**: The system must exploit external resources to show maps to the users.

  This is another requirement fulfilled by the mobile application, which will display the map thanks to Google Maps APIs. The streets are shown with different colors, as will be explained in R8.
- **R8**: The system must show the right colors on the map, given the number of approved reports for each street.
  - **MapsManager**: Given a set of streets, performs queries in the DB, in order to retrieve the number of violation for each of the streets. Then returns the color to highlight the street with, according to rules that associate no color to the streets with less violations and red to the ones with the most violations.
- **R9**: When a license plate is recognized, the system must show the number of approved reports for that car.
  - **ReportManager**: Once the string is obtained from the OCRManager, checks if in the DB are present some violations committed by the same vehicle.

- **R10**: If the same violation is reported twice, it counts only one time on the map.
  - **ReportManager**: Checks if in the DB is already present an identical violation, reported in the last 24 hours.
- **R11**: The system must assign to each report a unique identification number.
  - **ReportManager**: Assigns the unique identification number to each report.
- **R12**: Registration must be allowed only entering a valid identification number that is not already associated to an existing SafeStreets account.
  - **SignUpManager**: Checks if in the DB the same identification number is already present.
- **R13**: The system must guarantee that each municipality is covered by at least one authority.

  This check is done at data access level, implementing in DB a rule that requires at least one assigned authority for each municipality.
- **R14**: The system must notify the authorities when a new report is available.
  - **ReportManager**: After successfully inserting a new report in the DB, calls the NotificationManager.
  - **NotificationManager**: Alerts all the agents assigned to the municipality of the new report.
- **R15**: The system must suggest a possible intervention to the authorities of a municipality, according on the number of each type of violation.
  - **RecommendationManager**: Checks every day if the recommender system has some suggestions.
  - **NotificationManager**: Alerts all the agents assigned to the municipality for which a new intervention is suggested.
- **R16**: Login must be allowed only if both e-mail address and password are correct.
  - **LogInManager**: Checks if the e-mail address is already in the DB, then if the password hash matches.

Chapter 5

Implementation, integration and test plan

5.1 Implementation and test plan

To facilitate development, the system has been divided into subsystems, each of which corresponds to one of the above mentioned managers. They will be developed individually with a bottom-up approach and the order will be imposed by their importance to the system (how many sub-systems are depending from it) and for the customers. After the development of each subsystem, they will be tested to check their correctness and then some integration tests will be executed. The system will use also some external systems, in particular Google Maps to have maps always updated and a DBMS to store all the data.

Subsystem	Importance for customers	Importance for development
Sign-up & Login manager	Low	High
Report manager	High	High
Maps Manager	High	Low
History	High	Low
Evaluate	High	High
Suggestion	Medium	Medium
Notification Manager	Low	Low
Position Manager	Low	High

It is important to stress that the reports and evaluation functionalities have to be developed and tested as a first step, because all the others functionalities rely on these two (except the functionalities for the suggestion about improving security on the streets).

- **Sign up & login manager**: even if these two are not core functionalities of the system, they have a great importance in SafeStreets, because some functionality requires to keep trace in a unique way of who did what (for example history or suggestion); so this part has to be prioritized over the others (except "Evaluate manager" and "Report manager").

- **Report manager**: one of the core functionalities, it has to developed with maximum priority and tested as soon as possible, probably its more critic aspects are the OCR and the position. In order to be developed it needs a functioning "Position manager", otherwise it will not be possible to convert the GPS position into a street name to understand where the traffic violation has been committed. Even if the "Notification manager" is in theory a sub-component of this, its development is not so crucial for the correct functioning of the system, so it can be developed in a second moment.

- **Maps manager**: this component must be developed after the components necessary to ensure the creation and evaluation of reports ("Evaluate manager" and "Report manager'). It is not particularly important to choose the development order between the "Map manager" and "History manager", the important thing is that they are properly developed and integrated before developing the others. This allow the use of the "Visualize map" functionality, and seeing on the map the streets with most car accidents.

- **History manager**: this component must be developed after the components necessary to ensure the creation and evaluation of reports ("Evaluate manager" and "Report manager'). It is not particularly important to choose the development order between the "Map manager" and "History manager', the important thing is that they are properly developed and integrated before developing the other. This functionality allows the user to see the chronology of his reports.

- **Evaluate manager**: it is the second core functionality of the system. This functionality has to be able to correctly handle concurrency and various authorities that want to evaluate reports various reports at the

same time. After the correct development of both the core functionalities an integration test is necessary to check the correct functioning of the complete system.

- **Suggestion manager**: this is completely uncorrelated with the rest of the system, so it would be better to develop it as last thing, in this way the focus is on the core functionality of the system and the correct integration with the rest of the functionalities.
- **Position manager**: this component allows to get the road or location on the map of the user, it has the maximum priority, indeed it allows the system to understand where the report has been made.
- **Notification manager**: this component sends notifications to the users, it can be developed at last, on the top of the complete system because it is not a crucial aspect of the core functionalities.

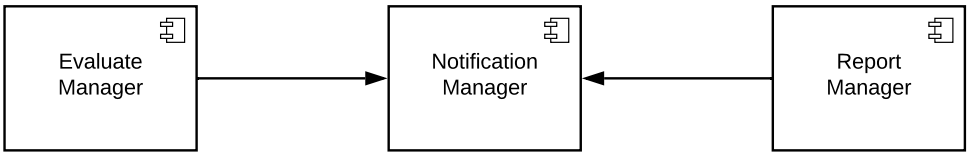
5.2 Integration

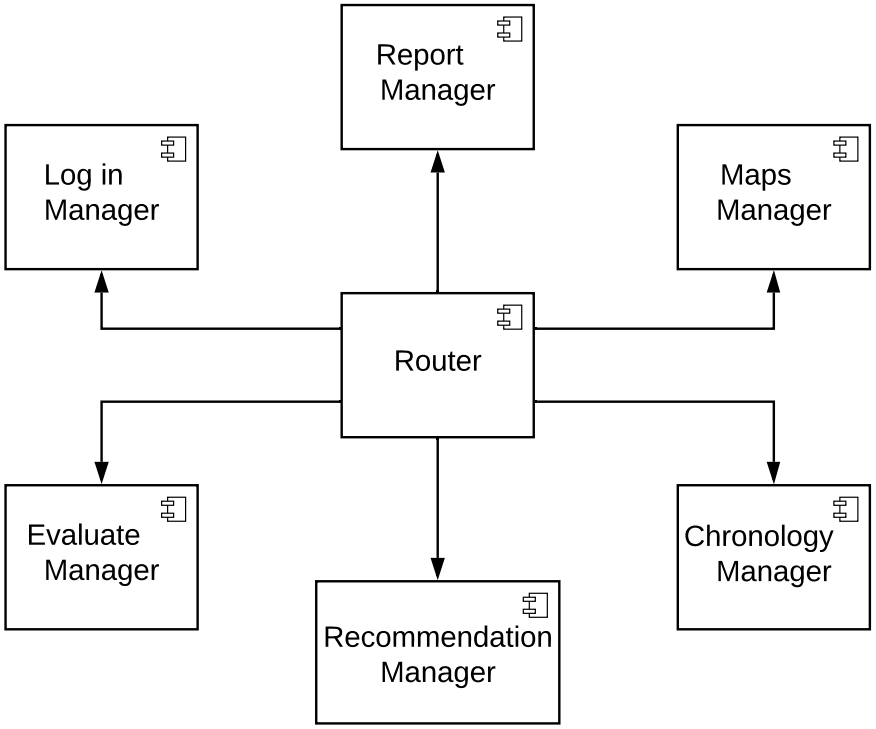
The following diagrams show which components will go through the process of integration for a further clarification. The arrows starts from the component which 'uses' the other one.

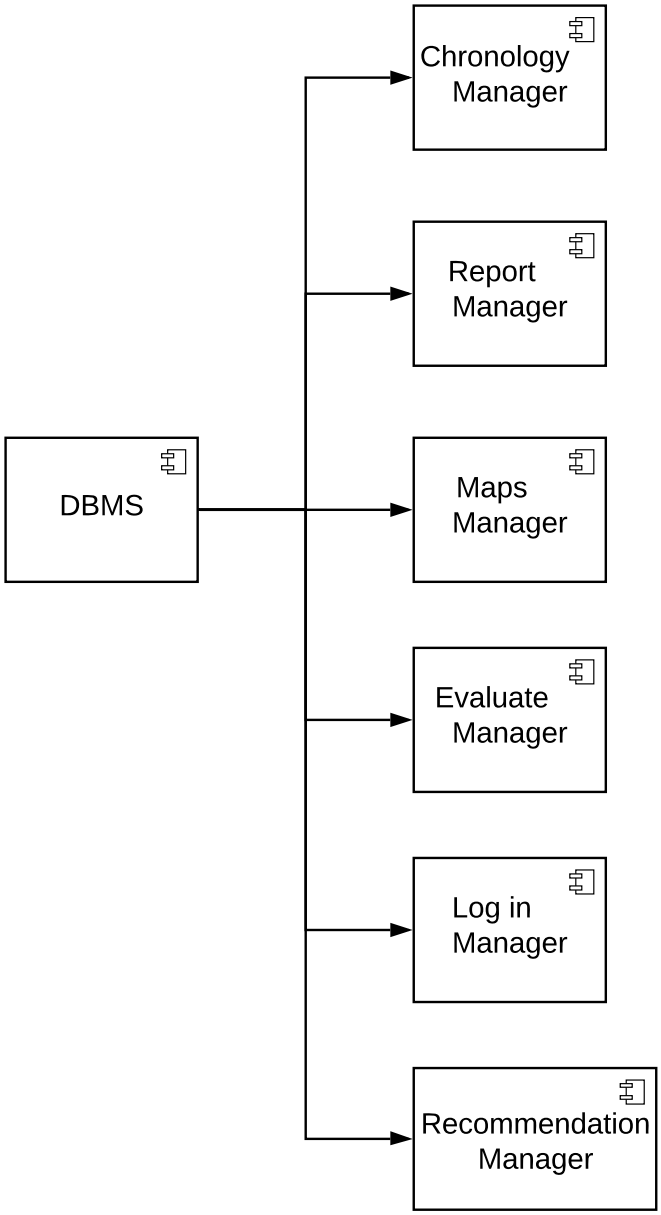
5.2.1 Internal component

All the components are implemented and unit tested. Subsequently some components are integrated and the integration is tested as well.


OCR
Manager
Report
Manager
Position
Manager


Evaluate
Manager
Notification
Manager
Report
Manager


Report
Manager
Log in
Manager
Maps
Manager
Router
Evaluate
Manager
Recommendation
Manager
Chronology
Manager


Chronology Manager
Report Manager
DBMS
Maps Manager
Evaluate Manager
Log in Manager
Recommendation Manager

Chapter 6

Effort spent

Chapter	Frangi (hours)	Fucci (hours)
Chapter 1	1	0.5
Chapter 2	13.5	17
Chapter 3	1	2
Chapter 4	2	4
Chapter 5	8	1.5
Total hours:	25.5	25

Chapter 7

References

7.1 Bibliography

- Course slides from Software Engineering 2 - *Professor Elisabetta Di Nitto*;
- SafeStreets Requirement Analysis and Specification Document - *Alberto Frangi, Tiziano Fucci*;
- 1016-1987, IEEE Recommended Practice for Software Design Descriptions - *IEEE*.

7.2 Tools

- StarUML 3.1.0 - to draw and export UML diagrams;
- `lucidchart.com` - to draw, share and export sequence diagrams;
- `draw.io` - to draw and export other diagrams;
- TeX Live 2019 - to write and organize this document;
- GitHub 2.23.0 - version control.