Politecnico di Milano

Computer Science and Engineering

Software Engineering 2

SafeStreets

Design Document

Authors
Simone Braga
Juan Calderon
Marzia Favaro

Professor
Elisabetta Di Nitto

December 9, 2019
Version 1.0

Contents

1 Introduction

1.1 Purpose

SafeStreets is a crowd-sourced application that intends to provide users with the possibility to notify authorities when traffic violations occur. The main target of the application are violations that can be easily captured by a camera (like, for instance, parking violations). SafeStreets intends also to provide users with the possibility to mine the stored information with different levels of visibility. Moreover, the application must cross the collected data with information coming from the municipality to provide suggestions on possible interventions to decrease the incidence of violations and accidents. In the end, the application must forward data about violations to generate traffic tickets, and must allow authorities to get statistics on issued tickets.

1.2 Scope

The services of SafeStreets must be developed through a mobile application. Everyone can use the mobile application, whose behavior is different depending on the user's privileges. Excluding the secondary actors (that interact directly with the back-end of the application with the purpose to exploit the front-end services), 3 types of user can be identified:

- Common user
- Authority
- Municipality user

SafeStreets must provide common users with the possibility to notify a traffic violation by taking a picture. Then the data about the violation is sent to the application server that checks its integrity before storing it in a database. Common users can access the data stored in the database with some restrictions. SafeStreets provides them with the possibility to select some filters for querying the data, and with a proper graphic interface to visualize the queried data.

Authorities are provided with the possibility to query without restrictions the data stored by SafeStreets. The application server must use a service responsible for the generation of traffic tickets. The results of the ticket generations are stored by SafeStreets and analyzed with the purpose to provide in-depth statistics. To do so, SafeStreets relies on a data warehousing system that gets the information from the primary database and analyzes it with data mining techniques. Authorities must be provided with the proper interfaces for this particular type of query.

Municipality users must be provided with the possibility to get suggestions on possible interventions to reduce the incidence of accidents and violations. To do so, SafeStreets relies on the data warehousing system described so far, that must apply data analysis techniques to identify possible solutions. SafeStreets can access information about accidents provided by the municipality to identify more precise interventions. Data from the municipality must be periodically checked for updates.

1.3 Definitions and acronyms

- **User**
  The consumer of the application. It includes common users, authorities and municipality users.
- **Common user**
  The user type that everyone can sign up as. It does not require any kind of verification.
- **Authority**
  The user type that authorities can get. It requires the verification of an activation code.
- **Municipality user**
  The user type that municipal employees can get. It requires the verification of an activation code.
- **Municipality Tickets Service (MTS)**
  Service offered by the municipality to generate traffic tickets from information about the violations.
- **Optical Character Recognition (OCR)**
  Software that converts text scanned from a photo in a machine-encoded text.
- **Query interface**
  The interface provided to the users to select some filters when requesting data.
- **Application Programming Interface (API)**
  An interface or communication protocol between client and server intended to simplify the building of client-side software.
- **Extract Transform Load (ETL)**
  The general procedure of copying data from one or more sources into a destination system which represents the data differently from the sources or in a different context than the sources.
- **User Experience (UX)**
  The person's emotions and attitudes about using a particular product, system or service.
- **Online Transaction Processing (OLTP)**
  The approach used to handle a large number of transactions.
- **Online Analytical Processing (OLTP)**
  The approach used to answer multi-dimensional analytical queries.

1.4 Revision history

Version	Release date	Description
1.0	December 9, 2019	First release

1.5 Document Structure

Section 1 contains a summary of the features of the application and a description of the purpose of this document. The description is slightly different from the one provided in the RASD, as it captures a more practical point of view of the application. It also contains all the necessary elements for the comprehension of this document.

Section 2 is the core of the document. It contains all the most important notions for the developing process of the application. It includes the description of the design architecture and the analysis of the chosen design patterns. Every subsection offers a different point of view of the application: in this sense, it can be considered as a "framework" where it is possible to access information with different levels of granularity.

Section 3 contains a large set of mockups that capture all the important aspects of the communication between the user and the application. Moreover, this section contains a formal description of the UX flow, which was not included in the previous section because it is not strictly part of the system architecture.

Section 4 contains a formal description of the mapping between the requirements and the components responsible for their exploitation.

Section 5 contains a description of how the project should be developed. It is a useful reference for the optimization of the developing process as it contains a forecast on how much the developing of every component is onerous. It also contains a plan for the testing and integration process.

Section 6 includes information about the number of hours each group member has worked for this document.

Section 7 includes the references to the tools used to draw up this document.

2 Architectural design

2.1 Overview

The purpose of this section is to provide an overview of the system architecture, on which all the assumptions later on in the document will be based. In particular, the reference system architecture is represented in figure 1.

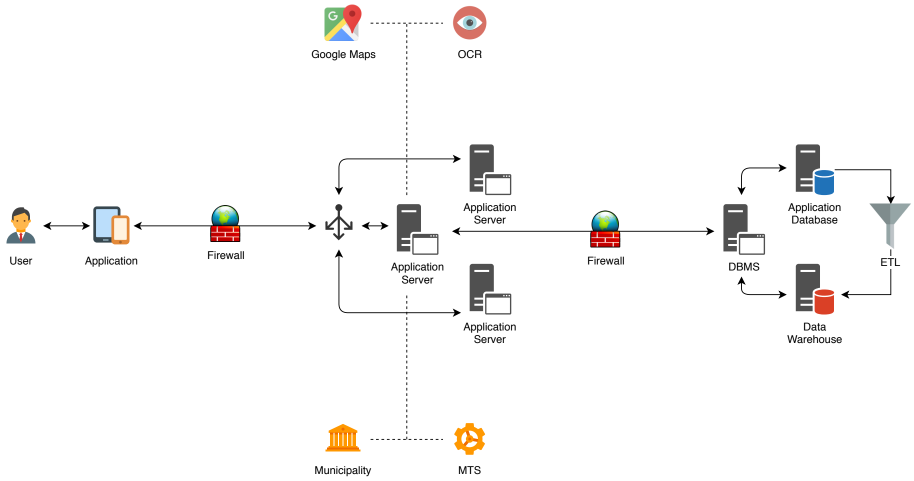


Figure 1: System architecture

The system is built over a 3-tier architecture, where the distinction between the three layers (presentation, application, data) is evident. Every layer corresponds to a node that can be developed independently from the others. This involves scalability and security of the system since every node can be upgraded without affecting the overall functionality. Moreover, keeping only one channel for the communication between the user and the application server involves a more compact architecture. Also, the user-application channel is the only one where a firewall is placed, to protect the application server from incoming dangers

Since the application database is centralized, it is possible to replicate the application server to achieve better performance. To fully take advantage of the server replication, a load balancer is placed between the user and the application server. It distributes the workload over the replicated nodes

The data node consists of the combination of a relational database and a data warehouse. The database must process a high number of transactions (OLTP policy). The data warehouse must handle complex queries and data mining over a large amount of data (OLAP policy). The information flow from the database to the data warehouse follows an ETL process. The physical implementation of the database and the data warehouse is not studied in deep since it is not the focus of the project.

2.2 Component view

Every node of the system consists of some components. The diagram in figure 2 aims to provide an overview of all the components of the world in which the application must work. Then there is a projection over every component that belongs to the application server, where the internal architecture and the information flow are described. Every functionality of the application is strictly independent of the others; for this reason, every one of them is fully exploited by a single (aggregated) component which is represented with the same name of the functionality.

Components inside the database server are not described since their physical implementation is not the core of this specification document. Components outside the application server are not described since they cannot be modeled by the developers. Components that interface with these external modules must agree to the needed communication standards.

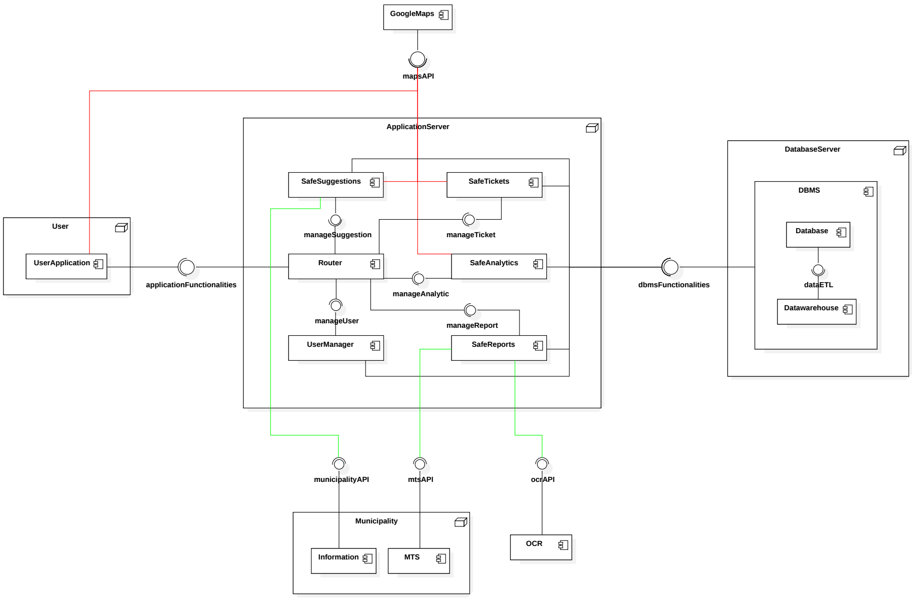


Figure 2: Component diagram

- **Router**
  Its purpose is to handle all the communications between the user and the application server. Using a specialized device to do this, minimizes the number of links between the client and the server. This involves a good level of scalability. The router manages all the messages, forwarding them to the responsible components. In particular, the router can recognize a user. This allows it to start the correct handling procedure on the correct component.

- **UserManager**
  It is responsible for the user-related operation. It allows the user to login and to register, and eventually verifies the activation code during the signup procedure. It is not described in detail since it is not the core of the application.

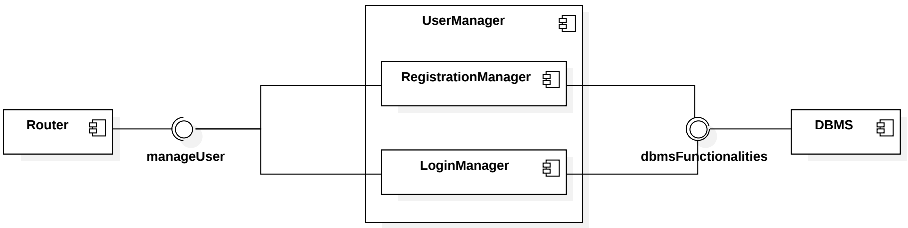


Figure 3: UserManager component

- **RegistrationManager** is responsible for the signup procedure.
- **LoginManager** is responsible for the login procedure.

- **SafeReports**
  It provides services to verify and store violation reports. It is split into several sub-components to avoid a centralized managing of communication with external tools. This involves better scalability since every component can be updated independently from the others.

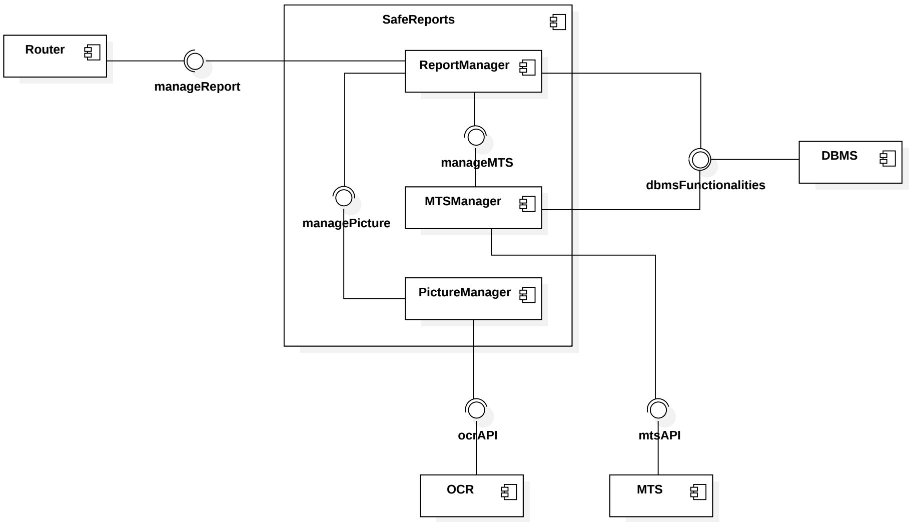


Figure 4: SafeReports component

- **ReportManager** performs all the necessary integrity checks to ensure that the chain of custody has never been broken. When it receives a report, it forwards the image to **PictureManager** and waits for a response. If the picture is not good, it notifies the user that the procedure must be repeated. If the picture is good, it waits for a confirmation from the user. If the user confirms, the component asks the DBMS to store the report. The DBMS returns a value that determines if the report was not duplicated. If it wasn't, the component forwards the report to **MTSManager**.
- **MTSManager**
  It converts the reports in a suitable format for MTS and forwards

them to the service. If a traffic ticket is generated, the ticket-related information answered by MTS is sent to the DBMS to be stored.

  - **PictureManager**
    It manages the communication with the OCR software. It forwards the picture to the OCR and interprets the answer. Once it is done, it responds to **ReportManager**.

- **SafeAnalytics**
  It is just an interpreter between the application and the DBMS. It consists of a single component that directly interfaces with the external modules.

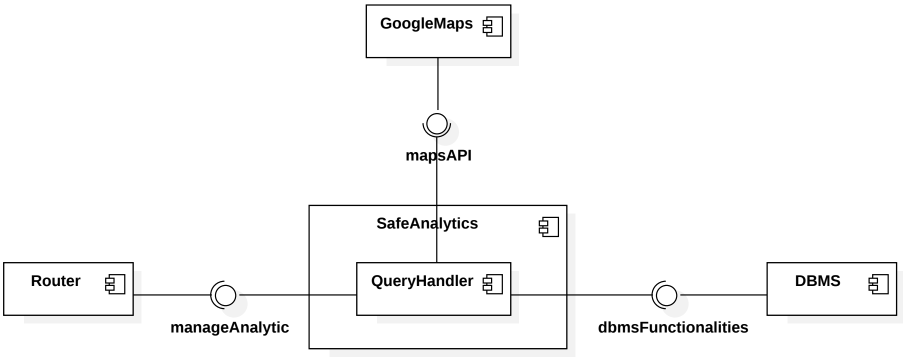


Figure 5: SafeAnalytics component

  - **QueryHandler** receives the queries from **Router**. Once a query is received, the component checks that the selected filters are coherent with the access right of the user who made the query, since the component has unrestricted access to the DBMS. What happens is that **Router** calls a user-specific function in the interface provided by the query handler (the identification of the user is done by **Router**). Once the query is answered, the data are sent back to the user.

- **SafeTickets**
  Its behavior is the same as **SafeAnalytics**. The only difference is the format of the data the component works with.

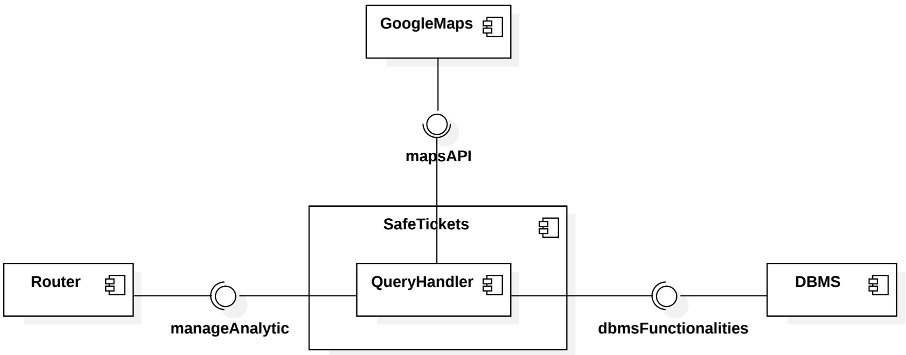


Figure 6: SafeTickets component

- **QueryHandler** has the same behavior as the sub-component of **SafeAnalytics**. The only difference is that it doesn't need to check the validity of the filters since it offers his services only to authorities.

- **SafeSuggestions**
It has the same purpose of **SafeAnalytics** and **SafeTickets**. Moreover, it manages the data extraction from the municipality information, periodically activating the update check. Then it forwards the data to the DBMS.

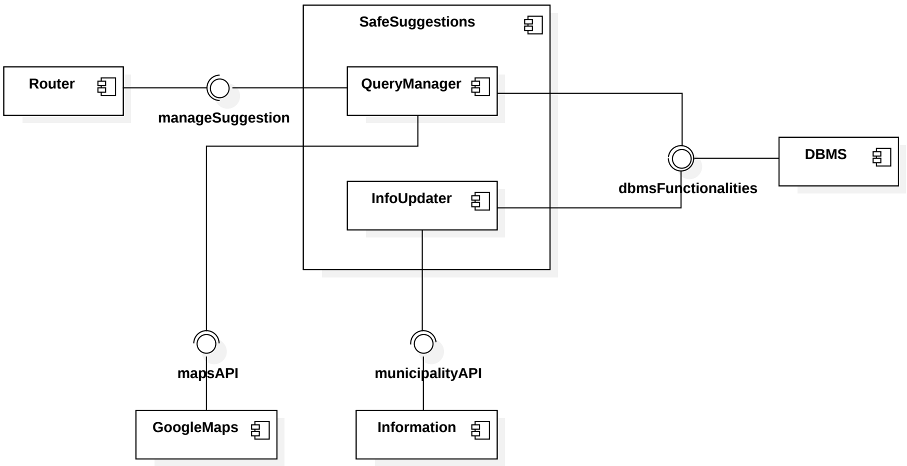


Figure 7: SafeSuggestions component

- **QueryManager** works exactly as **QueryHandler**. Refer to that description.
- **InfoUpdater** is a sub-component that works independently from the others. Its purpose is to periodically check for the existence of new information from the municipality. If there is new information, it gets it and forwards it to the DBMS. The analysis of the extracted information is up to the database system.

2.3 Deployment view

The composition of the system that exploits SafeStreets has already been discussed in the previous sections. This section will focus on the distribution of the functionalities over the physical nodes. The system is based on a 3-tier architecture, that involves a clear distinction between the three layers of the application. It is possible to see the application of this architecture yet in the component diagram. The diagram in figure 8 shows the clear distinction between the three layers of the application, which are the presentation layer (left-most tier), the application layer (central tier) and the data layer (right-most tier).

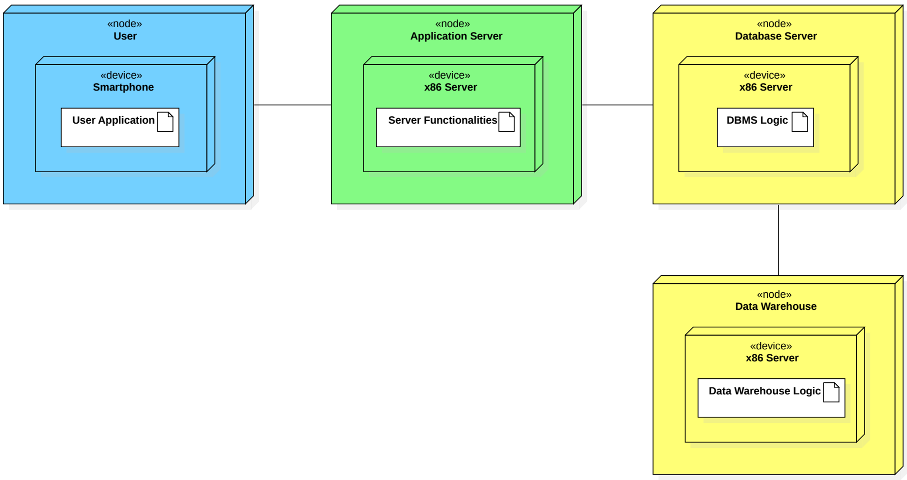


Figure 8: Deployment diagram

The presentation tier develops the user application, which only responsibility is to present a user interface through which the user can communicate with the application server. The application tier develops the application server, that exploits all the application logic but does not contain any data, that, instead, are stored in the database. The database is developed in the data tier, whose purpose is also to split on different nodes the application database and the data warehouse for data analysis and complex queries.

2.4 Runtime view

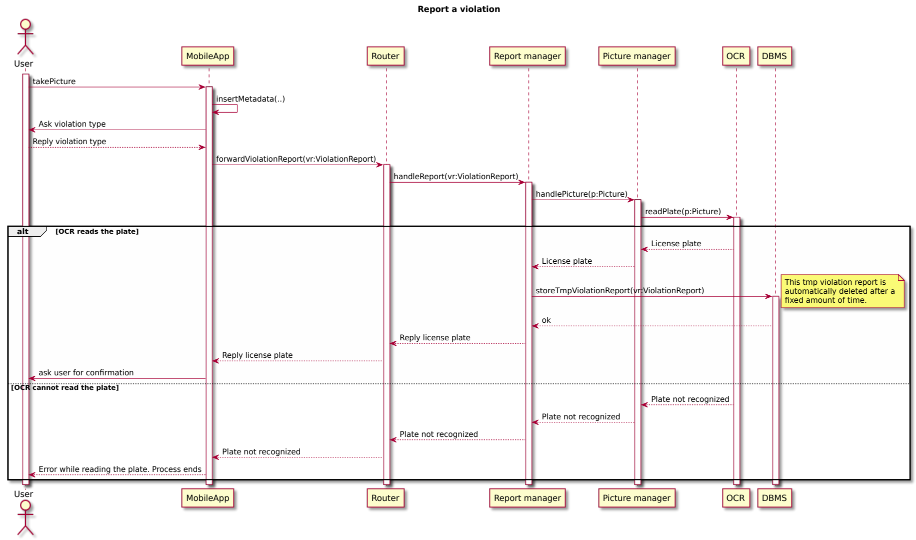


Figure 9: First part of SafeReport

The user takes a picture using the mobile app. The needed metadata (GPS coordinates, timestamp) are then automatically captured by the app. The user needs to select the type of violation from a predefined list. After selecting the violation type the violation report is generated and forwarded to the router. The router must determine the right component for the new violation report. The report is sent to the report manager a SafeReports subcomponent that can handle all the new requests for violation reports. The report manager wants to retrieve the license plate of the violation and to achieve that goal, it sends the picture contained in the violation report to the picture manager. The picture manager forwards the picture to the external OCR software. If the OCR is not able to read the license plate then an error message is sent in a cascade mode back to the user, who is asked to repeat the entire process. The process ends.
Otherwise, the license plate is returned to the picture manager and then to the report manager. Once the report manager has the license plate it sends the completed report to the DBMS. The violation report is temporarily stored in the database. After that, the DBMS sends an acknowledge to the report manager. Then the report manager sends the license plate to the router and from it to the app.
It is important to notice that the violation reports temporarily stored are deleted after a small fixed amount of time. When a violation report temporarily stored is automatically deleted it means that the user never confirmed the license plate of that violation report. This situation is better explained in the second part of the SafeReports sequence diagram.

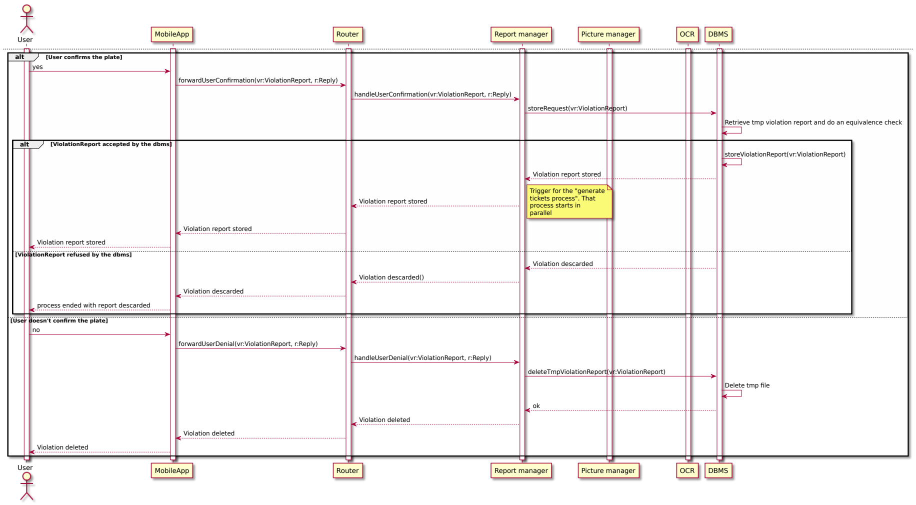

User
MobileApp
Router
Report manager
Picture manager
OCR
DBMS
alt [User confirms the plate]
yes
forwardUserConfirmation(vr:ViolationReport, r:Reply)
handleUserConfirmation(vr:ViolationReport, r:Reply)
storeRequest(vr:ViolationReport)
Retrieve tmp violation report and do an equivalence check
alt [ViolationReport accepted by the dbms]
storeViolationReport(vr:ViolationReport)
Violation report stored
Trigger for the "generate tickets process". That process starts in parallel
Violation report stored
Violation report stored
Violation report stored
[ViolationReport refused by the dbms]
Violation descarded
Violation descarded()
Violation descarded
process ended with report descarded
[User doesn't confirm the plate]
no
forwardUserDenial(vr:ViolationReport, r:Reply)
handleUserDenial(vr:ViolationReport, r:Reply)
deleteTmpViolationReport(vr:ViolationReport)
Delete tmp file
ok
Violation deleted
Violation deleted
Violation deleted
User
MobileApp
Router
Report manager
Picture manager
OCR
DBMS

Figure 10: Second part of SafeReports

The user needs to confirm or refuse the license plate. If the user doesn't confirm the plate then a *user denial* is sent to the report manager passing through the router. The report manager sends a request to the DBMS to delete the temporary violation report if any. The DBMS sends the acknowledge of the deletion to the report manager. The report manager sends the reply to the router and from it to the app. The process ends.

Otherwise, a *user confirmation* is sent to the router and then to the report manager. The report manager wants to store the new violation report and to do that it forwards the report to the DBMS. The DBMS tries to retrieve the corresponding violation report from the temporary data. After retrieving the temporary report the DBMS checks if there is no other equivalent violation report already permanently stored in the database. If something goes wrong during the retrieval or if the DBMS finds an equivalent report already permanently stored into the database then it doesn't update the database and it sends an error to the report manager. An error is sent to the user passing through the router and the mobile app. The process ends.

Otherwise, the DBMS is able to permanently store the new violation report in the database and then a positive reply is sent to the report manager. The report manager sends a positive reply to the router and then to the mobile app. The user is notified about the successful operation. The process ends.

We tried to emphasize the differences between storing violation reports temporarily or permanently. Temporary reports are reports that are waiting for a confirmation from the user. Permanent reports are reports that have already the user confirmation and are stored in the database forever.

We need to notice that every time the report manager is notified that a new violation report is permanently stored in the DBMS then the *generate tickets* process starts in parallel.

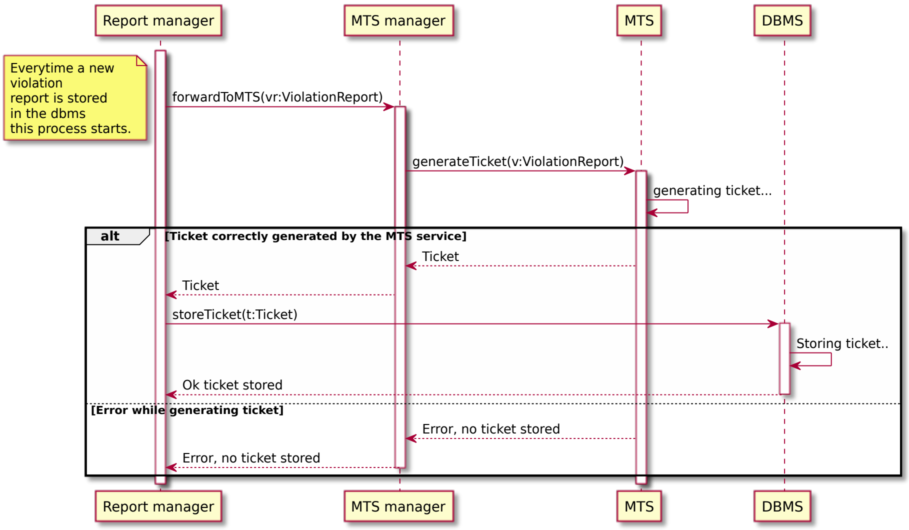

Report manager
MTS manager
MTS
DBMS
Everytime a new violation report is stored in the dbms this process starts.
forwardToMTS(vr:ViolationReport)
generateTicket(v:ViolationReport)
generating ticket...
alt
[Ticket correctly generated by the MTS service]
Ticket
Ticket
storeTicket(t:Ticket)
Storing ticket..
Ok ticket stored
[Error while generating ticket]
Error, no ticket stored
Error, no ticket stored
Report manager
MTS manager
MTS
DBMS

Figure 11: Generate tickets from reports

Every time a new violation report is permanently stored in the database this process starts. The report manager sends the stored violation report to the MTS manager. This process is done in parallel with the SafeReports process. Since MTS manager wants to generate a ticket it forwards the violation report to the MTS. If the MTS is not able to generate the ticket then it forwards an error to the MTS manager. The MTS manager propagates the error to the report manager and the process ends.

Otherwise, the generated ticket is sent to the MTS manager and then to the report manager. It forwards the new ticket to the DBMS to store the ticket. After the ticket is stored in the database the DBMS sends a confirmation to the report manager and the process ends.

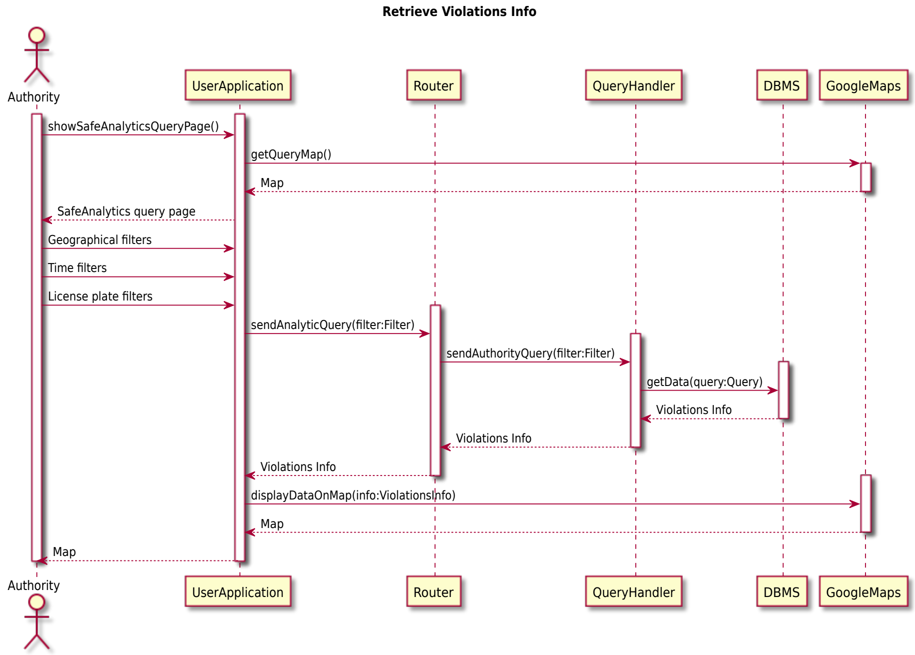


Figure 12: Get information about reported violations

Here is shown how authorities get access to the information about the reported violations. This diagram is very similar to the one for the same functionality for the common user, so instead of representing it twice, the differences will be highlighted in this description.
The user is shown the SafeAnalytics main page, which consists in a query interface, changing according to the type of user. While common users are asked to insert position and time ranges, authorities can also ask for specific license plates. The selection of the position filter requires to display a map, so Google Maps is involved.
All the query data provided are stored in a Filter object and the query is passed to the DBMS, to retrieve the required data from the DB. This does not happen in a single step, as the request must pass through the Router and the QueryHandler. The response is sent back to UserApplication, which displays it on a map thanks again to Gooogle Maps.

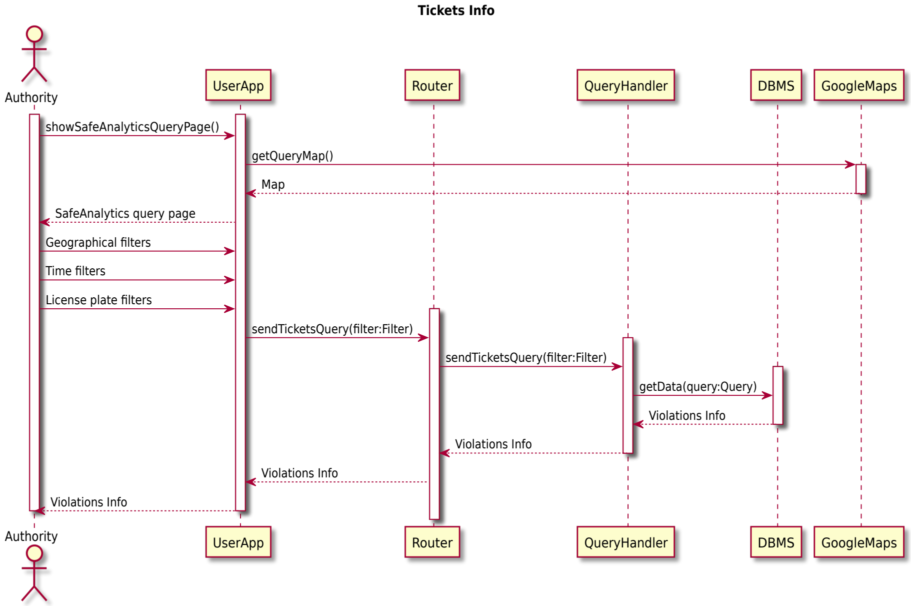


Figure 13: Get information about generated tickets

Authorities get an interface similar to the one for getting information about the violations to get the tickets data. In the same way as for the other authority queries, a request for the information is forwarded to the DBMS. The statistics are not shown in a map (so Google Maps is not involved anymore), and the data flows back on the same path of the request to the user.

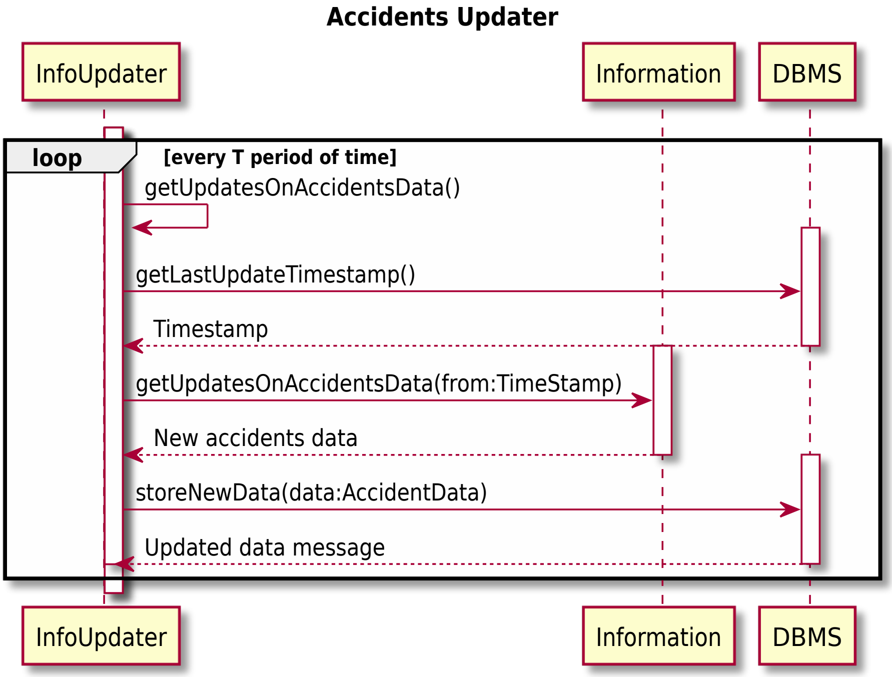


Figure 14: Update data from municipality

The system must periodically collect the accidents' data provided by the municipality. To do so, InfoUpdater contains a trigger to ask for new data. In order to not download the whole dataset provided by the municipality every time, a time stamp is saved to get the latest data only.

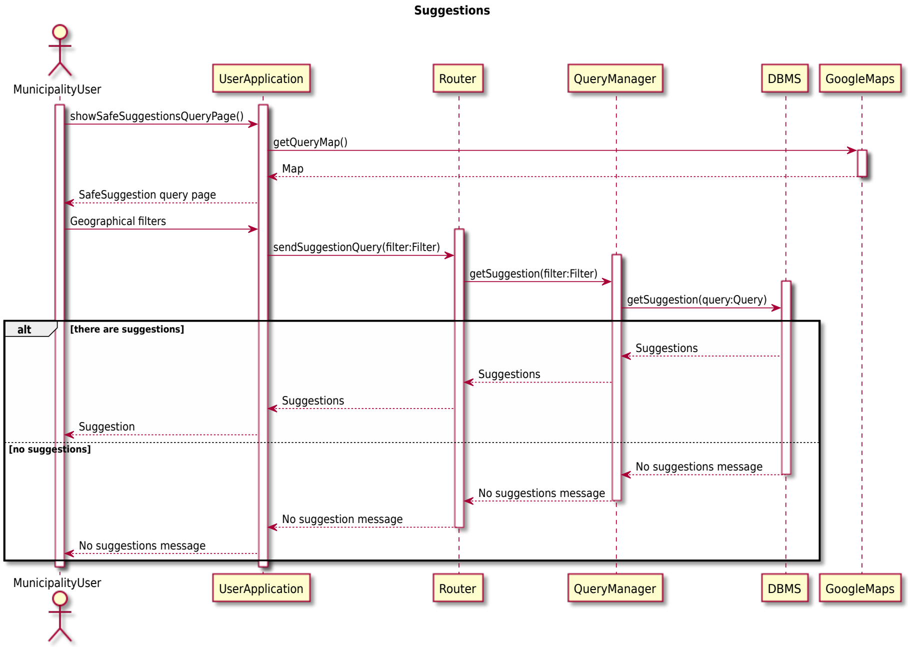


Figure 15: Get suggestions

The municipality user can get suggestions from a query page similar to the ones provided to the other users, but where the filters are only geographical. The DBMS gives an answer, which can be negative or a suggestion.

2.5 Component interfaces

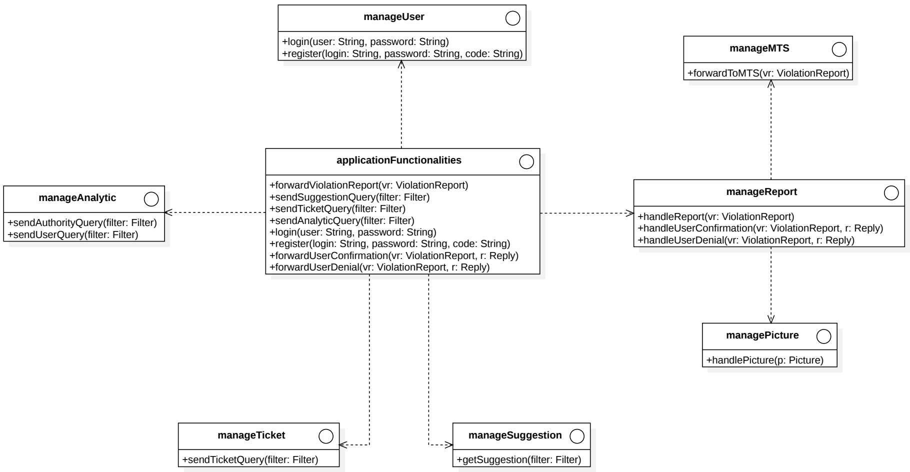

manageUser
+login(user: String, password: String)
+register(login: String, password: String, code: String)
manageMTS
+forwardToMTS(vr: ViolationReport)
applicationFunctionalities
+forwardViolationReport(vr: ViolationReport)
+sendSuggestionQuery(filter: Filter)
+sendTicketQuery(filter: Filter)
+sendAnalyticQuery(filter: Filter)
+login(user: String, password: String)
+register(login: String, password: String, code: String)
+forwardUserConfirmation(vr: ViolationReport, r: Reply)
+forwardUserDenial(vr: ViolationReport, r: Reply)
manageAnalytic
+sendAuthorityQuery(filter: Filter)
+sendUserQuery(filter: Filter)
manageReport
+handleReport(vr: ViolationReport)
+handleUserConfirmation(vr: ViolationReport, r: Reply)
+handleUserDenial(vr: ViolationReport, r: Reply)
managePicture
+handlePicture(p: Picture)
manageTicket
+sendTicketQuery(filter: Filter)
manageSuggestion
+getSuggestion(filter: Filter)

Figure 16: Component interfaces

In the component interfaces diagram, the main interfaces described in the component diagram are better described. Not all the interfaces have been presented, because they are related to external services.

applicationFunctionalities interface is the main one, as it allows the communication between the users and all the services, exposing its methods to all of the main sub-components of ApplicationServer.

- **forwardViolationReport** is part of the subsystem which deals with reporting the violations. The argument of the method is a ViolationReport, consisting in an object which contains all the information related to the violation.
- **sendSuggestionQuery**, **sendTicketQuery** and **sendAnalyticQuery** are different methods interacting with different services, but they all have in common the Filter parameter. Nevertheless, this is not to mean that all the filters are the same, indeed they depend on the type of user and service that is requested. They are expressed in the same formulation for compactness and to show their similarities. Take as an example sendAnalyticQuery: this method can be called by both common users and authorities to get data about the violations which, however, can get data in different levels of refinement and detail. The query filters' domain of the common user will be a subset of the one of the authority, but few differences apply to what the two types of users can do.
- **login** and **register** accept as input the user name and password. Moreover, register can accept the code string, which identifies the user as authority or municipality user.

Most of the other interfaces are self explanatory at this point, as their name well describes their functionality, their parameters are the same as those just discussed, and so they are not furthermore described. Conversely, it is worth to describe the **managePicture** interface, as it introduces a not yet described type of data. Here the method **handlePicture** deals with Picture which, however, is not a new type of data, but an object contained in ViolationReport consisting in the picture of the violation.

2.6 Selected architectural styles and patterns

2.6.1 Architectural styles

Client/Server The system's architecture is a typical client-server architecture. Our application handles a huge amount of confidential data (violation and accident reports, tickets, suggestions). The centralization of the information improves the security of confidential data. The client-server architecture also allows the system to have an efficient data processing capability independently from the location of the clients and the server. This characteristic is important since we have several clients spread among different regions and countries. The clear separation of the client and the server provides the advantage to have a high level of maintainability. Updates or changes into one subsystem doesn't directly affect the other.

Three tiers architecture The three-tier architecture is a logical consequence of the chosen client-server architecture. The separation of the client and the server is fundamental but it is not enough. We want to have a clear separation of the presentation layer, application/business layer and the data layer. This clear distinction over different tiers provides the opportunity to have multiple subsystems that can be developed in parallel and by different teams with different expertise areas (backend, frontend, etc). Moreover, changes in the rules of the application layer don't affect the presentation layer. In general, we have more flexibility thanks to the independence of the different layers.
These three layers presented have a one-to-one relationship with the mobile application, the application server, and the DBMS. Furthermore, since our mobile application is also used by common citizens we want to forbid them from having direct access to the database. With the selected architecture we have the application layer between the presentation layer and the data layer. We can easily verify and validate the requests before sending them to the database.

2.6.2 Patterns

MVC The most used architectural pattern for GUI applications is the model-view-controller. Starting from the well known MVC we tried to modify and adapt it to our SafeStreets app. One of the main differences with the classic MVC is that in our case the model doesn't update the view. In general, we don't have direct communication between the view and the model but every communication is intermediate by the controller as shown in figure 17

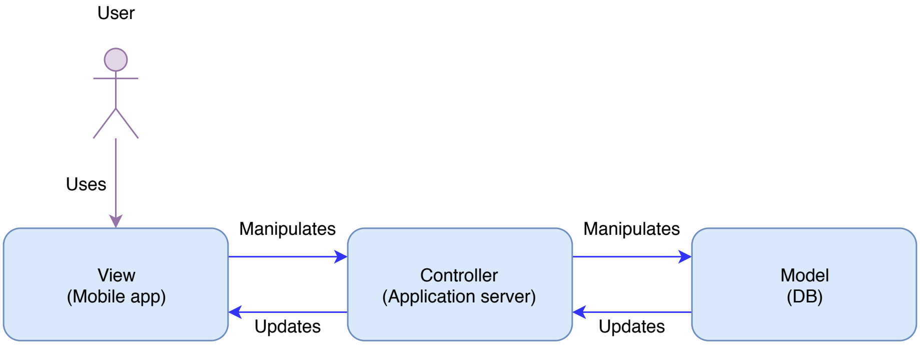


Figure 17: Modified version of the MVC.

With this modified version of the MVC, we follow the division imposed by the three-tier architecture centralizing and decoupling the user interface, the logic and the data of our application. The view represents the mobile application that contains mostly graphical logic. The view interacts with the controller through different types of requests (report requests or queries). We designed a fat controller since most of our logic resides on the application server. The model contains all the data of our application and is therefore represented by the DBMS. Changes to the application data are only made through the controller requests. The user *uses* the view to achieve his personal goals. The communication always starts from a view request and ends with a reply to the view.

RESTful The modified version of the MVC chosen fits perfectly with the RESTful architecture. The figure 18 shows the schema of the communication.
SafeStreets offers different services that can be cluster into two main categories: query services and report services. The nature of these services suggests the complete decoupling between the client and the server. In order to report or to query information, we do not need to maintain a stateful connection. The server reacts and replies to *violation report* requests and query requests. RESTful APIs are ideal because they are stateless. Each API call has all the necessary information to properly work without having to maintain a stable connection. This request/reply paradigm over HTTPS is really simple to develop and maintain.
The stateless connection forbids the application server from making requests to clients. This situation is critical during the SafeReports service. The user after sending the violation report should confirm or deny the license plate. We overcome this limitation transforming the confirmation of the user into a user request. Therefore the server never sends requests to clients but only replies to them.
SafeStreets is intended to provide services to a very large number of users (ideally all the people of one or more countries). Scalability is, therefore, a really important aspect. RESTful architecture allows the system to be scalable. The partition between the client and the server grants the almost independent evolution of the subsystems. Therefore, it is easier to modify the server and to accept an increasing number of requests. The scalability is also guaranteed by the stateless communication as previously stated. On the contrary, stateful communications with a huge number of users can lead to efficiency and availability problems.
With a RESTful architecture it is easier to completely modified and add clients. It is possible to add web applications or different types of views in a relatively small amount of time. The only constraint that the view needs to satisfy is the contract of the programming interfaces. The combination of RESTful architecture and the MVC pattern makes our architecture similar to the already well known ASP.NET MVC framework.

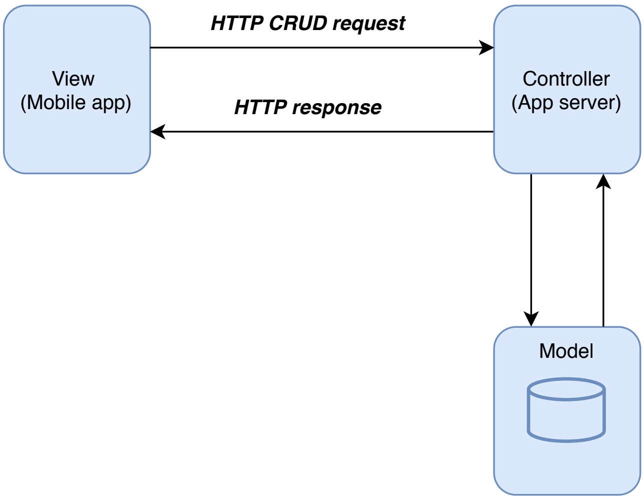


Figure 18: The focus is on the REST API requests from the mobile app to the application server

2.7 Other design decisions

2.7.1 Hybrid mobile app

The underline architecture of our app is a modified version of the MVC combined with a RESTful architecture. Our architecture is indeed very similar to the ASP.NET MVC a very well known architectural pattern intensively used for web applications. SafeStreets however provides a mobile application instead of a web application. Instead of writing a native mobile app we decided to develop a hybrid mobile application. Hybrid applications embed a platform-specific web browser control. The hybrid approach has a series of advantages.

- Low cost of developement.
- Same native app experience with simple web backend approach.
- High speed performances compared to web mobile apps.
- Sharing code with web applications.

It is relatively easy to add web application clients because both web and hybrids applications share most of the code. We adopted the thin client model in which the client contains only the graphical logic following a typical *https request response* paradigm.

2.7.2 Data architecture

The implementation of the database is not the focus of this document. However, it is important to list the main characteristic of the database to have a full understanding of the interaction between the application server and the DBMS. SafeStreets must handle simple and complex historical queries. Therefore, we need to have a separation of the information into a database and a data warehouse.

The database is in charge of simple and frequent queries like the SafeAnalytics queries. We adopted a relational database since the relational model has a series of advantages:

- Easy to use.
- Mature tools for the design and the development process.
- Large number of skilled personnel in relational database technologies.
- Integrity constraints defined for the ACID properties.

The relational database adopts an OLTP policy that consists of a large number of short and online queries. OLTP focuses on the integrity of the data in multi-access environments. The efficiency of an OLTP system is defined as the number of transactions per second (throughput). The throughput is very important since SafeStreets must handle queries from a huge number of users and it must reply in less than 3 seconds.

The data warehouse instead, must process historical data and must handle complex queries like the SafeSuggetions queries. We adopted a data warehouse since SafeStreets must create new suggestions from the violation and accident reports. The creation of new suggestions is a complex process that involves AI and data mining techniques (clustering, association rule learning, etc). The data warehouse is the sine qua non for mining the information and producing the expected results. The source data of the data warehouse comes from the relational database described before but it is not difficult to extend the source of information even to external heterogeneous data sources. The incoming data is pre-processed by the ETL, a module that extracts transforms and loads the data into the data warehouse. The ETL module is responsible for integrating all the heterogeneous data sources. The data warehouse uses an OLAP policy that consists of small but complex queries from aggregated and historical data. The efficiency of an OLAP system is defined as the response time.

3 User interface design

3.1 Mockups

This section contains the most significant mockups of our application. Some of them were already shown in the RASD document but they are included again to improve readability.

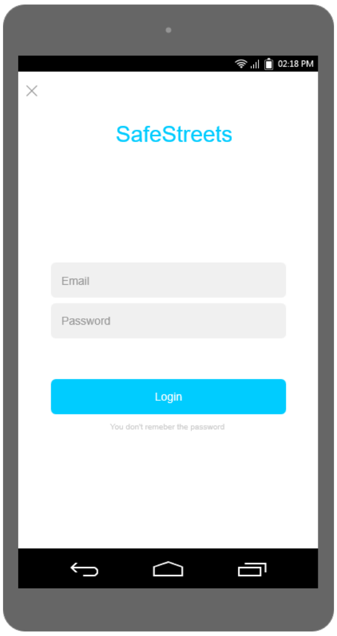


Figure 19: The login process for all types of users.

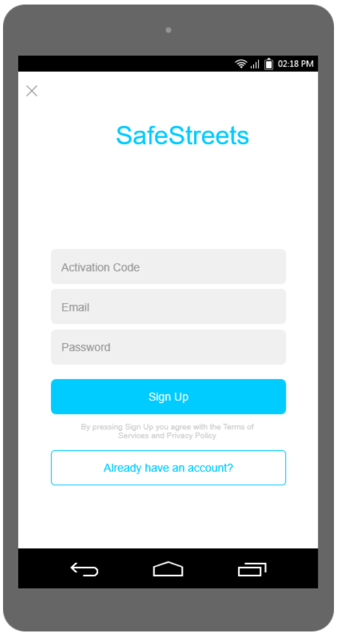


Figure 20: The sign-up process for the authority/municipality user.

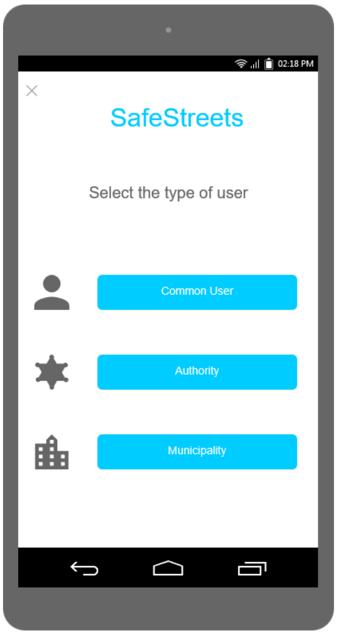


Figure 21: The choice between different types of users.

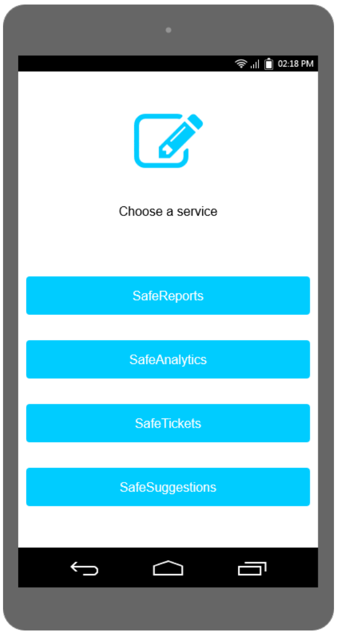


Figure 22: The menu page. The available services depend on the type of user as shown in the UX diagrams.

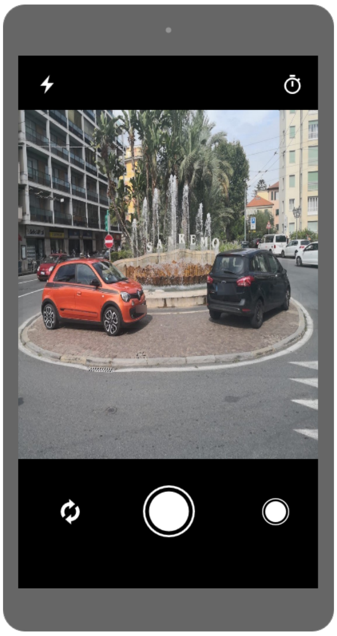


Figure 23: Take picture process.

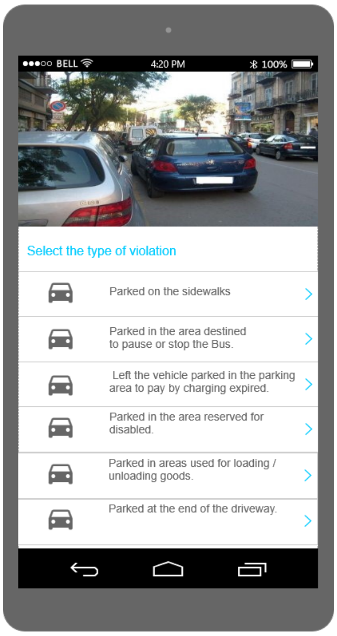
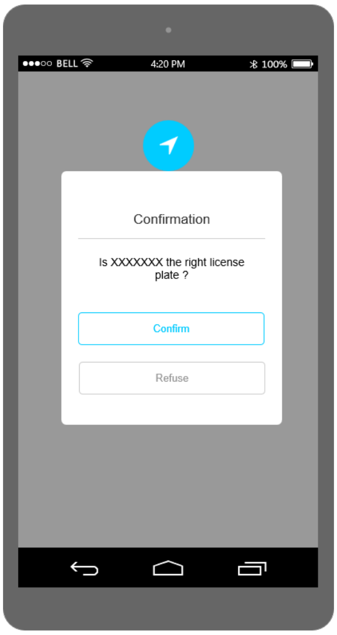
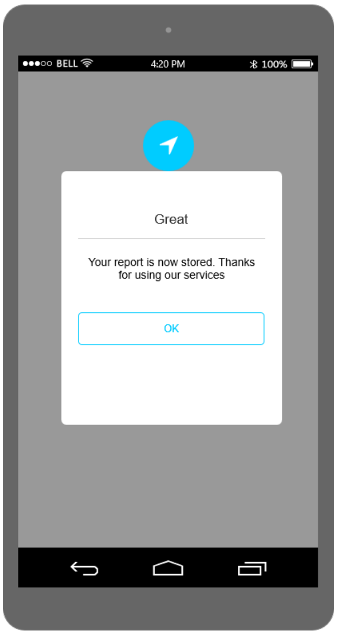

BELL
4:20 PM
100%
Select the type of violation
Parked on the sidewalks
Parked in the area destined to pause or stop the Bus.
Left the vehicle parked in the parking area to pay by charging expired.
Parked in the area reserved for disabled.
Parked in areas used for loading / unloading goods.
Parked at the end of the driveway.

Figure 24: The user needs to select the violation.



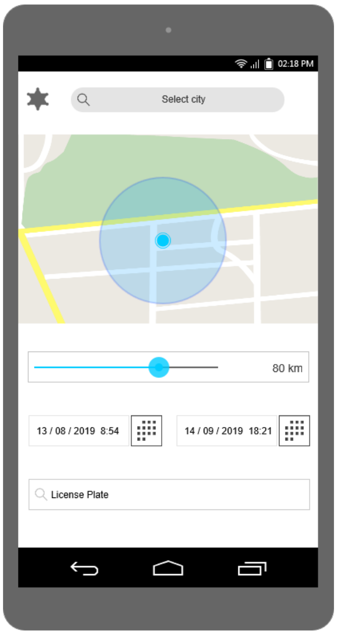
Is XXXXXXX the right license plate ?

Figure 25: The user needs to confirm the license plate.



Your report is now stored. Thanks for using our services

Figure 26: Confirmation of a successful report.

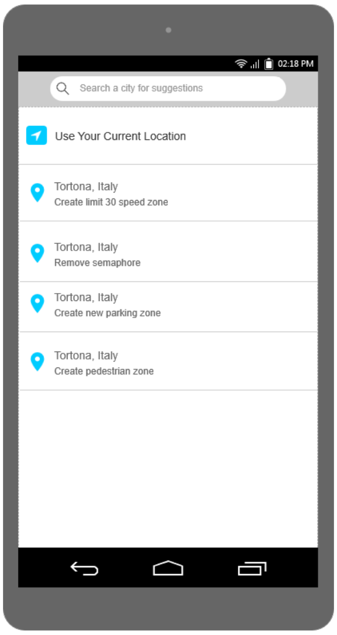


80 km

13 / 08 / 2019 8:54

14 / 09 / 2019 18:21

Figure 27: Filters. As shown in the UX diagram different users and services requires different filters



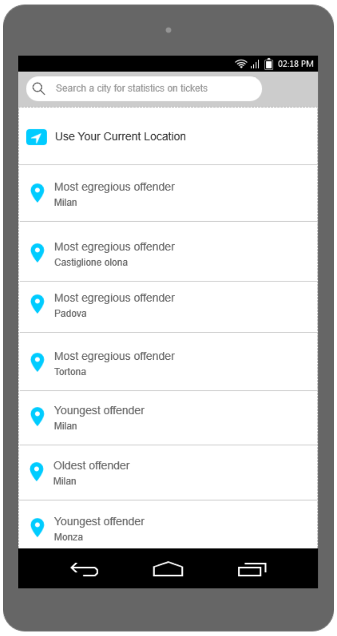
Tortona, Italy
Create limit 30 speed zone

Tortona, Italy
Remove semaphore

Tortona, Italy
Create new parking zone

Tortona, Italy
Create pedestrian zone

Figure 28: The results of a SafeSuggestions query.


02:18 PM

Search a city for statistics on tickets

Use Your Current Location

Most egregious offender
Milan

Most egregious offender
Castiglione olona

Most egregious offender
Padova

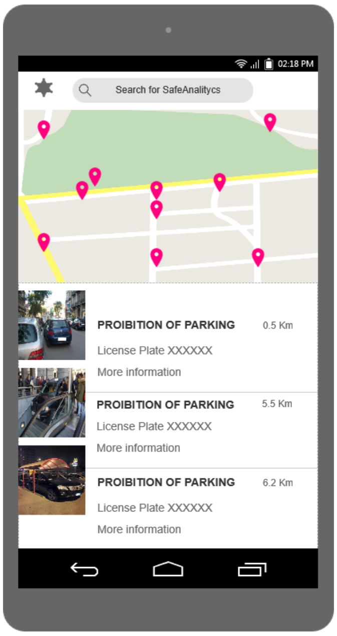
Most egregious offender
Tortona

Youngest offender
Milan

Oldest offender
Milan

Youngest offender
Monza

Figure 29: The results of a SafeTickets query.


02:18 PM

Search for SafeAnalitycs

PROIBITION OF PARKING 0.5 Km
License Plate XXXXXX
More information

PROIBITION OF PARKING 5.5 Km
License Plate XXXXXX
More information

PROIBITION OF PARKING 6.2 Km
License Plate XXXXXX
More information

Figure 30: The results of a SafeAnalytics query.

3.2 UX diagrams

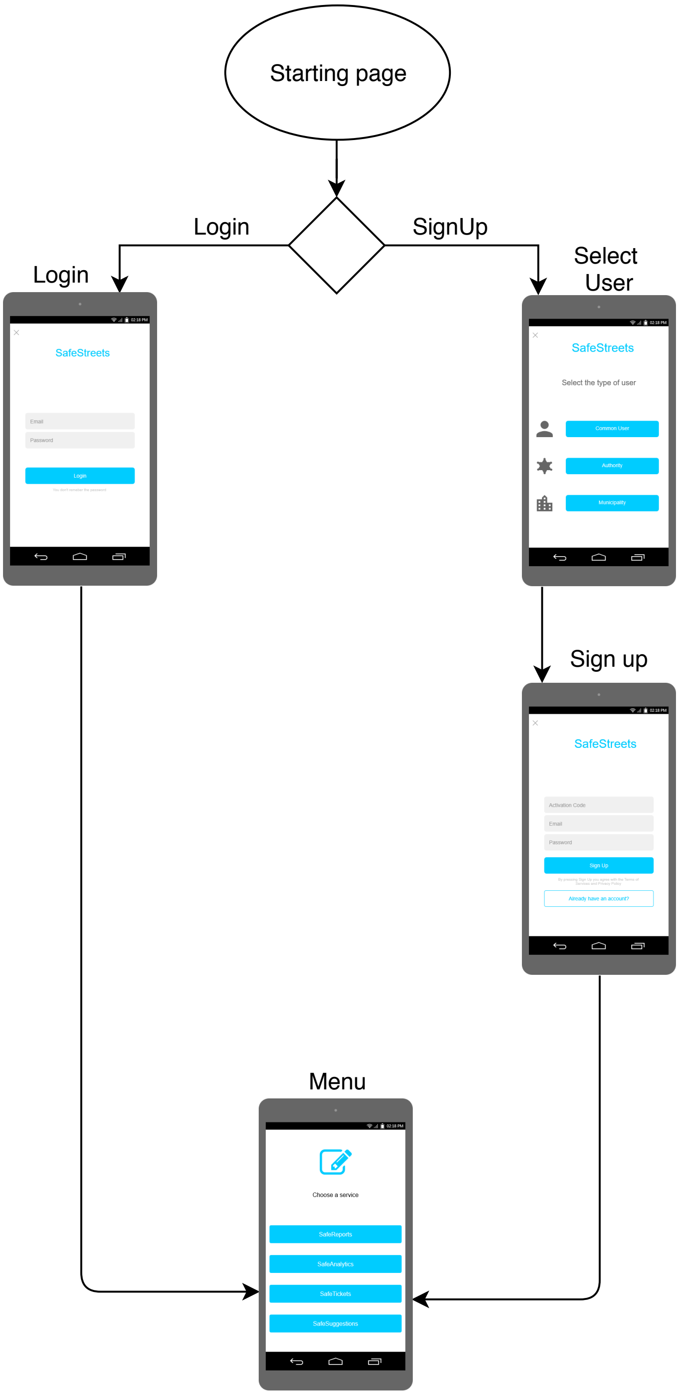
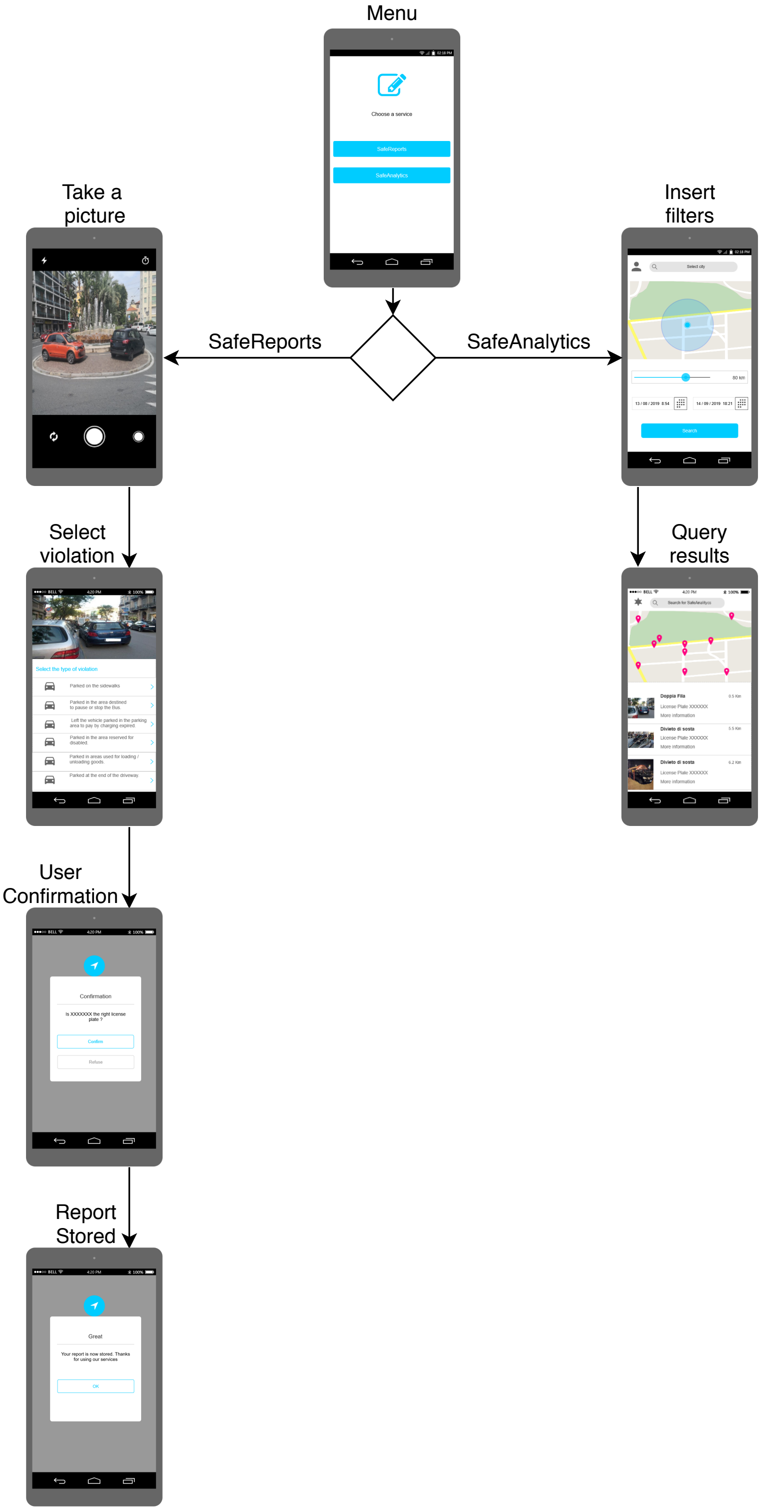


Figure 31: The user decides to login or sign-up.



Choose a service

Select the type of violation

Parked on the sidewalks

Parked in the area destined to pause or stop the Bus.

Left the vehicle parked in the parking area to pay by charging expired.

Parked in the area reserved for disabled.

Parked in areas used for loading / unloading goods.

Parked at the end of the driveway.

Doppia Fila 0.5 Km
License Plate XXXXXX
More information

Divieto di sosta 5.5 Km
License Plate XXXXXX
More information

Divieto di sosta 6.2 Km
License Plate XXXXXX
More information

Confirmation

Is XXXXXXX the right license plate ?

Great

Your report is now stored. Thanks for using our services

Figure 32: User choosing between SafeReports and SafeAnalytics.

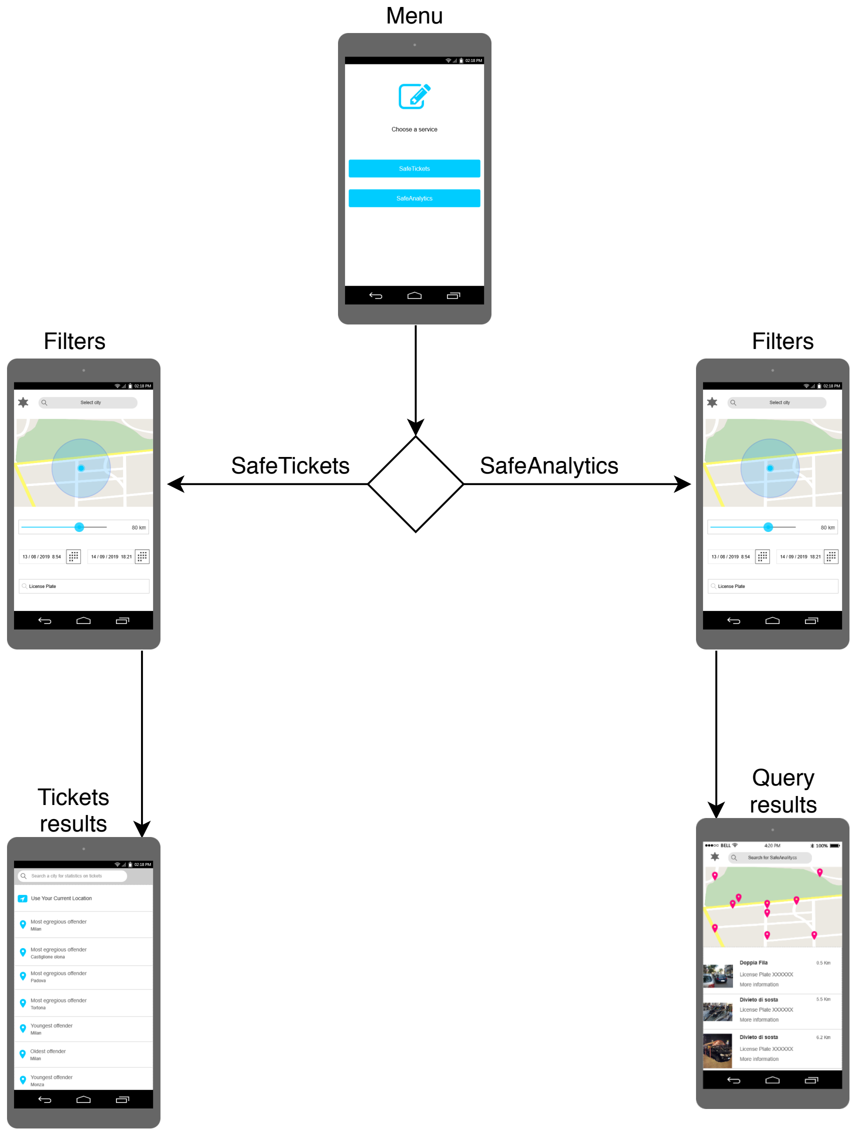


Figure 33: Authorities choosing between SafeTickets and SafeAnalytics.

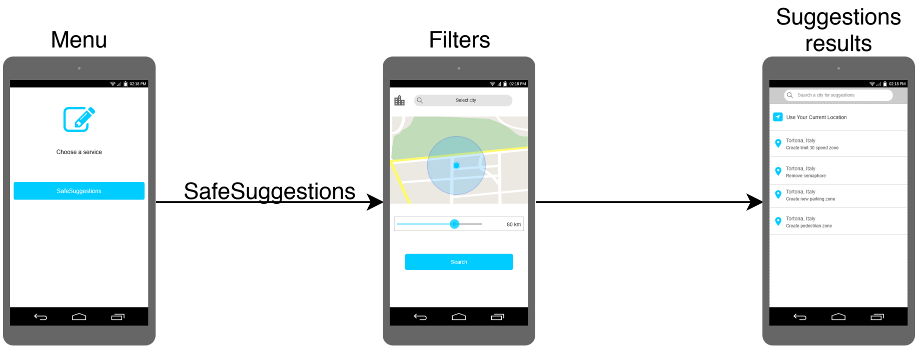


Figure 34: Municipality has only SafeSuggestions as service.

The UX diagrams represent the flow of the most important user interactions with the SafeStreets app. The login sign-up diagram is unique since the distinction between the different types of users is not relevant yet. The other distinct diagrams show the different services that are available for each type of user. The UX diagrams represent the transitions between the distinct mockups. Other less important interactions were omitted to improve the readability of the diagrams.

4 Requirements traceability

The design is thought to satisfy the requirements shown in the RASD. To clarify the motivations that led to the definitions of the components here are reported the links between requirements and components.

- **R1** When a picture is taken using SafeReports, a new violation record is generated.

  ReportManager and **PictureManager** are interested in the process. In particular PictureManager sends the picture to the ReportManager, which embeds the image with the other violation data.

- **R2** When a new violation record is generated, the current position of the user is added to the report.
  R3 When a new violation record is generated, the timestamp is added to the report.

  ReportManager ensures to get the position and the timestamp from the picture data, which is sent to this component together with the image from **PictureManager**.

- **R4** When a new violation record is generated, the photo is scanned by an OCR software to automatically detect the plate.

  PictureManager interacts with the OCR software to get the reading of the license plate and sends the result to **ReportManager**.

- **R5** If the OCR software fails in detecting the plate, the user is notified and asked to repeat the procedure.
  R6 If the OCR software detects the plate, the user is asked to confirm the violation report.

  In case of failure in the interpretation of the picture, **ReportManager** deals with the exception. The messages between this component and the user flow through the **Router** and of course occur inside the **UserApplication** (the only means of communication between users and the system).

- **R7** If the user confirms the violation report, it is sent to SafeStreets.
  R8 SafeStreets stores the information about the violation only if there aren't equivalent events already stored.

  DBMS deals with the duplicated violations and eventually discards them. If it considers the violation valid and receives the approbation of the user through **ReportManager**, it stores the data.

- **R9** SafeAnalytics allows common users to get data about violations selecting zone, time and type of violation.

  UserApplication is aware of the type of user who is interacting with the system (thanks to **UserManager**) and allows it to send (through the Router) just the allowed filters to the **QueryHandler**.

- **R10** SafeAnalytics anonymizes information before sending it to common users.

R11 SafeAnalytics allows authorities to get all the information stored by SafeStreets.

QueryHandler deals with the data received and filters it for the common user, while letting all the informations available for the authorities.

- **R12** SafeStreets must store data about accidents provided by the municipality when available.

  InfoUpdater manages the periodic update of the database adding the data acquired from Information.

- **R13** SafeStreets must analyze collected data crossed with data from the municipality to identify possible interventions.

  QueryManager gets the suggestions related to the user's query from **DBMS**, in particular Datawarehouse.

- **R14** SafeSuggestions allows municipality users to get suggestions provided by SafeStreets.

  UserApplication is aware of the type of user and allows municipality users to access SafeSuggestions' functionalities through **Router**.

- **R15** When the users send a violation report, its integrity is checked.
  R16 If the integrity check is not successful, the violation report is discarded.

- **R17** SafeStreets must forward every new stored violation report to MTS to generate traffic tickets.

  MTSManager receives the data about the new violations from **ReportManager** and forwards it to MTS

- **R18** When a new ticket is generated using MTS, ticket-related data are stored by SafeStreets.
  R19 SafeStreets must build statistics from stored data about issued tickets.

  The link between **MTSManager** and **DBMS** shows that the ticket data are stored in the database. Statistics are built inside the Datawarehouse.

- **R20** SafeTickets allows authorities to get information and statistics on issued tickets.

  Since the data about issued tickets is saved in the database, authorities can access it as they do for the violations data.

- The system must be able to serve a great number of users reporting a violation simultaneously.

  The database exploits an OLTP policy, that allows to manage a large number of transactions (the concurrency is internally handled by the DBMS).

- SafeAnalytics must be able to provide the data requested by both common users and authorities in less than 3 seconds, to provide the best experience.

  The data warehouse pre-processes the data according to an OLAP policy, in order to speed up the queries.

- The data about accidents provided by the municipality must be checked for updates at least every 5 minutes, to provide reliable and always updated suggestions.

  This can be achieved by setting the correct time T to activate the loop described in the AccidentUpdater sequence diagram, triggering **InfoUpdater** as frequently as needed.

5 Implementation, integration and test plan

This section contains the guidelines for the implementation, the testing and the integration for the development teams. Before diving into the specific implementation order it is useful to have an abstract view on the different functionalities available for the SafeStreets' customers. The table above shows for every functionality the importance for the customer and the implementation difficulty. The classification of features based on the importance and the difficulty allows us to give priority to some functionalities. Therefore, the developers can focus on the high priority functionalities and if necessary postpone the others.

Feature	Importance	Difficulty
Sign up and login	Low	Low
Common users must be allowed to report violations	High	High
Common users must be allowed to access data about violations	High	Medium
Authorities must be allowed to access data about violations	High	Medium
Stored violations must be forwarded to MTS to generate traffic tickets	Medium	Medium
Authorities must be allowed to access data about issued tickets	Medium	Medium
Municipality users must be allowed to get suggestions on possible interventions	Medium	High

SafeStreets offers different services but the core services are the ones offered by SafeReports and SafeAnalytics. This explain the high importance for the first three features in the table. All the other functionalities are important but not as much as the core functionalities. Almost all the SafeStreets features are simple queries. The implementation is not that heavy. However, there are two services that requires more logic.
SafeReports must stores the reports sent by the users appliyng a complex logic. Since this functionality involves many subcomponents the implementation becomes trickier. The other complex functionality is SafeSuggestions. SafeSuggestions is consider a high difficulty feature because in order to provide the results for the query a complex data mining process must be implemented. The sign-up and the login process are not the core of our application and they are an easy feature to implement.

5.1 Component integration

In this section, the order for the integration of the components will be described. This is useful for the planning of the implementation. If many components are implemented simultaneously, the order for the integration shown in this section avoids wasting time due to dependencies not considered.

5.1.1 Integration of the application server

The components inside the application server are quite independent. They can be developed in any order, except for **Router**, that must be developed as last since it uses the interfaces of the other components. The figure 35 shows the diagram for the integration.

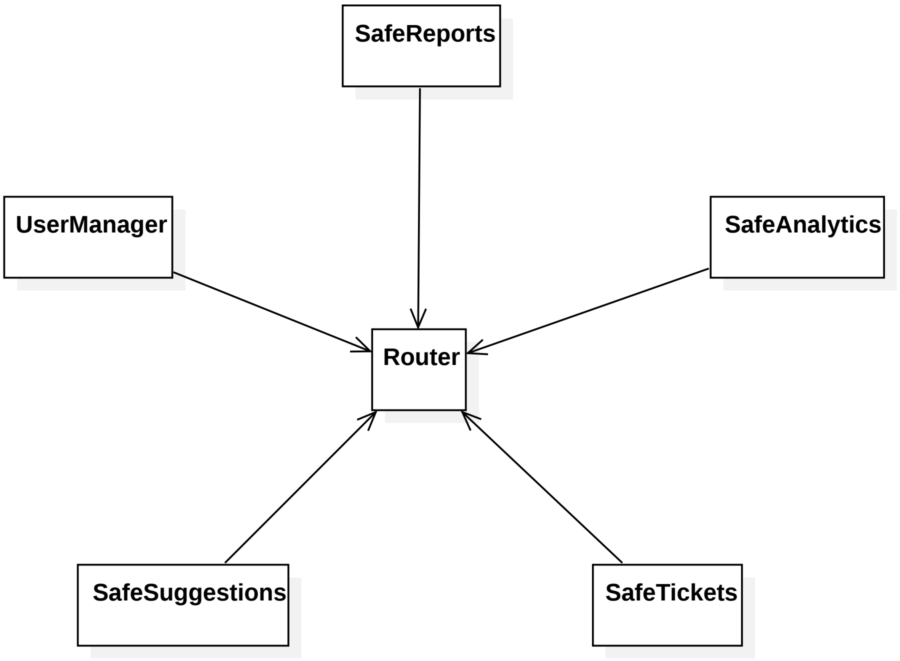


Figure 35: ApplicationServer integration

5.1.2 Integration of the internal components

The only internal components that consists of more than one sub-element are **SafeSuggestion** and **SafeReports**. The first one does not need any diagram since the two sub-components are independent one from another. The second one must follow the integration order of the diagram in figure 36.

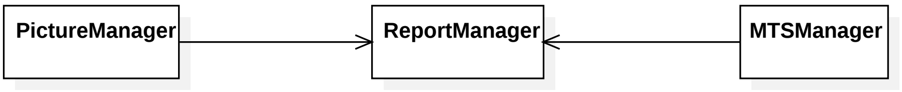


Figure 36: ReportManager integration

5.1.3 Integration with external components

The diagram in figure 37 shows the order for the integration of the application with the external components. Since the external component are not developed by SafeStreets itself, they are considered as pre-existing.

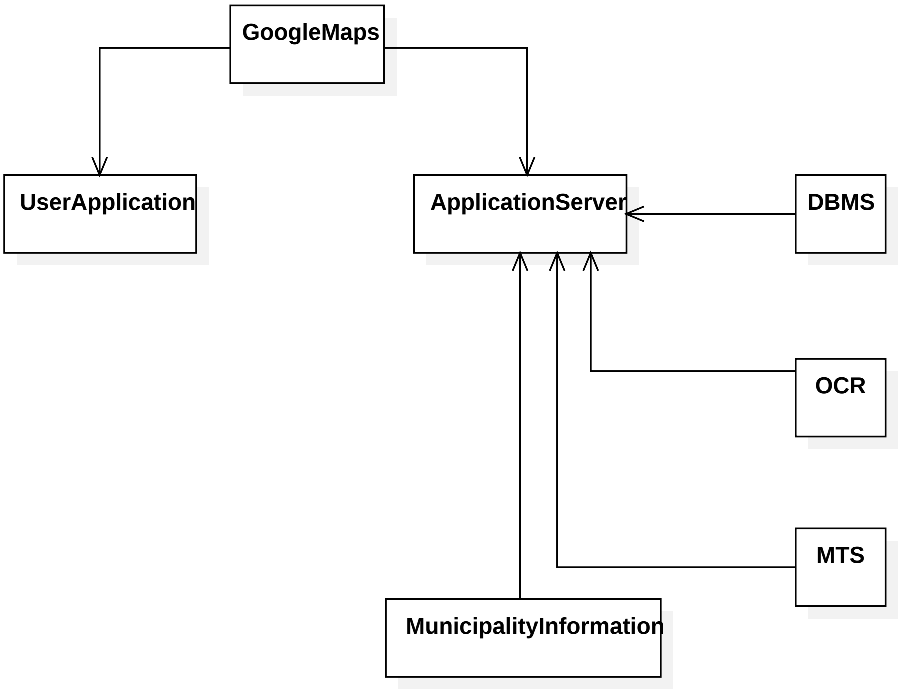


Figure 37: External components integration

5.2 Test plan

Testing will be done using a bottom-up approach. Every time a new component is implemented it must be tested with unit tests. This approach allows us to have single components that properly work. During the testing process, it might be necessary to develop stub or drivers that simulate the behavior of subcomponents that are not implemented yet. The entire testing process follows the famous V model. During the process, we test different parts of our system such as :

1. Single components or modules
2. Groups of related components
3. Subsystems
4. Entire system

5.2.1 Unit test

The single components are tested through the unit tests. The most used framework for unit testing is the well-known Junit test framework. Every public method of every class must be properly tested. The tests should cover all the edge cases for every functionality exposed by the component. The tests are written after the component is properly implemented to avoid the massive implementation of stubs typical of the test-driven approach.

5.2.2 Integration test

After testing all the single components, it is necessary to work on integration testing. The integration testing is made on groups of components or subsystems. Assuming the single components correctly working the main focus of the integration testing is the testing of the interfaces between the different modules since the dynamic behavior of the subsystems is encapsulated in the interfaces.

After testing the subsystems the entire system needs to be tested. We are going to test the integration of the different functionalities subsystems (SafeReports, SafeSuggestions, SafeTickets, SafeAnalytics) with all the different users that our application handles. With the entire integration testing, we need to discover the limits of our system such as the throughput, the response time and all the non-functional requirements already described in the RASD. After having a good correspondence between our goals and the developed systems we can release a beta version of the application. We let a small number of users try the beta version to test the robustness of the system and to have useful feedback for the final release.

6 Effort spent

Task	Braga	Calderon	Favaro
Introduction	2	2	2
Architectural design	28	28	28
User interface design	6	6	6
Requirements traceability	4	4	4
Implementation, integration and test plan	8	8	8

7 References

- Specification document ”Mandatory Project Assignment” 2019-2020
- IEEE Std 830-1998 IEEE Recommended Practice for Software Requirements Specifications
- UML diagrams:
  `https://www.uml-diagrams.org/`