Politecnico di Milano

Software Engineering II project

SafeStreets

Design Document

Authors:
Matteo Pacciani
Francesco Piro

Professor:
Matteo Giovanni Rossi

December 9, 2019

version 1.0

Contents

List of Figures

List of Tables

1 Introduction

1.1 Purpose

This document completely describes the software design and architecture of the SafeStreets system. The entire description we are going to provide is totally based on what we have defined in the *Requirements Analytics and Specification Document* [2] but we have tried to maintain an independence between the two documents by giving further descriptions of what we have already said but in a better manner for the aim of this document.

It is very important that this document gets completely read and understood before starting with the development of the system and to stick with its details as much as possible as they are thought and engineered as a whole in order to obtain what the system's actual purpose is.

1.2 Context

SafeStreets is a *crowd-sourced* application that intends to provide a way to notify authorities when a traffic violation occurs. Nowadays systems tend to aim the *crowdsourcing* always more, this way of managing the information in fact reflects different positive aspects that can be used to provide a lot of useful functionalities with a cost that is almost null. Consequentially, several discussions and opinions try to describe which is the best practice to develop a crowd-sourced application. After having considered several alternatives (for example [1]) we decided to build our system with the aspects we thought to be the most important, grasped from one possibility to the other. As a result we have obtained a solid system based on totally agreeable assumptions that would be really developed in the "real world".

1.3 Scope

The aim of the system is to achieve with outsourcing a simple way to notify traffic violations and use this information to retrieve interesting data for both its customers: users and authorities. Hence we can think as the main objective of SafeStreets the one of the notification and then describe the other functionalities in the basic and advanced services.

The system will describe the notification functionalities in terms of *parking violations* but is thought for further extensions in order to deal with the other types. Users are the clients allowed to report a violation and authorities will be notified whenever a new one will be reported. This process takes place entirely inside the application where both customers, once recognized, will be able to benefit only of the services related to their role. As we have already said SafeStreets is a crowd-sourced application; thanks to this property the big amount of data that is going to be managed will be used to provide additional functionalities based on the processes of mining and crossing this information. We call the *basic functionality* the one related to the data mining used to retrieve statistical data interesting for both the users and authorities. It is important to highlight that this functionality considers the role of each customer in order

to provide him a way for retrieving the data with a precise level of visibility. As completely described in the section related to the functionalities provided by the application (2.1.1) the authorities in fact benefit of an additional service that allows them to retrieve the information about the parking violations in the most detailed way (for example with the entire description of an infraction that contains also the data about who reported it and the plate of the vehicle). The *advanced functionality* instead is related to the concept of **safety** that can be attributed to a street. This functionality is based on the crossing process between the data stored in SafeStreets system related to the violations and the one of the accidents possibly provided by a municipality. The safety of each street our application is able to retrieve is strictly bound with an additional service that wants to find a suggestion for a possible intervention for a street that has been marked as *unsafe*.

1.4 Glossary

1.4.1 Definitions

- **DBMS:** is the software system that allows to manage efficiently the data stored in the database of the system.
- **Posta Elettronica Certificata:** is a technology that allows to send email with a legal approach.
- **Crowdsourcing:** is a sourcing model in which individuals or organizations obtain goods and services from a large, relatively open and often rapidly-evolving group of internet users.
- **REST:** is an architectural style for communication based on strict use of HTTP request types.
- **UX diagram:** helps to visualize the steps a user takes to complete a task or achieve a goal on a site or app.

1.4.2 Acronyms

- **RASD:** Requirements Analytics and Specification Document
- **DBMS:** Database Management System
- **EMS:** Email Management System
- **ACI:** Authority Common Interface
- **PEC:** Posta Elettronica Certificata
- **API:** Application Programming Interface
- **IRI:** Image Recognition Interface
- **MI:** Map Interface
- **GPS:** Global Positioning System
- **JSON:** JavaScript Object Notation

- **OS:** Operating System
- **HTTPS:** HyperText Transfer Protocol over Secure Socket Layer
- **TCP:** Transmission Control Protocol
- **IP:** Internet Protocol
- **REST:** Representational State Transfer
- **IOS:** iPhone Operating System

1.4.3 Abbreviations

- **IIT:** Implementation, Integration and Testing
- **R:** Requirement
- **e.g.:** exemply gratia

1.5 Document Structure

The document is structured in a double linked way in order to provide an easier and quicker navigation in particular for the several images that can be easily retrieved in this way by going right at the toc and search for a new one.

Moreover the document is structured as now briefly described:

1. **Introduction:** gives a first description of the problem and tries to motivate how in this document it is going to be faced first with the purpose and second with the context. The scope is used to point out the main features of the system as they can be always clear in mind while reading the document. The section ends with the glossary.
2. **Architectural Design:** this is the core of the entire document. This section starts with a high level description of the architecture of the system and then continues always more in details considering the components and then the interfaces that have been identified. The deployment of the system is also taken into account right after having defined the components and the database view while the interfaces are precised in the next section. The entire section terminates with the precise description of the architectural styles and patterns (sec 2.7) used to define the entire system.
3. **User Interface Design:** this section aims to a precise description of the interfaces that will conduce the interaction of the clients with the system. As these mockups were already presented in the RASD we want here: first to describe with the UX diagrams how the client navigates the application in order to obtain the functionality he is requiring; second to give a precise identification of each mockup to explain why it needs to be designed in this way in order to provide the functionality it is supposed for.
4. **Requirements Traceability:** once the description of the software design and architecture is clear we identify the relation between the requirements identified in the RASD document [2] and the components that allow to realize them.

5. **Implementation, Integration and Test Plan:** finally in this section everything gets together as a plan to develop the entire system needs also to be clearly identified and described. In order to do so, in this section, we first give an overview of the main features we considered to decide the plan. Then, in the next subsection we identify the conditions that are needed to be met before starting with the development process. The utility tree (Figure 59) is presented before the real description of the plan as we thought it would be helpful to understand further motivations provided for the order of the IIT plan. Thanks to the **Use Relation Hierarchy** diagram of Figure 60 we used the relations to establish the order and then described the plan by matching each phase of *Implementation, Integration and Testing* with the relative number.

6. **Effort Spent:** this section has been used to keep track of the hours spent to complete the document. The first table defines the hours spent together while taking the most important decisions, the seconds instead contains the individual hours.

7. **References:** includes all the references used to define the document.

2 Architectural Design

In the following section we describe precisely the architecture of our system. Starting with the highest possible view we first give an illustration of all the systems that SafeStreets is going to interact with. Differently than how we did in the RASD [2] document we need here to precise the rule of the systems in order to obtain a clear description of the communication between with the one we are designing.

After a first global description we start focusing on the system, initially with a clear listing of all the components needed to obtain the functionalities of the application and then with their deployment (2.4.2) and runtime (2.5) utilization.

The section concludes first with the precise definition of the architectural patterns used to deploy all the components identified and finally with other design decisions (2.8) that have been chosen in the architectural design process.

2.1 Overview

2.1.1 General Context

To get involved in the problem, we start in section with the highest level of visibility possible of our system in order to understand which are the other systems it needs to interact with. SafeStreets is a crowd-sourced application that intends to provide services to two types of customers: **users** and **authorities**. Hence the crowd is composed by the users who provide the system information thanks to the core functionality offered: the *notification of a traffic violation*.

To understand which systems are involved in each of the functionalities offered by SafeStreets, that will now be called as already mentioned in the RASD document [2], we need to focus on their definitions:

- **Access Functionality:** service that allows customers to register and login to the system in order to be recognized.
- **Core Functionality:** service provided only to the users to allow them to notify a traffic violation.
- **Basic Functionalities:** mining services over the information provided by the users redistributed to the customers with different levels of visibility.
- **Advanced Functionalities:** services that provide the safety of an area chosen by the customers and some suggestions for possible interventions for the streets that are found to be dangerous.

In the following picture we illustrate the **context viewpoint** (Figure 1) as we can later describe how each of the systems identified are needed in order to obtain the functionalities just presented.

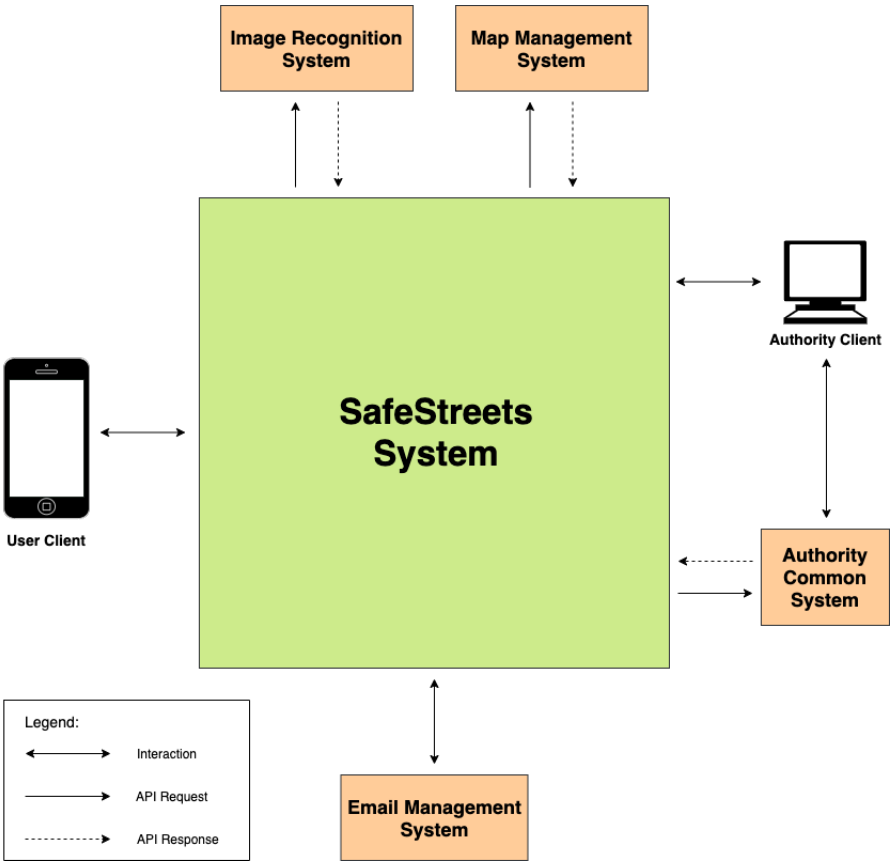

Figure 1: Context Viewpoint

Now that we know which are the systems needed, before dealing with the modules that will consider the communication with each of them let's understand first for which functionality they are needed for. The following table lists the three types of functionalities and a tick (✓) is used to express that the functionality needs the external systems services in order to be realized.

System / Functionality	Image Recognition	Map Management	Email Management	Authority Common
Access			✓	✓
Core	✓	✓		
Basic		✓		
Advanced		✓		✓

Table 1: System Mapping Table

2.1.2 Composition Diagram

With the definition of the systems used by SafeStreets we are now able to highlight the modules used in order to benefit of their services. In the following diagram (Figure 2) we divide the system in three different types of areas, each one containing the modules considered critical for that competence.

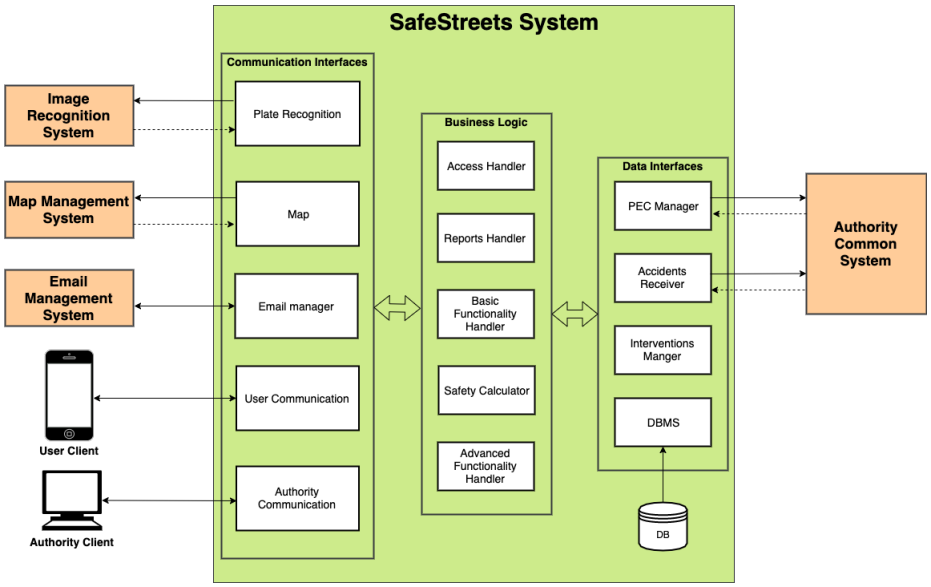

Figure 2: Composition Diagram

As we see can see in the picture above, the system is composed of three areas, each one specialized on a particular behavior.

Communication Interfaces Are the interfaces that allows our system to interact with externally active agents or systems that provide services needed by the business logic. In fact, as we notice in the left hand-side of the diagram we have:

- **Image Recognition System:** provides the methods to deal with the recognition of the plate of the vehicles
- **Map Management System:** provides the methods to deal with all the issues related to the geographical positioning and map interaction
- **Email Management System:** provides the minimal email services that SafeStreets needs to consider in order to provide the recovery of the credentials and more important the recognition of the authorities
- **User Client:** is the client used by a user in order to benefit of SafeStreets' functionalities
- **Authority Client:** is the client used by an authority in order to benefit of SafeStreets' functionalities

All the modules defined here will bring inside the system all the services and requests coming from the outside as the ones in the business logic can process them and provide the functionalities.

Business Logic Now we have the modules that are thought to deal with the functionalities that the system has to provide. As we can see also in this case, we have a correspondence between the functionalities described in the previous section and the modules now listed:

- Access Handler
- Reports Handler
- Basic Functionality Handler
- Safety Calculator
- Advanced Functionality Handler

Data Interfaces The last interfaces instead are separated from the initial ones because they provide a way to the system to access the external data he needs to use in order to provide its functionalities. As we can see in the right hand-side of the diagram we only have the **Authority Common Interface**: it provides the methods to retrieve the data of the accidents that took place in a certain city and the list of all the PEC addresses of the authorities.

2.2 Component View

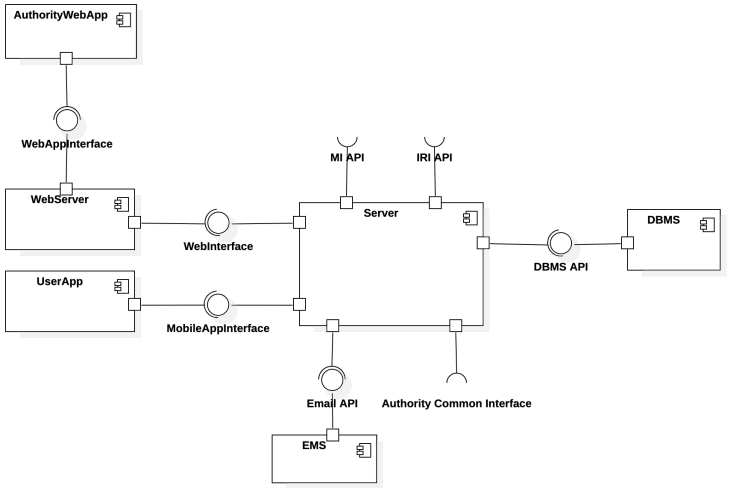

Figure 3: High Level Component Diagram

With the high-level description of the system provided in the previous sections we are now able to determine which are the components that will be considered in the architecture in order to accomplish the functionalities that each of the modules identified are going to provide. In fact it is straightforward the mapping between the component diagram (Figure 2) and the picture above (Figure 3). The **communication interfaces** area is mapped with all the interfaces provided by the components that provide or receive services from the

outside; the **business logic** is realized by the server component where all the computation to provide the functionalities of the system takes place, while the **data interfaces** are the ones that brings to the system both the DBMS and the authority common interface's services.

Before describing in detail the most important components of the system it is now the moment to precisely describe what are the components presented in this picture and what are the services provided by their interfaces:

- **Server:** it is the business logic of the entire system. Inside this component, that will be precisely described in the next section, takes place all the computation that allows to provide the customers the functionalities that SafeStreets has defined as its goals. The two interfaces provided by this components are the ones that allow the client systems to benefit of these functionalities; considering to have both a mobile application and a web app we need one interface directly for the mobile software while another interface for the web server that will provide to the browser of the authority the functionalities offered.
- **Web Interface:** it is the interface that allows the web server to access the methods that provide the functionalities once a request arrives from an authority.
- **Web Server:** it is the component that deals with all the issues related to the realization of a web app that needs to be displayed on the browser of the authority. In fact it provides the interface that represents the endpoint where the browsers can access the system's functionalities.
- **Web App Interface:** it is the interface that allows the browsers to benefit of the services of SafeStreets with a web app technology.
- **Authority Web App:** it is the component that represents the browser used by the authority whenever it decides to access the system.
- **Mobile App Interface:** it is the interface that allows the mobile applications to access the functionalities provided by our system.
- **User App:** it is the component that represents the mobile software installed in the users' devices which accesses the services of SafeStreets thanks to the interface just presented.
- **MI API:** it is the interface that provides the methods to the system in order to deal with the map and geographical issues:
  - Allows to retrieve the street providing a GPS position
  - Allows to retrieve the best possible path between two positions
  - Allows to display the colored map for the safety functionality
  - Allows to store in the system all the streets and cities
- **IRI API:** it is the interface that provides the methods to the system in order to deal with the recognition of a vehicle's plate. Remember that the algorithms accessible from this API can be helped with the information

provided by the user about the plate's number and, more important, they always "have the last word": this means that whenever a result is found for the plate it will be considered correct and storable by the system; in all the cases when no result is found by the algorithms the notification will be discarded.

- **Email API:** it is the interface that provides the methods to the system in order to deal with the minimal email sending and receiving services needed. An email system in fact is necessary both for the process of code verification to recognize the authorities and also for the credential recovery of both the customers.
- **Authority Common Interface:** it is the interface that provides the methods to the system in order to deal with the PEC addresses and the accidents receiving. This interface is thought to be provided by a common system used by every authority, where:
  - we can retrieve all the PEC addresses of each authority, in order to do the checks in the recognition process
  - we can retrieve all the accidents that took place in the area of an authority's competence. It is assumed that the exchange of the accidents' data develops in this way: each authority publishes its accidents on the common system as we can periodically try to retrieve them.
- **DBMS API:** it is the interface that provides the methods to the system in order to deal with the data management process. Starting from the access one, the notification process and all the others that need to require or store information from the database of the system (in the following section it will be clear which are the components that necessarily need to access the DBMS interface).

2.2.1 Server Component

From now on we are going to blow up the high-level component view just presented, starting with the most complex component that is the server and continuing first with its internal ones and in the end with the others that we have to take into account while defining the architecture of our system (WebServer and UserApp). In all the diagrams we are going to illustrate, the interaction between the components and the interfaces provided is carried on by means of the lollipop-socket notation. We will then continue by precisely describing each of these components to understand how they allow to obtain the functionalities of the system by providing and using the interfaces identified.

The diagram of the next page (Figure 4) highlights all the components that compose the server and describes both their internal and external interactions: we can see in fact the interfaces that we defined in the previous section how are provided or used by these components and how new interfaces are needed to make them interact.

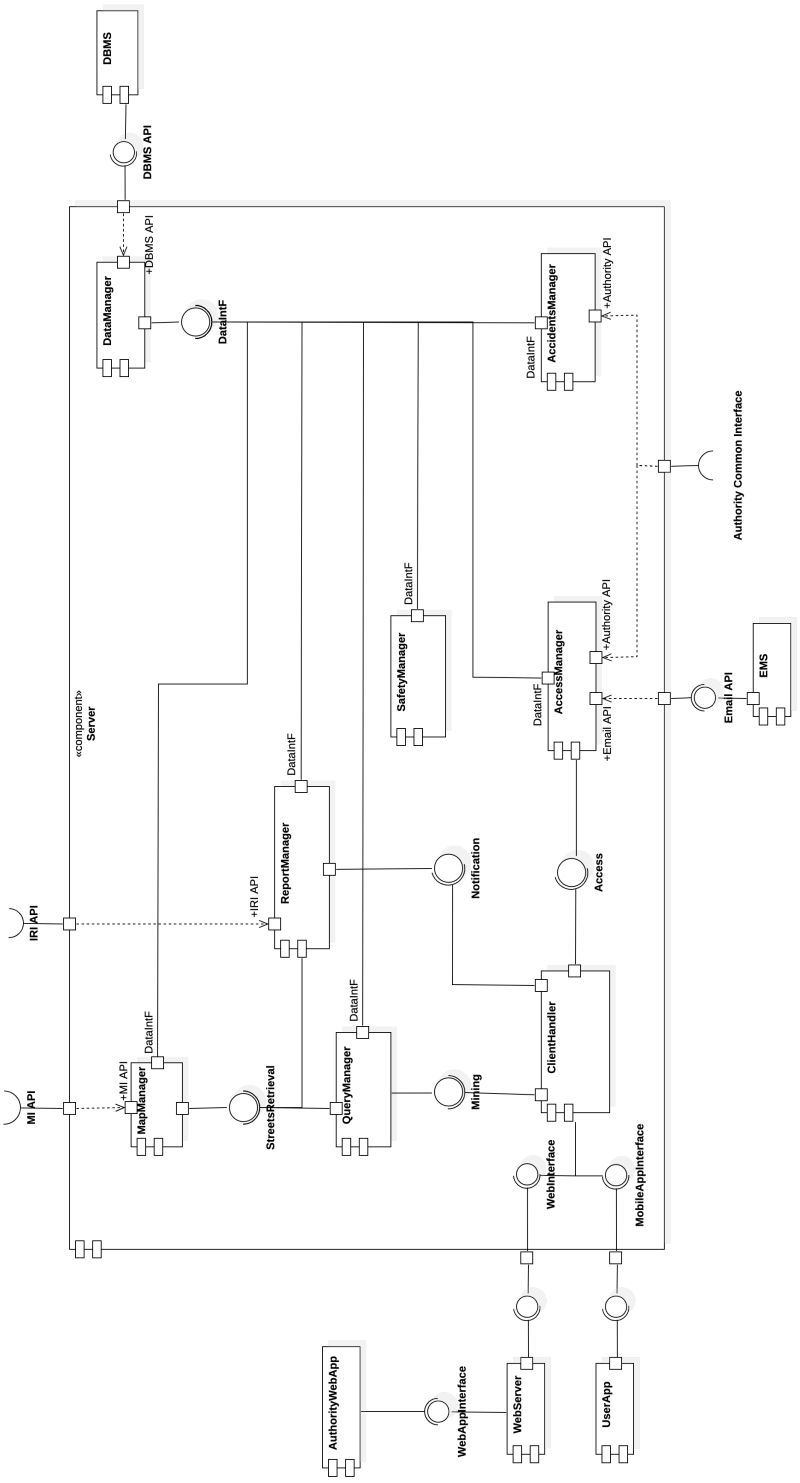

Figure 4: Server Component Diagram

2.2.2 Client Handler Component

The *ClientHandler* (Figure 5) is the component that manages all the requests coming from the customers. As we see two interfaces are provided, one for the mobile application (**MobileAppInterface**) and the other for the web server, that manages the web application (**WebInterface**). These two interfaces are the ones that provide the result of each request once it has been managed by the logic of the system which starts its computation in this component.

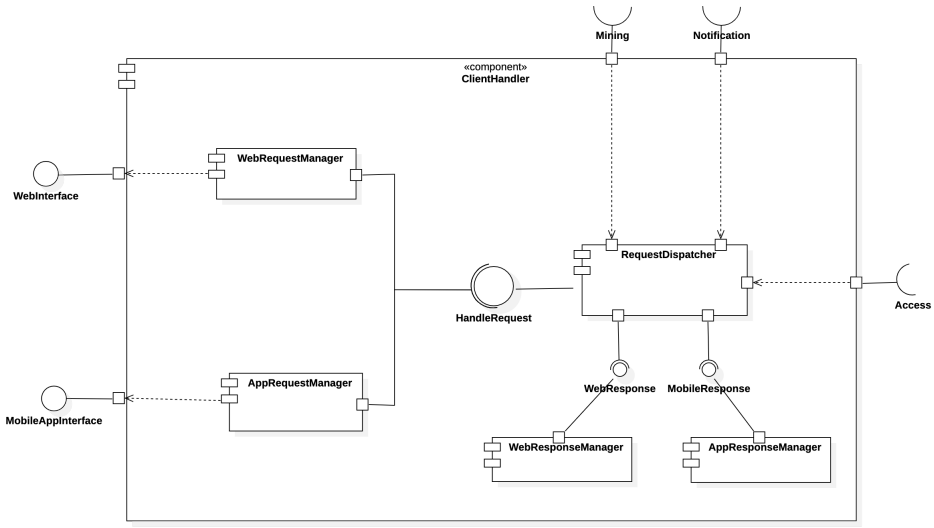

Figure 5: Client Handler Component Diagram

The components that allow the *ClientHandler* to process a request, first receiving it and then dispatching it to the related functionality are:

- **WebRequestManager:** it is the component that manages all the requests that come from the web application, hence from the web server that uses the **WebInterface** realized by this component.
- **AppRequestManager:** it is the component that manages all the requests that come from the mobile application, hence from the device that uses the **MobileAppInterface** realized by this component.
- **RequestDispatcher:** it is the component that allows to handle each type of request that comes either from the web app or the mobile one. In fact, thanks to the **HandleRequest Interface** the components previously described can deploy the request to this one that, depending on the type of the request, will need to use different services in order to provide the functionality required. As a consequence, we can see several interfaces that are used by this component:
  - **Mining Interface:** allows to benefit of the methods provided by the *QueryManager* component that is the one which manages all the queries related to the basic and the advanced functionalities.

- **Notification Interface:** it is the interface that allows to manage the notification process once a request to report a parking violation is sent by a user.
- **Access Interface:** it is the interface that allows to deal with all the issues related to the access to the system. This means: the login, the registration, the credential recovery, the recognition of the authorities...
- **WebResponse Interface:** it is the interface that allows to process the information received from the logic of the system as it can be used by the *WebServer* to be displayed on the web app.
- **MobileResponse Interface:** it is the interface that allows to process the information received from the logic of the system as it can be used by the device to be displayed on the mobile app.

2.2.3 Access Manager Component

The *AccessManager* (Figure 6) is the component that manages all the issues related to the access of the customers. As we have said several times when defining the goals and requirements of SafeStreets, in fact, it is fundamental to recognize the clients and thus to provide a registration process.

The components described in the following diagram are the ones that allow to provide the access functionalities of:

- Registration
- Login
- Credentials Recovery

In order to provide the methods that allow to realize these functionalities through the **Access Interface**, this component has also to benefit of the external services provided by the **Email API** and the **Authority API**. The interface that allows to access and manage the data stored in the system is obviously fundamental for all the checks needed and to store new customers in the application.

The components that allow the *AccessManager* to process an access request, meaning one of the three functionalities listed before are:

- **Email Manager:** it is the component that uses the methods provided by the **Email API** in order to have a way to send the emails with the recognition code and to offer the service of the credentials recovery. This component provides the **mail Interface** to expose the methods related to its service.
- **PEC Manager:** it is the component that deals with all the issues related to the PEC addresses. In fact it uses the methods provided by the **Authority API** in order to know which are the existing addresses of all the authorities and then uses the **mail Interface** to provide through the **PEC Interface** the methods that allow to send the verification code in the registration process.

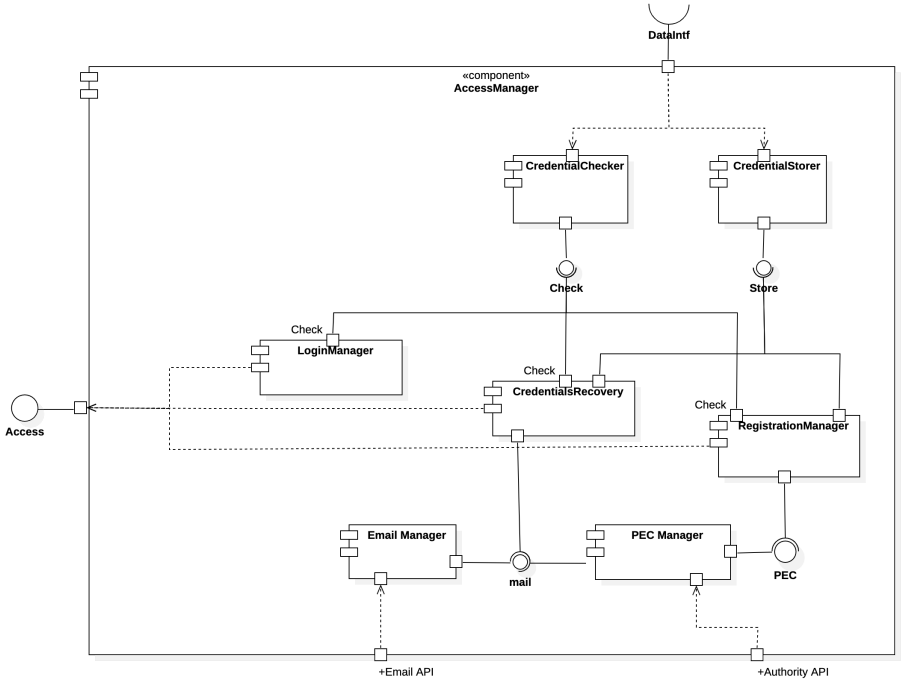

Figure 6: Access Manager Component Diagram

- **LoginManager:** it is the component that deals with the login process. It realizes the **Access Interface** by providing the methods for the login and has always to interact with the *CredentialChecker* component in order to verify the credentials of the logging customer. In this way, thanks to the **Check Interface**, this component will control the credential of either the user or the authority and decide if to authenticate them or not.

- **CredentialRecovery:** it is the component that deals with the credentials recovery process. It realizes the **Access Interface** by providing the methods for the recovery and thus it needs to interact with both the *CredentialChecker* and the *CredentialStorer* components. Thanks to the **Check Interface** and the **Store Interface**, in fact, it will be able to control the identity of the customer who is requiring to recover its credentials and store its possible new information. Obviously the **mail Interface** is also needed in order to send the email to the client that is requiring to recover its credentials.

- **RegistrationManager:** it is the component that deals with the registration of the customers. It realizes the **Access Interface** by providing the methods to register both the user and the authority and thus needs to use the **Check Interface** and the **Store Interface**. In fact, this component will need to check for the new credentials to be feasible and then to store the new data related to the registering customer. The **PEC Interface** is used for the authority registration process to send the verification code

and check the address.

- **CredentialsChecker:** it is the component that deals with all the checks needed in the access functionalities previously described by providing the **Check interface**. Obviously to perform these checks it needs to access the database of the system and thus the methods provided by the **DataIntF Interface**.
- **CredentialStorer:** it is the component that deals with the storing of all the data used in the access functionalities by providing the **Store Interface**. Obviously in order to store the information in the database it needs to use the methods provided by the **DataIntF Interface**.

2.2.4 Query Manager Component

The *QueryManger* (Figure 7) is the component that manages the requests related either to the basic or the advanced functionalities (see section 2.1.1 for the precise functionalities description) coming from the customers with the realization of the **Mining Interface**. Hence, this component needs first to identify the request, then it checks its filters with the **Check Interface** and finally, once identified the functionality, it realizes the respective query with the relative interface provided by the correct component:

- Interventions Interface
- Safety Interface
- Reports Interface
- Dangerous Interface
- Frequency Interface

Once the query is ready it is prepared and sent to the component that will execute it through the **Send Interface** realized by the *QuerySender*.

The components that allow the *QueryManager* to process the requests related to a functionality are:

- **FunctionalityIdentifier:** it is the components that realizes the **Mining Interface** by providing the methods related to the processing of a request that deals with one of the basic or advanced functionalities. This component first needs to recognize which is the functionality requested as it can check it and then prepare it to be executed. Thanks to the **Check Interface** it controls if the filters in the request are feasible with the ones related to the functionality recognized and then it calls the method that builds the query from the correct interface between the ones presented before.

- **UnsafeQuery:** it is the component that manages the query related to the safety functionality. Hence it will select the correct area and retrieve the safety of all the streets required by the customer as they can be displayed

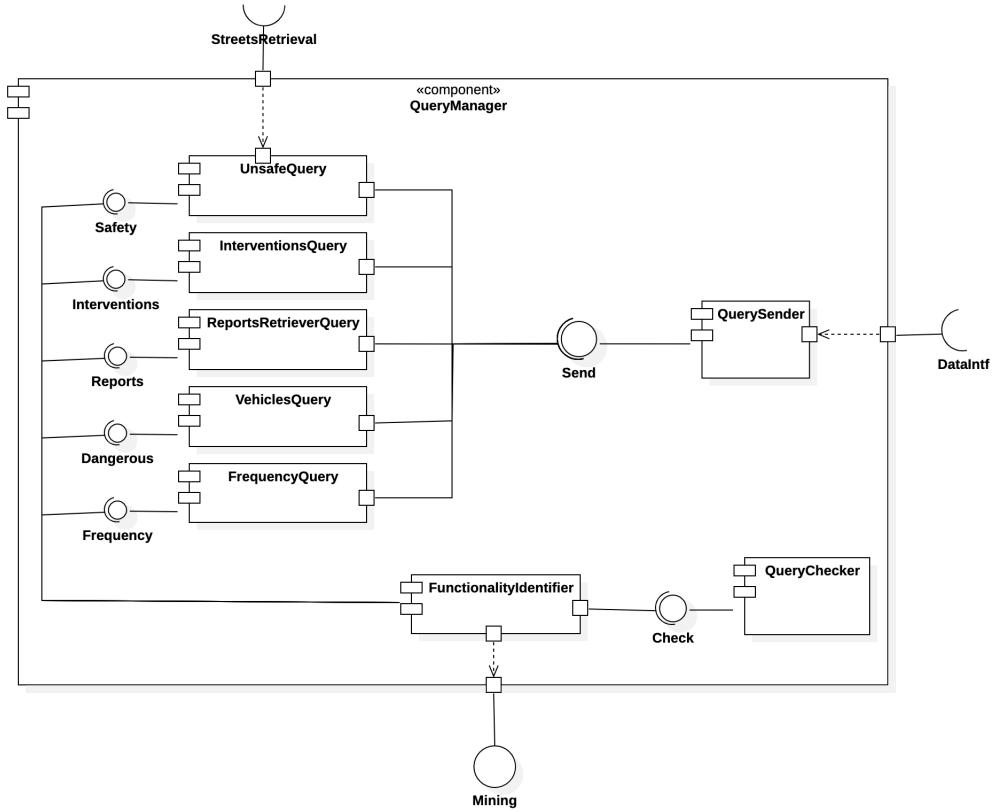

Figure 7: Query Manager Component Diagram

on the map. This component realizes the **Safety Interface** and needs to use the methods provided by the **StreetsRetrieval Interface**. The path retrieval and the JSON file that will be used to display the map are the result of an invocation of one of the methods of the last interface presented.

- **InterventionsQuery:** it is the component that manages the query related to the suggestion for possible interventions functionality. Hence it will prepare the query over the city selected and retrieve all the suggestions for it. This component realizes the **Interventions Interface** used by the *FunctionalityIdentifier*.

- **ReportsRetrieverQuery:** it is the component that manages all the functionalities that require to retrieve the reports from a request made by an authority. For this reason it handles both the *Check Unread Reports* and *Find Reports* functionalities provided to the authorities, that are nothing more than a query over the database retrieving the violations.

  – **Check Unread Reports:** it is the functionality that allows the authority to be notified of new parking violations. Thanks to the

unread flag that is stored in the system whenever a new notification is accepted we are able to retrieve for every authority its notifications and display them for this functionality.

- **Find Reports:** it is the functionality that provides the authority with a different level of visibility the retrieval of the notifications. In fact, while the user is able to see only some statistics thanks to the basic functionalities, in this way the authorities are able to see in detail each of the reports of their competence.

This component realizes the **Reports Interface** used by the *FunctionalityIdentifier*.

- **VehiclesQuery:** it is the component that manages the query related to the dangerous vehicles functionality. Hence it will prepare the query with the filters already checked and retrieve the type of dangerous vehicles. This component realizes the **Dangerous Interface** used by the *FunctionalityIdentifier*.
- **FrequencyQuery:** it is the component that manages the query related to the violations frequency functionality. Hence it will prepare the query with the filters already checked and retrieve the streets with most violations. This component realizes the **Frequency Interface** used by the *FunctionalityIdentifier*.
- **QuerySender:** it is the component that allows to send the query to the *DataManager*. Thanks to the **Send Interface**, in fact, the five components just presented will send the query they have prepared to the *DataManager* who will execute it over the database of the system.

2.2.5 Report Manager Component

The *ReportManager* (Figure 8) is the component that manages the notification process of the system, hence the one that we called the core functionality of SafeStreets. We have to remark that the notification process starts when a report arrives from a user and then the methods of the **Notification Interface** are invoked and ends when all the information about the report are stored in the system. At the end of the process each notification stored is marked as unread as it can be retrieved by the competent authority whenever it requires to benefit of the check unread reports functionality.

As we have already said in the description of the functionality we will need to complete the data provided by the user with some controls and additions that will be performed thanks to the three interfaces that we can see are used by this component:

- StreetRetrieval Interface
- IRI API
- DataIntF Interface

The components that allow the *ReportManager* to handle the notification process are:

- **ReportValidator:** it is the main component that deals with the notification process of the user-side. Once a report is notified by a user, all the information related to it will be delivered to this component by the *ClientHandler* thanks to the **Notification Interface**. Before storing the information, it has to be managed in order to add the missing details and validate its content; these two further considerations are managed with the methods provided by these interfaces:
  - **HandlePosition:** it is the interface that retrieves the street providing the GPS position of where the parking violation occurred.
  - **PlaceRecognition:** it is the interface that retrieves the plate of the reported vehicle providing the images of the infraction and the possible additional number inserted by the user. It is important to remark once more that the result of the methods provided by this interface will determine whether to discard or not the notification.
  - **Check:** it is the interface that allows to check if there exists already the same notification in the system. As we have already said, it is useless to store twice the same notification, in the case of a duplicate the current one will be discarded.
  - **Store:** it is the interface that allows to store the notification once the data completion and the validity have been carried out by the components.

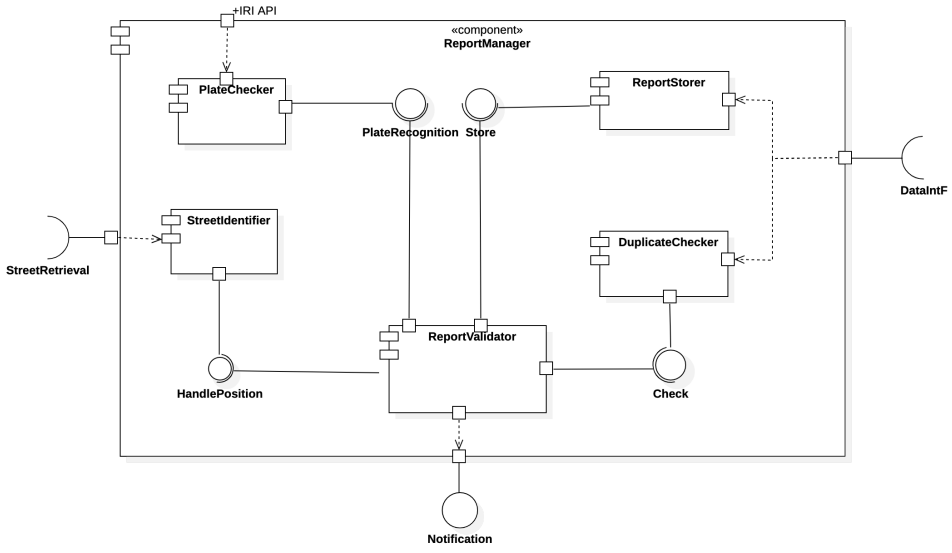

Figure 8: Report Manager Component Diagram

- **Street Identifier:** it is the component that manages to retrieve the street where the violation occurred using the GPS position provided in the notification. This component realizes the **HandlePosition Interface** thanks

to the methods provided by the **StreetsRetrieval Interface**. These are the real ones that execute the algorithms provided by the external map management system.

- **PlateReader:** it is the component that runs the algorithms provided by the IRI API in order to recognize the plate of the vehicle reported. It realizes the **PlateRecognition Interface** that is used to benefit of this result, fundamental to validate the notification.
- **DuplicateChecker:** it is the component that manages to check whether the reported violation is already stored in the database or not. It realizes the **Check Interface** which provides the methods to perform this additional control before storing the notification.
- **ReportStorer:** it is the component that manages to store the notification once it is ready (remember the unread flag). It realizes the **Store Interface** and uses the methods provided by the **DataIntF Interface** in order to store the data relative to the reporting notification.

2.2.6 Map Manager Component

The *MapManager* (Figure 9) is the component that deals with all the map issues. We have already presented them while describing the MI API (section 2.2); we are going to see now the components that manage these issues thanks to the methods provided by the external map management system. The functionalities of SafeStreets will be managed by the other components using the methods provided by the **StreetsRetrieval Interface** realized by this component.

The components that allow the *MapManager* to handle all the geographical and map issues inside the system are:

- **StreetFinder:** it is the component that allows to retrieve the street where the parking violation occurred once the GPS position is provided. It realizes the **StreetsRetrieval Interface** thanks to the methods provided by the **GPS Interface**.
- **GPSPositionHandler:** it is the component that runs the algorithms provided by the MI API in order to realize the **GPS Interface**.
- **MapRequest:** it is the component that deals both with the retrieval of the best path between two positions provided by the customer and the mapping of each street with the correspondent safety. The shortest path is managed with the methods provided by the **Path Interface** while the mapping is carried by the the methods of the **Map Interface**.
- **ShortestPathHandler:** it is the component that runs the algorithms provided by the MI API in order to realize the **Path Interface**.
- **MapUpdater:** it is the component that allows the system to have the information of all the streets and cities it has to manage inside its database. It Thanks to this information provided by the MI API, this component will periodically ask if some updates have occurred on the maps and in

case store them inside the database thanks to the interaction with the **DataIntF Interface**.

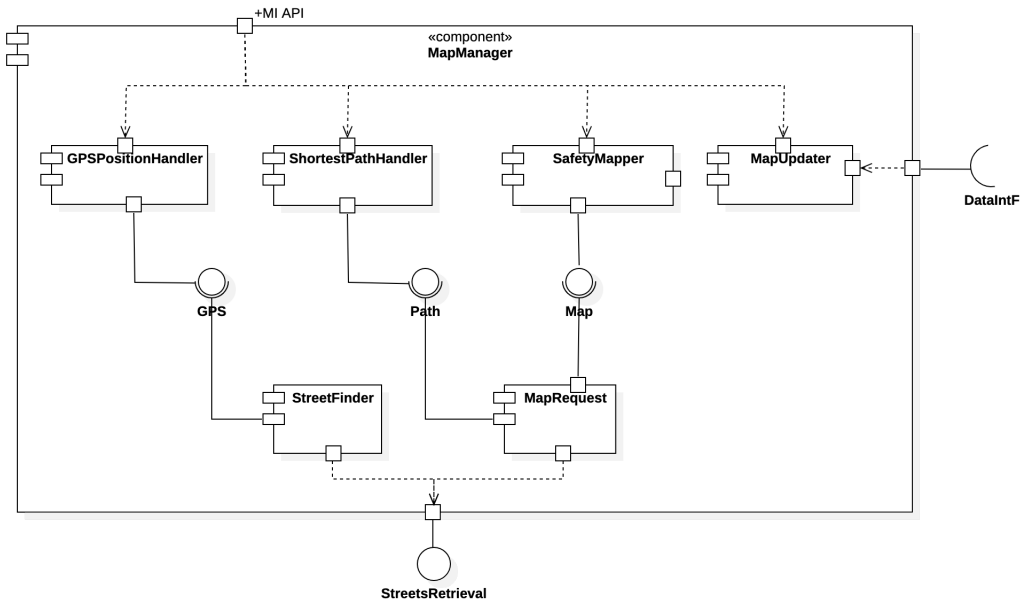

Figure 9: Map Manager Component Diagram

- **SafetyMapper:** it is the component that runs the algorithms provided by the MI API once it has retrieved all the information about the safety of the streets requested by the customer. It realizes the **Map Interface** that allows to benefit of the methods that return the data of each street together with its safety in order to be displayed to the client. Hence, this is the component that receives the JSON file containing all the information needed to display a map. It parses this data and stores as it is precisely described in the data view section (2.3).

2.2.7 Safety Manager Component

The *SafetyManager* (Figure 10) is the component that deals with the safety calculus that has to be carried out every day considering the previous thirty days in order to have the data for the safety functionality. As precisely described in the RASD document [2] we have defined a clear way to determine the safety of each street. In order to carry on this calculus we have to consider this component that periodically will run its algorithms in order to update the safety.

As we see, no interface is provided by this component, as we have just said in fact it is independent as it needs to run periodically by just retrieving the data over the notifications and accidents and update the safety of the streets.

It is important now to describe once more how the safety update works as we can also understand how the components that compose the *SafetyManager* interact in order to obtain this functionality. The system dynamically changes day per day the safety of the streets, considering the parking violations data relative of the previous thirty days, thanks to two different kind of thresholds. For each street if at least one threshold is exceeded it means that it is **unsafe** otherwise it will be considered **safe**. The thresholds are:

- **Intervention Threshold:** each intervention is defined with a threshold. On every street, every intervention would be possibly chosen as a suggestion. Each type of parking violation or accident, if occurred in a certain street, augment the threshold of the interventions linked with that type (hence the same violation or accident may augment at the same time two different interventions). If an intervention's threshold exceeds it means that the street is no longer **safe** and that its safety has to be updated. In this way counting each day for every street if a threshold for at least one intervention is exceeded allows to determine both the safety of a street and also the possible interventions to be suggested.
- **Sum Threshold:** in order to consider also the case when no accident is linked to any intervention (because of the fact we are just considering the parking violations and the accident may be caused by another reason, e.g.: the speed limit) for each street is also defined another threshold that independently from the type determines a street to be **safe** or not. This threshold augments whenever an accident or a parking violation occurs in the street and thus it is simply determined as the sum of the accidents and parking violations. In this way, counting each day for every street if the sum threshold is exceeded, it allows to determine the safety independently from the type of the accidents that have occurred.

Now that we know better how the safety process works we are able to describe the components that compose the *SafetyManager* in order to provide this functionality:

- **SafetyUpdate:** it is the core component that deals with all the computation related to the calculation of the safety that we have said to happen periodically every day. In order to determine whether an interface is exceeded or not it uses the **InterventionsSafety Interface** and the **SumSafety Interface** that allow to determine for a street if one of the thresholds have exceeded. In order to access the data of the reports and the accidents the component uses the **Reports Interface** and the **Accidents Interface** to carry out the computation. Hence every day a new block of reports and accidents will be considered in the computation and the oldest one will be discarded; once the safety has been determined, the *SafetyUpdate* stores the information thanks to the **Store Interface** and its functionality will be needed the next day.
- **InterventionsManager:** it is the component that realizes the **InterventionsSafety Interface** providing the methods to determine the safety of a street thanks to the thresholds of the interventions linked to the parking violations and the accidents that occurred in that street.

- **SumManager:** it is the component that realizes the **SumSafety Interface** providing the methods to determine the safety of a street thanks to its sum threshold; it counts the number of parking violations and accidents that occurred in that street.

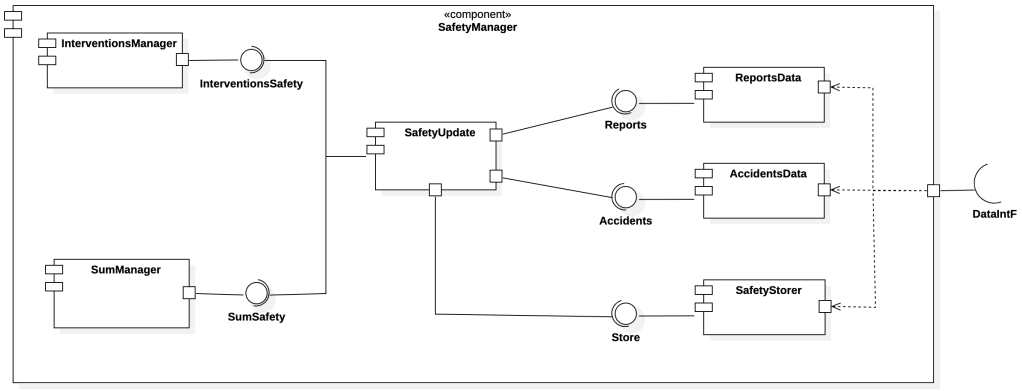

Figure 10: Safety Manager Component Diagram

- **ReportsData:** it is the component that allows to obtain the information about the type of the parking violations that occurred in a certain street by providing the **Reports Interface** as they can be considered for the safety update.
- **AccidentsData:** it is the component that allows to obtain the information about the accidents that occurred in a certain street by providing the **Accidents Interface** as they can be considered for the safety update.
- **SafetyStorer:** it is the component that is able to store the new information about the safety of each street once it has been updated. It realizes the **Store Interface** and needs to use the methods provided by the **DataIntF Interface** in order to complete the storing action.

2.2.8 Accidents Manager Component

The *AccidentsManager* (Figure 11) is the component that deals just with the management of the accidents that have been stored by the authorities onto the common interface. As we see only, two components are needed: one that receives the accidents by using the methods provided by the **Authority API** and the other which stores the accidents that have been retrieved.

The components that allow the *AccidentsManager* to handle the process of receiving the accidents provided by the authorities are:

- **Accidents Receiver:** it is the component that uses the methods provided by the **Authority API** in order to retrieve all the accidents that each authority has published. In this way, by updating this type of data periodically we will be able to have always all the information we need to

determine the safety of the streets. Once the accidents data is retrieved, this component is ready to store it in the database thanks to the **Store Interface**.

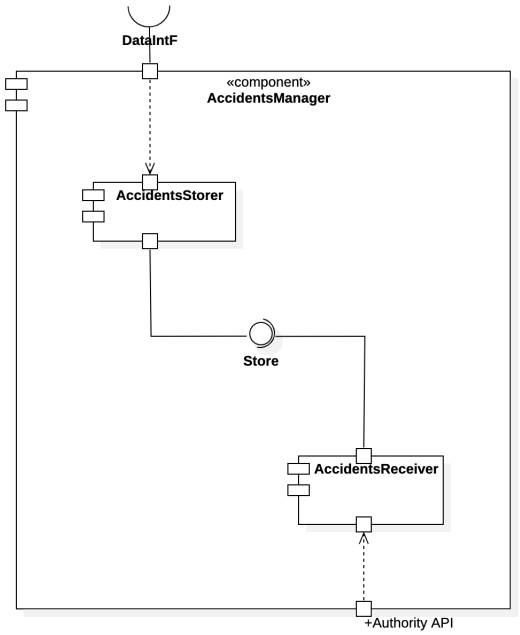

Figure 11: Accidents Manager Component Diagram

- **AccidentsStorer:** it is the component that deals with the storing process of the accidents. It realizes the **Store Interface** and needs to use the methods provided by the **DataIntF Interface** in order to store the accidents in the database of the system.

2.2.9 Data Manager Component

The *DataManager* (Figure 12) is the component that deals with all the management of the data needed by the system in order to perform its functionalities. As we can see in the general diagram of the server (Figure 4), almost every component needs to use the interface provided by this one.

Hence the *DataManager* realizes only the **DataIntF Interface** which provides the methods for the management of all the information and uses the methods provided by the **DBMS API** in order to perform on the database the storing or requesting actions needed.

The components that allow the *DataManager* to handle all the operations that involve the management of the data stored in the system's database are:

- **SafetyManager:** it is the component that provides all the methods that allow to manage the data relative to the mapping of the safety on the streets required, realizing the **DataIntF Interface**.
- **CustomersManager:** it is the component that provides all the methods that allow to manage the data relative to the customers (in particular for the access process), realizing the **DataIntF Interface**.
- **ViolationManager:** it is the component that provides both methods that allow to manage the data relative to the notification process and the ones relative to the basic functionalities, realizing the **DatIntF Interface**.
- **InterventionsManager:** it is the component that provides all the methods that allow to manage the data relative to the interventions, realizing the **DataIntF Interface**.
- **AccidentsManagerInterface:** it is the component that provides all the methods that allow to manage the data relative to the accidents, realizing the **DataIntF Interface**.

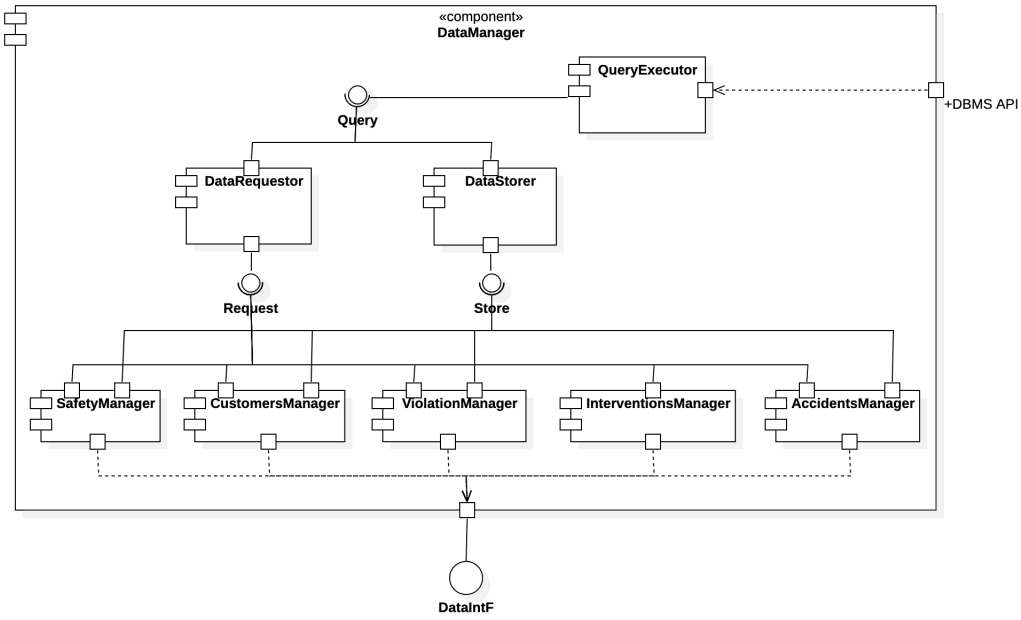

Figure 12: Data Manager Component Diagram

- **DataRequestor:** it is the component that allows to build a query that will obtain as a result the information required by one of the managers previously described. It realizes the **Request Interface** and needs to use the **Query Interface** in order to execute the query that has been built.
- **DataStorer:** it is the component that allows to build a query that will store in the database the data provided by the component that invokes

its methods. It realizes the **Store Interface** and needs to use the **Query Interface** in order to execute the query that has been built.

- **QueryExecutor:** it is the component that finally executes the query received either from the *DataRequestor* or the *DataStorer*. It realizes the **Query Interface** and needs to use the methods provided by the **DBMS API** in order to manage all the data stored inside the database of the system.

2.2.10 Web Server Component

The *WebServer* (Figure 13) is an independent component, out of the server, that deals with the interaction between the web app and the business logic of the system. It realizes the **WebAppInterface** in order to provide the endpoint for the authorities to access SafeStreet with the web application displayed on their browser. It needs to use the methods provided by the **WebInterface** in order to establish the interaction between the authorities and the system.

The components that allow the *WebServer* to handle the interaction process between the requests coming from the web app and the business logic are:

- **Presentation:** it is the component that deals with the graphical interface that is needed to provide the authorities the access to the system. It realizes the **WebAppInterface** as it defines all the methods and scripts the browser will use to display the information. In order to manage the interaction, it handles the actions performed by the authority with the **Commands Interface** and sends the requests to the system as they can be processed thanks to the **Requests Interface**.

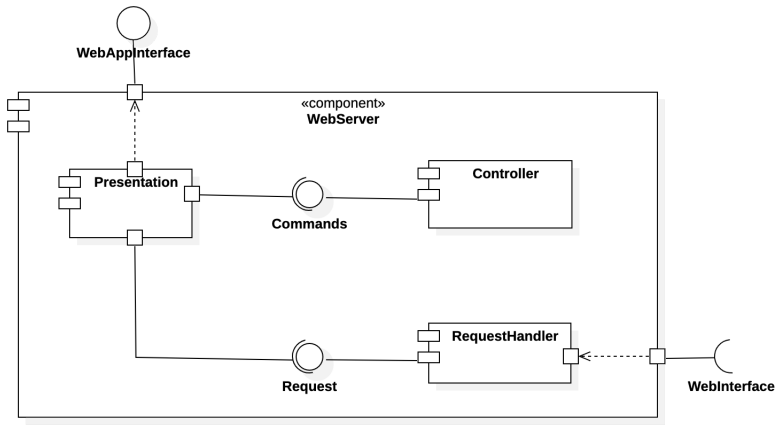

Figure 13: Web Server Component Diagram

- **Controller:** it is the component that deals with the logic needed to handle the actions performed by the authority. By providing the **Commands Interface** it allows to manage the flow of the web application displayed on the browser of the authority.

- **RequestHandler:** it is the component that deals with all the requests coming from the web application. Thanks to the **Request Interface** realized, it is able to process the incoming request and send it to the business logic with the methods of the **WebInterface** as the functionality required will be displayed in the web application.

2.2.11 User App Component

The *UserApp* (Figure 14) is the other independent component that will be installed on the device of the users in order to provide the interaction with the system. Differently from the *WebServer* this is already the component that will manage the actions performed by the user as it represents the mobile software application. For this reason it only needs to use the methods provided by the **MobileAppInterface** to send the requests to the system as they can be processed.

The components that allow the *UserApp* to handle the interaction process between the requests coming from the mobile app and the business logic are:

- **Presentation:** it is the component that deals with the graphical interface that is needed to provide the users the access to the system. As we have just said the realization of this component is the interface that allows the user to interact with the system; in order to do so it needs to use the methods provided by the **Commands Interface** and the **Request Interface**.

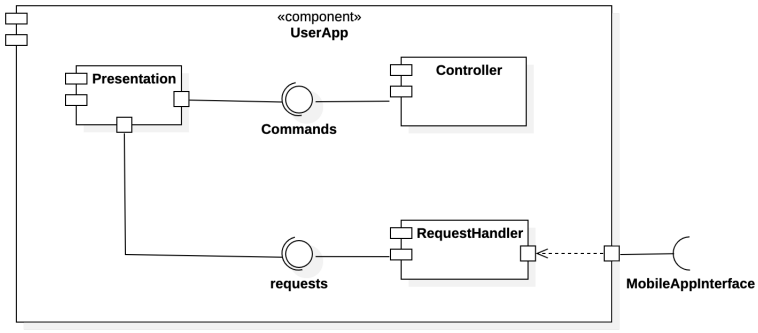

Figure 14: User App Component Diagram

- **Controller:** it is the component that deals with the logic needed to handle the actions performed by the authority. By providing the **Commands Interface** it allows to manage the flow of the mobile application displayed on the device of the user.
- **RequestHandler:** it is the component that deals with all the requests coming from the mobile application. Thanks to the **Request Interface** realized, it is able to process the incoming request and send it to the business logic with the methods of the **MobileAppInterface** as the functionality required will be displayed in the application.

2.3 Database View

Before giving the description of how the components now presented will be deployed in our system, it is also important to define a model that allows to manage the huge amount of data we expect to deal with. As we will also motivate later, in section 2.7, we decided to use a relational approach to store the information we need to manage. In order to do so we decided to use a class diagram model rather than an entity relational one as we could give a continuous representation of what we already presented in the RASD document [2]. The class diagram of next page, in fact, has been updated with more detailed considerations that we were now able to consider, being at the design level.

As we can see in the picture (Figure 15) some classes and relations have been updated, in particular for what concerns the maps issues and the safety ones. In particular:

- The classes that we are going to use to store the position issues have been updated. We removed the **Position** class as it is not relevant anymore while we added the **Cross** class that is important for the transportation of the data relative to a map to be highlighted. The **Path** class has also been updated to consider a possible way to store the information relative to a path provided by the MI. The **Street** and **City** classes are still considered as we have said that we need to store in our system their information in order to deal with the functionalities the system has to provide.
- The most important update is the **Map** class. This class is the result of the invocation of the methods that deal with the *UnsafeStreets* functionality. What happens, in fact, when a request of this type arrives to the business logic is: first retrieve in the database all the requested streets and their safety, second, provide them to the methods of the MI that will return the JSON file containing all the relevant information for a map to be displayed. Thanks to this class, we will easily parse the file and store its values in the class attributes.
- The **Safety** class has also been removed as it was used in the initial class diagram to define more precisely which were the relations that allowed to determine its value for the streets stored in the system. We decided now to simplify the safety considerations, even to reduce the complexity of the algorithms that will determine it, by storing in each street a value that is *true* if the street is safe, otherwise *false*.

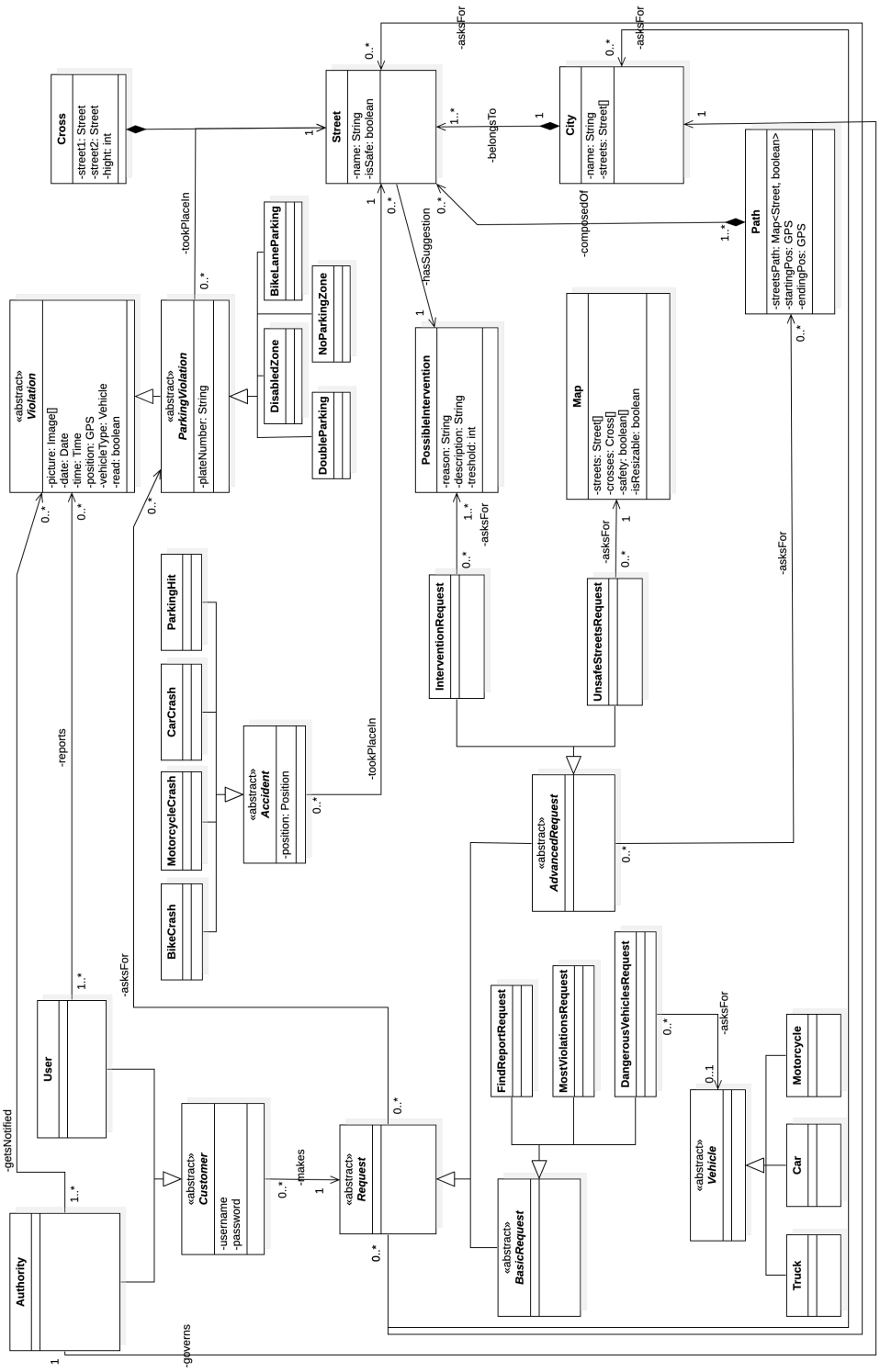

Figure 15: Class Diagram Model

To conclude with this section that aims to provide an overview of which type of data will flow inside our system, we want to give a practical example of how the objects of the **Map** class will be instantiated once we receive as a result the JSON file from the MI. What we want to achieve is a one to one correspondence between the file and the class as we can store all the information with just one method invocation of some parsing library. Hence, what we aim to is something like presented in the following picture:

Figure 16: Map-Class and Json-File Correspondence

Thanks to this information we will be able to use the map objects and display for our clients the safety they are requiring, in fact:

- The *streets* are the ones that have to be displayed
- The *crosses* establish how the streets are disposed
- The *safety* array specifies the safety of the crosses
- The *isResazible* attribute specifies if the map is resizable or not

2.4 Deployment View

As we have seen in the overview (2.1) and component view (2.2) sections, we have designed a system that needs to interact with many agents. In order to define a crowd-sourced application where a lot of information needs to be managed as it can be used to provide different services, we decided to use a **client-server** architectural approach. As we may have guessed in the high-level component diagram of Figure 3 the business logic is represented by the server component while the clients are designed with two different technologies in order to have the best application software to make the customers interact with the system.

The client-side system is composed of:

- **Mobile Application:** for the users
- **Web Application:** for the authorities

Hence, to make this decision possible we need to consider a web server that will manage all the interactions between the web application and the internal system, while we can make directly interface the mobile application with the internal system thanks to the software application installed on the devices of the users.

To accomplish the considerations just presented we decided to design the **client-server** paradigm with a **four-tier architecture** in order to have a clear separation between the clients and the servers that manage them; the decoupling of the database shown in the composition diagram (Figure 2) enforces once more this decision. All the communications between the system and the customers (both users and authorities) are handled via HTTP messages in a RESTful way (HTTPS is shown in the diagram for obvious security reasons). All the considerations just reported about some architectural and communication choices will be clearly precised in the devoted section (2.7).

Let's focus now on the illustration of the four-tier architecture as we can understand how the components that we have previously described are mapped with the tiers of the system. Then, in the next section we first illustrate the decision tree we used while designing the system as we can later give the complete description of the deployment.

The following picture (Figure 17) illustrates the tiers of the system needed in order to deploy the components identified in the previous section.

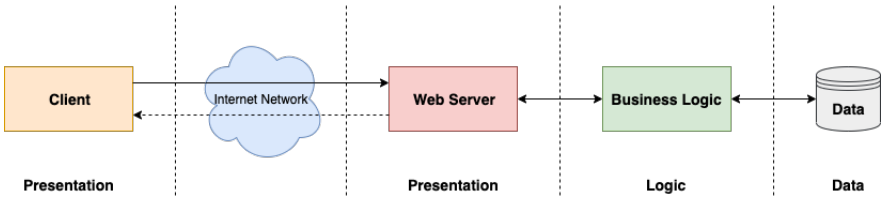

Figure 17: Four Tier Architecture Diagram

Components Mapping Now that we know which are the tiers and which are the components, we are able to map them together as we can clearly understand how the design is structured before starting with the deployment diagram.

- **Client:** are the components that allow the interaction of the customers with the system in order to benefit of its functionalities.
  - AuthorityWebApp
  - UserApp
- **Web Server:** it is the layer that allows the interaction between the web application used by the authorities and the system.
  - WebServer
- **Business Logic:** is composed of all the components that allow the computation needed to provide the functionalities of SafeStreets. Hence it is made of all the components inside the server one that are:
  - ClientHandler
  - AccessManager
  - QueryManager
  - ReportManager
  - MapManager
  - SafetyManager
  - AccidentsManager
  - DataManager
- **Data:** is the layer that deals with the management of all the data needed by the system in order to provide its services.
  - DBMS

The diagram of next page (Figure 18) provides a graphical illustration of what we have just said by coloring the corresponding layer with the same color of the components highlighted in the general component diagram already seen.

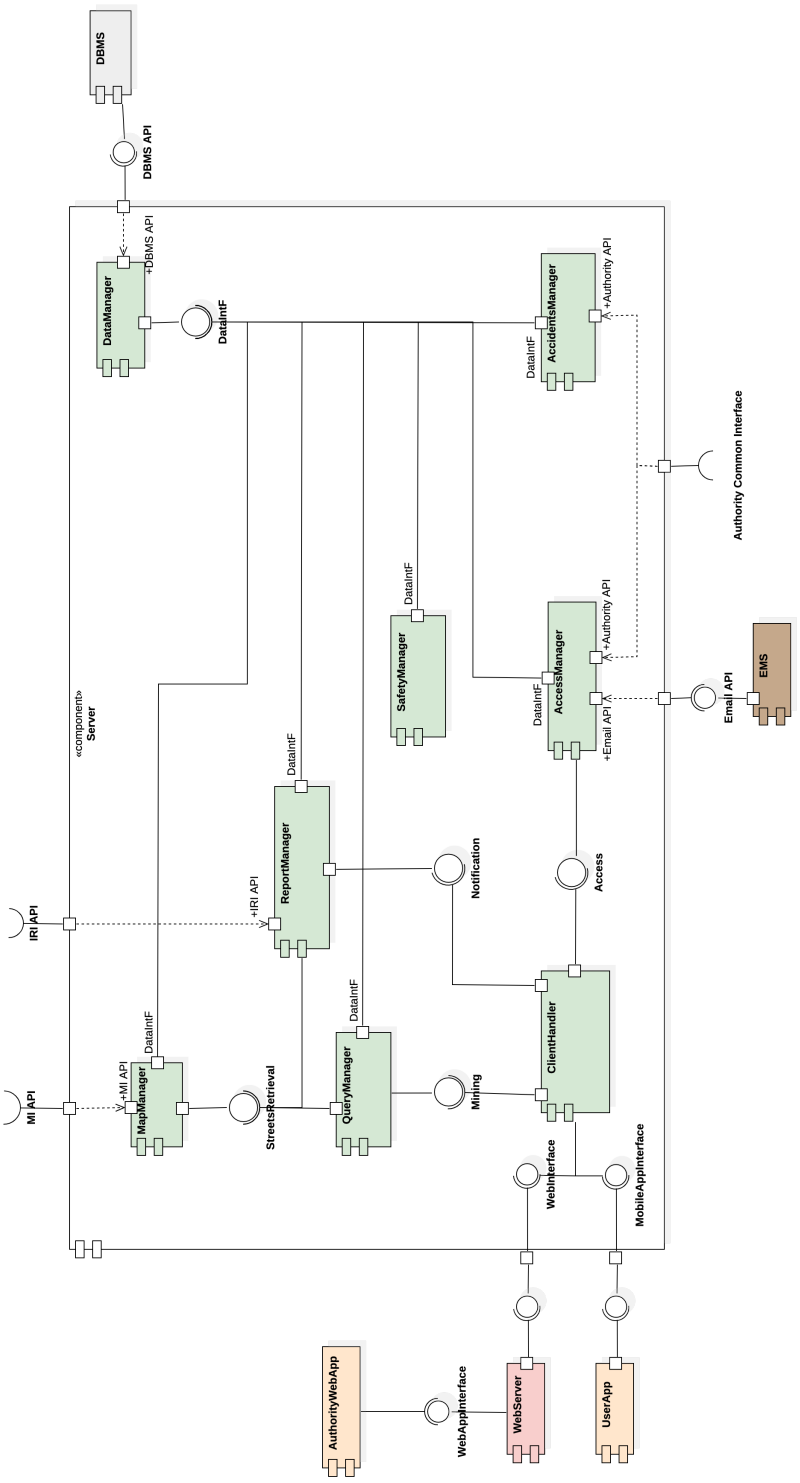

Figure 18: Components Mapping Diagram

2.4.1 Decision Tree

Now that we have established how our system is going to be deployed at the highest level possible, before providing the most detailed deployment diagram we present the decision tree we developed while defining the design process as we can justify the diagram of the next section. As we can see in the picture (Figure 19) we started with a **client-server** decision and then refined it with further more detailed decisions.

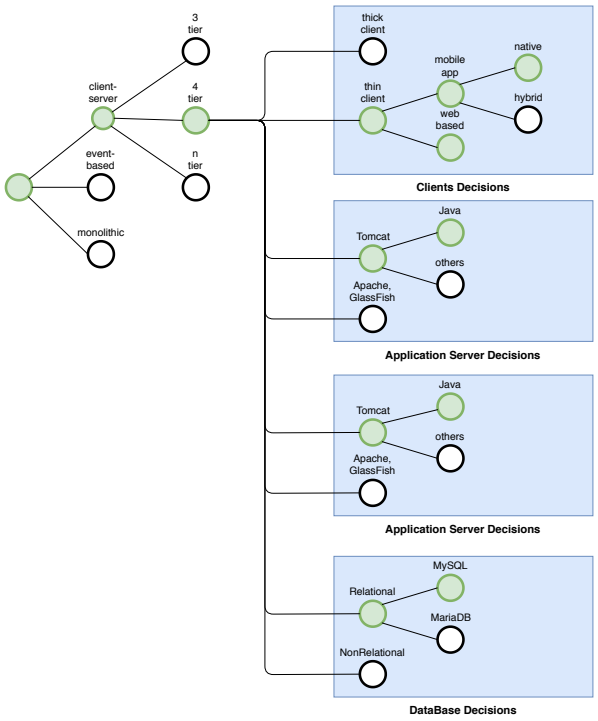

Figure 19: Decision Tree

2.4.2 Deployment Diagram

In the diagram of the following page (Figure 20) we illustrate a precise description of how and where the components identified are deployed. We still use the colors related to the components now for the nodes that involve them.

The communication is presented only between the nodes and not for the components as we already have a precise description of their interfaces that realize the interaction in the component view section (2.2). The **HTTPS** and **TCP/IP** protocols are used to connect respectively: a connection to a REST API or to a service providing system.

All the motivations for the architectural styles and patterns but also the implementation choices that we can see in the picture will be clearly motivated in the devoted section (2.7).

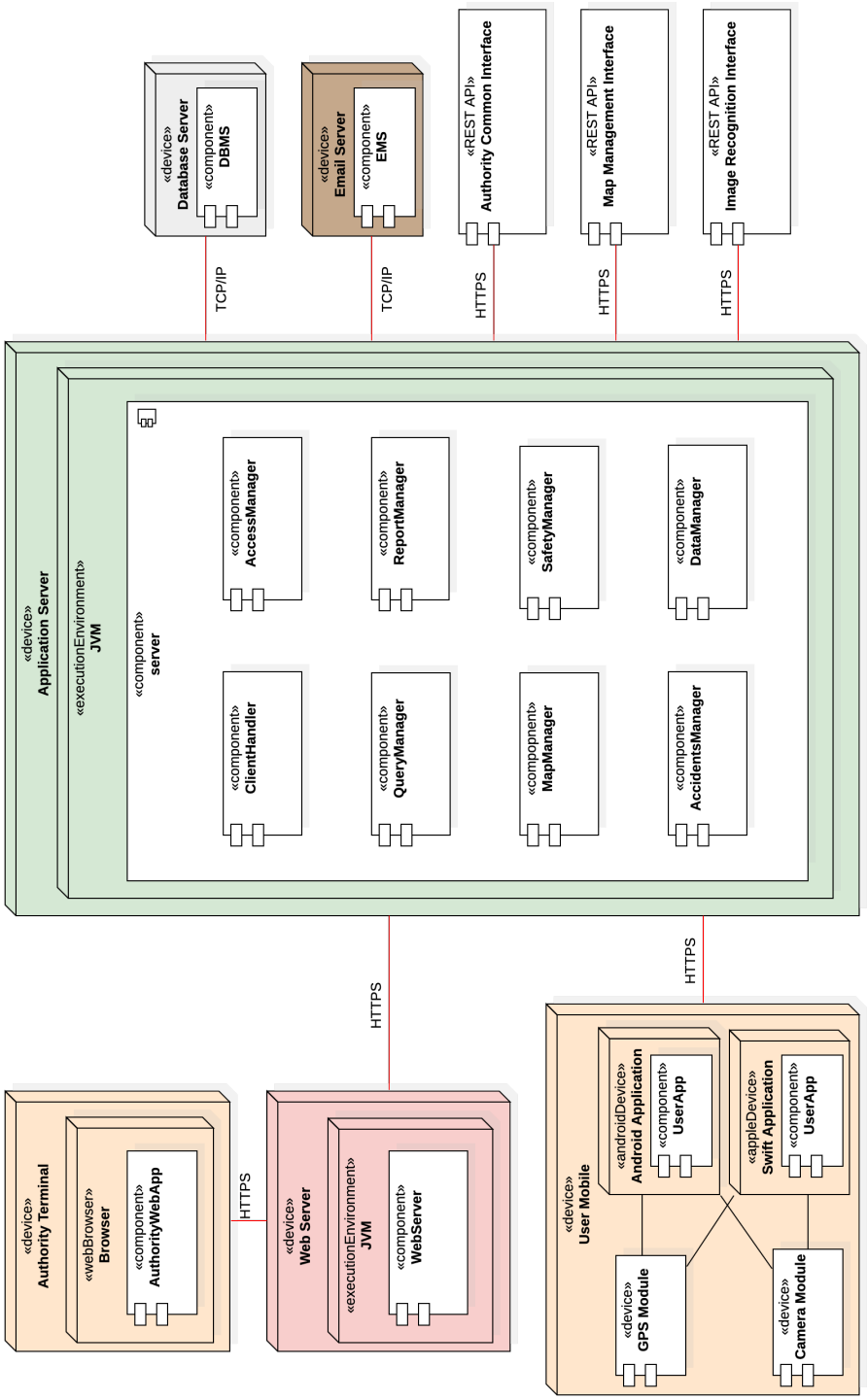

Figure 20: Deployment Diagram

2.5 Runtime View

In this section we are going to present the most important runtime views of the interactions between customers and SafeStreets. All the methods used in the following diagrams will be precisely described in the next section (2.6) where each of them will be mapped to the respective interface that realizes it. For the sake of simplicity we decided to use the first level of components to describe the interactions, in this way we obtained more clear diagrams and their internal correspondence can be easily found with the satisfactory description of every sub-component that we identified in the component view section (2.2).

The *Runtime View Diagrams* of this section are presented first for the **users** and **authorities**. Then, for the common functionalities, we decided to show only the interaction between the UserApp and the system to keep the diagrams clear (the ones for the authorities would need an additional interaction with the WebServer before interfacing with the system) and lastly two critical managers are shown in order to present how the components related to the safety and the map issues work.

2.5.1 User

- **Login:** In this sequence diagram (Figure 21), the process of user login is shown. The authority login is not shown in the runtime view section, since it is basically the same. In fact, both users and authorities log in with a username and a password. This process is actually quite simple: the UserApp asks the server to log in by passing username and password. ClientHandler receives the request and forwards it to the AccessManager component, that will ask DataManager for a simple query in the database. If no user with that username is found, an error message is given to the client, otherwise the check of the password will be performed. If the password inserted and sent by the user is the same as the one retrieved from the database, the login will be considered as logged in and then informed. The difference for the authorities is that their login requests will come from the AuthorityWebApp, instead of using the UserApp, thus the WebServer will be involved too, between the AuthorityWebApp and the ClientHandler.

- **Registration:** This sequence diagram (Figure 22) pictures the interaction through which a user registers to SafeStreets. After UserApp has asked him all the registration data (username, password, and other information), it will hand them to the server, hidden into a User object. ClientHandler will forward the request to AccessManager, that is the same component that manages the login interactions. AccessManager will first check if there is an already existing user who registered with that same username and in case will send back an error. If not, it will ask DataManager to store the user information into the database of the system.

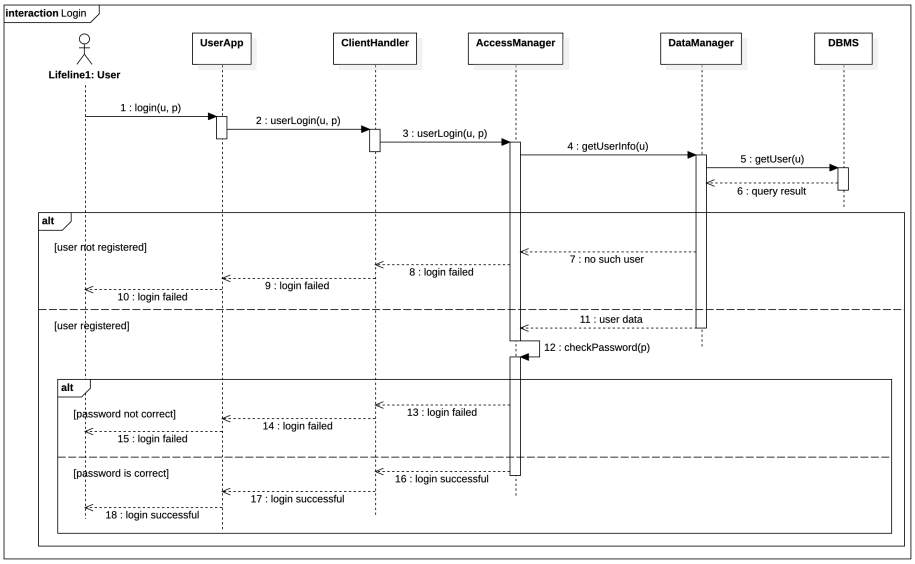

Figure 21: Login Runtime View

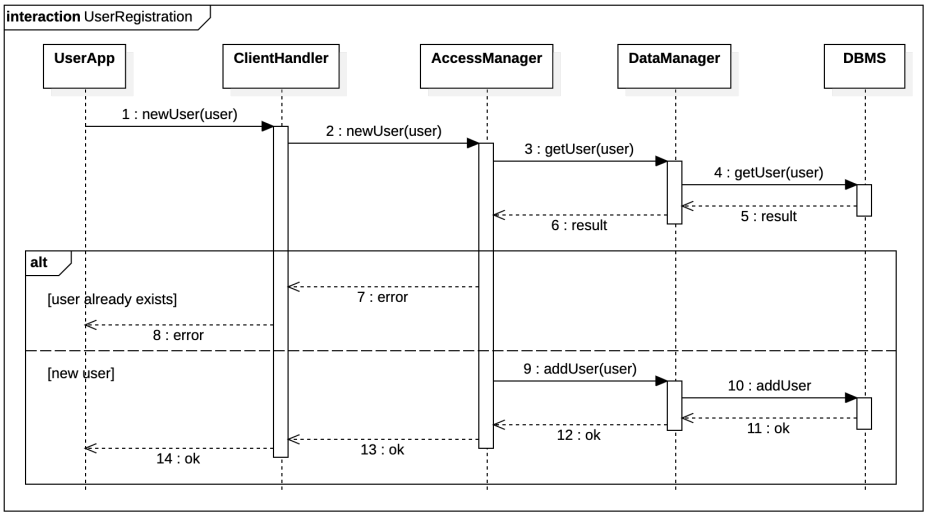

Figure 22: User Registration Runtime View

- **Report Violation:** Here (Figure 23), it is shown how a violation is reported by users to the system. UserApp sends all the information contained in a Violation object (pictures, location, ...). ClientHandler forwards the call to ReportManager, the component designed to handle the reports. It will indeed ask IRI for the plate number, providing it with the pictures sent by the user. Then, it will ask MapManager for the name of the street of the violation. MapManager of course needs to ask the external MI to accomplish this, and it only needs the provided location. When all the information is gathered, the report is ready to be stored into the database. To do so, ReportManager will ask DataManager to interact with the DBMS and update the related table(s).

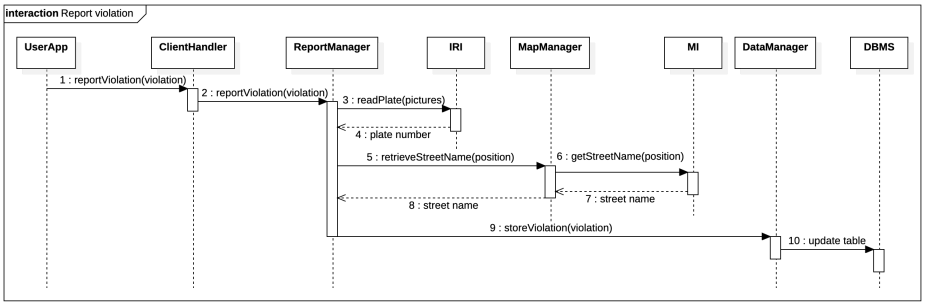

Figure 23: Report Violation Runtime View

2.5.2 Authority

The following are the interactions that involve authorities. In the sequence diagrams, all of these interactions start with a call from the WebServer but of course, the component that really start them is the AuthorityWebApp, that ask web pages to the WebServer. The interaction between the AuthorityWebApp and the WebServer is not shown here for the sake of simplicity.

- **Registration:** It is now described (Figure 24) the process through which an authority registers to SafeStreets. This is a little bit more complicated process with respect to the user one, because the system requires the authority to be real and recognized. The assumption here is that there's a completely secure PEC address assigning mechanism. The ClientHandler forwards the WebServer request to AccessManager. The request contains of course all the necessary information for the registration. AccessManager will then check whether there is already an authority registered for the city contained in the registration request. If a authority already exists, the request is rejected and an error message is sent back to the client, otherwise AccessManager will check whether the specified PEC address actually corresponds to the authority of the city contained in the request. This is to avoid that an authority registers for a different city. To do that, it needs to ask the Authority Common Interface that is external with respect to SafeStreets. If the addresses do not match, an error message is sent to the client and the procedure is aborted. If the specified address

is correct for that city, SafeStreets needs now to verify that the actual person that is making the request through the AuthorityWebApp is able to access the PEC address and actually read the received emails. In order to do that, AccessManager generates a code that will be sent by PEC through EmailAPI. AccessManager now waits for the authority to insert, through the AuthorityWebApp, the received code. Then, it will check that the generated code and the inserted one coincide. If they do not coincide, an error is sent to the AuthorityWebApp and the registration is aborted. If the inserted code is correct, AccessManager will ask DataManager to store the information into the database.

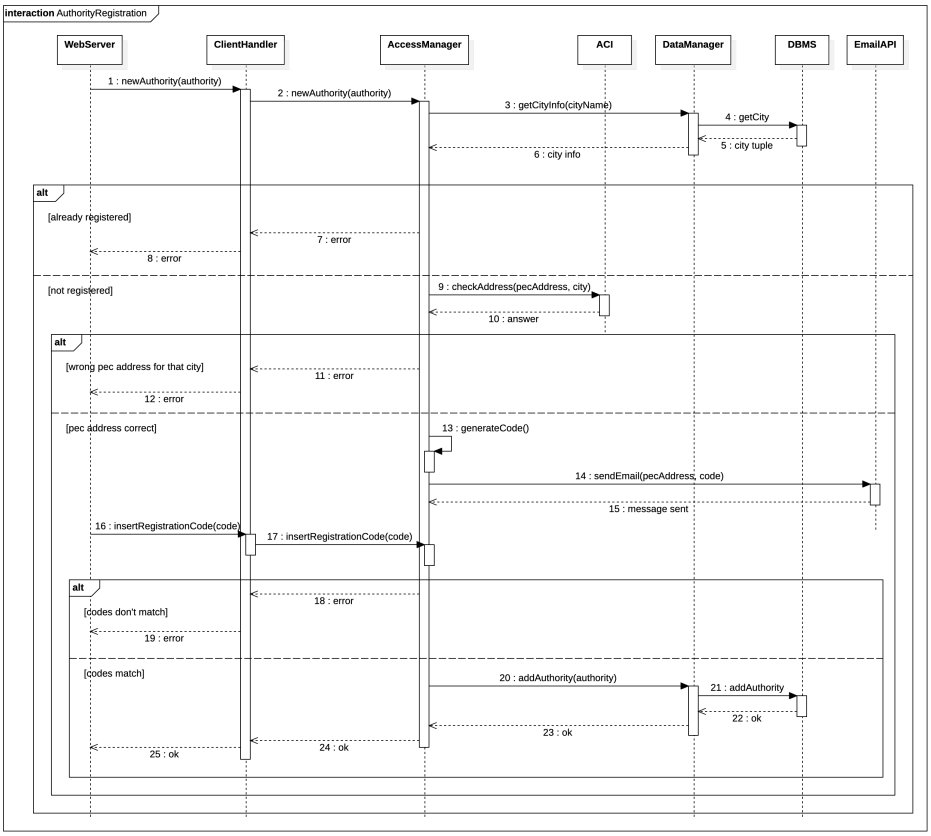

Figure 24: Authority Registration Runtime View

- **Check Unread Reports:** In this sequence diagram (Figure 25), it is shown how an authority is able to retrieve all the unread reports sent by users to SafeStreets. The WebServer will ask ClientHandler to forward the request to the right component, that in this case is again QueryManager. In fact, its job is to call the right method of DataManager to obtain only the reports that have the "read" flag set to false. DataManager will as usual execute the actual query into the DBMS. Finally, QueryManager will send the result back to the WebServer, that will forward it to the AuthorityWebApp.

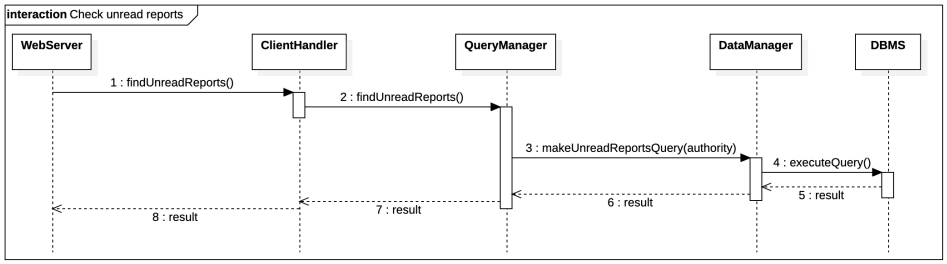

Figure 25: Check Unread Reports Runtime View

- **Find Reports:** Here (Figure 26) it is described the interactions through the components when the authority asks for some past reports applying some filters. The request is made by the WebServer and is forwarded to QueryManager by ClientHandler. After having checked if the specified filters are correct and eventually returning an error to the AuthorityWebApp, QueryManager will ask DataManager to interact with the DBMS and execute the query that will return all the reports located in the authority's city that match the specified filters.

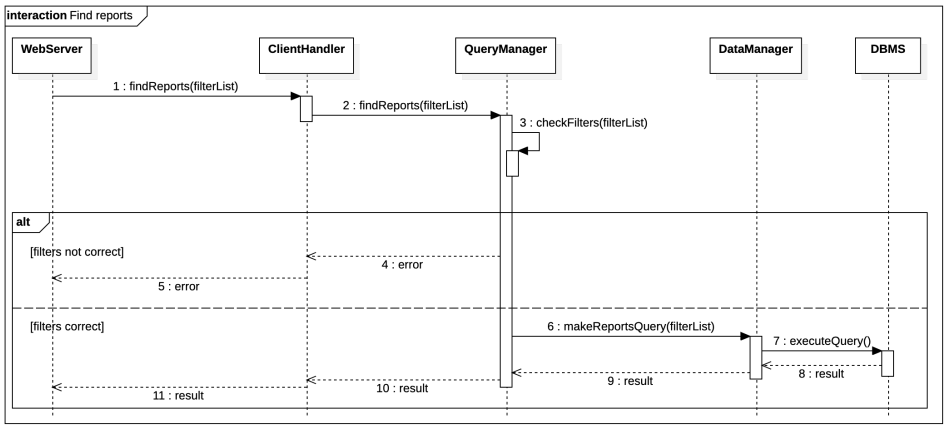

Figure 26: Find Reports Runtime View

2.5.3 Common Functionalities

These functionalities are available to both users and authorities. Therefore, they're reported here.

- **Find Streets With Highest Number of Violations:** It is now shown (Figure 27) how a possible query for the streets with the highest number of violations is made. The filterList object represents all the filters that are selected by the user, like time slots and types of violation. These filters are widely described in the RASD. The component designed to handle these queries is of course QueryManager. It will check the correctness of

all the used filters (e.g. at least a type of violation must be included). If some constraint is not fulfilled, an error is sent to the user, otherwise QueryManager will prepare the query and forward it to DataManager, that will actually execute it on the DBMS. The result is then back propagated through the involved components and to the UserApp.

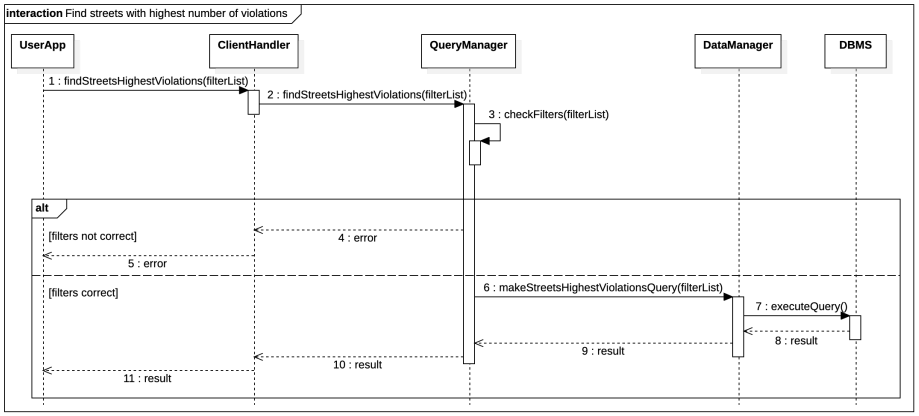

Figure 27: Violations Frequency Runtime View

- **Find Dangerous Vehicles:** This interaction (Figure 28) is basically the same as the previous one, thus it won't be explained again here. The only difference is that the query designed by QueryManager will be specific for finding the types of vehicle that made the most violations.

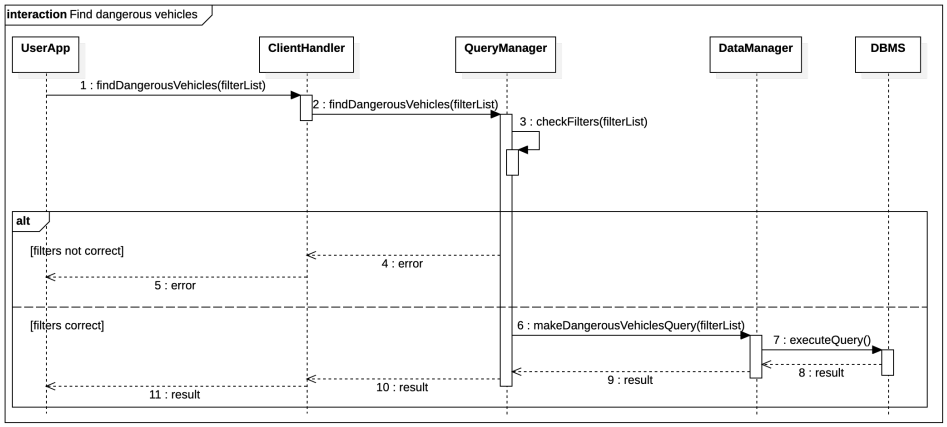

Figure 28: Dangerous Vehicles Runtime View

- **Find Unsafe Streets:** In this interaction (Figure 29) it is shown how the UserApp requests maps with coloured streets. The request made by the UserApp is forwarded by ClientHandler to QueryManager. Depending on the option selected by the user, three different methods could be called:

the one with a city as input, the one with starting and ending points, or the one with a single street. In the first case, QueryManager will look for all the streets of the selected city in the DBMS, going through DataManager. In the second case, a path needs to be generated, thus QueryManager will ask MI through MapManager for a path, that will include a certain number of streets. In the last case, no action is required. Then, for all the gathered streets, QueryManager will search in the database for their safety, asking DataManager to execute the actual query. Then QueryManager will ask MapManager for the generation of the map that, according to the gathered infomation about the safety of the involved streets, will be coloured as described in RASD. If path option was selected, the path will be passed to MapInterface so that the generated map will show it. This map will be finally sent back to the UserApp and displayed.

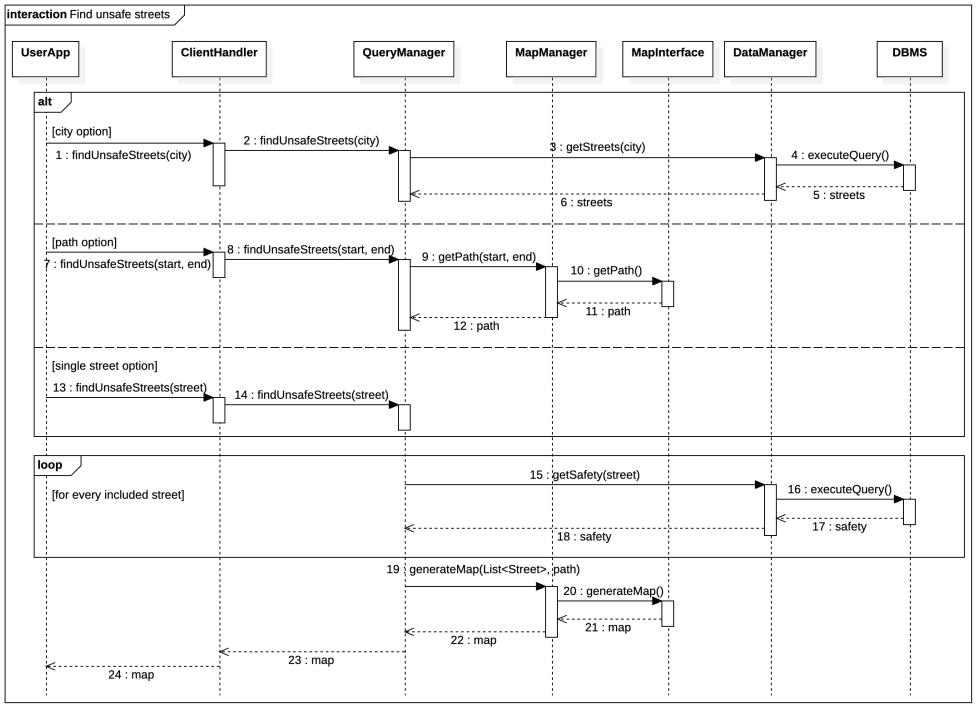

Figure 29: Unsafe Streets Runtime View

- **Find Urgent Interventions:** This interaction (Figure 30) is basically the same as "find streets with highest number of violations", thus it won't be explained again here. The only difference is that the query designed by QueryManager will be specific for urgent interventions.

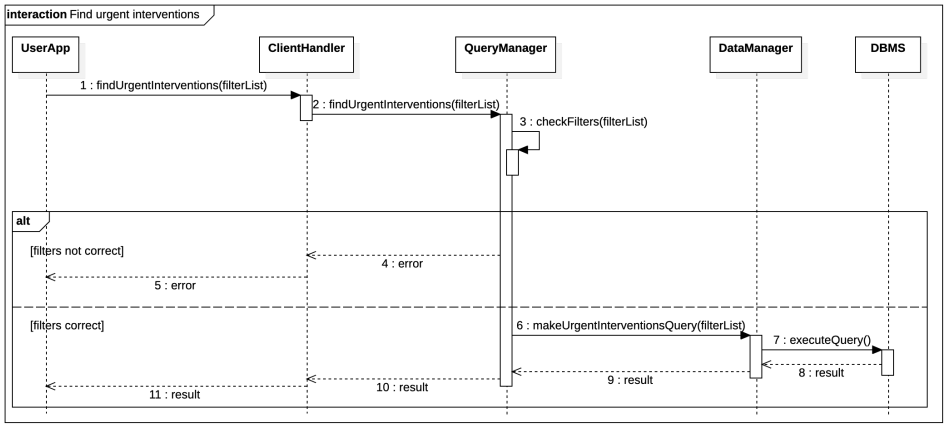

Figure 30: Urgent Interventions Runtime View

2.5.4 Safety Manager

- **Update Safety:** This (Figure 31) is the description of the process of updating the safety estimation of every registered street, along with the suggested possible interventions. The component designed to do this operation is SafetyManager. It asks DataManager to get all the violations and accidents information from the DBMS. Then it will calculate the safety estimation and all the possible interventions for that street as described in RASD. Finally, it will store safety estimation and all those possible interventions in the database, so that it will be possible to always retrieve the most updated information without actually calculating it for every request that is made.

- **Update Accidents:** Here (Figure 32) it is described how SafeStreets uses the Authority Common Interface to retrieve data about accidents. For every city with a registered authority, AccidentsManager will ask ACI for these data and then will ask DataManager to store them into the database.

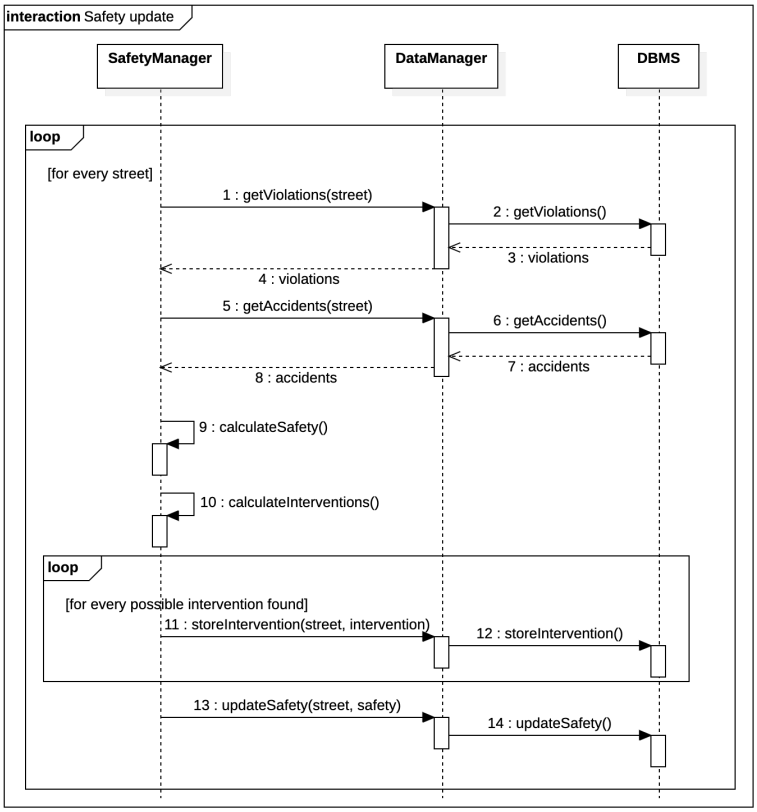

Figure 31: Update Safety Runtime View

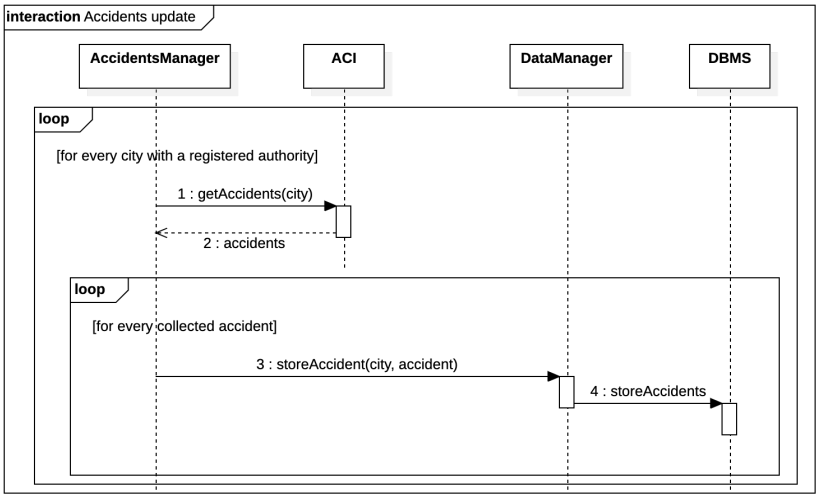

Figure 32: Update Accidents Runtime View

2.5.5 Map Manager

The map manager has to deal with a particular component that deals with the updating of the information related to the maps it has stored in the databases. This is the diagram (Figure 33) describes the updating process of the map thanks to the component designed for this operation: the *MapManager*. It asks the external MapInterface for an updated map of the whole territory considered by SafeStreets and then stores it in the database. What is stored is actually a simpler version of the map: all the system needs is in fact the collection of cities and streets and the relationships among them.

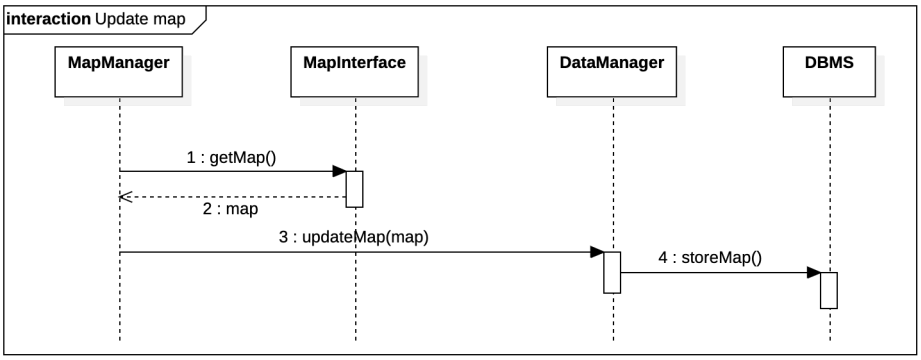

Figure 33: Update Map Runtime View

2.6 Component Interfaces

Figure 34 shows the component interfaces that are inside the application server. For every interface, the exposed methods are shown. Only the internal interfaces are shown. All the methods used in the sequence diagrams of the RuntimeView section must be shown here. For what concerns the specific methods, their input parameters and returned value, they are only meant to give an idea of what they should do, but we are open to changes at the implementation level.

2.6.1 WebInterface

This is the interface with all the methods exposed by the ClientHandler component that will be called by the WebServer. This methods have the only purpose of forwarding requests to the right component of our system that will handle them. The parameters and the returned values are the same as in the interfaces of the components to which the requests are forwarded to. Therefore, they won't be described here.

2.6.2 MobileAppInterface

Same as above, except that this interface is specific for the requests that come from the UserApp. Some methods are duplicated in these two interfaces because

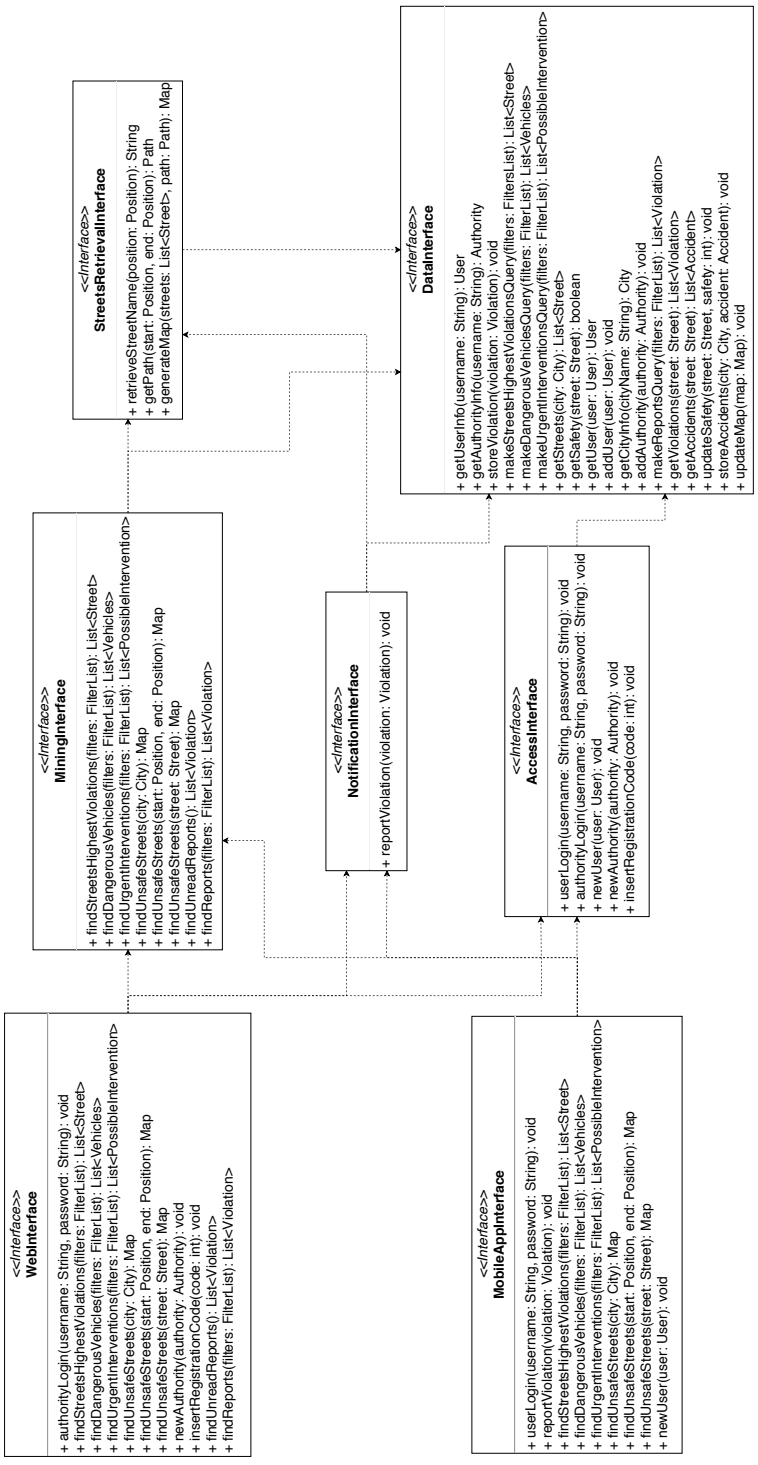

Figure 34: Component Interfaces Diagram

they realize functionalities that both users and authorities are supposed to get advantage of.

2.6.3 MiningInterface

The main purpose of the methods exposed in this interface is to prepare the queries that will be then forwarded to the DataManager component. Queries, that are the means through which mining functionality is provided by our system, usually have a bunch of filters that are here represented by the objects of type FilterList. It is a fictional Class that stands for a multitude of other classes: for example, a user can filter by City, Street, ViolationType and so on. Thus, we can imagine this object at the implementation level having some fields, that could be scalar in case of City, or arrays when users want to include some different types of violation in their queries. The unused fields (for example if a user doesn't want to filter by time slot) will be left empty, for example set to null. Other types of queries, implemented with the overloaded method findUnsafeStreets() for example, are a little bit more simple so the decision is to use the actual object used as parameters. findUnreadReports() and findReports() are actually quite similar: the first is a simpler version of the second because it has no filters as parameters and will include in the returned list of violations only those that have the "read" field set to false.

2.6.4 NotificationInterface

Despite the fact this interface only exposes one method, it is extremely important because it fulfills the reports notification functionality, that is the very core of our system. The idea is that the UserApp will create an object of type Violation and will then send it to the server and finally to this method, that is able to process it. This method will interact with IRI and MapManager to prepare the final Violation object that will be stored into the database.

2.6.5 AccessInterface

These methods are useful for registration and login purposes. They're actually quite simple and self-explanatory. The objects of types User and Authority have all the information gathered during the registration phase, and the methods newUser() and newAuthority() use them to store all the information in the database.

2.6.6 StreetsRetrievalInterface

The methods of this interface are exposed by the MapManager component, whose main purpose is to interact with the external MapInterface and for example ask for street names, paths and maps. Our system is not able to do all of this by itself, so this is the reason why we need an external interface like MapInterface. The objects of type Position are created either by getting the user's GPS positon through the GPS module of the user's device or when the user inserts the position manually. The generateMap() method requires a detailed explanation. It has two object as input, a list of object of type Street and a Path. Depending on the case, only one of those parameters will be set, while the other one will be left null. The goal of this method is handing all the

necessary information to the MapInterface so that it could generate a colored map according to the streets safety data. Of course, if path option is selected by the user, the previously calculated path will have to be shown in the generated map and this is the reason why it appears as a parameter. In addition to this, an object of type Path already includes a list of the involved streets plus other information that is needed to link all the streets in a particular way (e.g. right and left turns etc.), therefore when a Path object is passed, generateMap() won't need the list of streets.

2.6.7 DataInterface

This interface contains the methods exposed by the DataManager component. This component is in charge of communicating with the DBMS. In this way, the whole system is decoupled from the actual DBMS. All of these methods will be implemented by calling the methods exposed by the DBMS that are not modeled here for the sake of simplicity.

2.7 Selected Architectural Styles and Patterns

In this section we are going to describe precisely which are the architectural choices we did while designing the system and why we decided to choose them. As we saw in the deployment view section (2.4) we started with a **client-server** approach and then realized it with a **four-tier architecture**. We now need to describe why we decided these two styles that work good together but also how the elements that really deploy the system can communicate, hence we want also to describe precisely the communications that we highlighted in red in the deployment diagram (Figure 20).

2.7.1 Client-Server

Our system, as described in the context viewpoint of Figure 1, needs to interact with several external systems. It is true that in general SafeStreets needs to interact with systems that provide him services in order to realize its functionalities, but it is also true that SafeStreets aims to be a **crowd-sourced** system that is going to manage interactions with several users (who provide him the data) and several authorities. Thanks to these considerations, after studying different alternatives, we decided to choose a **client-server** architectural approach in order to manage easily the interaction with the mobile application of the users and to have a centralized server that is able to provide the services of the system to any customer that requires them.

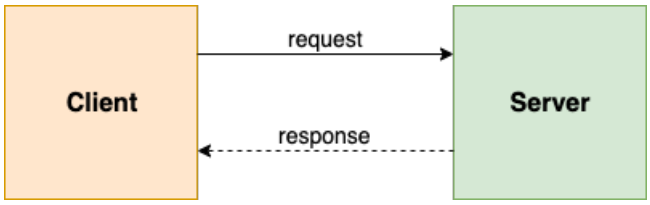

Figure 35: Client Server Architecture

Thanks to the client-server approach we also simplify the way in which we communicate with the external clients taking the decision of using the same technology of communication. In fact, as we will describe precisely in the further section 2.7.3, we are going to build our system in a RESTful way as we can have a simple communication protocol with HTTPS for the **Web Application** and we can integrate it with the **Mobile Application** in order to have the same APIs. This decision has several motivations even if it may not look the best choice to manage the interaction with the mobile application; the first is the one just presented related to the development and interfacing to a single API technology, another is connected with an implementation and integration aspect (as we will see in section 2.7) that is: as we can see in the plan identified by the diagram of Figure 60 the implementation and integration (obviously testing too) of the **User App** and **Web Server** will take place in parallel. In this way, as we will need at this moment to define also the interfaces that provide the communication, we will develop first a common REST API on which the two different interfaces will specialize in order to connect to the related device.

The communications with the other systems depend on the services they provide, as we can see in the deployment diagram (Figure 20) some of the services are still provided in a RESTful way while others, related in particular to the interaction with the *database server* and the *email server* consider instead a classic interfacing related to the management of data and the using of an email service.

2.7.2 Four Tier Architecture

Selected the **client-server** approach we needed to decide how to manage the layers of our system. We chose a multilayered architecture first to benefit of its advantages such as: decoupling, reuse, flexibility, migration... Second to have a way to realize a centralized business logic that allows to manage all the complex management of the data we need to take care of. Thanks to the different layers on which our system is designed we also separate the data as it can be managed more easily rather than having its complexity added inside our principal server. A multilayered architecture allows also to adopt the **thin-client** approach by letting only the presentation layer to be managed on the devices of the clients.

For SafeStreets we decide to choose a **Four Tier Architecture** as we need to take into account (as we see from the devices highlighted in the deployment diagram of Figure 20) two different approaches for the client interaction: **mobile application** for the users and **web application** for the authorities. Hence the necessity of a Web Server needs also to be considered in order to manage the web side of the clients. For this first architectural decision of the client's separation and to maintain the **thin-client** approach we need to have two different tiers to handle the **Presentation Layer**. In order to consider the management of the information (fundamental resource for our crowd source based system) we also need to separate the business logic from the data. The two remaining layers will be separated, hence one tier is needed to deal with the **Logic Layer** and another for the **Data Layer**.

We now present once more the picture of our four tier system as we can

easier compare it with what we have just said.

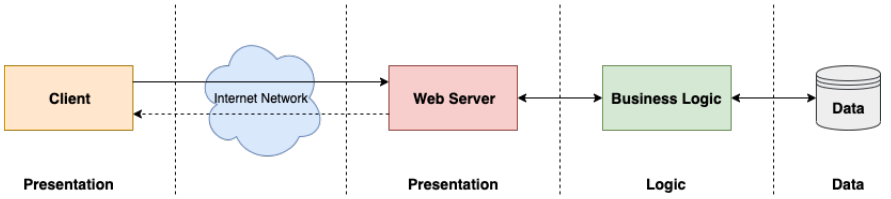

Figure 36: Four Tier Architecture

To be completely precise, our system would also need an additional tier, on which the **script engine server** would be running. This layer would allow to manage the dynamic generation of the web pages for the authority web app. To focus on the most critical aspects of the design process we decided not to represent also this layer to keep quite simple diagrams and descriptions. However we should not completely overlook this consideration as it may be needed in particular for the generation of the map in the *Unsafe Streets* functionality.

2.7.3 RESTful Architecture

We have already motivated the choice of this approach in the client-server section (2.7.1) in particular for how our system is going to provide its services to the clients. The communications with the external services do not need a precise motivation, we only need to stick with the decisions of the system that is providing the functionality and adapt our communication modules.

Now we want to present some further motivations and advantages that the RESTful architecture allows us to obtain. The principle on which REST is based, that is the stateless interaction of requests between the clients and server, is really helpful to us in order to manage the huge amount of data we expect to receive for the notification process. As we have already said, in this way, we will be also able to manage both the requests coming from the users and the authorities with one single interface that is adapted for the **mobile application** side and the **web application** side.

In conclusion this communication architecture based on HTTP is perfectly embraced in our four tier architecture and allows also an easier way to manage the information in a precise standard. In particular we expect the JSON format of the result provided by the **Map Interface** when it return us the data relative to the map that has to be displayed to our clients for the *Unsafe Streets* functionality.

2.8 Other Design Decisions

In this subsection we want to provide additional details on what we have presented in the entire section. In particular we will focus on the motivations for the **relational database** chosen and on a brief description of the two approaches used to realize a **native mobile application**.

2.8.1 Relational Database Approach

We decided to consider a relational approach in particular for the way in which the data has to be managed because of the functionalities of the system. In fact, without focusing on the clients that can be managed in a relational database without any problem, we have seen how several relations are established between the entities that compose our system. In particular the most important are the ones related to the safety calculus: first the relations between the interventions and the streets and also between the accidents and the streets allow to determine the safety; second the relations between the **safety** and the street allows us to process this data and send it to the clients as it can be highlighted for the *Unsafe Streets* functionality.

Considering now some more implementation aspects, as we can also see in the decision tree (Figure 19), we decided to choose MySQL as the DBMS to interact with because it is the most popular relational one and provides a complete and clear documentation on how to benefit of its services.

2.8.2 Native Application Approach

We decided to consider a native approach for the development of the **mobile application** for the users in order to provide a better experience of interaction with the system but also to obtain the best communication and responsiveness between the client and the server. Nowadays the mobile applications are developed with two different languages depending on the OS of the device: for **Android** we will use *Android Studio* while for **IOS** the native language **Swift**.

Android Studio it is the official IDE for Google's Android operating system, designed specifically for Android development. It is available for every operating system and perfectly documented with a huge community. In addition it is based on *Gradle* that helps a lot in the dependency management and thus makes our result more extensible. In fact, it is important to remark that we are designing an extensible system: as we saw in the **class-diagram** of the RASD document [2] the most expected extension we expect is the possibility to report not just parking violations but also other kinds of the general traffic violations.

Swift it is the official programming language developed by Apple in order to replace the old Objective-C paradigm on which all the IOS applications were based on. It is designed to work with the most popular Apple's frameworks and provides several performances and safety trade-offs. We choose it as the native approach to develop a mobile application for IOS as it is still highly documented and its community is getting bigger and bigger.

3 User Interface Design

This section describes precisely the mockups already presented in the *SafeStreets: Requirements Analytics and Specification Document* [2] in order to understand how the main functionalities will be offered. The first subsection focuses on the illustration of the flow of both the mobile and the web applications designed for the users and the authorities by using two **UX diagrams**.

As the design of the application defines two different interfaces, one for the users and one for the authorities, the following subsections will present each the related interfaces. These mockups are meant to precisely define how the customers will interact with the system to benefit of its functionalities, their design is still minimal not to reduce too much the style and fantasy of the developers. In fact it has to be considered that the **Mobile Application** for the users will possibly have two different styles related to the *Android* and *Apple* layouts while the **Web Application** can be designed with any kind of technology preferred.

Both the interactions are structured over a *Client-Server* paradigm, hence all the actions performed by a client are managed by the respective logic that allows to display the graphical interface and whenever a request is needed, its execution will be deployed to the server. Once the answer is received back, the modules for the presentation will display its information to the customers.

3.1 UX Diagrams

Before going deep in details that describe the interfaces of the applications designed for both the users and the authorities, it is very useful to first understand how these interfaces interact one with the others by means of a flow graph that leads from one mockup to the other. Thanks to this kind of diagrams, called UX diagrams, we are now able to understand in which way either the user or the authority will have to interact with the application in order to benefit of the services provided by SafeStreets.

3.1.1 User Mobile Application

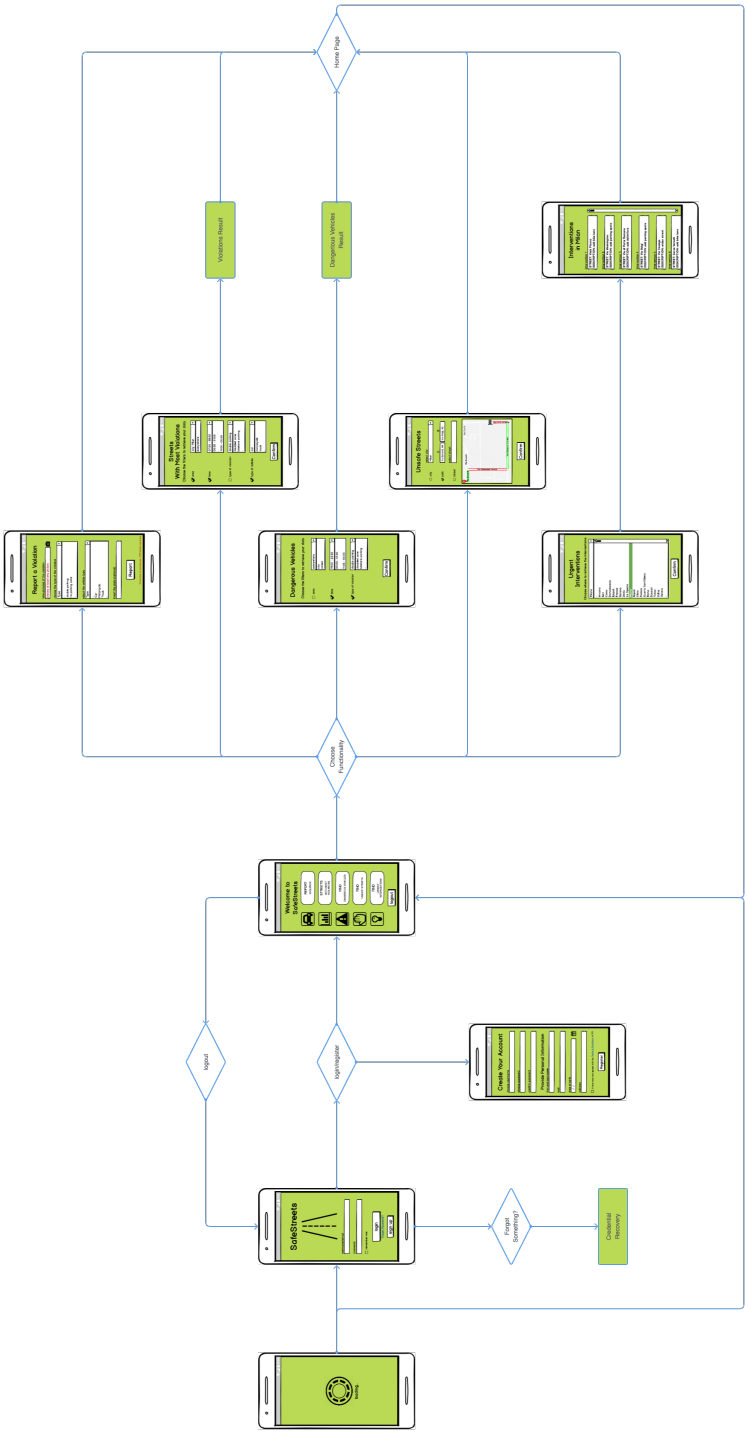

Figure 37: Mobile Application UX Diagram

3.1.2 Authority Web Application

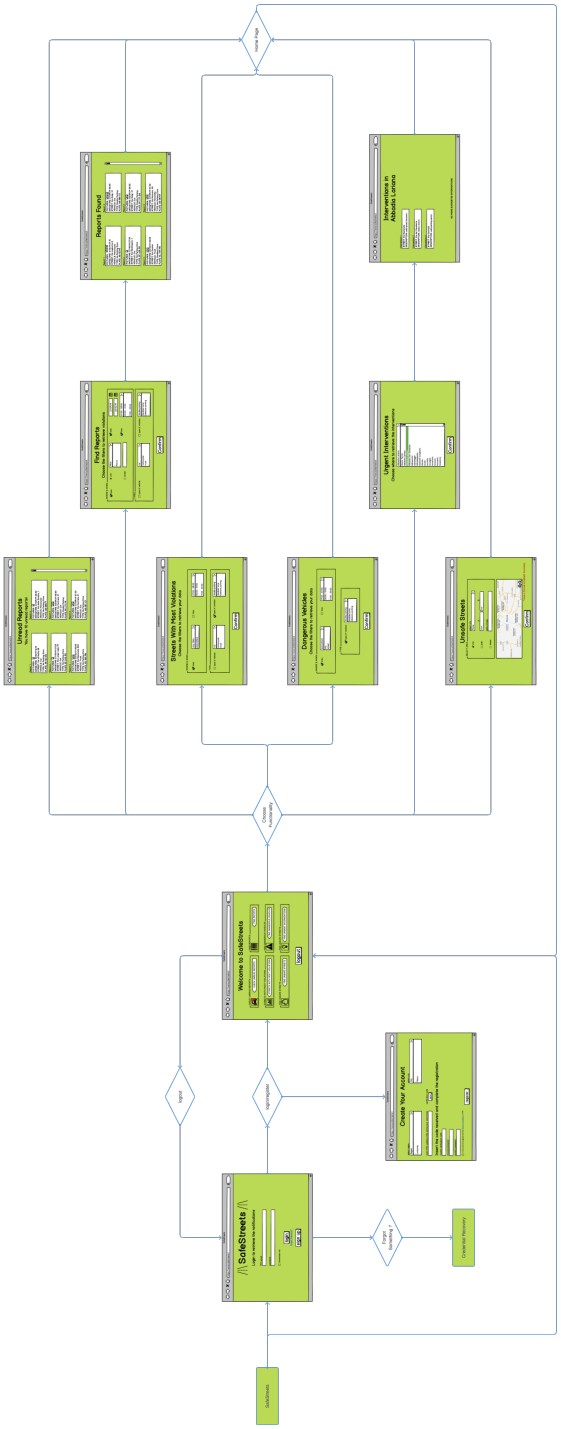

Figure 38: Web Application UX Diagram

3.2 User Mobile App

As previously said the following mockups are just to define which functionalities each interface needs to present. Its layout is left unspecified to allow the creativity of the developer to consider also the two possible different layouts. However it is important to say that the **Mobile App** will be used by the users, citizens of any age, in possible uncomfortable moments. Hence it has to be intuitive and charming in particular to allow a quick way to report a notification. The interfaces that display the information relative to the mining functionalities has also to be easy to understand as it can be understood by any kind of user.

3.2.1 Login & Registration

Login The first mockup (Figure 39) is the initial page every user will find the first time he uses SafeStreets. A user can interact with the system only if authenticated. Hence the first time every user will choose the button **sign up** for registering to the system and interact with the page described in the next paragraph.

If a user is already registered but has not chosen to be remembered, he will need to **login** to SafeStreet every time by providing either its username or email and his password. Once recognized he will be redirected to the home page of the application described in the next section.

This interface leads also to the possibility of recovering the credentials in case a user forgets its password; the mockup relative to the link **Forgot Password?** in the application is not presented as it will simply ask for the users' email in order to send him his new password (the username recovery is not needed as the access can be always performed with the email).

Registration The registration (Figure 40) allows the user to create his account in order to be authenticated whenever he chooses to use SafeStreets. Thanks to this interface he will be able to provide first the information related to his new account and then his personal details. At the end of the page, before confirming the registration it is mandatory that the user accepts the **Terms and Conditions** of SafeStreets.

The button **register** will add the new user to the system and redirect to the login interface where the client can now access the system with the information provided in the previous module.

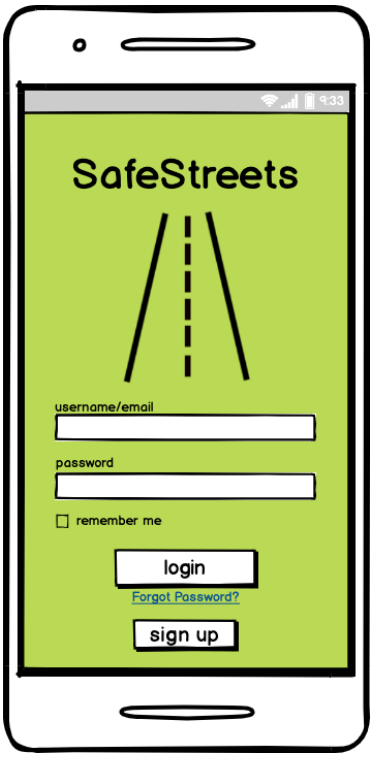

Figure 39: User Login

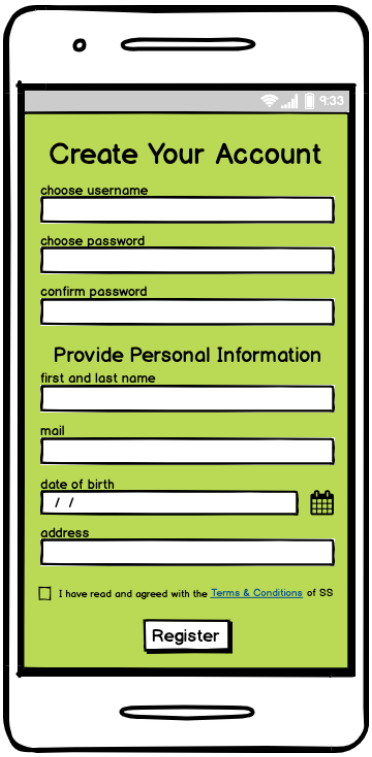

Figure 40: User Registration

3.2.2 Home Page

This page (Figure 41) is the core of the interfaces, once logged the user will reach it and after each action too. From the **Home Page** a user is able to perform all the functionalities allowed to him, offered by SafeStreets:

- Report Violation
- Streets With Most Violations
- Find Dangerous Vehicles
- Find Unsafe Streets
- Find Urgent Interventions

The **logout** button is used to log out of the application and thus the redirection will lead to the login page.

Figure 41: User Home

3.2.3 Report Violation

This interface (Figure 42) allows the entire process of notification to be held. It provides the user a way to insert all the possible information related to a parking violation in order to notify the authority. First the user is asked to take at least one picture of the violation and after to provide two other mandatory details: the type of the violation and the type of the vehicle he is reporting. In the end the plate's number can be also inserted, as we said this information is not mandatory because it is thought that the *Image Recognition Algorithms* can determine it even without this help.

It is important to remember that the device that is reporting the violation must have enabled its GPS to complete a notification. The position will be in fact added to the details of the notification once the user has pressed the **Report** button as the system can automatically retrieve the street where the infraction occurred.

The entire data received whenever a user performs a report is the one that needs to be processed by the system before storing it. In fact it is necessary that no incorrect parking violation is taken into account by the system that wants to notify the authority whenever a new one is received.

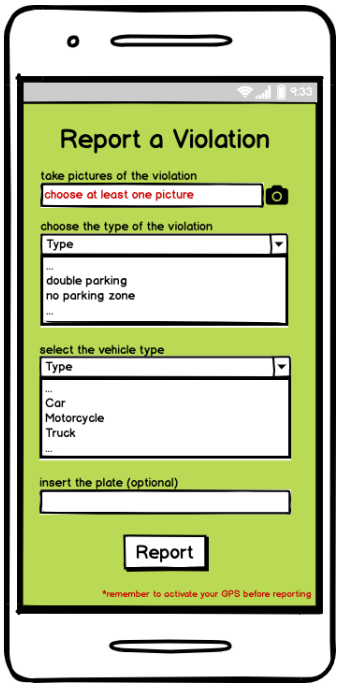

Figure 42: User Notification

3.2.4 Basic Functionalities

These mockups are the ones that allow the user to benefit of the basic functionalities, such as: mining the number of violations in a particular place or the vehicles that commit the most violations. Both the interfaces display the filters that a user can choose while mining the information, remember that the type of filters allowed for each functionality are different, depending on how the goals of the system have been specified.

Streets With Most Violations This page (Figure 43) is used to mine the information about the frequency of the violations that have already occurred. Data can be retrieved with different levels of aggregation thanks to the possible filters:

- **area:** by city or everywhere
- **time:** time slots of one hour each
- **type of violation:** different types of parking violations possible
- **type of vehicle:** different types of vehicle possible

Selected the filters and the parameters for each filter chosen, the user confirms the request with the **Confirm** button and receives the list of all the streets contained in the parking violations that match the parameters, ordered by the number of infractions.

The mockup related to the result of this functionality is not presented because it simply has to display the list of the streets found and allow the user to get back to the home page.

Dangerous Vehicles This page (Figure 44) is used to mine the information about the vehicles that commit the highest number of parking violations. As well as in the first functionality we have filters to retrieve the data, but only the ones that are interesting for this kind of request:

- **area:** by city, street or everywhere
- **time:** time slots of one hour each
- **type of violation:** different types of parking violations possible

Selected the filters and the parameters for each filter chosen, the user confirms the request with the **Confirm** button and receives the list of all the type of vehicles contained in the parking violations that match the parameters, ordered by the number of infractions.

The mockup related to the result of this functionality is not presented because it simply has to display the list of the type of vehicles found and allow the user to get back to the home page.

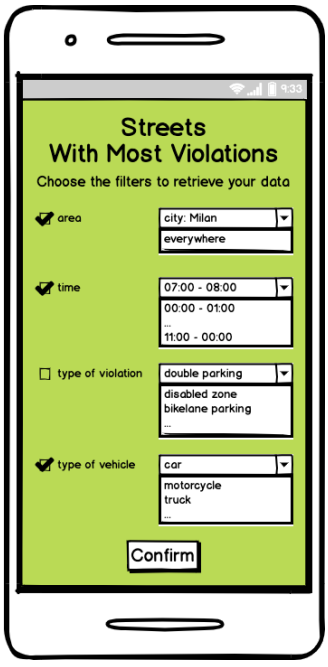

Figure 43: User Violations Frequency

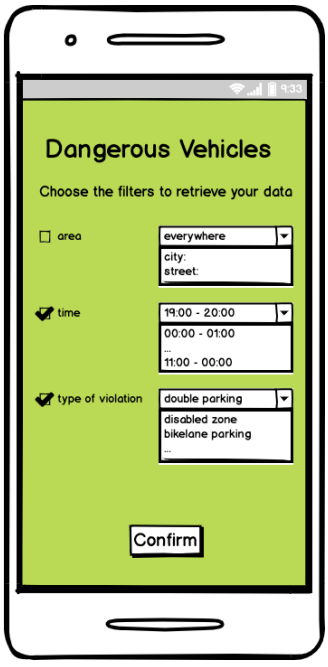

Figure 44: User Dangerous Vehicles

3.2.5 Advanced Functionalities

These mockups are the ones that allow the user to benefit of the advanced functionalities, such as: finding the safety of the streets and receive a suggestion for a possible intervention that can be considered in order to enhance the safety of a dangerous street. In this case we have multiple filters for the safety, related to the area the user decides to highlight while for the suggestions there is only one filter that selects the city of interest.

Unsafe Streets This page (Figure 45) is the only one that reflects the safety functionality, in fact whenever the user confirms the request with the **Confirm** button, the map on the bottom will be highlighted in green and red depending on the safety of the streets of interest. The selection of the area can be done in three different ways thanks to the filters provided by the interface:

- **city:** list of all the Italian cities
- **path:** two cells allow to select the starting (A) and the ending (B) point, the system will highlight the streets in the best path from A to B
- **street:** free choice of the user, a comma has to divide the street from the city

Once the user has chosen the area to highlight the safety and pressed the **Confirm** button, the map will be updated focusing on either the street or the path or the city, in this last case it will be displayed the view of the city with a high scale; in this last case a high view of the city will be displayed and the user will be able to zoom in and out to select the places it prefers.

Figure 45: User Unsafe Streets

Urgent Interventions This page (Figure 46) allows the user to retrieve the suggestions for the possible interventions in the city he selects. The **confirm** button will lead to the result providing a list of all the interventions found in that city.

In this case we provide also the page of the results (Figure 47) as it is important to show which kind of information this request will display. As we see in the result, for each intervention it is provided the street and the suggestion, in case a street has multiple suggestions, two interventions in the same street but with different suggestion will be presented.

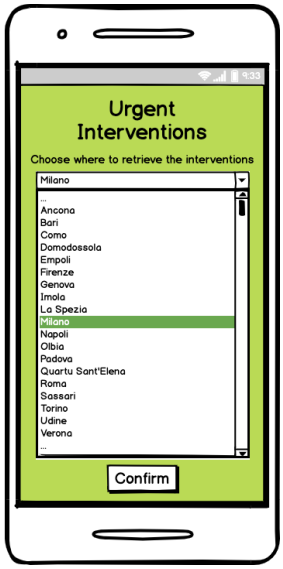

Figure 46: User Interventions

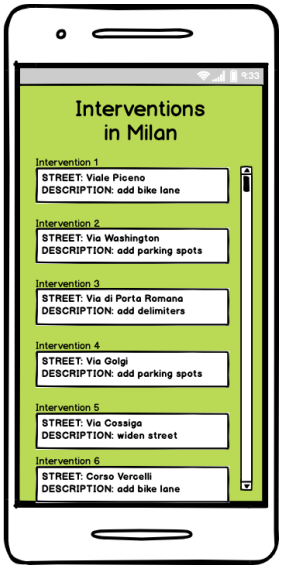

Figure 47: User Interventions Result

3.3 Authority Web App

Also in this case the layouts of the following mockups are only used to precisely describe how the functionalities of SafeStreets are provided to the authority. In this case we can consider the fact that the authorities do not necessarily need an interface that is very charming and pleasant while it is still very important to be intuitive. As in the mobile application, we have also here an interface that should be considered carefully in order to provide a clear description of the parking violations that have been reported to the authority (**??**).

3.3.1 Login & Registration

Login The first mockup (Figure 48) is the initial page each authority will find the first time he uses SafeStreets. Authorities can interact with the system only if authenticated. Hence the first time every authority will choose the button **sign up** for registering to the system and interact with the page described in

the next paragraph.

If an authority is already registered but has not chosen to be remembered, it will need to **login** to SafeStreets every time, by providing its PEC email and the password. Once recognized it will be redirected to the home page of the application described in the next section.

This interface leads also to the possibility of recovering the credentials in case the authority forgets its password; also in this case, the mockup related to the link **Forgot Password** is not presented as it will simply ask the authority's email in order to send him its new password (it is supposed that authorities will never forget their PEC address).

Figure 48: Authority Login

Registration The registration (Figure 49) allows the authority to create its account in order to be authenticated whenever it chooses to use SafeStreets. Thanks to this interface it will be able first to retrieve its city and provide the PEC email. Once the authority decides to send the verification code with the **send** button, the code will arrive to the PEC address of the authority that will insert it in the interface, choose the password and finally register. It is important to remember that also the authorities have to accept the **Terms and Conditions** of SafeStreets.

The button **register** will add a new authority to the system and redirect to the login interface where the client can now access the system with the information provided in the previous module.

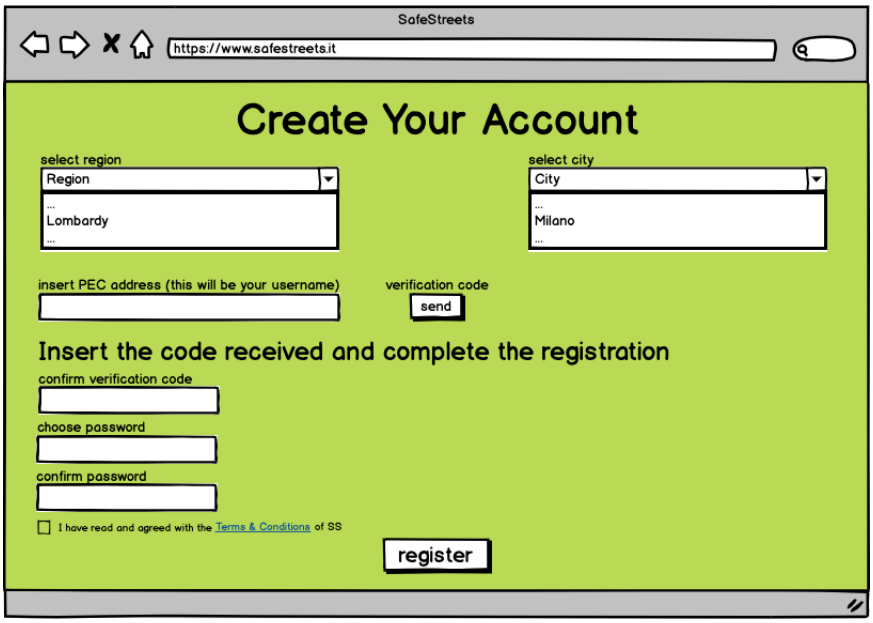

Figure 49: Authority Registration

3.3.2 Home Page

This page (Figure 50) is the core of the interfaces, once logged the authority will reach it and after each action too. From the **Home Page** an authority is able to perform all the functionalities allowed to it, offered by SafeStreets:

- Check Unread Reports
- Find Reports
- Streets With Most Violations
- Find Dangerous Vehicles
- Find Unsafe Streets
- Find Urgent Interventions

The **logout** button is used to log out of the application and thus redirection will lead to the login page.

Figure 50: Authority Home

3.3.3 Unread Reports

This page (Figure 51) is the one that allows the notification of the authorities. As already described in the notification process, whenever a new report is accepted by the system it will be marked as unread for the authority that manages the relative street. This allows to notify the authority with all the reports that are not considered until the last check: if at least one report is present, the red exclamation icon will be displayed in the home page (3.3.2) and here a list of all the unread reports will be presented.

Each report is described with the highest number of details possible: the ones provided by the users and those obtained by the system when validating the notification.

Entering in this interface for the system means that the authority reads all the reports, hence after the redirection back to the home page, each notification will be marked as **read** and the red exclamation mark will not be present anymore: the list of unread reports is now empty.

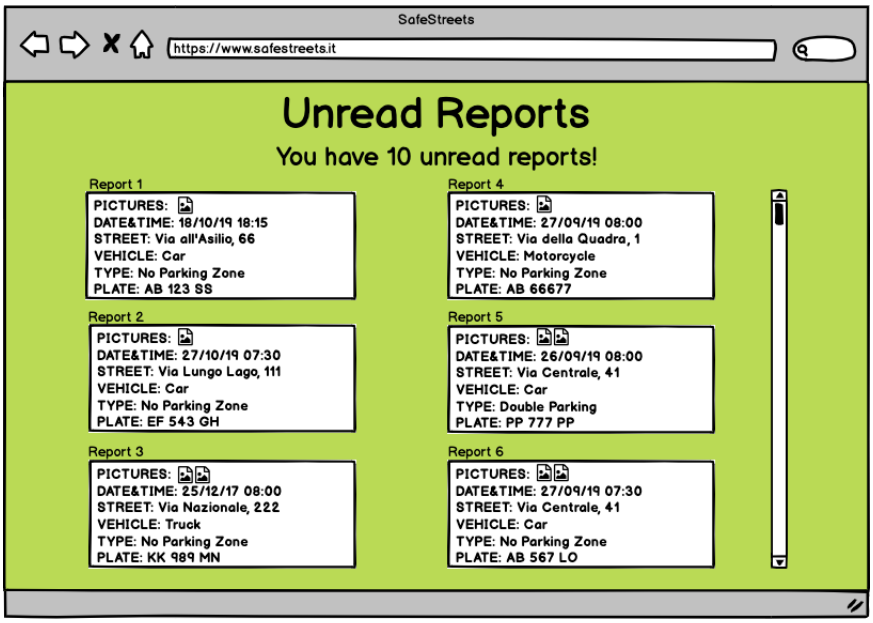

Figure 51: Authority Notification

3.3.4 Find Reports

This interface (Figure 52) is an authority only page that allows it to retrieve any kind of report stored in the system, related to its competent area (if a different city is selected, the system will not provide any result displaying an alert of selecting the correct city). This functionality in fact is given only to authorities in order to provide them a different level of visibility of the reports.

Thanks to this interface the authority is able to retrieve all the reports that match the filters selected:

- **area:** by city and street
- **date:** periods of time are defined with an initial and a final date
- **time:** time slots of one hour each
- **type of violation:** different types of parking violations possible
- **type of vehicle:** different types of vehicle possible

Once the authority has chosen the parameters and pressed the **Confirm** button, a list of all the reports that match the filters will be displayed as the result (Figure 53). In this page the authority is able to see the details of each notification as they are stored in the system, hence with the highest level of visibility possible.

Figure 52: Authority Find Reports

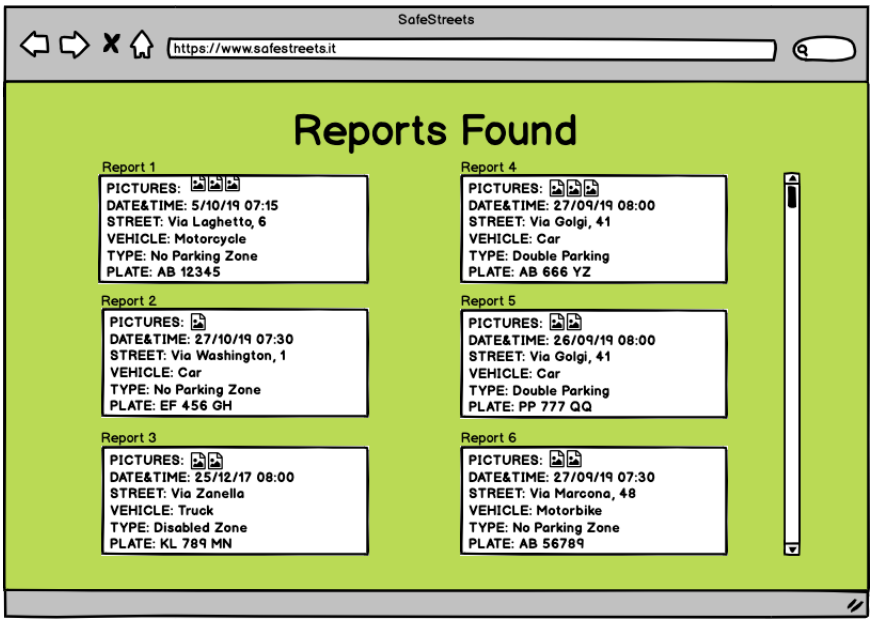

Figure 53: Authority Reports Found

3.3.5 Basic Functionalities

These mockups are the ones that allow the authority to benefit of the basic functionalities, such as: mining the number of violations in a particular place or the vehicles that commit the most violations. The functionalities are the same of the users but presented in a web designed page: filters will be displayed for both pages, depending on the request that can be performed in that functionality.

Streets With Most Violations This page (Figure 54) is used to mine the information about the frequency of the violations that have already occurred. Data can be retrieved with different levels of aggregation thanks to the possible filters:

- **area:** by city or everywhere
- **time:** time slots of one hour each
- **type of violation:** different types of parking violations possible
- **type of vehicle:** different types of vehicle possible

Selected the filters and the parameters for each filter chosen, the authority confirms the request with the **Confirm** button and receives the list of all the streets contained in the parking violations that match the parameters, ordered by the number of infractions.

The mockup related to the result of this functionality is not presented because it simply has to display the list of the streets found and allow the user to get back to the home page.

Figure 54: Authority Violations Frequency

Dangerous Vehicles This page (Figure 55) is used to mine the information about the vehicles that commit the highest number of parking violations. As well as in the first functionality we have filters to retrieve the data, but only the ones that are interesting for this kind of request:

- **area:** by city, street or everywhere
- **time:** time slots of one hour each
- **type of violation:** different types of parking violations possible

Selected the filters and the parameters for each filter chosen, the authority confirms the request with the **Confirm** button and receives the list of all the type of vehicles contained in the parking violations that match the parameters, ordered by the number of infractions.

The mockup related to the result of this functionality is not presented because it simply has to display the list of the type of vehicles found and allow the user to get back to the home page.

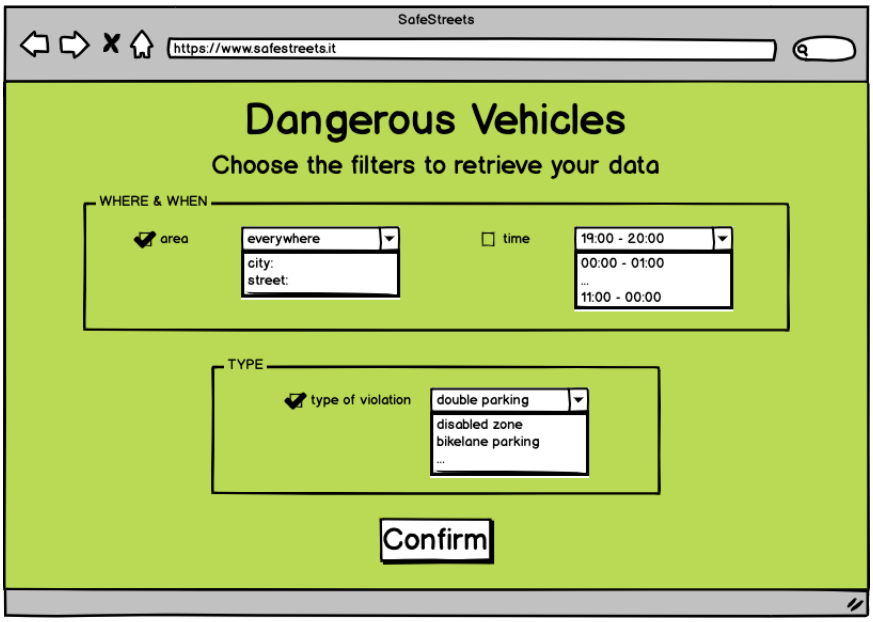

Figure 55: Authority Dangerous Vehicles

3.3.6 Advanced Functionalities

These mockups are the ones that allow the authority to benefit of the advanced functionalities, such as: finding the safety of the streets and receive a suggestion for a possible intervention that can be considered in order to enhance the safety of a dangerous street. Also in this case the functionalities are the same of the users' and thus we will have filters for the area of the unsafe streets and one only to choose the city where to retrieve the interventions.

Unsafe Streets This page (Figure 56) is the only one that reflects the safety functionality, in face whenever the authority confirms the request with the **Confirm** button, the map on the bottom will be highlighted in green and red depending on the safety of the streets of interest. The selection of the area can be done in three different ways thanks to the filters provided by the interface:

- **city:** list of all the Italian cities
- **path:** two cells allow to select the starting (A) and the ending (B) point, the system will highlight the streets in the best path from A to B
- **street:** free choice of the user, a comma has to divide the street from the city

Once the authority has chosen the area to highlight the safety and pressed the **Confirm** button, the map will be updated following on either the street or the path or the city; in this last case a high view of the city will be displayed and the authority will be able to zoom in and out to select the places it prefers.

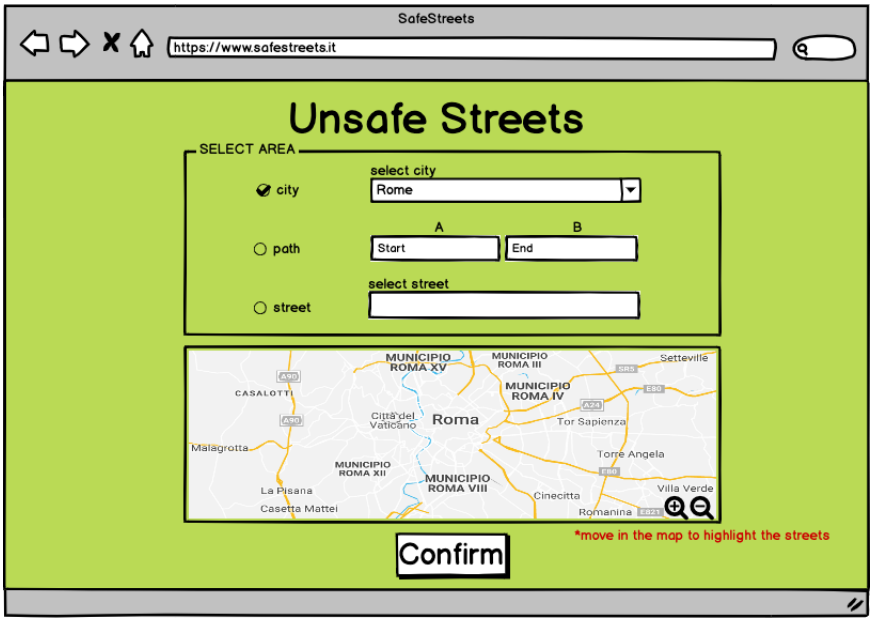

Figure 56: Authority Unsafe Streets

Urgent Interventions This page (Figure 57) allows the authority to retrieve the suggestions for the possible interventions in the city it selects. The **Confirm** button will lead to the result providing a list of all the interventions found in that city.

In this case we provide also the page of the results (Figure 56) as it is important to show which kind of information this request will display. As we

see in the result, for each intervention is is provided the street and the suggestion, in case a street has multiple suggestions, two interventions in the same street but with different suggestion will be presented.

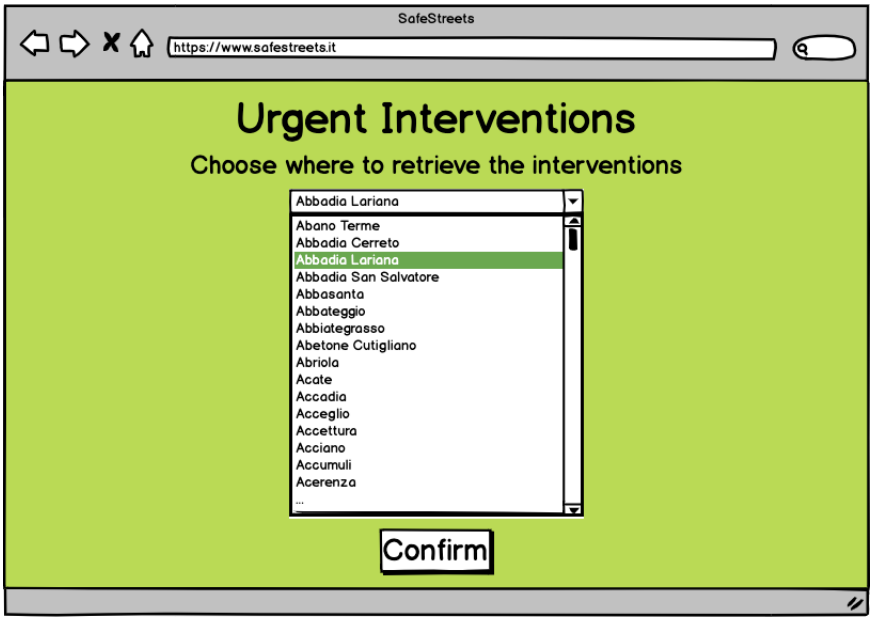

Figure 57: Authority Interventions

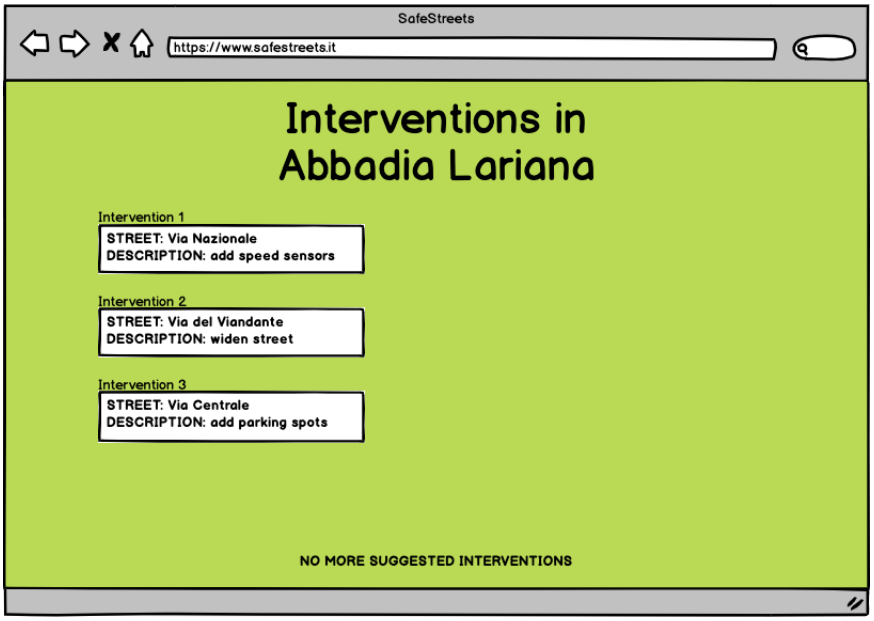

Figure 58: Authority Interventions Result

4 Requirements Traceability

In this section it is shown how the requirements specified in the RASD are mapped to the components defined in this document. It is important to remark that we still consider only the components of the first level, the sub-components they are realized with, in fact, have been sufficiently described to understand the precise functionality they need to provide in order to stick with the requirements.

R1 The system must allow users to register.
AccessManager

R2 The system must allow authorities to register.
AccessManager

R3 The system must allow the user to log in.
AccessManager

R4 The system must allow authorities to log in.
AccessManager

R5 The system must guarantee that each username is unique.
AccessManager

R6 The system must save the customers registration data.
AccessManager

R7 The system must prevent users to use special characters in their username.
AccessManager

R8 The system must allow users to take pictures of the violation they want to report.
UserApp

R9 The system must retrieve date and time while the user reports a violation.
UserApp

R10 The system must retrieve the GPS position while the user reports a violation.
UserApp

R11 The system must allow users to insert the type of violation they're reporting.
UserApp

R12 The system must allow users to insert the type of the vehicles they're reporting.
UserApp

R13 The system must allow users to insert the license plate number of the vehicle they are reporting.
UserApp

R14 The system must read the license plate from every picture sent by the user.
ReportManager

R15 The system must store the data received along with the name of the street where the violation occurred.
ReportManager, MapManager

R16 The system must be able to retrieve the name of the street from the GPS coordinates.
MapManager

R17 The system must be able to show authorities all the violation reports sent by users.
ReportManager

R18 The system must be able to mine the stored violation reports.
QueryManager

R19 The system must allow the customer to filter by city.
QueryManager

R20 The system must allow the customer to filter by street.
QueryManager

R21 The system must allow the customer to filter by path.
QueryManager

R22 The system must allow the customer to filter by type of violation.
QueryManager

R23 The system must allow the customer to filter by time slot.
QueryManager

R24 The system must allow the authority to filter by date intervals.
QueryManager

R25 The system must allow the customer to filter by type of vehicle.
QueryManager

R26 The system must be able to sort streets by number of violations.
QueryManager

R27 The system must be able to sort types of vehicle by number of violations they make.
QueryManager

R28 The system must be able to provide different levels of visibility.
AccessManager

R29 The system must be able to store and manage data about accidents, if provided by the authority.
AccidentsManager

R30 The system must be able to cross accidents data with violations one.
AccidentsManager

R31 The system must be able to determine whether a street is safe or not.
SafetyManager

R32 The system must be able to access all the available maps.
MapManager

R33 The system must find a path between two given points in a map.
MapManager

R34 The system must be able to color the streets in a map according to their safety.
MapManager

R35 The system must be able to determine the most urgent interventions in a street.
SafetyManager

5 Implementation, Integration and Test Plan

In this section we are going to provide a precise plan of how the implementation, integration and test plan should be realized in order to deal with all the characteristics and features of our system, now that we have a complete and clear description of all its aspects.

First we provide a brief introduction of some entry conditions that must be met before starting with this plan, in particular with also some details of the tools we suggest to carry the process with. Second, to accomplish some of the requirements of the system and to justify some other decision taken while defining the plan, we illustrate with an **utility tree** the importance of the features we expect SafeStreets to provide. Finally, in the plan definition section (5.0.4) we give the precise description of the *Implementation, Integration and Test Plan*, along with a concise correspondence of the **use relation hierarchy** diagram (Figure 60).

5.0.1 Strategy Overview

The system we have identified has its main complexity in the component that deals with the computation of all the requests coming from the users whenever a functionality is required. This is the reason why, while describing the component view section (2.2), we did not stop to identify only the components that compose the server one, but we deepened also inside each of them to understand how each of their functionality could be carried out. In this section we want to identify a plan to be followed while building the entire system, hence it is obvious that we will start from the *Server Component*, in particular with the component that has to deal with the most critical feature inside of it. The plan will take in consideration only the first level of components the server is composed of, in fact, the more specific ones have been identified to provide an additional separation as it can be easier, for the team that is going to implement it, to chose how to proceed in their realization. This freedom inside the development of each component allows to obtain a better result having already identified the specific modules that will compose it; however, the implementation and testing of each component must always follow the plan that is later described.

Thanks to what we have just identified we now list the main features we have considered for the *Implementation, Integration and Test Plan*:

- **Bottom-Up and Top-Down:** we mainly focused on a bottom-up approach to define the plan because we decided to start from the basic components that compose the most critical one (the server) and then started to proceed upwards ending with the interactions of the entire system with the external ones. A top-down approach is also used in the cases where the realization of a component precedes the one that allows to manage its data. For example, the integration of the AccidentsManager component with the ACI is going to be done at the end of the plan as we can create mocked accidents to develop the main functionality that is the storing inside the database as they can be used for the safety calculus.

- **Core Functionality:** we consider the notification process the core functionality of our system, hence the most important to be first implemented, tested and integrated. Its realization will be held only after the *DataManager Component* that allows us to manage the data of the system as soon as possible; the data needed to test the system will be mocked exploiting the top-down approach.
- **Decoupling of Components:** we have defined our plan with an interaction between the components that is mainly defined by the functionalities we want to realize. This means that the components needed to realize different functionalities can be considered first at the step of the realization of a functionality and left uncompleted until the others functionalities are taken into account.
- **Reliability of External Systems:** while considering the realization of the interaction with the external systems we do not take in consideration the testing of the methods provided by their interfaces as they should be already tested and verified by their providers.

5.0.2 Entry Criteria

The following conditions have to be met before entering the *Implementation, Integration and Test Plan* as its guideline can be easily followed and meaningful results are obtained.

A fundamental criterion on which the entire plan has to rely on is that the implementation goes in parallel with the testing. This consideration allows to retrieve the problems of the system as soon as possible nd to reduce the cost of their correction if found later; the percentage of testing for each module implemented has to be at lest 85% in order to proceed with its integration.

Another important issue related to the top-down approach often used in order to start developing the most critical components related to the most important functionalities of the system, is that we need to be able to generate feasible **stubs** that can be used to simulate the real data the system is going to manage. Hence some random generators may be also needed in order to consider all the possible cases in which the system can be stressed.

Before starting with the identification of the plan we want to give some guidelines related to some very useful tools that can be used when the process of development and testing of the system really starts.

Development Tools We consider *Java* as the main language the system has to be developed with. Thanks to its enormous popularity lots of documentations and libraries can be found to solve almost every problems we could encounter. In particular it fits also very well with the communication among different platforms and devices as in our case. In fact several editions have been released in order to deal with all the possible types of systems that can be developed.

Testing Tools Testing tools are fundamental in order to proceed with a correct and meaningful testing process while developing the system. Several tools are needed in order to consider all the testing issues, between all of them we suggest:

- **JUnit:** it is a framework that allows to define the test for each of the parts developed in a Java application. It is very well documented and allows to be integrated with other testing tools that we are going to use.
- **Mockito:** it can be integrated to *Junit* in order to realize tests with "fake" objects that are used as a stub for the functionalities in which they are needed. This framework can be very helpful for the realization of the components that need data not already present in the system.
- **Jacoco:** it can be also integrated with *JUnit* and is used to keep track of the coverage of the lines implemented. It is fundamental to have a precise criterion to establish when a component is sufficiently tested as it can be integrated.

5.0.3 Utility Tree

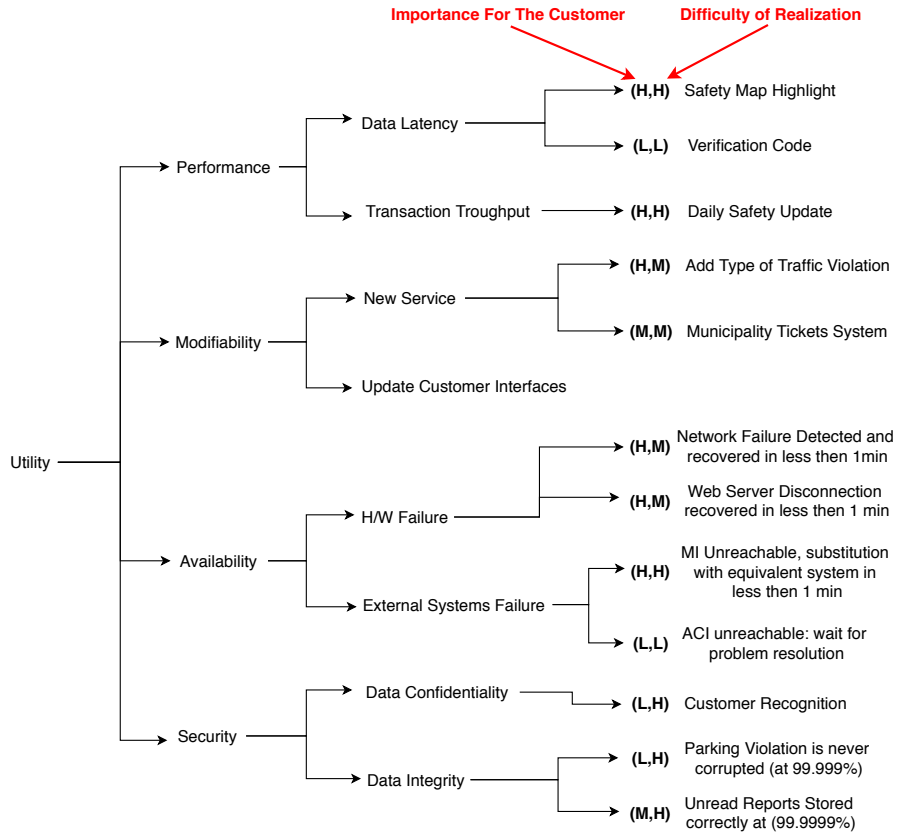

Figure 59: Utility Tree

5.0.4 Plan Definition

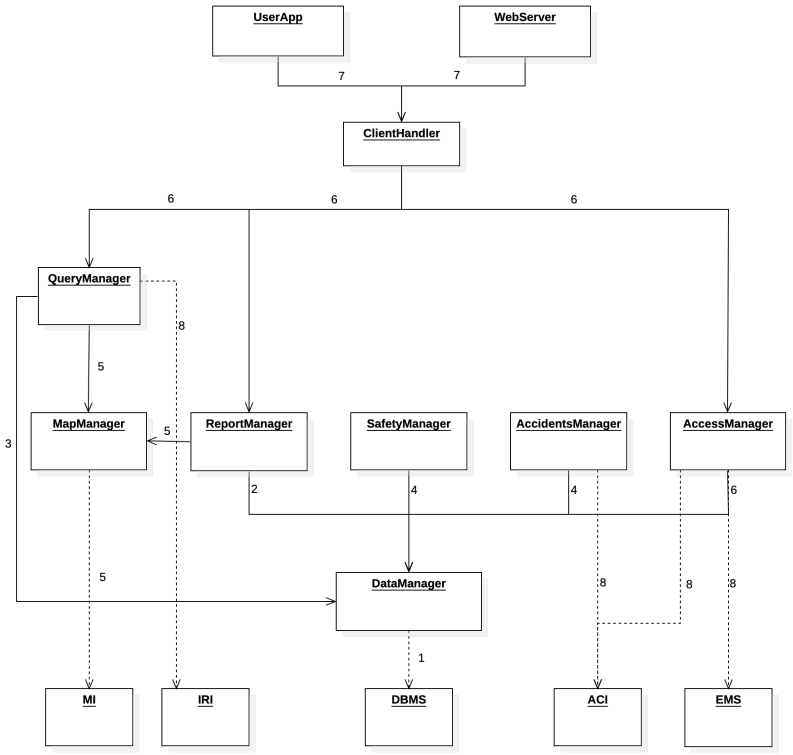

Figure 60: Use Relation Hierarchy Diagram

1. The first component we plan to implement is DataManager. This choice is due to the fact that this particular component is responsible of all the interactions with the database and it is needed by the majority of all the other components in the ApplicationServer, being the very leaf of the tree, as shown in the diagram. Then it will be integrated with the DBMS and thus tested.

2. Then, the ReportManager component will be implemented because its role is quite crucial for the whole system. It provides in fact the core functionality, allowing users to notify authorities of violations. Its integration with the DataManager will follow as soon as possible. It is important to highlight the fact that ReportManager needs MapManager only to retrieve the name of the street from the position of a reported violation. Because of this, it is quite simple to generate a stub for MapManager and test ReportManager anyway. The components designed to handle the interactions with external interfaces are planned to be implemented later on, mainly because they're not that "intelligent" and external APIs are well documented.

3. Next step is implementing QueryManager, another extremely important

component, and probably the most complex one. It needs to be fully implemented and tested as soon as possible, and integrated with DataManager because of course queries rely a lot on the DBMS. With regard to MapManager, the same thing as point 2 applies here: the difference is that now MapManager is needed for path finding and map retrieval and thus is a little bit more crucial. This is the reason why the integration of MapManager with the interacting components will have to be done in the very following steps.

4. In order to complete the development of the most important functionalities, we now plan to implement and test AccidentsManager and SafetyManager, and integrate them with DataManager. They are actually not too complex components, so this operation should be quite fast. The idea is to integrate AccidentsManager and ACI later on.

5. It is now time to complete the implementation of MapManager and integrate it with the two components that use it (ReportManager and QueryManager) and with the external MapInterface. Excluding the interactions with the remaining external interfaces and the registration/login services, the system is now ready to provide all the functionalities.

6. Now that the internal logic is complete, we plan to realize the components that interact with clients. Therefore, ClientHandler and AccessManager will be implemented, tested and integrated together and with all the components they interact, that are ReportManager, QueryManager and DataManager. Thanks to the section related to the user interfaces (3) we give the developers an illustration of how the interaction with the clients have to occur in order to provide the functionalities of the system. As already said in that section, the graphical interface can be modified and beautified with the skills of the developers as long as they stick on the way in which the functionalities have to be provided.

7. Since ClientHandler is now fully tested, we could integrate it with the UserApp and the WebServer.

8. As the last step, we plan to integrate all the remaining external interfaces with the internal components that use them: QueryManager will be integrated with IRI, AccessManager and AccidentsManager will be integrated with the AuthorityCommonInterface, and finally AccessManager with EMS.

6 Effort Spent

6.1 Teamwork

Task	Teamwork's Hours
Introduction	1
Planning Architecture	30
Architectural Overview Design	10
Choosing Patterns	5
IIT Plan	5

Table 2: Teamwork effort

6.2 Individual Work

Task	Matteo Pacciani's Hours
Introduction	0.5
Component View	2
Deployment View	2
Runtime View	30
Component Interfaces	5
Styles and Patterns	0.5
Other Design Decisions	2
User Interface Design	1
Requirements Traceability	5
IIT Plan	5

Table 3: Matteo's effort

Task	Francesco Piro's Hours
Introduction	1
Component View	30
Deployment View	5
Runtime View	2
Component Interfaces	10
Styles and Patterns	5
Other Design Decisions	1
User Interface Design	15
Requirements Traceability	1
IIT Plan	7

Table 4: Francesco's effort

Appendices

A Revision History

- **v.1.0** December 9, 2019

B Software and Tools used

- LaTeX as document preparation system
- Git & GitHub as version control system. The repository of the project is here
- Draw.io for the high level diagrams
- Balsamiq for the mockups
- StarUML for the UML diagrams
- FlowMap for the UX diagrams

7 References

[1] Shrikanth N. C. Gurdeep Virdi Alex Kass Anitha Chandran Shubhashis Sengupta Anurag Dwarakanath, Upendra Chintala and Sanjoy Paul. Crowdbuild: A methodology for enterprise software development using crowdsourcing. Technical report, 2015.

[2] Matteo Pacciani and Francesco Piro. *SafeStreets: Requirements Analytics and Specification Document*. Politecnico di Milano - Software Engineering 2 Project, 2019.