POLITECNICO
MILANO 1863

Software Engineering 2 Project

Politecnico di Milano AA 2019-2020
Computer Science and Engineering

DD - Design Document

Professor:

Elisabetta Di Nitto

Group Components:

Fiozzi Davide (Matricola: ***945107***)

Frantuma Elia (Matricola: ***945729***)

Freddi Eleonora (Matricola: ***945113***)

Contents

1. Introduction

1.1 Purpose

The purpose of this document is to provide an overall guidance to the architecture of the software product concerning the SafeStreets required system, and therefore it is primarily addressed to the software development team.
In this document all the SafeStreets System's components are illustrated, including runtime processes and design choices. In particular, the following topics are discussed:

- High-level architecture;
- Main components, including interfaces and deployment;
- Runtime behavior;
- Employed Design Patterns;
- User interfaces;
- Requirements Traceability;
- Implementation, Integration and Testing plan.

1.2 Scope

The purpose of the SafeStreets' initiative is to raise citizens' awareness about the violations that are committed every day, allowing them to make reports for a more civilized city.
In addition, for a municipality, a service like SafeStreets can help to determine which areas are considered unsafe and those with high rate of violations, so authorities can act on more accurate data and help to make the municipality optimally controlled.
Finally, for authorities, being able to leverage on the service provided by SafeStreets can help to reduce the time needed to emit traffic tickets and to invest it in more important, resource-intensive activities.
In detail, SafeStreets S2B allows:

1. users to have the possibility to notify the system when traffic violations occur, and in particular parking violations;

2. to cross system's data with those made available by the municipalities, like information of local accidents, to identify potentially unsafe areas, and let the system suggest possible interventions to the involved municipality;

3. local police of a municipality to search for violations in SafeStreets database and generate traffic tickets based on those reported by registered users.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

- Customer: a generic registered SafeStreets customer (can be user, authority or a SafeStreets operator).
- User: a normal application customer who notifies the authorities of a traffic violation.
- Authority: a favoured customer of the application who intervenes as a result of a traffic violation. It can be distinguished into municipality or local police.
- Violation: a set of photo and metadata that describes an attitude that is not in accordance with the rules of the road.

1.3.2 Acronyms

- GPS: Global Positioning System
- RASD: Requirement Analysis and Specification Document
- DD: Design Document
- OS: Operating System
- API: Application Programming Interface
- PC: Personal Computer
- S2B: Software To Be
- DB: Database
- DBMS: Database Management System
- DMZ: DeMilitarized Zone
- UI: User Interface
- OCR: Optical Character Recognition
- UX: User Experience
- SQL: Structured Query Language

1.3.3 Abbreviations

- Rn: nth requirement

1.4 Revision history

- Version 1.0: First release

1.5 References

- Specification document: “SafeStreets Mandatory Project Assignment”
- Document “DD to be analyzed - A.Y. 2019-2020”

1.6 Document Structure

Chapter 1: Here an introduction of the design document is given. This chapter contains the purpose and the scope of the document, as well as some details on used terms in order to provide a better understanding.

Chapter 2: this chapter deals with the architectural design of the application. It gives an overview of the architecture and it also contains the most relevant architecture views:

- Component view;
- Deployment view;
- Runtime view;
- Interaction of the component interfaces.

Some of the used architectural designs and designs patterns are also presented here, with an explanation of the purpose of their usage.

Chapter 3: this chapter specifies the user interface design. The mock-ups presented in the RASD document and UX diagrams are here visualized.

Chapter 4: the requirements traceability is analyzed in this chapter; in detail, is described how the requirements identified in the RASD are linked to the design elements (defined in this document).

Chapter 5: briefly describes the implementation plan, the integration plan and the testing plan, specifying how all these phases are thought to be executed.

Chapter 6: contains a detailed report of the effort spent by each group member while working on this project.

Chapter 7: this is the last chapter and includes references to the documents, texts and resources used to write this document.

2. Architectural Design

2.1 Overview

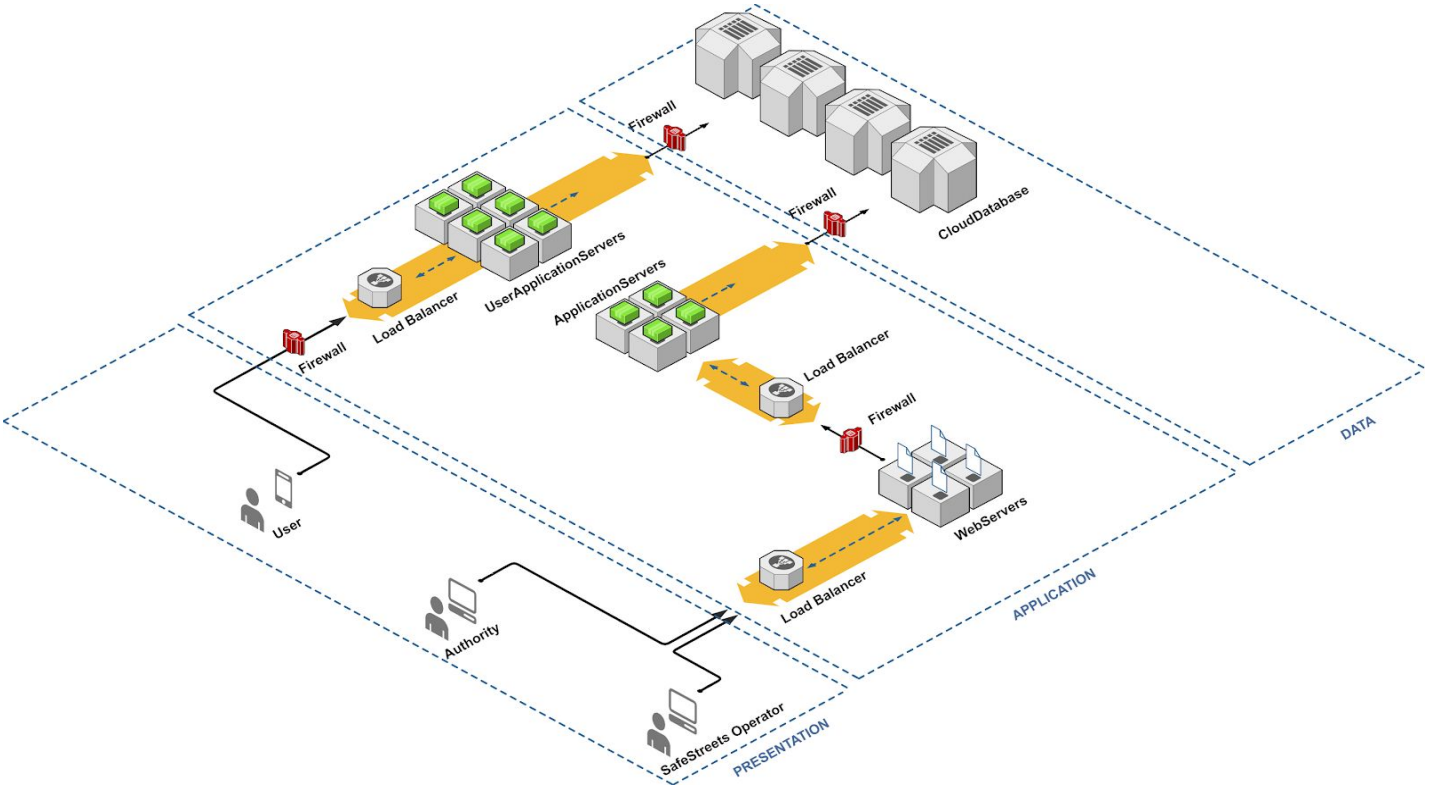

Figure 1 - System Architecture

ApplicationServers and UserApplicationServers encapsulate all the SafeStreets service logic. In order to provide 99.9% availability and reliability, servers must be replicated. This approach requires the adding of a load-balancing system to distribute the workload among the nodes. The replication shall follow the cloning with shared disk configuration, in order to distribute computational load and to replicate data for security reasons. Both scalability properties and fault tolerance of the service are increased with this configuration. Treated data must be protected; firewalls are installed before and after the application tier to create a DMZ (Demilitarized Zone). The functionality of the firewalls to be installed is standard, so they will not be described in detail in this document

UserApplicationServer and ApplicationServer handle different types of requests. The latter replies to Authorities and SafeStreets operators requests, the former to all actions concerning users' application. The expected workload of these two types of servers are different; in particular, the UserApplicationServer is expected to have more interactions than the

ApplicationServer. It's necessary to have a higher number of replications for the former in order to increase the throughput of served requests and reduce the overall latency.
All data concerning the SafeStreets service is saved in a cloud database. The reasons for this choice will be explained in detail on the following pages.

2.2 Component View

The following diagram shows all system components and their interactions. For a better identification, the components belonging to the UserApplicationServer have been colored blue, those belonging to the ApplicationServer have been colored red.

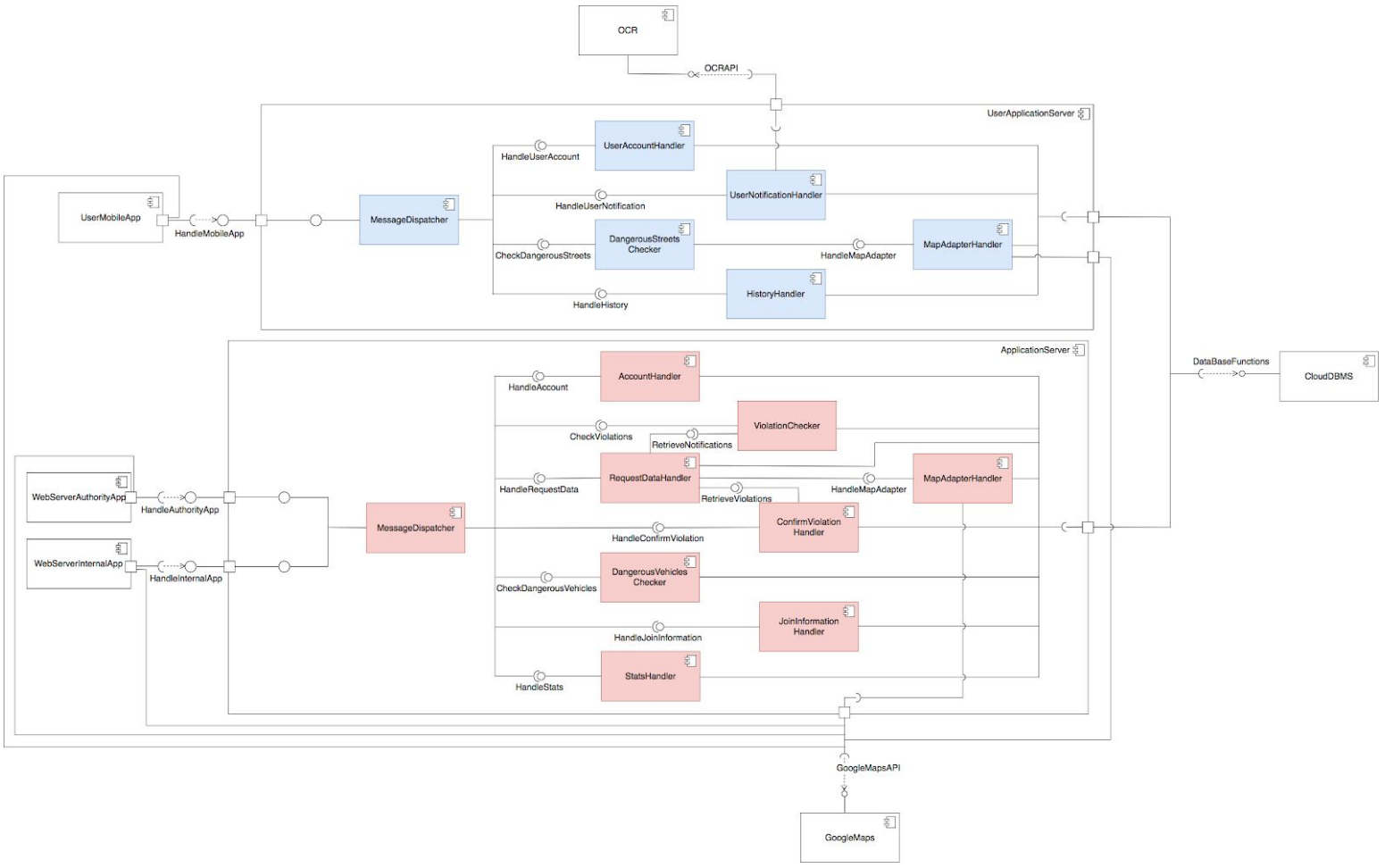

Figure 2 - Component Diagram

Each component and the functions it offers are explained here in detail.

UserApplicationServer:

- ***MessageDispatcher***: receives all the user's client messages and route them to the correct component of the UserApplicationServer. When the request is satisfied, *MessageDispatcher* sends a requestResponse to the client.

- ***UserAccountHandler***: This component contains all the procedures that allow the registration, login and logout of a user. It interacts with the CloudDBMS to save user information regarding registration, and to authenticate a user, checking the email and password when logging in. It also checks whether the user has made 3 or more notifications in which no violations have been reported; in this case it refuses the login of the corresponding user.

- ***UserNotificationHandler***: receives and handles users' reports. It interacts with the OCR service in order to obtain the license plate of the reported vehicle; if the license plate is not recognized, the report is automatically discarded, otherwise the report is saved and waits to be analyzed. The user is instantly notified about the operation outcome. It also interacts with the CloudDBMS to save notifications that have not been discarded; these notifications will subsequently be analyzed by a SafeStreets operator.

- ***HistoryHandler***: manages a user's request to view the history of reports he made. It interacts with the CloudDBMS to retrieve information relating to all notifications made by the user involved.

- ***DangerousStreetsChecker***: this component handles users' requests to view streets that have been classified as dangerous. It retrieves from the database the data concerning dangerous streets of the city indicated by the user as search filter. In order to provide a user friendly visualization, interacts with the *MapAdapterHandler.*

- ***MapAdapterHandler***: this component adapts the data of the dangerous streets requested by the user to a GoogleMaps map. Here the interaction with the external service is made thanks to service APIs. This component is activated only by *DangerousStreetsChecker.*

ApplicationServer:

- ***MessageDispatcher***: receives all the authorities and SafeStreets operators requests and routes them to the correct component of the ApplicationServer. When the request is satisfied, *MessageDispatcher* sends a requestResponse to the client.

- ***AccountHandler***: handles the registration and login of the authority or SafeStreets operator.

- ***ViolationChecker***: this component receives a request from a SafeStreets operator to analyse the notifications reported by users. It retrieves the not yet analyzed notifications and submits them to the operator who made the request. When a violation is confirmed or rejected by the operator, its state is saved on the CloudDB thanks to this component.

In addition, if the violation is rejected, the counter of the rejected violations of the user that made the report must be increased (by one). A trigger allows to set to true a flag for the user involved when he makes 3 or more notifications in which no real violations of the Highway Code have been detected. This flag is checked when the user logs in.

- ***DangerousVehiclesChecker***: receives a request from an Authority operator to get vehicles that have been classified as dangerous and retrieves the related information from the CloudDB.

- ***MapAdapterHandler***: this component adapts the data of the dangerous streets requested by the Authority operator to a GoogleMaps map. Here the interaction with the external service is made thanks to service APIs. This component is activated only by *RequestDataHandler.*

- ***JoinInformationHandler***: component used to merge the information provided by municipalities with that saved in the SafeStreets database. Once the information has been entered, it saves it in the SafeStreets' cloud database.

- ***RequestDataHandler***: component used to generate direct queries to the cloud database. In detail, this component is used for the following functions:

    - Retrieve notifications to be analysed by a SafeStreets operator;
    - Retrieve violations to be confirmed by an authority operator;
    - Retrieve dangerous streets;
    - Retrieve dangerous vehicles;
    - Advanced search request.

  This component is also used by *ViolationChecker* and *ConfirmViolationHandler* components to retrieve from the database respectively the notifications made by users (that have yet to be analyzed) and the violations saved (previously analyzed by a SafeStreets operator). It also uses *MapAdapter* if the retrieved data needs to be adapted to a portion of a map in order to be easily understandable.

- ***ConfirmViolationHandler***: component dedicated to the management of the confirmation of violations by the authorities. It receives the request to display a list of violations and saves in the database the response: the violations that have been confirmed (and therefore a traffic ticket has been issued) and those that have been discarded.

- ***StatsHandler***: component devoted to the formulation of complex queries to obtain statistics on the data saved in the SafeStreets' cloud database.

2.3 Deployment View

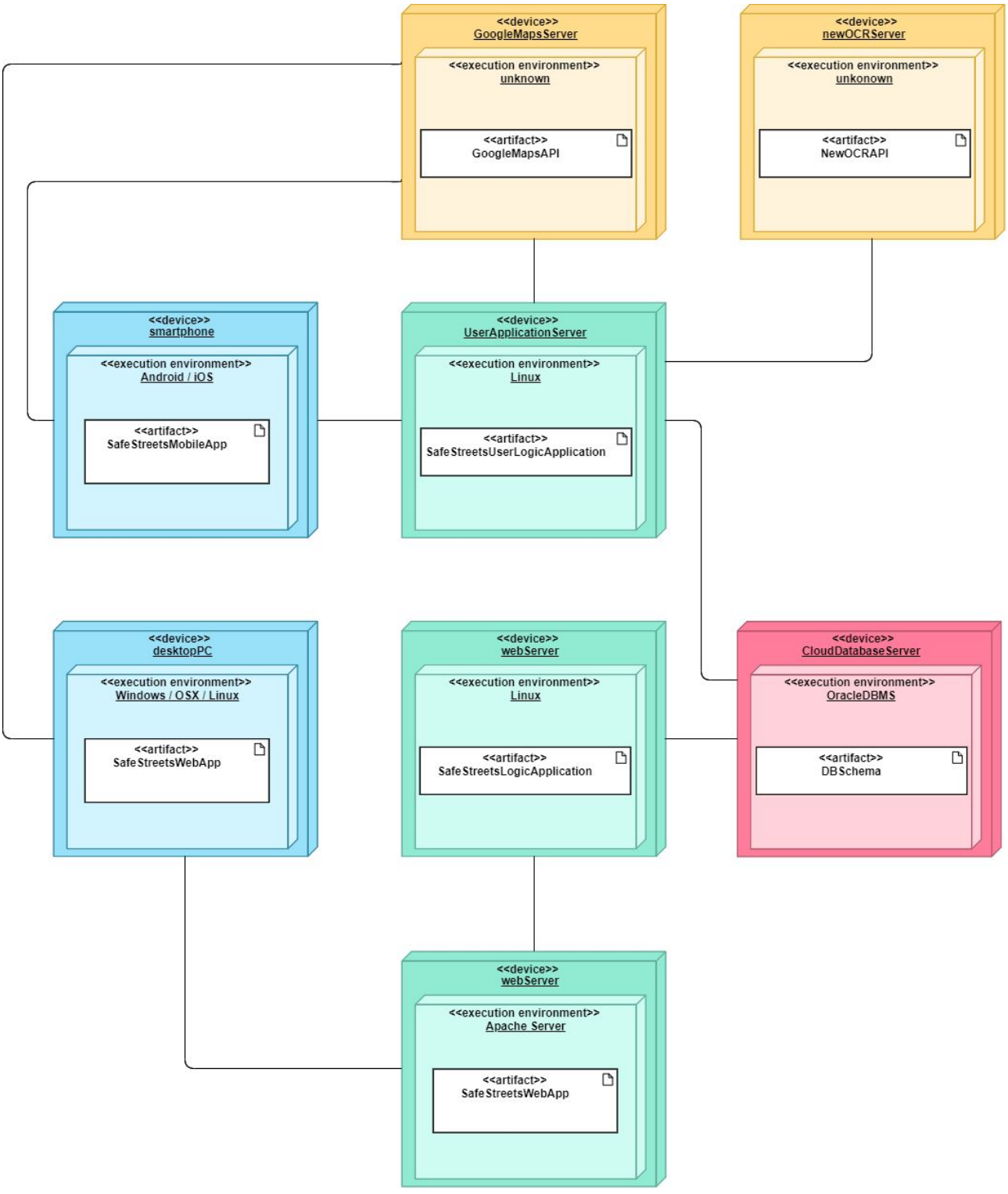

Figure 3 - Deployment Diagram

The deployment diagram shows the architecture of the system as deployment of software artifacts to deployment targets. In detail these nodes are considered:

- *Smartphone*: the smartphone communicates with Application layer when a user requests a SafeStreets service and communicates with external service "Google Maps" through APIs. In particular, when the smartphone Application needs to obtain access to the map, Google Maps Server provides the requested portion of the map.
- *DesktopPC*: authorities and SafeStreets operators log into SafeStreets services with a proprietary WebApp. In order to access, they must use a Desktop PC with an installed browser.
- *ApplicationServer*: this node contains all SafeStreets service logic, concerning Authorities and SafeStreets operators requests. The requests arrive from node "WebServer", that handles SafeStreets WebApp. When a request arrives, this node uses the access to the cloud database in order to make all his actions persistent.
- *UserApplicationServer:* this node contains all SafeStreets service logic, concerning users' requests. It accept all users' requests made from users' smartphones. In order to work properly, this node needs to access Database.
- *WebServer*: this node provides access to SafeStreets services to authorities and SafeStreets operators through a Web Application. No business logic is found in this node.
- *GoogleMapsServer*: this node provides access to external map services through APIs, used by smartphones, desktop PCs and application servers.
- *NewOCRServer*: this node provides access to external Optical Character Recognition (OCR) service through proprietary API. This service is used by UserApplicationServer in order to recognize a license plate in a notification photo.
- *CloudDatabaseServer*: the Application layer relies on the DataServer to make all his actions persistent. Data layer deals with data manipulation, insertion and deletion of all types of data concerning SafeStreets organization.

2.4 Runtime View

Below are reported the sequence diagrams, already exposed in the RASD, analyzing in detail the interaction between the different components and showing the messages that are exchanged between them.

Visitor Sign Up

This diagram shows the interaction and set of messages exchanged between the different components that are responsible for the registration process of a user.

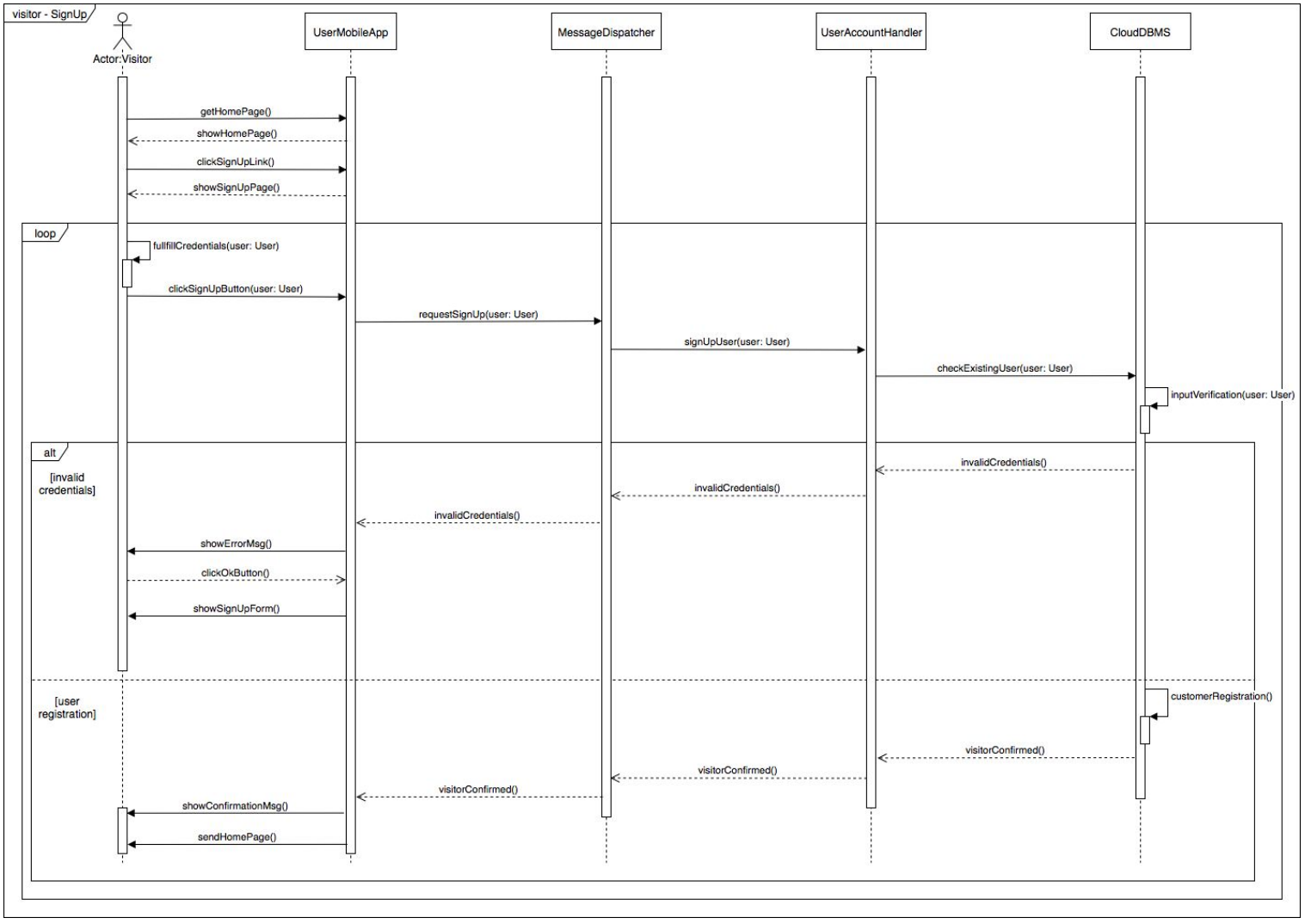

Figure 4 - Sequence Diagram: Visitor Sign Up

The diagram shows the possibility that a user may enter invalid credentials. For example, it is not possible to enter a fiscal code already used by another registered user. The loop continues until the user's registration is successful. Since the diagram representing the registration by an operator of the authorities is very similar to this one, it has been omitted.

User Login

The following diagram shows the flow of messages that are exchanged between components during a user's login process.

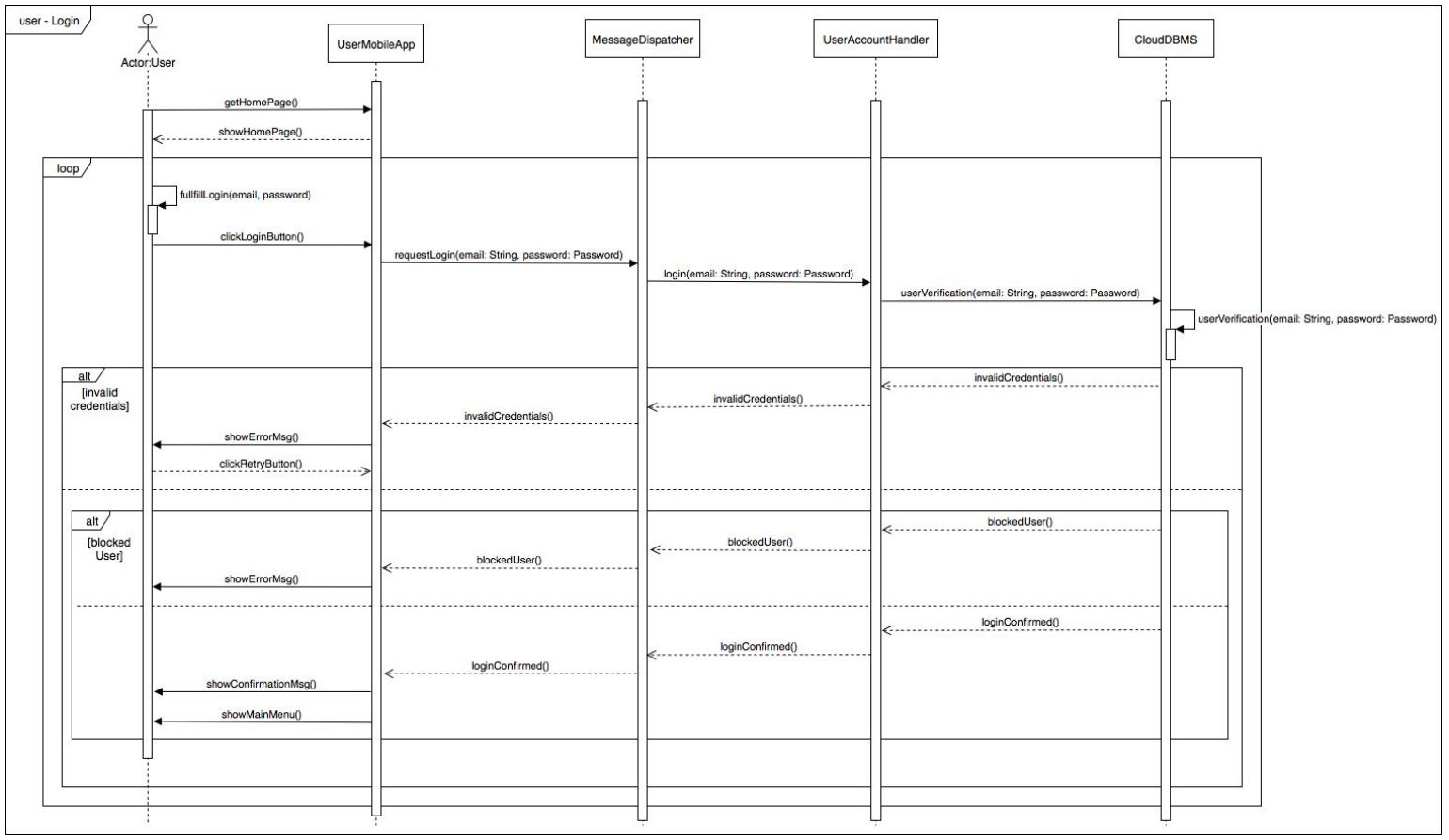

Figure 5 - Sequence Diagram: User Login

The diagram showing the login of the authorities and SafeStreets operators is similar, so it has not been reported.
In the diagram above you can identify the case in which a user is "blocked" (user who has made 3 or more notifications without reporting a real violation of the Highway Code); it will not be able to login.

User reports a violation

The diagram below shows the messages exchanged between components when a user notifies a traffic violation.

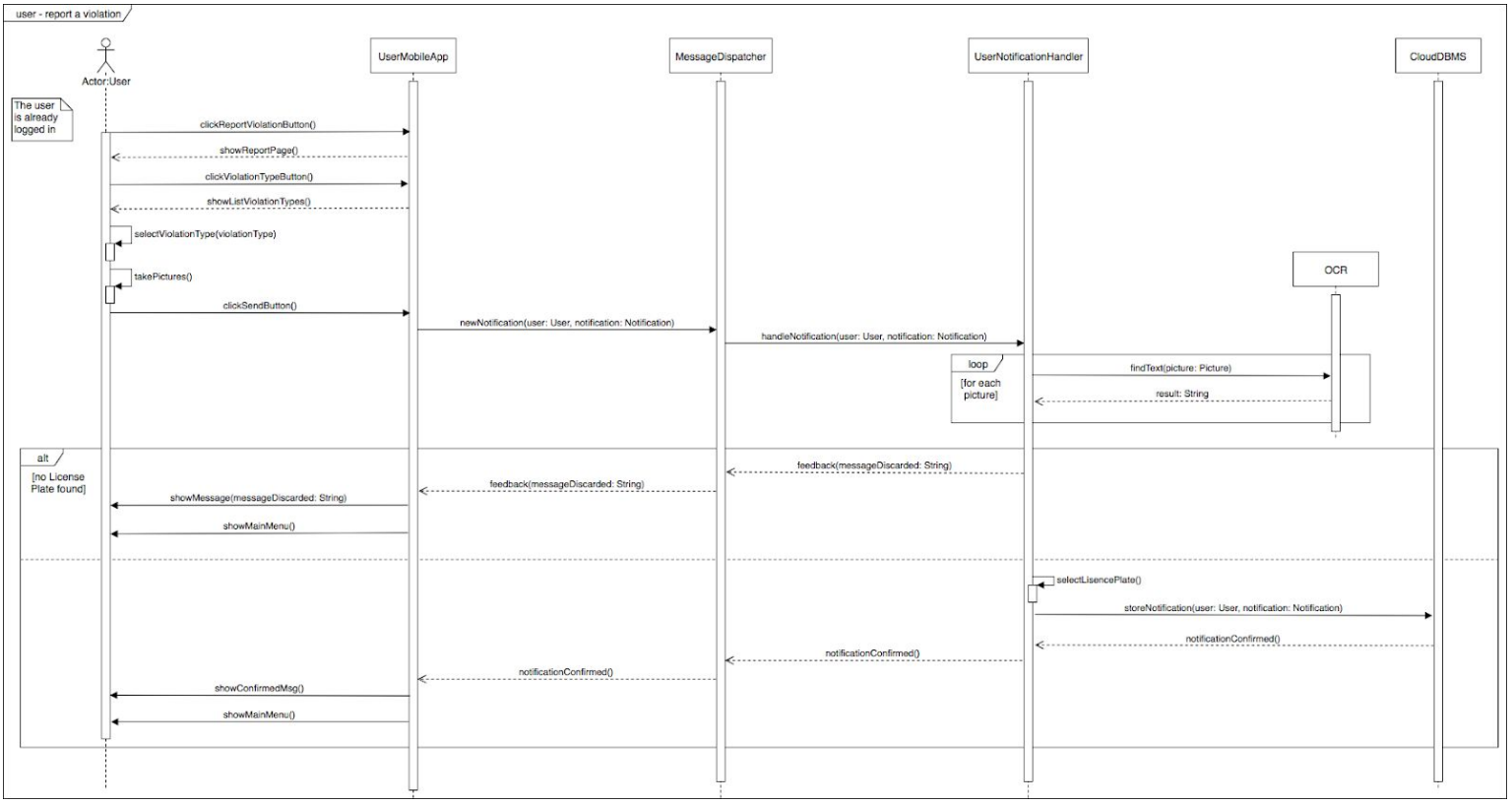

Figure 6 - Sequence Diagram: User reports a violation

The loop involving the UserNotificationHandler and OCR components is necessary because the search for a license plate is made for each picture that is sent by the user. The result UserNotificationHandler receives from OCR indicates if at least one license plate has been found. If no license plate is found, the notification reported by the user is discarded, otherwise all the relevant information is saved in the database. In both cases, the user who made the notification receives feedback.

User searches for dangerous streets

This diagram shows the interactions and the set of messages exchanged between the components involved in a user's request for streets classified as dangerous.

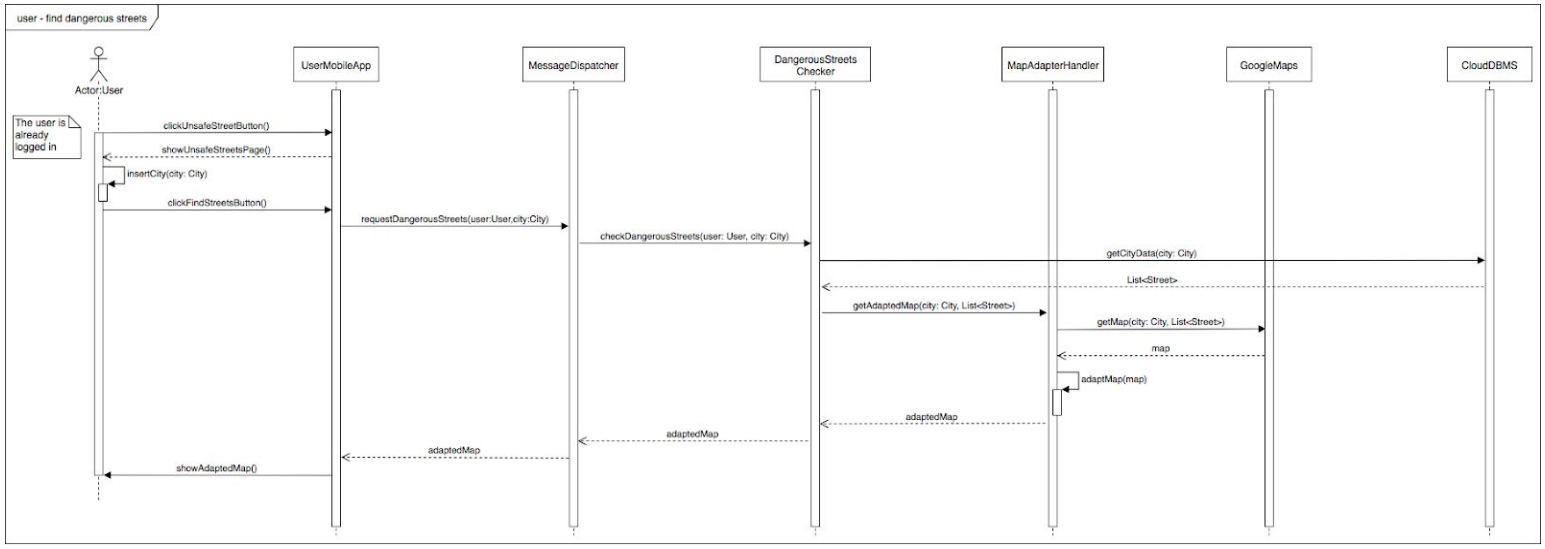

Figure 7 - Sequence Diagram: User searches for dangerous streets

The diagram for the search for dangerous streets by an authority operator is very similar to the one presented above. It's very similar also the sequence diagram for the search of dangerous vehicles by an operator of the authorities. Therefore, these two diagrams are omitted.

SafeStreets operator analyzes notifications

The following diagram shows the flow of messages that are exchanged between components when a SafeStreets operator wants to analyze the notifications made by users.

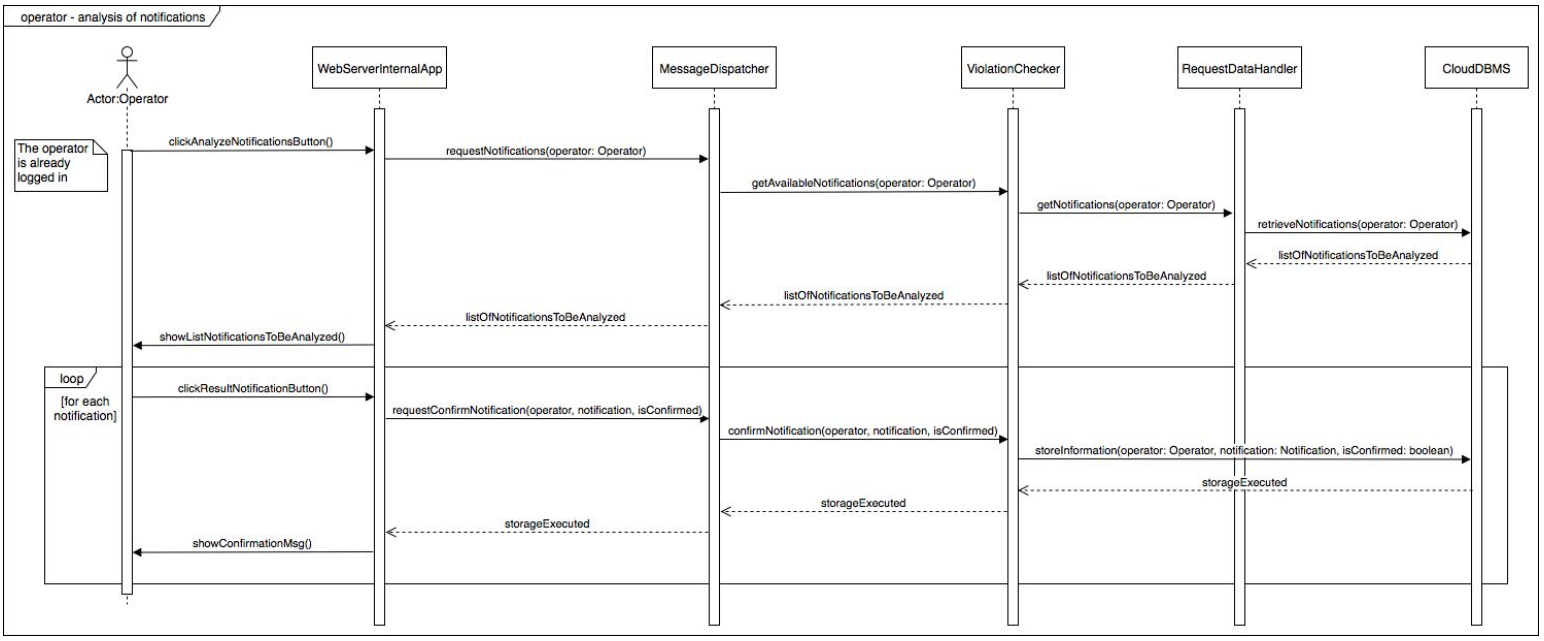

Figure 8 - Sequence Diagram: A SafeStreets operator analyzes the notifications made by users

The messages *requestConfirmNotification(...)* and *confirmNotification(...)* contain the *isConfirmed* attribute, a Boolean indicating whether the corresponding notification has been confirmed by the operator or not. In both cases, the relevant violation information is saved in the cloud database. If the notification is confirmed, it becomes a true Violation.
All violations must then be confirmed by an operator of the authority, which in case will generate the respective traffic ticket to be sent to the person responsible for the violation of the Highway Code.
The loop continues for all notifications that the SafeStreets operator wants to analyze.

Authority analyzes violations

The following diagram shows the flow of messages that are exchanged between components when an authority operator wants to analyze the violations saved by SafeStreets.

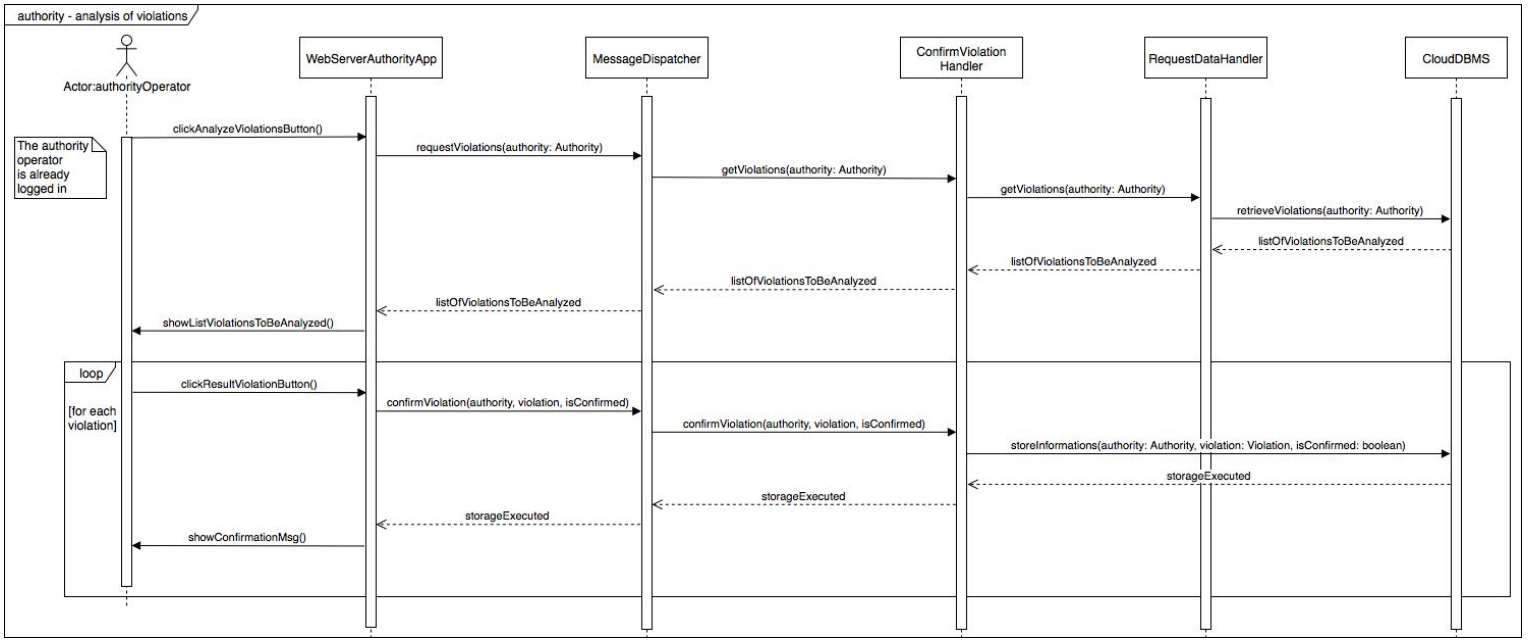

Figure 9 - Sequence Diagram: An authority operator analyzes the violations saved by SafeStreets

The messages *confirmViolation(...)* contain the *isConfirmed* attribute, a Boolean indicating whether the corresponding violation has been confirmed by the authority operator or not. In both cases, the relevant violation information is saved in the cloud database.
The loop continues for all violations that the authority operator wants to analyze.
In this diagram the messages related to the generation and emission of the relative traffic ticket, in case of confirmed violation, have been omitted.

2.5 Component Interfaces

2.5.1 User Application Server

The following figure shows the methods belonging to the component interfaces of the UserApplicationServer represented in the Component Diagram.

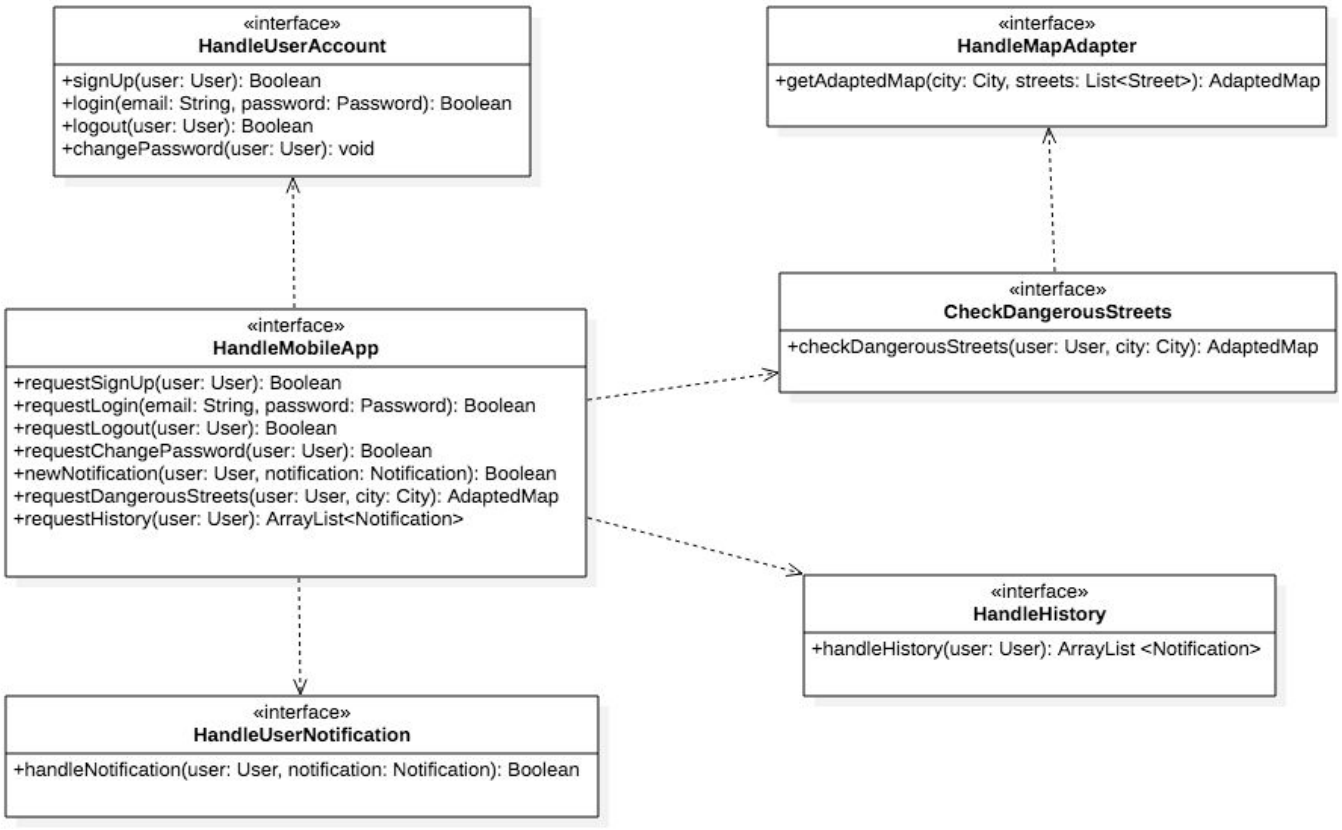

Figure 10 - Component Interfaces: UserApplicationServer

The names of the methods defined in the interfaces above are self-explanatory and reasons for their use can be understood thanks to the sequence diagrams in section 2.4.
The *Password* class is required to ensure the security of the password forwarding. This class contains a *hashPassword()* method that allows to share the hashed password and not to save the password in clear text in the database.

2.5.2 Application Server

The following diagram shows the methods belonging to the ApplicationServer component interfaces represented in the Component Diagram.

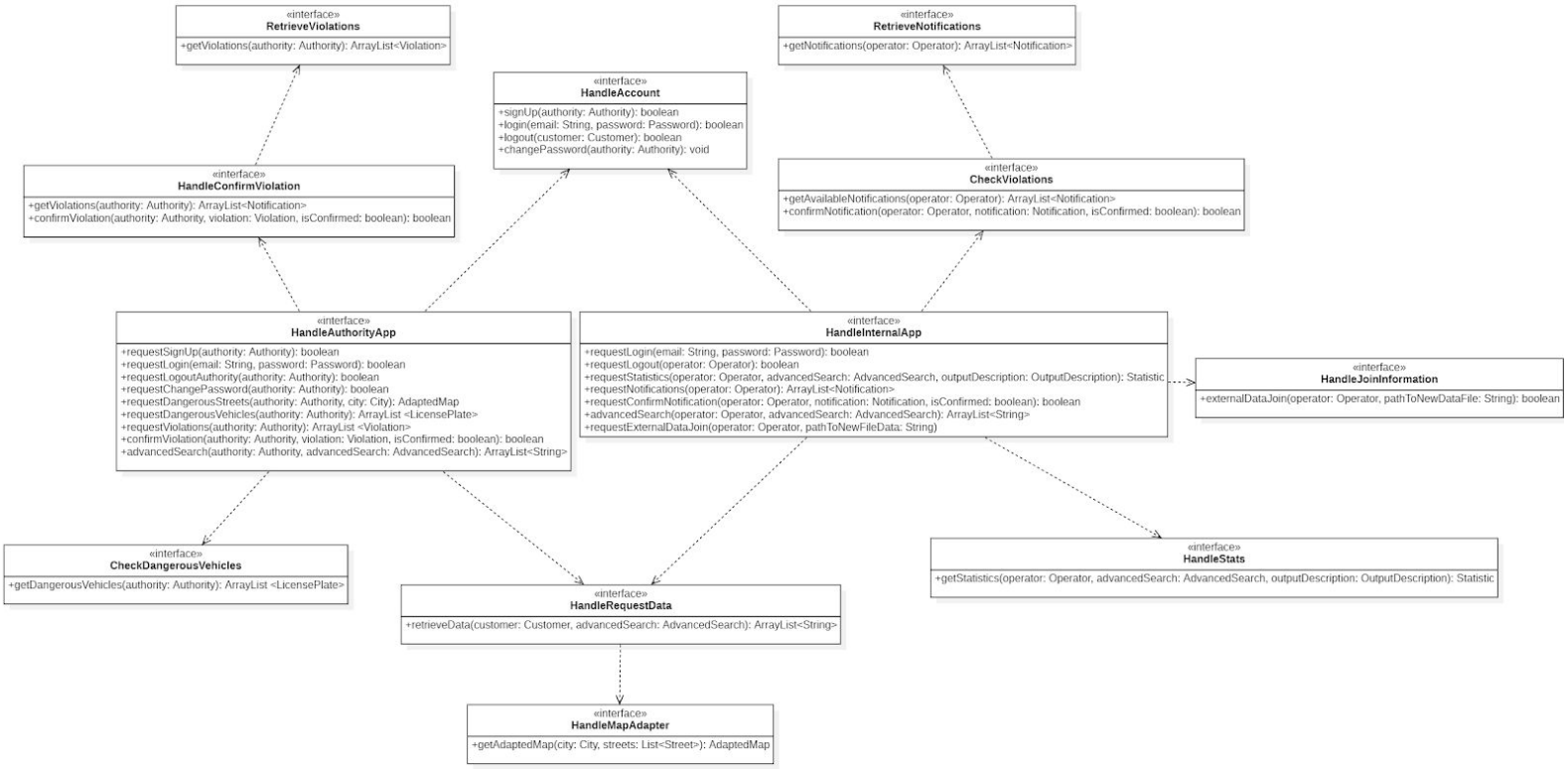

Figure 11 - Component Interfaces: ApplicationServer

The names of the methods defined in the interfaces above are self-explanatory and the reason for their usage can be understood thanks to the sequence diagrams in section 2.4.
The *Password* class is required to ensure the security of the password forwarding. This class contains a *hashPassword()* method that allows to share only the hashed password and not to save the password in clear text in the database.
For what concerns the *getStatistics(...)* method, the *AdvancedSearch* parameter is used to filter the data on which the operator needs to build statistics. The *OutputDescription* class is used to communicate the output format which means that it is used to indicate whether a graph or table is required for the final representation; it must also indicate whether it is required that the results shall be ordered and, if so, on which parameters this sorting should be based.

2.6 Architectural Styles and Patterns

2.6.1 Architectural Styles

- *UserApplicationServer* and *ApplicationServer*: The classic three-tier architecture has been adapted to SafeStreets' business logic and its benefits have been enhanced. In particular, the ApplicationServer has been divided into two parts because it is expected that the workload of the servers used to receive requests from users is much greater than the one of the servers that deal with managing authorities and operators' requests. This choice leads to a greater modularity of the application level and therefore allows to differentiate the need for scalability according to the growth in use of the service. Users are expected to increase rapidly and thanks to the modularity of the application servers, it is possible to optimize resources and infrastructure costs.

- *CloudDatabaseServer*: The choice to use a cloud-based database leads to important advantages:

    - Simplified administration
    - High scalability
    - High availability and durability
    - High levels of security
    - Reduced management costs

  The cloud-based database solution also allows external and secure management of user password recovery, via SMTP servers.

- *WebServer*: the WebServer must be implemented with a three-tier architecture. In addiction of the usual advantages of modular software with well-defined interfaces, the three-tier architecture is intended to allow any of the three tiers to be upgraded or replaced independently in response to changes in requirements or technology.

- *RESTful architecture*: The architecture that must be used is the RESTful, the objective pursued is to reduce the coupling among client and server components as much as possible. Moreover, we want to be able to update the server software without touching the client software and viceversa, this is done in order have the possibility of bug fixing in both part without interferring with the working one. In order to be a true RESTful API, the system must adhere to the following six REST architectural constraints:
    - **Use of a uniform interface (UI).** Resources should be uniquely identifiable through a single URL, and only by using the underlying methods of the network protocol, such as DELETE, PUT and GET with HTTP, should it be possible to manipulate a resource.
    - **Client-server based.** as explained so far, there should be a clear delineation between the client and server. UI and request-gathering concerns are the client's

domain. Data access, workload management and security are the server's domain. This loose coupling of the client and server enables each to be developed and enhanced independent of the other.

- **Stateless operations.** All client-server operations should be stateless, and any state management that is required should take place on the client, not the server.
- **RESTful resource caching**. All resources should allow caching unless explicitly indicated that caching is not possible.
- **Layered system.** REST allows for an architecture composed of multiple layers of servers.
- **Code on demand.** server will send back static representations of resources in the form of XML or JSON. However, when necessary, servers can send executable code to the client.

Concerning the client-server architecture, the authority and the operator will use the appropriate web app, which communicates with the web server, which in turn is connected with the application server.
The user will use the native MobileApp to access directly the application server.
Finally, the application server will also acts like a client when it comes the time to query the CloudDBMS for data.
The aftermentionied communications, since a RESTful architecture is used, will exploit the HTTP protocol, using TSL when dealing with sensitive data in order to guarantee the security and the reliability of the connection and also to authenticate the identity of the communicating parties.
The data are transmitted using XML is used because it is suitable for managing data interchange between client, server and database.

2.6.2 Design Patterns

- *Layered architecture:* The three-layer approach allows to limit coupling and functional duplication. The distribution of application responsibilities across multiple objects leads to complex functionalities achieved through the collaboration of more elements with less complexity. This permits to write more robust and maintainable code.

- *MVP:* Model-View-Presenter Pattern for the smartphone APP
  The MVP pattern allows separating the presentation layer from the logic. Everything about how the UI works is agnostic from how it is represented on the Android/IOS device. In this way the code is easily extensible and maintainable.

2.7 Design Decisions

In order to guarantee optimal results, user-friendly graphical interfaces and simple implementation it has been decided to adopt useful external services:

- *GoogleMaps:* SafeStreets' MobileApp and WebApp need an integration with a map service in order to provide a no-compromise and immersive user experience. The best

option for this purpose is to use GoogleMaps through its APIs. GoogleMaps is a highly available and reliable service, and its rich set of functionalities is to be taken into account for possible future expansions of the SafeStreets' environment.

- *NewOCR:* using an AI-based OCR service permits solid results and extremely easy implementation. The NewOCR API allows to request to recognize a plate directly from a photo, with a single method call. The result of the computation is a string, a highly manipulable data type.

- *CloudDatabaseServer:* With a database-as-a-service, SafeStreets do not have to install and maintain the database. Instead, the service provider takes responsibility for installing and maintaining the database, and the cost is charged proportionally to the use of the service.

It is expected that there will be a large amount of data to be handled; to make data easily available and manageable, the cloud database must be based on relational models.
In fact:

- Relational databases are based on a relatively simple data model to implement and manage. Thanks to relational database models, it is easy to map a large amount of information.

- The relational database model uses well-defined rules for eliminating redundancy. Normalization requirements must be implemented consistently, in order to allow the elimination of redundant data.

- Normalized relational databases allow for non-contradictory data storage, contributing to data consistency. Relational database systems provide automatic defining and verifying of integrity constraints.

- The language for formulating queries is the well-known SQL, used globally and strongly standardized.

The relational model of the cloud database must be based on this Class Diagram:

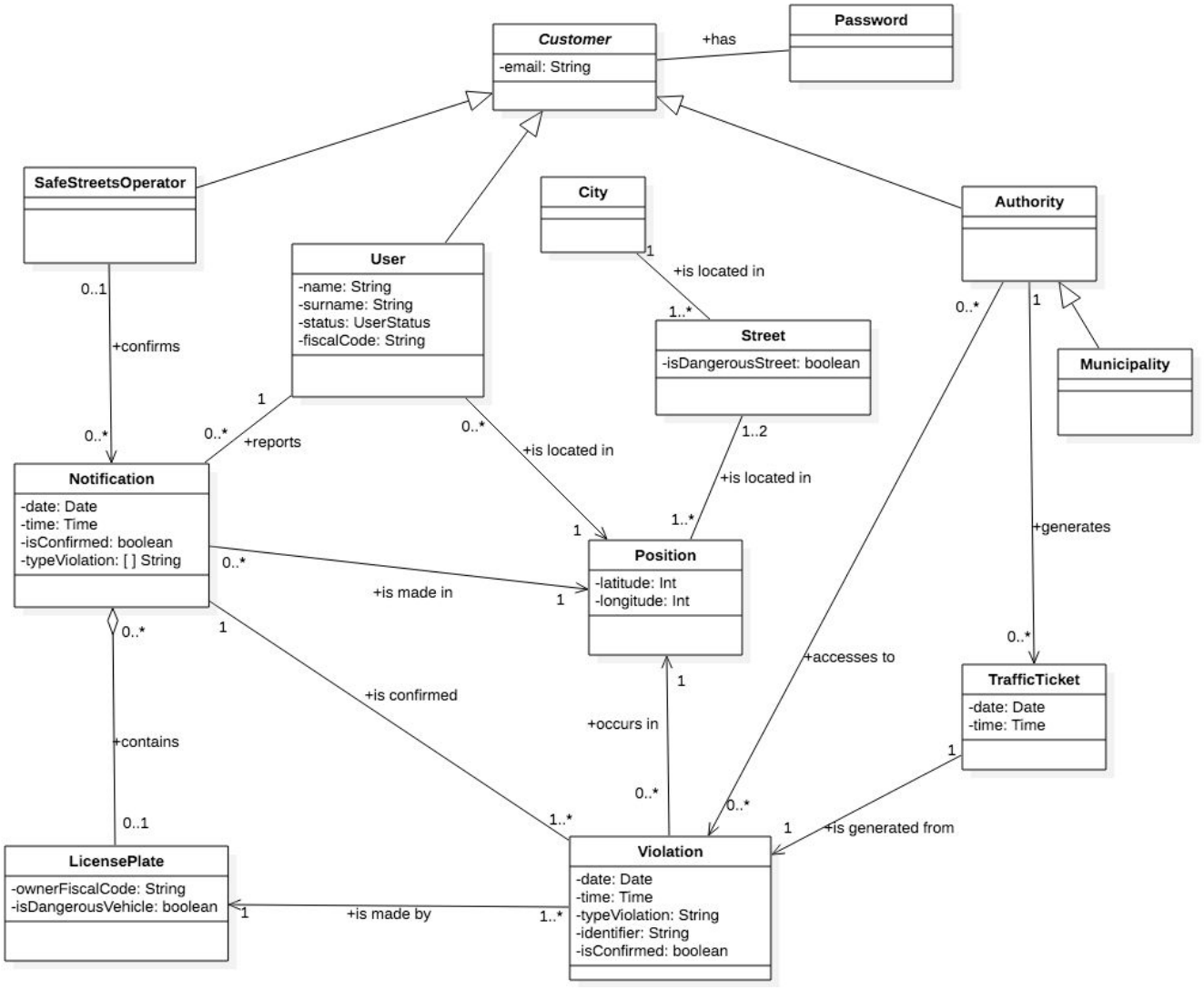

Figure 12 - Class Diagram

It contains the essential attributes of the classes evidenced, the more important ones of the model of the system and not the whole set of classes that will be useful to define it.

3. User Interface Design

3.1 User Interfaces

Mock-ups of both User and Authority applications were presented in the RASD Document in section *3.1.1. User interfaces*. Below is presented a new mock-up that represents the Analyze Notifications action, performed by a SafeStreets Operator.

The main frame shows the list of notifications that SafeStreets has collected and that still need to be analyzed by operators. For each notification it is possible to change the License Plate and the Violation Type associated with the report. If the notification is valid, the operator shall click "Confirm Notification" button, otherwise "Reject Notification".

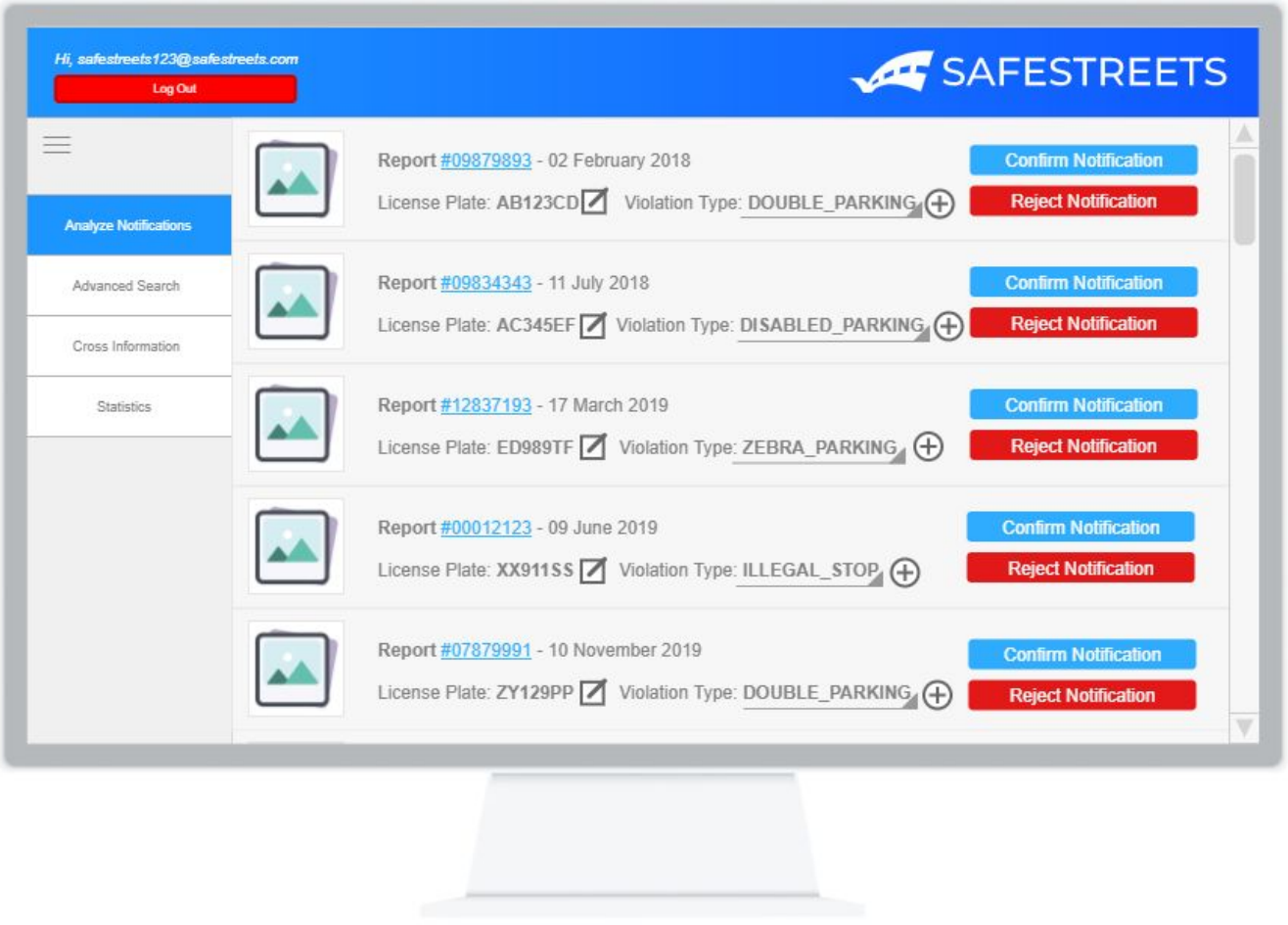

Figure 13 - SafeStreets Operator WebApp

3.2 UX Diagrams

Below we have provided some UX diagrams showing how and which actions can be performed by the different actors, in accordance with the mockups reported in the RASD.
To avoid misunderstandings, we provide an explanation of the symbols used: the oval symbolize the start or the end in a flowchart; the rectangle symbolize the screens of the application; the parallelogram symbolize the actor's input (including buttons); the diamond symbolize decisions; the arrow symbol is used to represent a flow direction.

In the first diagram it is possible to identify the actions that a visitor can perform in the application. The initial screen is the login one, from which it is possible to access the registration page if the visitor has never registered. If the actor has to log in, he can enter his credentials and, if correct, access the main menu, or he can request a new password if forgotten. In addition, the "Main Menu" screen that follows the login has been inserted as a terminator for a better understanding of the different graphs. The following ones show the actions that the different logged in actors can perform starting from the main menu.

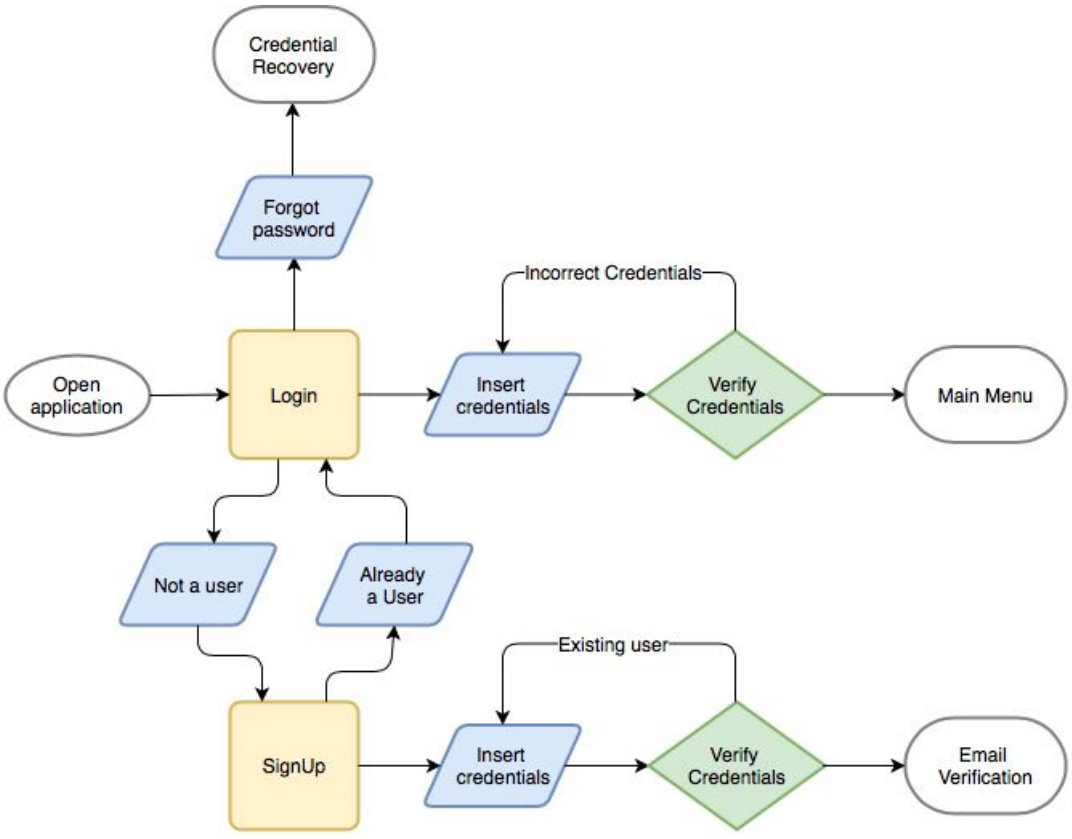

Figure 14 - UX Diagram: User and (SafeStreets and authority) operators

The following diagram shows the actions that a logged in user can perform (starting from the main menu screen). A user can report a violation (by entering the different photos that report it and selecting the type(s) of violations committed), look at the history of the reports made by him or view the streets classified as dangerous in the city he entered as a search parameter.

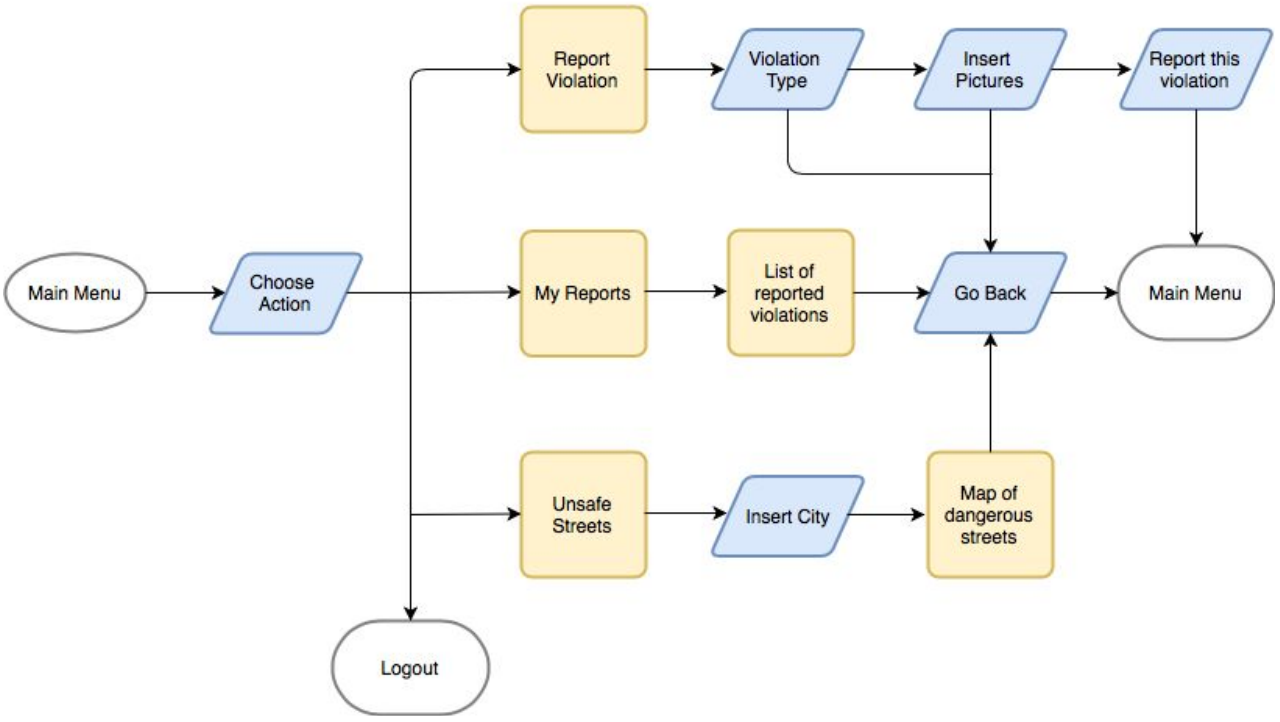

Figure 15 - UX Diagram: User

The following diagram shows the actions that an operator of the logged in authority can perform (from the main menu screen). An authority operator can:

- analyze violations saved by SafeStreets by confirming or rejecting them;
- search for vehicles that have been classified as dangerous;
- search for roads classified as dangerous of a city entered as a search parameter by the operator himself;
- do an advanced search, having the possibility to enter parameters such as city, date, license plate, type of violation.

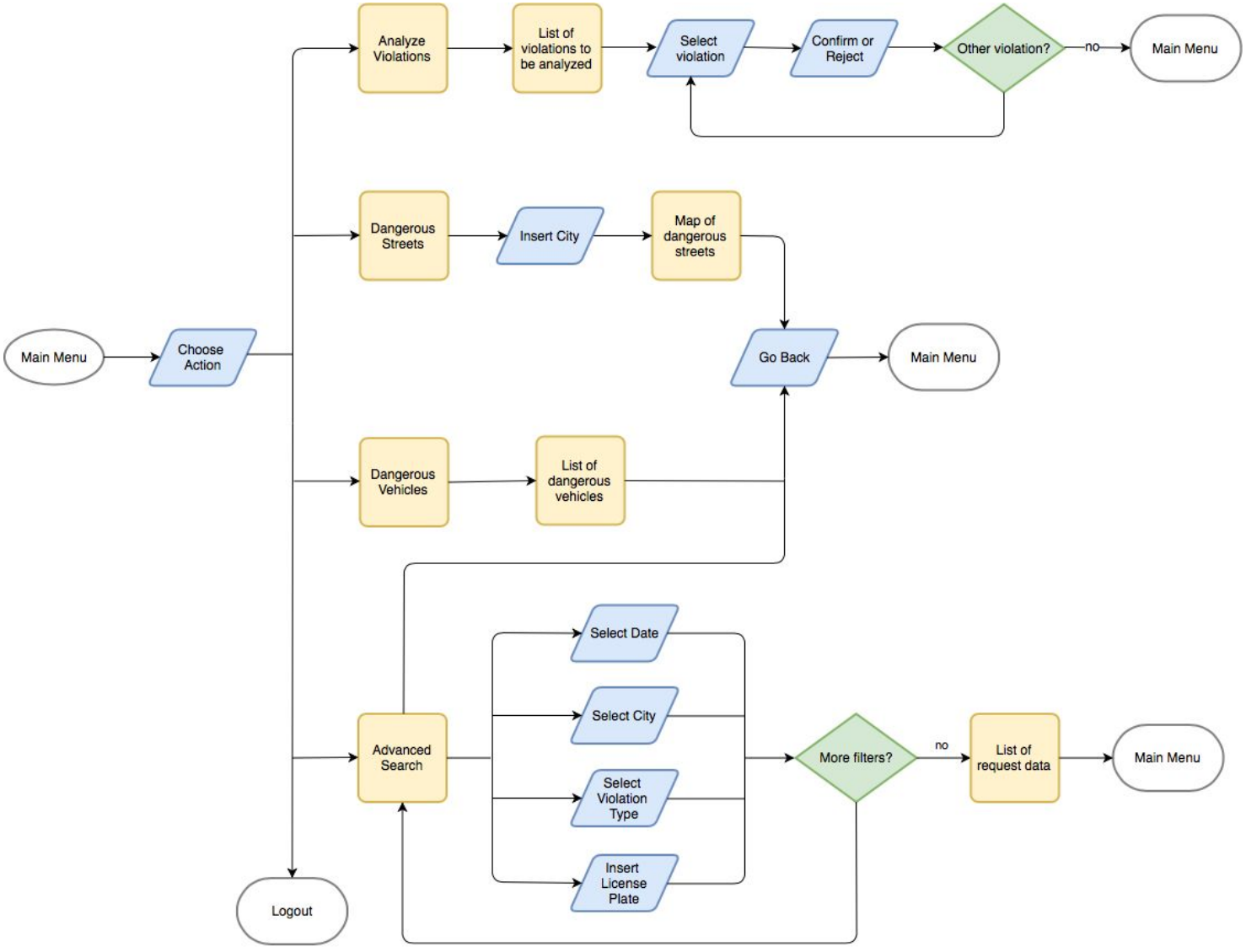

Figure 16 - UX Diagram: Authority operator

The following diagram shows the actions that an operator of the logged in SafeStreets operator can perform (from the main menu screen).
A SafeStreets operator can:

- Analyze the stored notifications reported by users;
- Generate statistics on saved data;
- Perform advanced searches, having the possibility to enter parameters such as city, date, license plate, type of violation;
- Cross information given by municipalities.

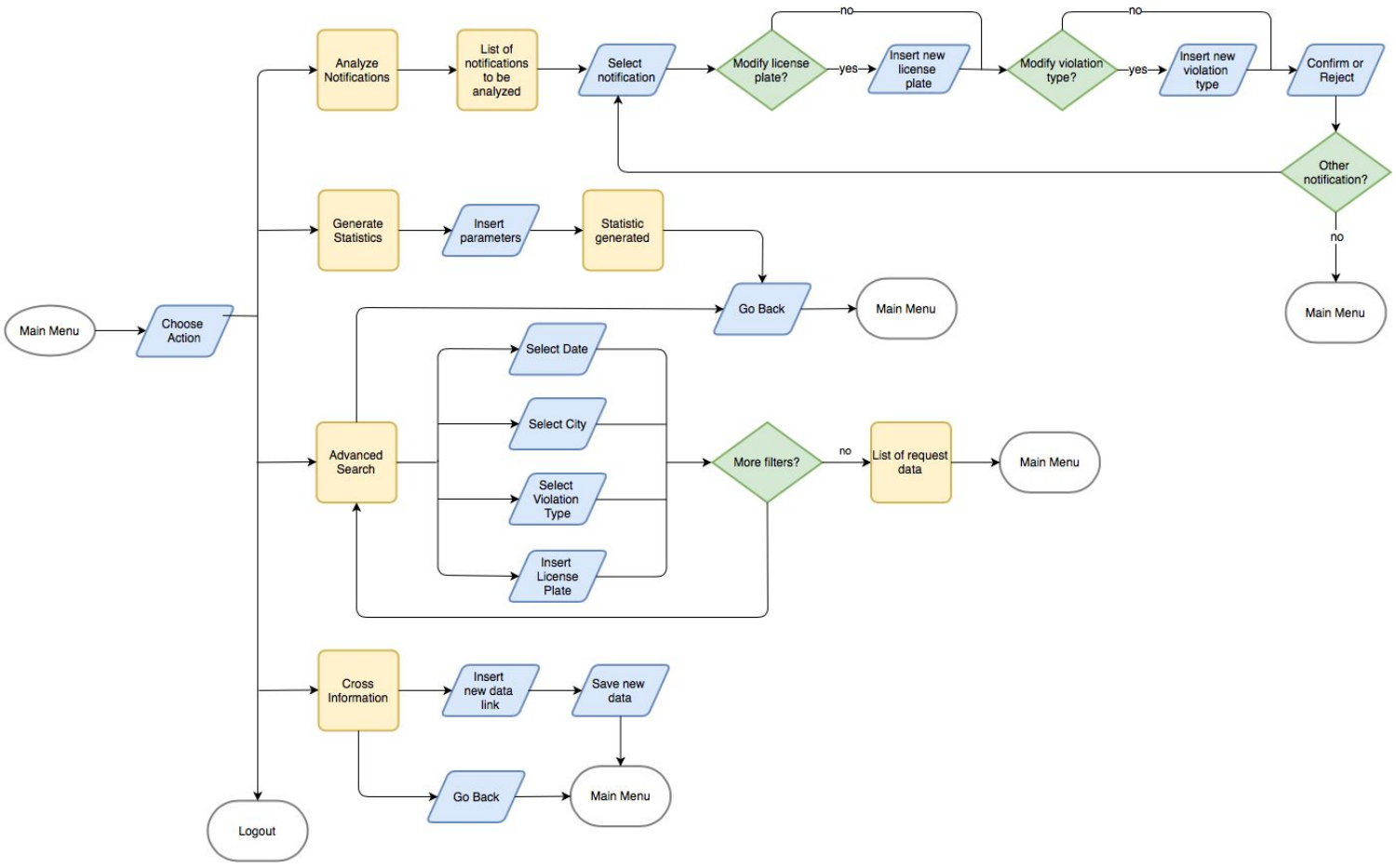

Figure 17 - UX Diagram: SafeStreets operator

4. Requirements Traceability

The entire design must ensure that the system is able to enforce all the requirements defined in the previous document (RASD) and consequently that the system is able to achieve the goals that have been established.
This chapter shows a mapping between the established requirements and the design components illustrated in the DD.

- R1: The system should be available 24/7.
    - For this requirement all defined components are required.
- R2: The system offers the possibility to register, with the insertion of e-mail, password and fiscal code.
    - UserAccountHandler
- R3: The email used must be different for each account.
    - UserAccountHandler
- R4: The system should allow registered users to exploit its features.
    - UserAccountHandler
    - UserNotificationHandler
    - HistoryHandler
    - DangerousStreetsChecker
    - MapAdapterHandler
- R5: The system should allow users to take a picture instantly and send it.
    - UserNotificationHandler
- R6: The system offers a list of possible types of violations to report.
    - UserNotificationHandler
- R7: The system must save the information about the notified violations.
    - UserNotificationHandler
    - ViolationChecker
- R8: The system must allow to verify the correctness of the notification made.
    - ViolationChecker

    - ConfirmViolationHandler

- R9: An algorithm allows to recognize license plates involved in a notified violation.
    - UserNotificationHandler
- R10: A user is blocked if he makes more than 3 incorrect reports (there is no traffic violation in the report).
    - UserAccountHandler
    - ViolationChecker
- R11: The system has to ask the authorities which parameters to use to filter data.
    - RequestDataHandler
    - DangerousVehiclesChecker
- R12: The system can receive the data made available by the municipalities.
    - JoinInformationHandler
- R13: The data to be integrated must be made homogeneous (data saved by SafeStreets and data coming from the municipality).
    - JoinInformationHandler
- R14: The system offers the possibility to the employees of the authorities to register by entering the working email and a password.
    - AccountHandler
- R15: The system must allow access to the stored data.
    - For this requirement all defined components are required

5. Planning

The system is divided into different subsystems :

- UserMobileApp
- WebServerAuthorityApp
- WebServerInternalApp
- UserApplicationServer
- ApplicationServer

External Systems:

- OCR
- GoogleMaps
- CloudDBMS

The implementation of these subsystems must follow a bottom-up approach, but since there is a very low level of coupling between the components of different subsystems, this approach can be combined to a strategy that also consider the importance and the difficulty of implementation of the different functionalities that the system provides.
Since in this document only the UserApplicationServer and the ApplicationServer have been described with detail, even in this section the attention will be focused on these two components, according to the fact that the integration and testing with the other subsystems will take place at a later time.
The external systems are implemented yet and they will be used as they are so it isn't needed to implement and unit test them.
The following table lists the features available for the costumer and the internal operators:

Features	**Importance for the Costumer**	**Difficulty of implementation**
Sign Up and Log In	Low	Low
Notify a violation	High	High
Access to History	Medium	Medium
Find Dangerous streets	High	Medium
Stats Reading	Medium	Medium

Information Crossing	Medium	High
Analyse a Violation (both operators and authorities)	High	Medium
Find Dangerous Vehicles	High	Medium
Extract Customized Data	High	High

5.1 Implementation Plan

For what concerns the implementation of the components belonging to UserApplicationServer and the ApplicationServer, it must be considered that since the two subsystems are independent, it is possible to develop their components in a parallel way and this is the order that must be followed:

5.1.1 UserApplicationServer

- **Notify a violation:** For providing this feature is needed to implement the "UserNotificationHandler", which is the only component in the UserApplicationServer that is thought for this feature.
- **Find dangerous streets:** To provide this function, the components that must be implemented are "MapAdapterHandler" and "DangerousStreetsChecker" in this order, because the first one is used by the second one and the order is important since we are following a bottom up approach, after that the two components must be integrated.
- **Access to history:** To guarantee this function is enough to implement the HistoryHandler.
- **Sign Up and Log in:** This functionality is the one in charge of managing the users' accounts and in order to make it available the "UserAccountHandler" component must be implemented

5.1.2 ApplicationServer

- **Extract Customized Data:** This is the core function for the ApplicationServer and it is made available thanks to the implementation, in this order, of "MapAdapterHandler" and "RequestDataHandler", with the purpose of following the bottom up approach, and consequently the integration of the two.
- **Analyse a violation (Operator):** For providing this functionality to the operators, the "ViolationChecker" component needs to be implemented, after it has been implemented it needs to be integrated with the "RequestDataHandler" which at this point must already be implemented.
- **Analyse a violation (Authority):** This functionality needs to be provided to the authorities and in order to do that the "ConfirmViolationHandler" component must be developed and integrated with the already implemented "RequestDataHandler".

- **Find dangerous vehicles:** To guarantee this function the “DangerousVehiclesChecker” component must be implemented
- **Information crossing:** This feature is provided thanks to the implementation of the component called “JoinInformationHandler”
- **Stats reading:** This feature is made available implementing “StatsHandler” that is the inly internal component thought for this function.
- **Sign Up and Log in:** This functionality is the one in charge of managing the Operator’s accounts (Both from SafeStreets and authorities) and in order to make it available the “AccountHandler” component must be implemented.

In the end, for both subsystems, the “MessageDispatcher” must be implemented (his duty is only to call functions from other components), it is the last component to be developed because it is on top of hierarchy since it’s used only from outside and it exploit the function of the other internal components. After the implementation, its integration with the rest of the components in the application server must be performed.

5.2 Integration Plan

In this paragraph it will be explained, in a graphical way, how the different components must be integrated, for each one of these graphs must be clear that the involved components are already full implemented and unit tested, in each graph the arrows are directed from the “User component” to the “Used component”.

For what concerns the external components, since they are provided from the outside and that can be a worthless effort to develop stubs for them, because we can rely on their correctness, so they’re integration with the internal component must be done as a first step.

The next graph explain how the external OCR is used by the system and with which component it must be integrated.

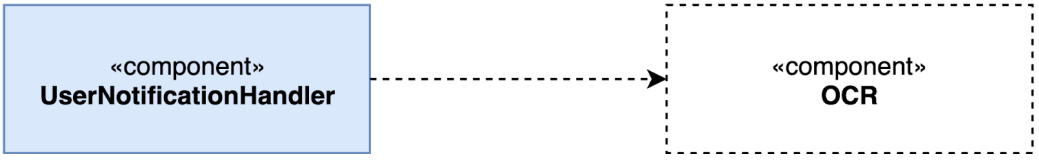

Figure 18 - OCR Integration

The following graph is used to illustrate which components needs to be integrated with the GoogleMaps component, it is important to specify that the components that aren't connected can be implemented independently, and more precisely, they must follow the order described in the previous section, so for what concern this graph, each component must be implemented with GoogleMaps as soon as his development and unit test is completed.

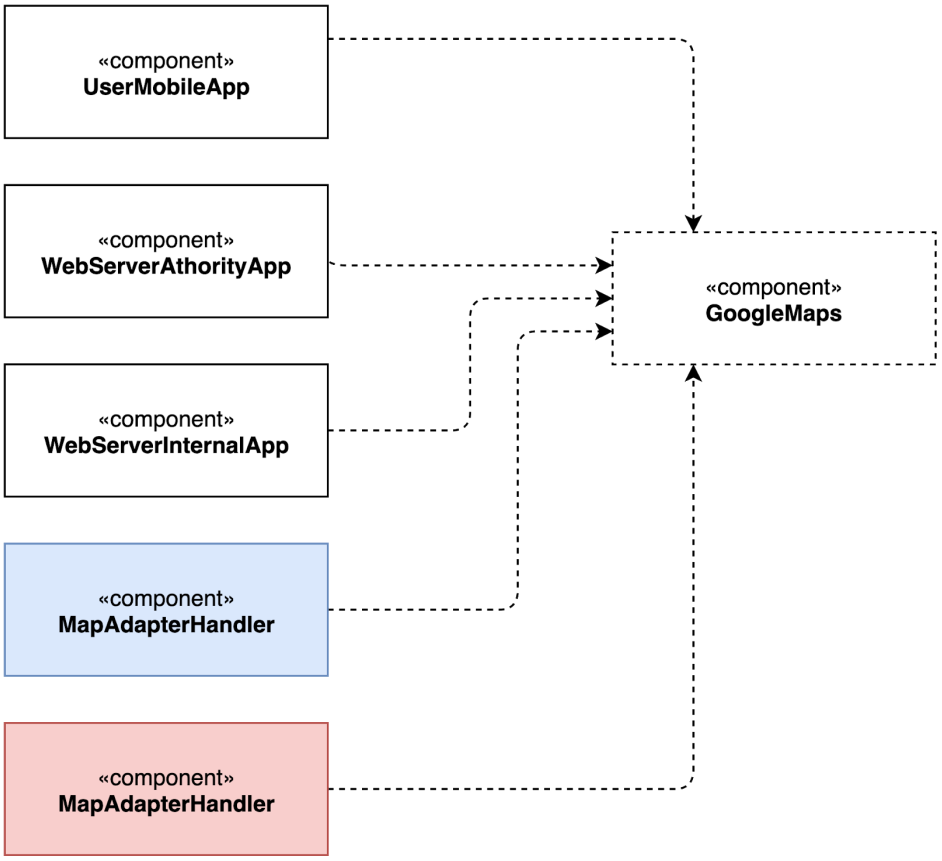

Figure 19 - GoogleMaps integration

This graph is used in order to explain which are the components that must be integrated with the CloudDBMS.

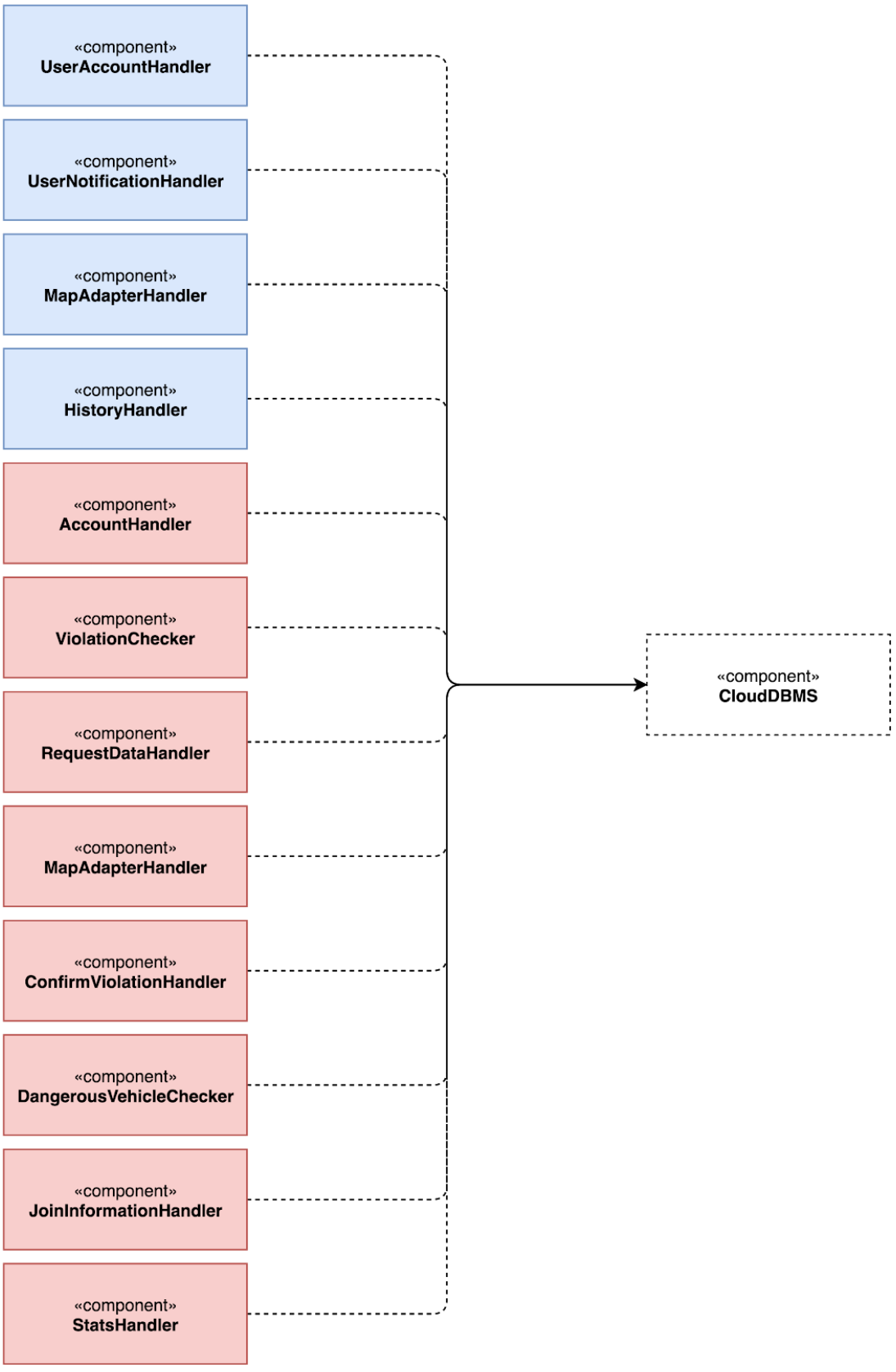

Figure 20 - CloudDBMS integration

For what concerns the integration of the UserApplicationServer's components, the following order of the graphs reflects the real order that must be used to integrate the components:

This first graph points out that the first two components to be integrated are "DangerousStreetsChecker" and "MapAdapterHandler" because the first uses the second's features.

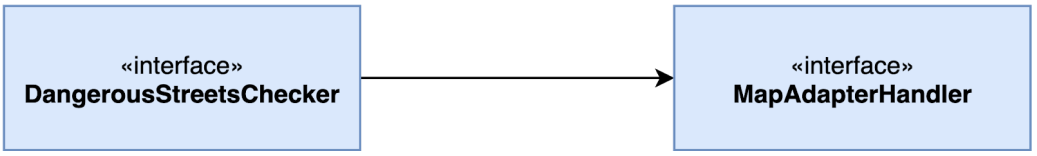

Figure 21 - DangerousStreetsChecker

The following graph is the last one regarding the internal components of the UserApplicationServer and it is about the MessageDispatcher, it must be integrated with the pointed component as soon as they're implemented and tested, in the end, the Arrow from "UserMobileApp" indicates that "MessageDispatcher" will be the connection point for the future subsystem integration.

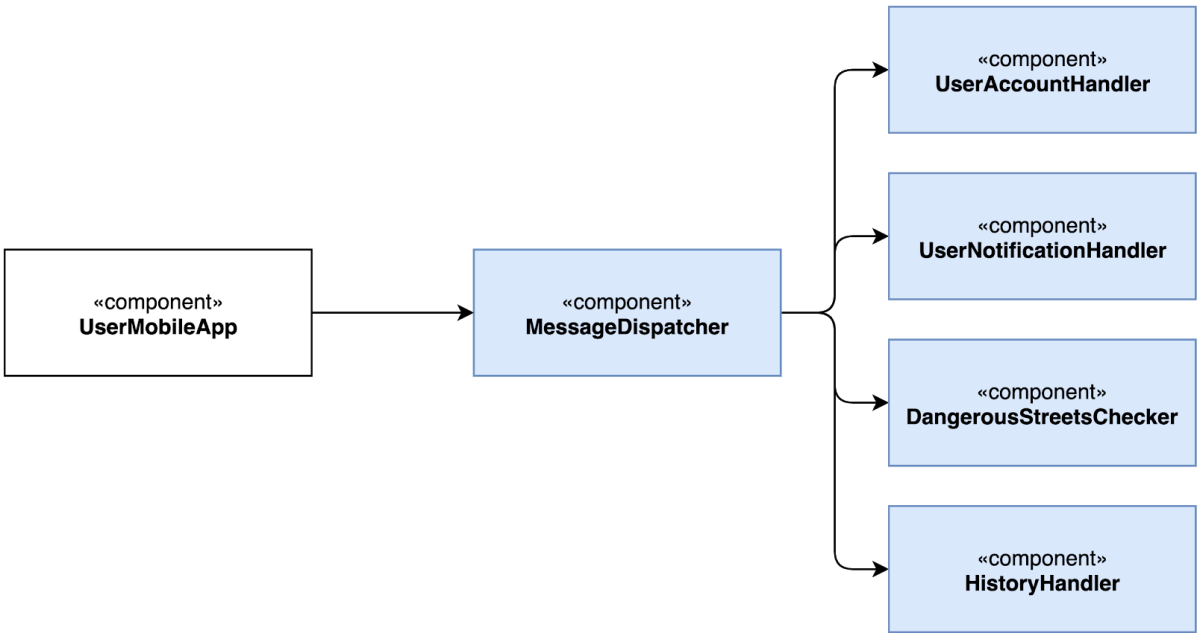

Figure 22 - MessageDispatcher Component integration

Finally, with regard to the ApplicationServer's component the order to be followed is the one of these two listed graphs:

In this graph the integration must be done "Backward", namely the first two component to be integrated are the RequestDataHandler and the MapAdapterHandler, after them the other two components can be integrated as soon as they're ready.

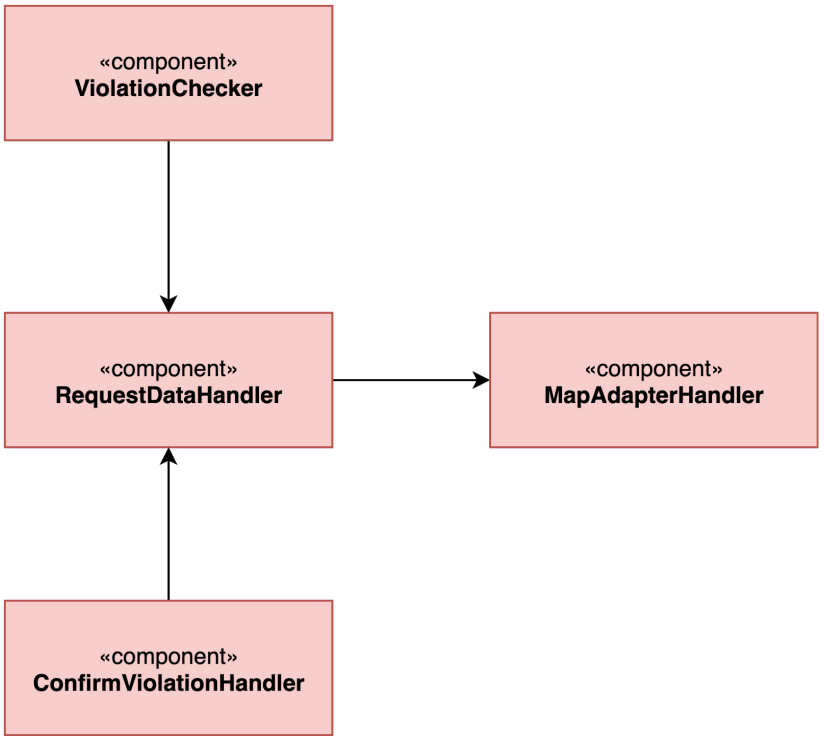

Figure 23 - RequestDataHandler integration

This graph represents the integration of the MessageDispatcher for the ApplicationServer it must be integrated with the pointed component as soon as they're implemented and tested, in the end, the arrows from "WebServerAuthorityApp" and "WebServerInternalApp" indicates that "MessageDispatcher" will be the connection point for the future subsystem integration.

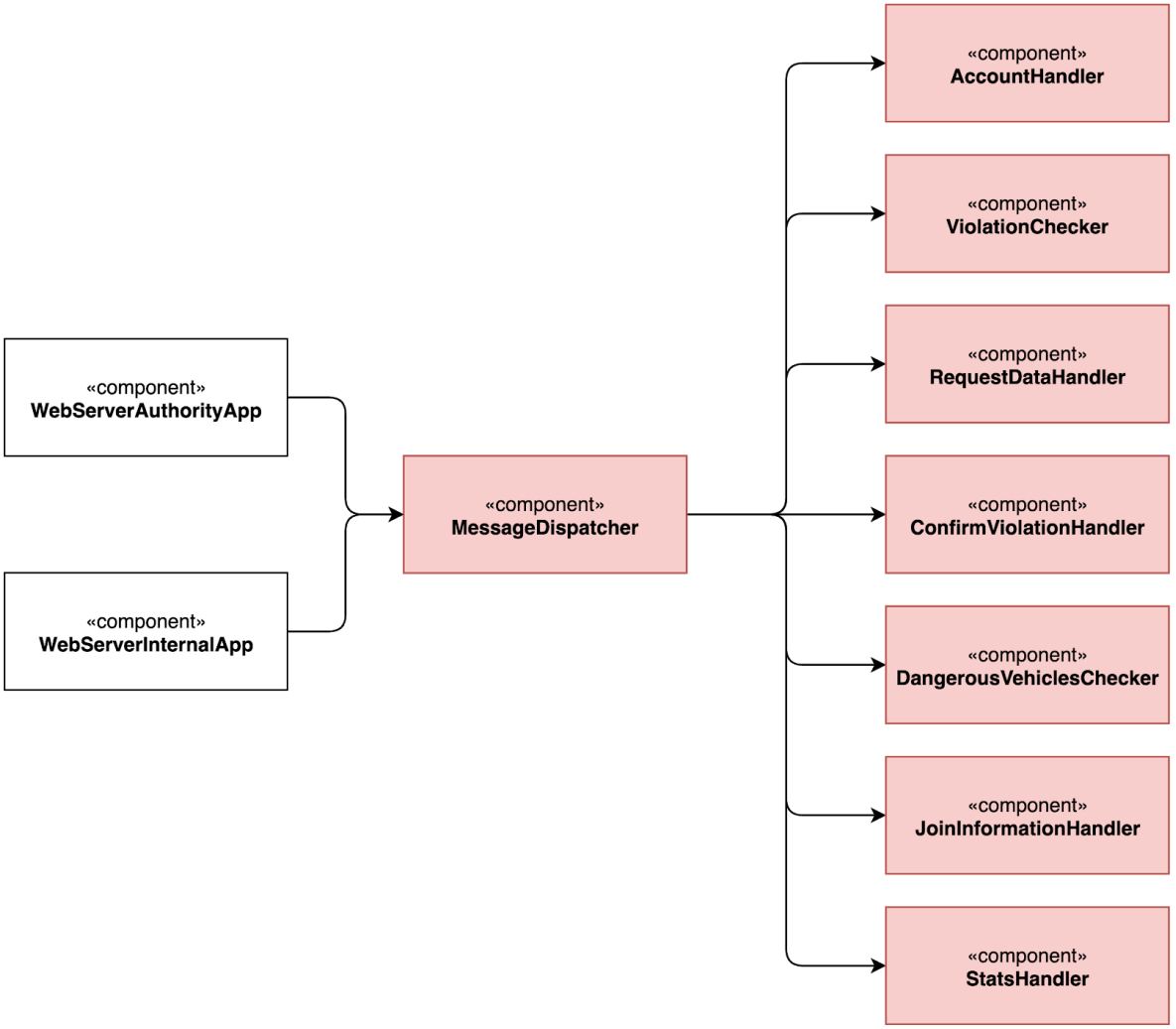

Figure 24 - MessageDispatcher integration

5.3 Test Plan

The Verification and Validation phase must begin as soon as the development phase begins, this means that the Unit Tests must be done in parallel with the implementation of the components, using a whitebox approach, and also the Integration Tests must be start in the moment in which the integration itself begins, in this case using a blackbox approach. This incremental testing plan will ensure that the majority of bugs will emerge in the early phases of production so it will be easier to correct them.

Once the system is completely integrated, it must be overall tested for verifying that functional and non-functional requirements hold. In particular, the load testing must be done very carefully, because the S2B will have to cope with great quantity of data, but also stress and performance testing must be performed, because they help to identify bottlenecks that affects response time and how the system reacts to failures.

6. Effort Spent

6.1 Fiozzi Davide

TASK	**HOURS**
Introduction	1
Overview and Component View	6,5
Overview figure and description	2
Component Interfaces	3
Deployment View	3,5
Arch Styles, Patterns and Design Decisions	4
New Mockup	1
Revision	8
Total	29

6.2 Frantuma Elia

TASK	**HOURS**
Overview and Component View	6,5
Component Interfaces	2
Arch Styles, Patterns and Design Decisions	2
Planning	10
Revision	3
Total	23,5

6.3 Freddi Eleonora

TASK	**HOURS**
Overview and Component View	6,5
Component Diagram	1
Component Interfaces	3
User Interface Design	4
Runtime View	5
Requirements Traceability	0,5
Revision	5
Total	25

7. References

- Specification document: “SafeStreets Mandatory Project Assignment”
- Document “DD to be analyzed AY 2019 2020”
- https://www.casestudy.club/journal/ux-flowchart

Used Tools

- www.draw.io: for component diagram, system architecture, UX diagrams, sequence diagrams
- StarUML 3.1.0: for component interfaces diagram, class diagram
- https://diagrams.visual-paradigm.com/#proj=0&type=DeploymentDiagram: deployment diagram