POLITECNICO
MILANO 1863

SOFTWARE ENGINEERING II PROJECT

SafeStreets

DD – Design Document

Authors:

Matteo POZZI
Sara SACCO
Andrea VENTURA

Professor:

Matteo G. ROSSI

December 9, 2019

version 1.0

Index

1. Introduction

1.1 Purpose

SafeStreets is a mobile application that relies on the help of lawful citizens to make life in the streets less stressful and more organized. As opposed to the RASD, the purpose of this document is to provide a description of the design of the application with enough completeness to allow the development process to proceed with an understanding of what needs to be built and how.

1.2 Scope

In this section, we refer to what has been previously stated in the RASD, providing a general overview of the scope of SafeStreets.

The intent is to create an application that gives people the ability to report and notify violations, e.g. vehicles parked in the middle of bike lanes, or in spots reserved to people with disabilities, to the designated authorities. In particular, citizens should be able to register as users by providing meaningful credentials, so as to avoid wasteful data such as fake accounts, and a way to verify them, e.g. their fiscal code. Once successfully logged in, users should be able to send pictures as proof of vehicles parked illegally and attach additional information to provide authorities with a starting point for the reviewing process, such as the date, the time, the type of violation which is to be reported and the place in which it has occurred, which can be retrieved through the geographical position of the user itself.

SafeStreets stores the information provided by its users and employs it by identifying and highlighting the zones which are found to be subject to the highest amount of violations, making them visible to both authorities and citizens.

Furthermore, SafeStreets wants to exploit its own data by combining it with information about accidents and analyzing it in order to identify zones or streets whose safety could be improved by making interventions, possibly suggesting viable solutions as well. This functionality is developed in collaboration with a third party, i.e. the municipality, meaning its usefulness will depend on the possibility of the municipality itself to share its data and match it with the interface SafeStreets developed for the functionality.

Lastly, SafeStreets strives to assist the local police in generating traffic tickets, and possibly build various statistics of interest. To ensure the effectiveness of this service, it is necessary that the exchange of sensible data which must occur between SafeStreets and the municipality cannot be tampered with in any way, e.g. modifying the picture of the violation at hand. To avoid this scenario, SafeStreets only accepts as reliable information pictures that have been taken within the application itself.

In the following diagram (Figure 1.1), we define the boundaries of SafeStreets by identifying and distinguishing between World and Machine phenomena, with particular attention to the shared ones.

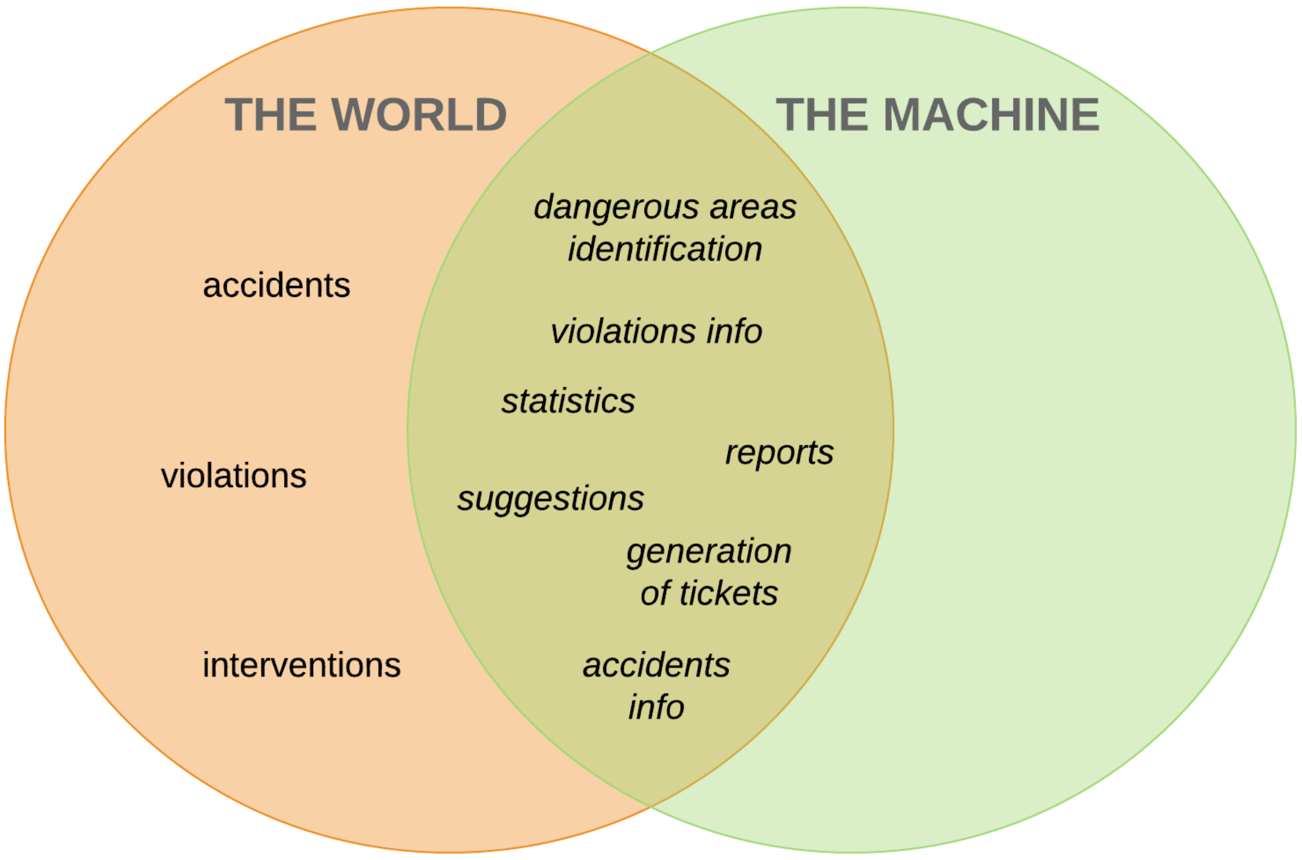


Figure 1.1: World and Machine phenomena.

1.3 Definitions, acronyms, abbreviations

- Definitions:

User: a general actor which is registered into the application; all users can consult statistics about violations and highlight unsafe areas;

Authority: a user which receives complaints and is able to identify actual violations among them. It has the power to punish the culprits with traffic tickets;

Citizen: a user which is not an authority, he can send reports about violations;

Violation: a violation of traffic laws, in particular parking violations;

Accident: a traffic event involving two or more vehicles where people got injured or caused damages to the vehicles

Report: a notification sent by a citizen to indicate violations, containing all the meaningful information about it;

Traffic ticket: a sanction which force an offender of a violation to pay an amount of money, can be generated by authorities;

Unsafe area: an area in which many violations and accidents have been reported;

Statistics: a collection of data about issued traffic tickets for each kind of violation occurred in a certain area.

Suggested intervention: a suggestion made by a system manager to be possibly applied in order to avoid future violations of a certain type.

- Acronyms:

RASD: Requirements Analysis and Specifications Document;

DD: Design Document;

REST: Representational State Transfer;

API: Application Program Interface.

- Abbreviations:

[**Rn**]: n-th requirement.

1.4 Revision history

- **Version 1.0 –** December 9, 2019
    - First Release

1.5 Reference documents

- Specification Document: “SafeStreets Mandatory Project Assignment”

1.6 Document structure

The document at hand is composed of 5 chapters, plus an appendix:

1. Introduction: it includes the goal of the project and an analysis of the world and shared phenomena;
2. Architectural design: here we provide a description of the components used in the application and their interactions;
3. User interface design: this section includes a general overview of how the user interfaces of the application will look like;
4. Requirements traceability: it provides an explanation of how the requirements defined in the RASD map to the design components described in this document;
5. Implementation, integration and test plan: here we identify the implementation plan, the integration plan and the test plan, specifying the order in which each component has to undergo each of the three steps;
6. Appendix: an accessory part that contains a quantitative description of the effort each member put into the completion of the document;

2. Architectural design

2.1 Overview

Considering the fact that data storage plays a major part in the system to be designed, we opted for a three-tier architecture, which is defined as a client-server architecture in which functional logic, data access, computer data storage and user interface are developed and maintained as independent modules on separate platforms. The choice was made to give the IT infrastructures more scalability and flexibility, and in particular to lighten the burden server-side by distributing it on two different nodes. The three tiers involved are:

- Tier One: a light application layer and the presentation layer for the client;
- Tier Two: the application layer for the server;
- Tier Three: the data layer.

Each layer previously mentioned has the following characteristics:

- Presentation layer: it is responsible for displaying information to users and enabling these to input data that needs to be sent to the application layer;
- Application layer: it receives data sent by users and controls the different functionalities of the system by performing detailed processing, sending responses to the previous layer if necessary;
- Data layer: it stores data, keeping it independent from the logic of the application layer.

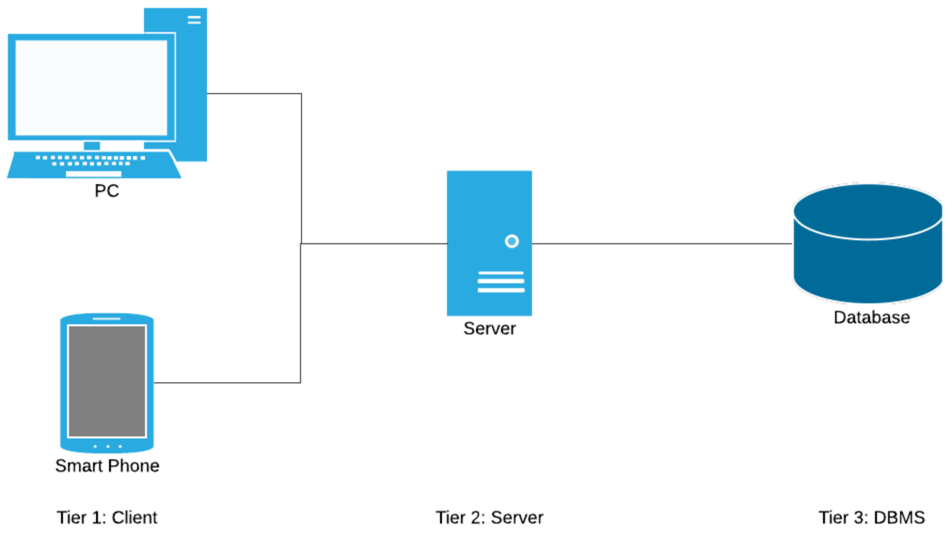


Figure 2.1: Three-Tier Architecture.

2.2 Component view

Taking into account what has been established in the previous diagrams, we have hereunder identified the high level components present in our system.

- ***Client:***

  All the clients listed down below communicate with the Server by making HTTP requests to the RESTful API.

  - *CitizenMobileApp*

    A native mobile application developed for the main mobile platforms, i.e. iOS, Android, meant to be used by Citizens to perform submissions and consult map information on the go.

  - *CitizenWebApp*

    A web application developed for browsers through which Citizens can visualize information more easily, though reports cannot be submitted through here.

  - *AuthorityMobileApp*

    A native mobile application developed for the main mobile platforms, i.e. iOS, Android, meant to be used by Authorities.

  - *AuthorityWebApp*

    A web application developed for browsers through which Authorities can visualize information more easily and provide their own data, i.e. reports about accidents, through a designated user interface.

  - *AdministratorApp*

    An application meant exclusively for users in charge of managing the system.

  - *PictureAnalyzer*

    It is the component responsible for analysing pictures and determining whether or not they contain readable license plate numbers. As such, it is only present in the *CitizenMobileApp*.

- ***Server:***

  It receives and handles requests from the Client through the RESTful API.

  - *Router*

    It is responsible for redirecting all incoming requests received from the Client to the right component server-side.

  - *AuthenticationManager*

    It manages user data, and the registration and authentication processes. It has direct access to the DBMS.

  - *ReportManager*

    It is responsible for processing all data regarding reports submitted by users. It has access to the DBMS for both storing and querying purposes, as data about reports needs to be updated (in case of status changes) and analysed for different functionalities.

  - *AreaManager*

    It handles information about locations, by labelling them as *unsafe* or with *high frequency of violations*, when the right conditions are met. It communicates with the Google Maps API to retrieve map data, and with the DBMS to store and retrieved the processed data about relevant locations.

  - *StatisticsProvider*

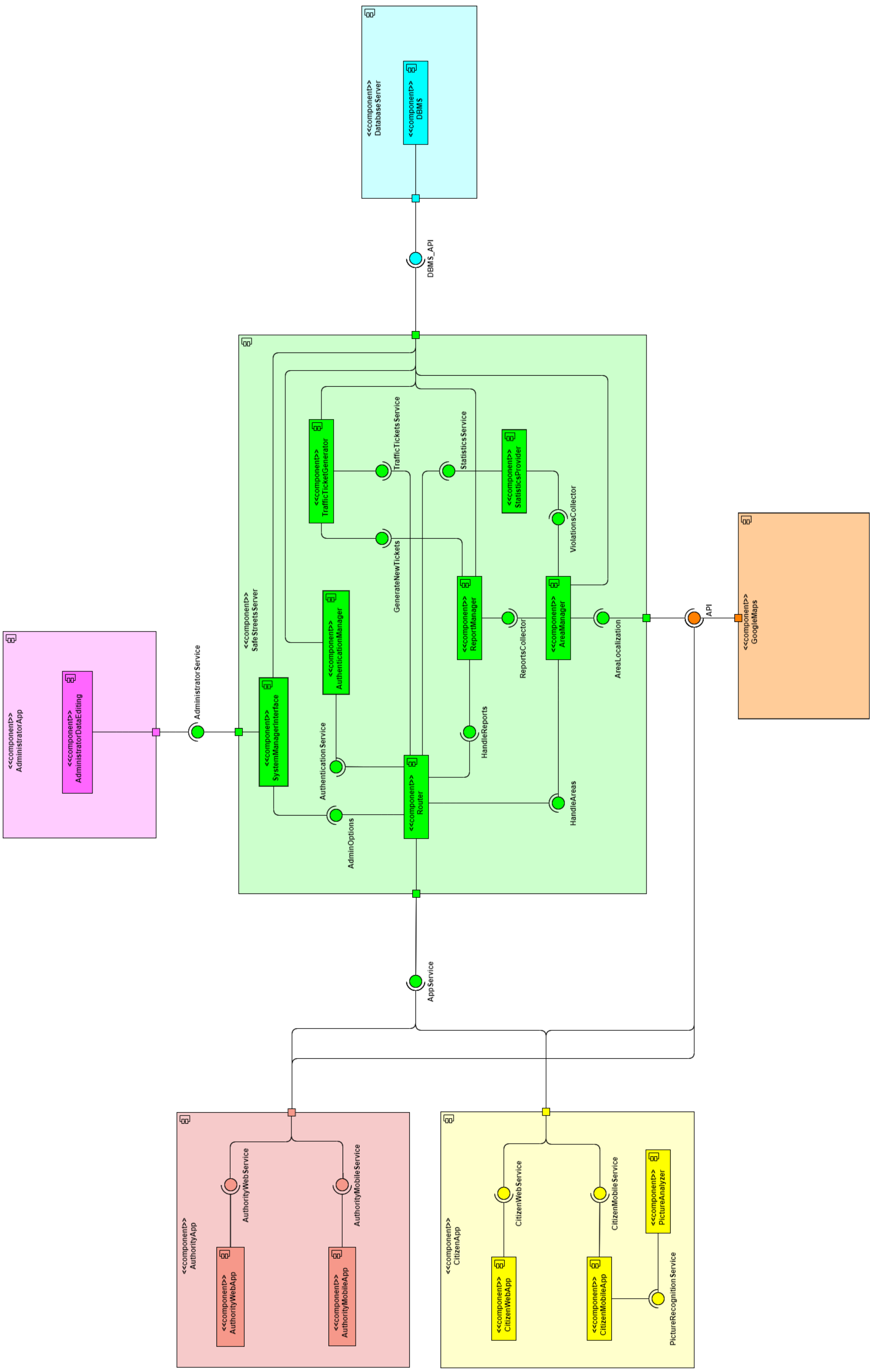
    This component is given the task of calculating statistics by processing data stored by the system. By consequence, it has direct access to the DBMS.

  - *TrafficTicketGenerator*

    It interfaces with the Municipality's external system by providing information about tickets that could possibly be issued, based on reports data.

  - *SystemManagerInterface*

    It manages data input from the *AdministratorApp*.



<<component>> Database Server
<<component>> DBMS
DBMS_API
<<component>> SafeStreetsServer
<<component>> TrafficTicketGenerator
TrafficTickets Service
Statistics Service
<<component>> StatisticsProvider
ViolationsCollector
GenerateNewTickets
<<component>> AuthenticationManager
<<component>> ReportManager
ReportsCollector
<<component>> AreaManager
AreaLocalization
API
<<component>> GoogleMaps
<<component>> AdministratorApp
<<component>> AdministratorDataEditing
Administrator Service
<<component>> SystemManagerInterface
Authentication Service
HandleReports
<<component>> Router
HandleAreas
AdminOptions
AppService
<<component>> AuthorityApp
AuthorityWeb Service
AuthorityMobile Service
<<component>> AuthorityWebApp
<<component>> AuthorityMobileApp
<<component>> CitizenApp
CitizenWeb Service
CitizenMobile Service
<<component>> PictureAnalyzer
<<component>> CitizenWebApp
<<component>> CitizenMobileApp
PictureRecognition Service

Figure 2.2: Component diagram.

2.3 Deployment view

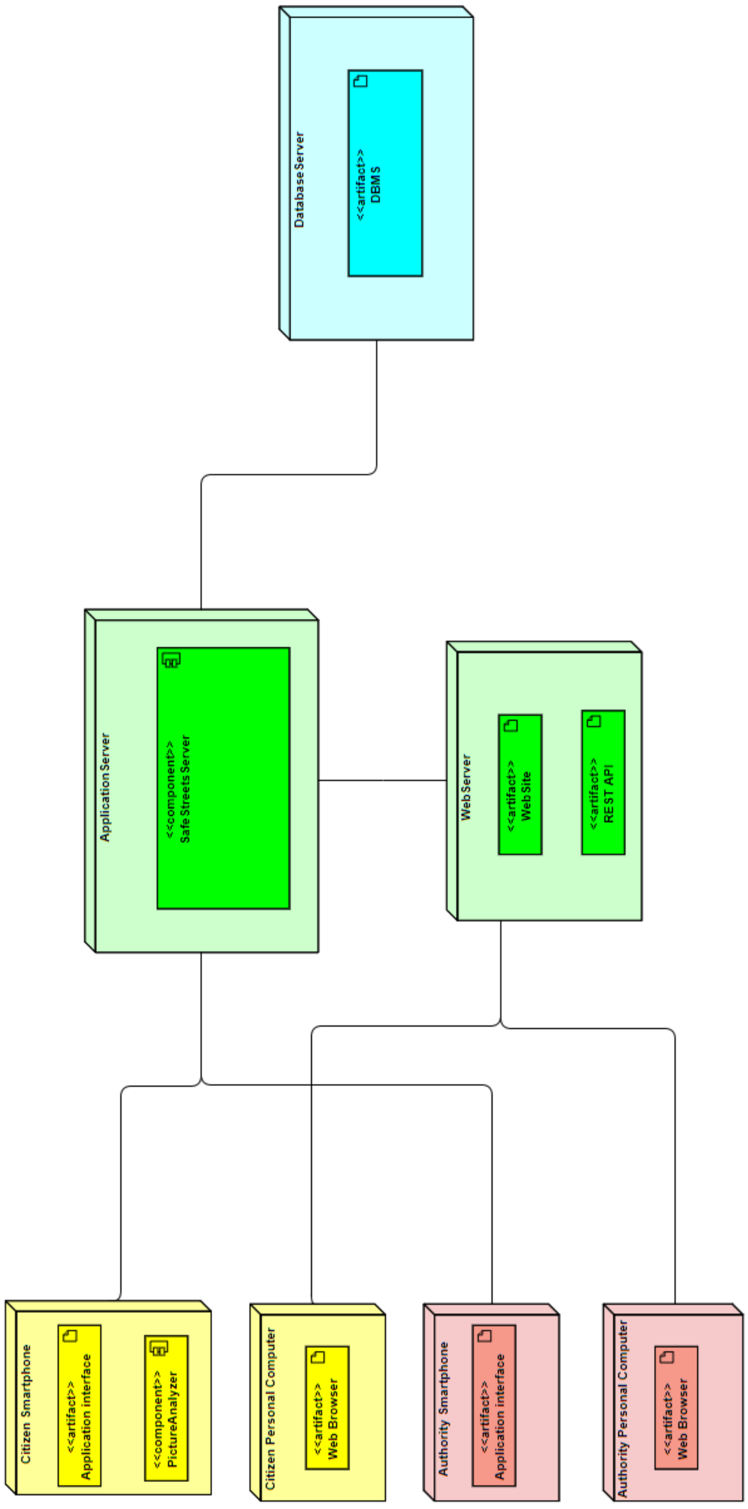


Figure 2.3: Deployment diagram.

2.4 Runtime view

In this section we resume the sequence diagrams already presented in the Requirements document and we expand the macro component of SafeStreets with a more detailed description using the subcomponents of the system. The diagrams are self-explanatory, eventual additional information that may be useful for a better understanding of the behaviour are inserted as notes inside the diagrams near the concerned operations.

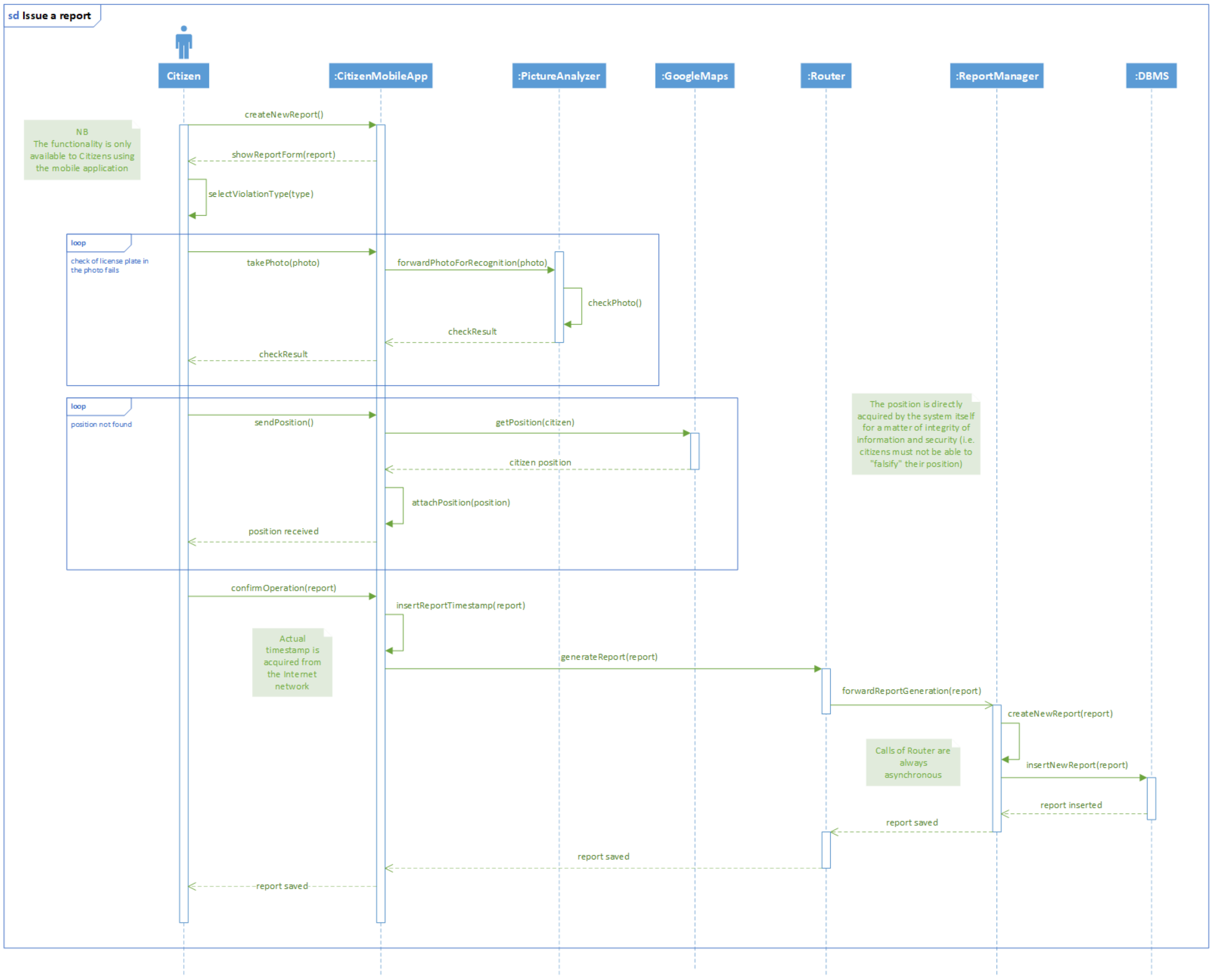

sd Issue a report
Citizen
:CitizenMobileApp
:PictureAnalyzer
:GoogleMaps
:Router
:ReportManager
:DBMS
createNewReport()
NB The functionality is only available to Citizens using the mobile application
showReportForm(report)
selectViolationType(type)
loop
check of license plate in the photo fails
takePhoto(photo)
forwardPhotoForRecognition(photo)
checkPhoto()
checkResult
checkResult
loop
position not found
sendPosition()
getPosition(citizen)
The position is directly acquired by the system itself for a matter of integrity of information and security (i.e. citizens must not be able to "falsify" their position)
citizen position
attachPosition(position)
position received
confirmOperation(report)
insertReportTimestamp(report)
Actual timestamp is acquired from the Internet network
generateReport(report)
forwardReportGeneration(report)
createNewReport(report)
Calls of Router are always asynchronous
insertNewReport(report)
report inserted
report saved
report saved
report saved

Figure 2.4: Issue a report.

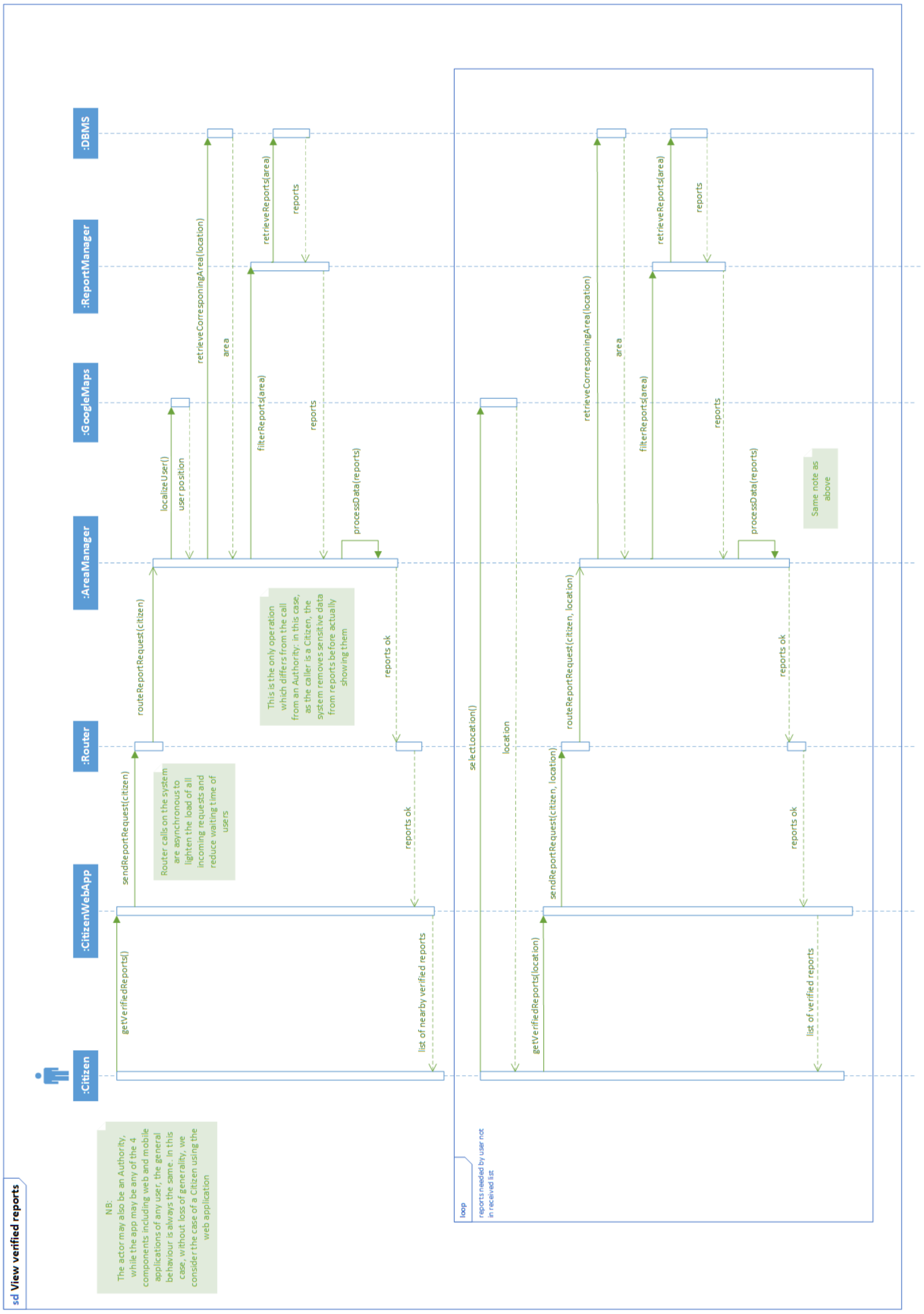

sd View verified reports
:Citizen
:CitizenWebApp
:Router
:AreaManager
:GoogleMaps
:ReportManager
:DBMS
getVerifiedReports()
sendReportRequest(citizen)
routeReportRequest(citizen)
localizeUser()
user position
retrieveCorresponingArea(location)
area
filterReports(area)
retrieveReports(area)
reports
reports
processData(reports)
reports ok
reports ok
list of nearby verified reports
Router calls on the system are asynchronous to lighten the load of all incoming requests and reduce waiting time of users
This is the only operation which differs from the call from an Authority: in this case, as the caller is a Citizen, the system removes sensitive data from reports before actually showing them
NB: The actor may also be an Authority, while the app may be any of the 4 components including web and mobile applications of any user, the general behaviour is always the same. In this case, without loss of generality, we consider the case of a Citizen using the web application
loop
reports needed by user not in received list
selectLocation()
location
getVerifiedReports(location)
sendReportRequest(citizen, location)
routeReportRequest(citizen, location)
retrieveCorresponingArea(location)
area
filterReports(area)
retrieveReports(area)
reports
reports
processData(reports)
reports ok
reports ok
list of verified reports
Same note as above

Figure 2.5: View verified reports.

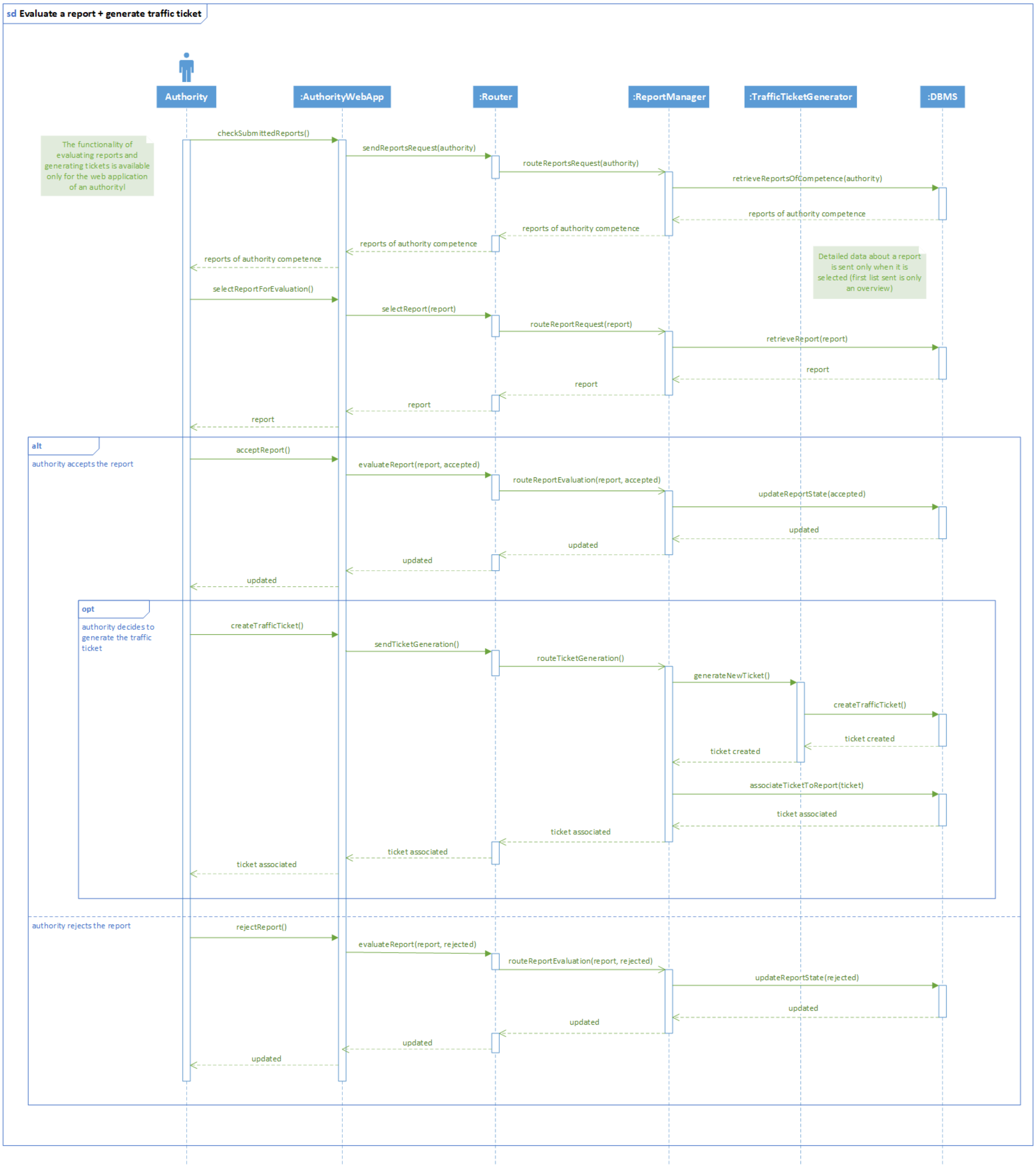

sd Evaluate a report + generate traffic ticket
Authority
:AuthorityWebApp
:Router
:ReportManager
:TrafficTicketGenerator
:DBMS
The functionality of evaluating reports and generating tickets is available only for the web application of an authorityl
checkSubmittedReports()
sendReportsRequest(authority)
routeReportsRequest(authority)
retrieveReportsOfCompetence(authority)
reports of authority competence
reports of authority competence
reports of authority competence
reports of authority competence
Detailed data about a report is sent only when it is selected (first list sent is only an overview)
selectReportForEvaluation()
selectReport(report)
routeReportRequest(report)
retrieveReport(report)
report
report
report
report
alt
authority accepts the report
acceptReport()
evaluateReport(report, accepted)
routeReportEvaluation(report, accepted)
updateReportState(accepted)
updated
updated
updated
updated
opt
authority decides to generate the traffic ticket
createTrafficTicket()
sendTicketGeneration()
routeTicketGeneration()
generateNewTicket()
createTrafficTicket()
ticket created
ticket created
associateTicketToReport(ticket)
ticket associated
ticket associated
ticket associated
ticket associated
authority rejects the report
rejectReport()
evaluateReport(report, rejected)
routeReportEvaluation(report, rejected)
updateReportState(rejected)
updated
updated
updated
updated

Figure 2.6: Evaluate report and generate traffic tickets.

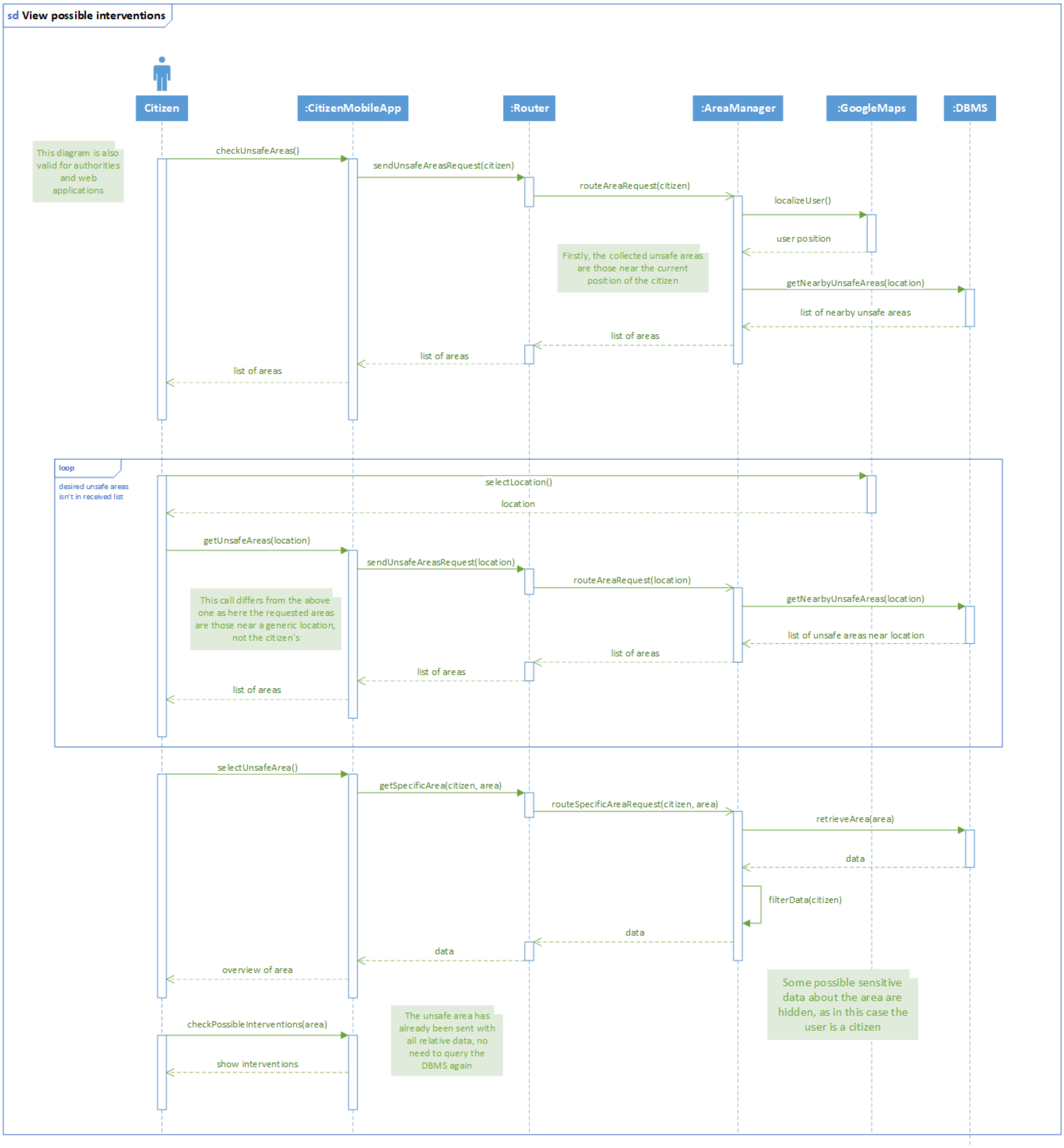

sd View possible interventions
Citizen
:CitizenMobileApp
:Router
:AreaManager
:GoogleMaps
:DBMS
This diagram is also valid for authorities and web applications
checkUnsafeAreas()
sendUnsafeAreasRequest(citizen)
routeAreaRequest(citizen)
localizeUser()
user position
Firstly, the collected unsafe areas are those near the current position of the citizen
getNearbyUnsafeAreas(location)
list of nearby unsafe areas
list of areas
list of areas
list of areas
loop
desired unsafe areas isn't in received list
selectLocation()
location
getUnsafeAreas(location)
sendUnsafeAreasRequest(location)
routeAreaRequest(location)
getNearbyUnsafeAreas(location)
This call differs from the above one as here the requested areas are those near a generic location, not the citizen's
list of unsafe areas near location
list of areas
list of areas
list of areas
selectUnsafeArea()
getSpecificArea(citizen, area)
routeSpecificAreaRequest(citizen, area)
retrieveArea(area)
data
filterData(citizen)
data
data
overview of area
Some possible sensitive data about the area are hidden, as in this case the user is a citizen
checkPossibleInterventions(area)
The unsafe area has already been sent with all relative data, no need to query the DBMS again
show interventions

Figure 2.7: View possible interventions for unsafe areas.

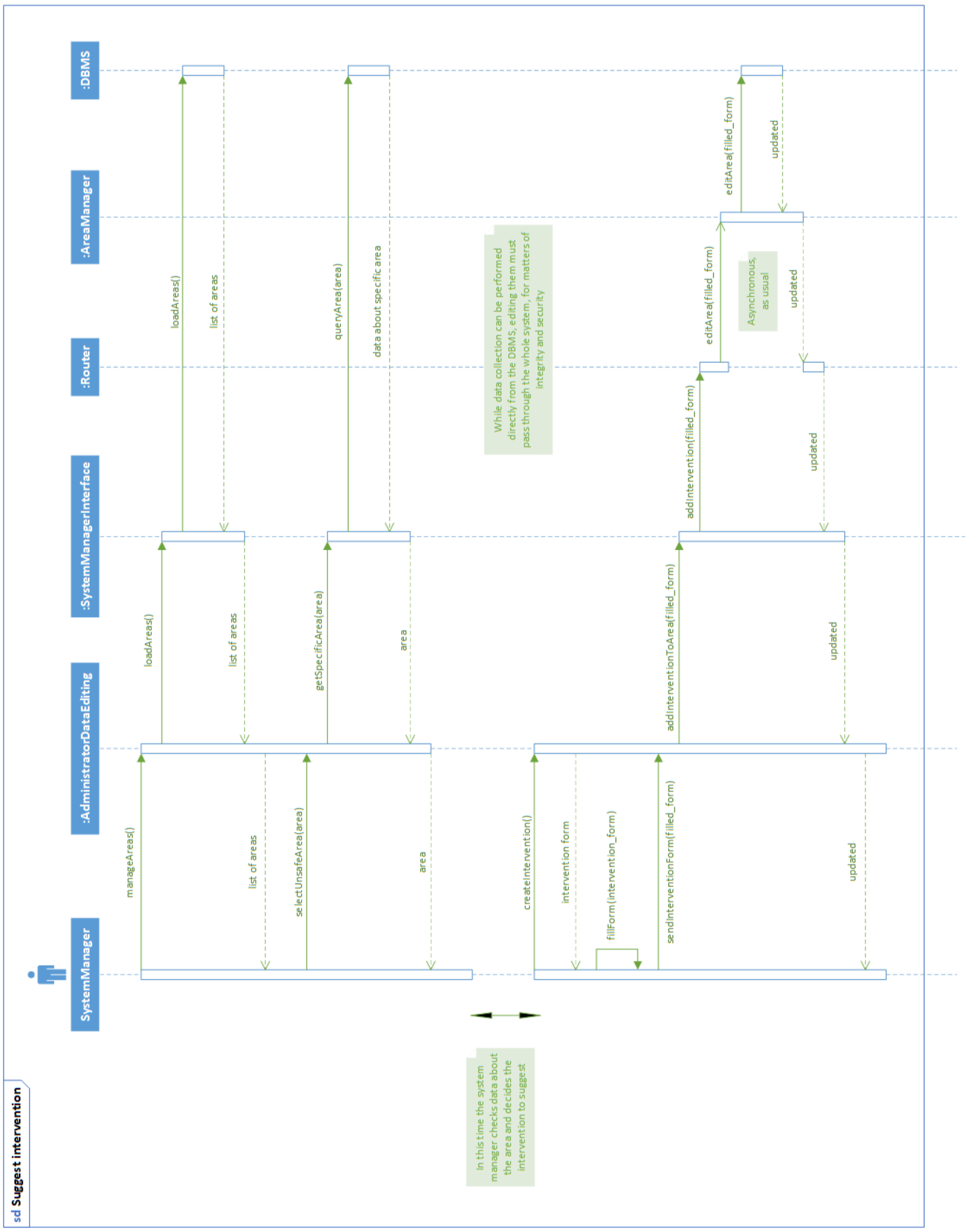
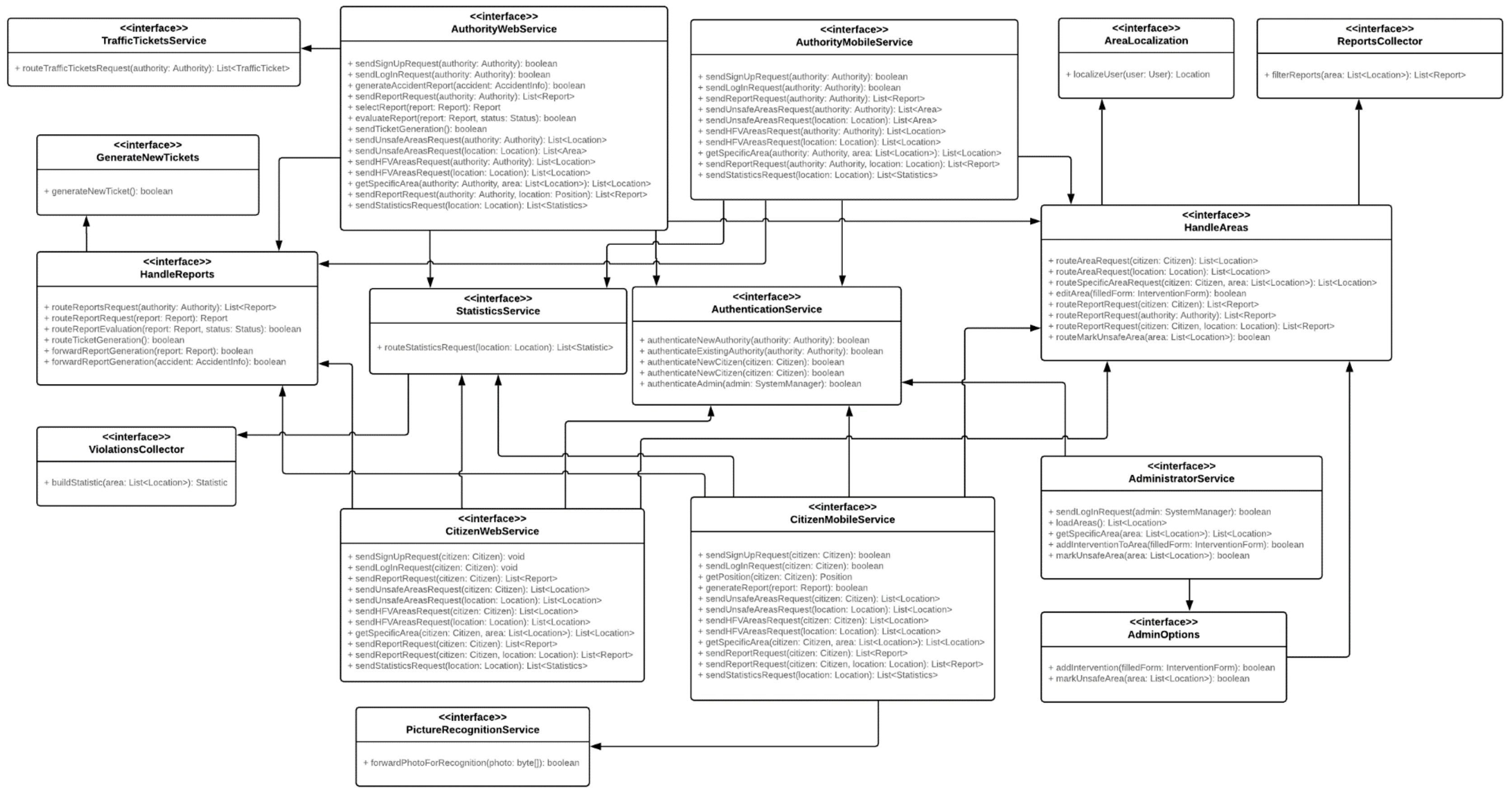

sd Suggest intervention
SystemManager
:AdministratorDataEditing
:SystemManagerInterface
:Router
:AreaManager
:DBMS
manageAreas()
loadAreas()
loadAreas()
list of areas
list of areas
list of areas
selectUnsafeArea(area)
getSpecificArea(area)
queryArea(area)
data about specific area
area
area
While data collection can be performed directly from the DBMS, editing them must pass through the whole system, for matters of integrity and security
In this time the system manager checks data about the area and decides the intervention to suggest
createIntervention()
intervention form
fillForm(intervention_form)
sendInterventionForm(filled_form)
addInterventionToArea(filled_form)
addIntervention(filled_form)
editArea(filled_form)
editArea(filled_form)
updated
Asynchronous, as usual
updated
updated
updated
updated

Figure 2.8: Suggest interventions for unsafe areas.

2.5 Component interfaces



<<interface>> TrafficTicketsService
+ routeTrafficTicketsRequest(authority: Authority): List<TrafficTicket>
<<interface>> AuthorityWebService
+ sendSignUpRequest(authority: Authority): boolean
+ sendLogInRequest(authority: Authority): boolean
+ generateAccidentReport(accident: AccidentInfo): boolean
+ sendReportRequest(authority: Authority): List<Report>
+ selectReport(report: Report): Report
+ evaluateReport(report: Report, status: Status): boolean
+ sendTicketGeneration(): boolean
+ sendUnsafeAreasRequest(authority: Authority): List<Location>
+ sendUnsafeAreasRequest(location: Location): List<Area>
+ sendHFVAreasRequest(authority: Authority): List<Location>
+ sendHFVAreasRequest(location: Location): List<Location>
+ getSpecificArea(authority: Authority, area: List<Location>): List<Location>
+ sendReportRequest(authority: Authority, location: Position): List<Report>
+ sendStatisticsRequest(location: Location): List<Statistics>
<<interface>> AuthorityMobileService
+ sendSignUpRequest(authority: Authority): boolean
+ sendLogInRequest(authority: Authority): boolean
+ sendReportRequest(authority: Authority): List<Report>
+ sendUnsafeAreasRequest(authority: Authority): List<Area>
+ sendUnsafeAreasRequest(location: Location): List<Area>
+ sendHFVAreasRequest(authority: Authority): List<Location>
+ sendHFVAreasRequest(location: Location): List<Location>
+ getSpecificArea(authority: Authority, area: List<Location>): List<Location>
+ sendReportRequest(authority: Authority, location: Location): List<Report>
+ sendStatisticsRequest(location: Location): List<Statistics>
<<interface>> AreaLocalization
+ localizeUser(user: User): Location
<<interface>> ReportsCollector
+ filterReports(area: List<Location>): List<Report>
<<interface>> GenerateNewTickets
+ generateNewTicket(): boolean
<<interface>> HandleAreas
+ routeAreaRequest(citizen: Citizen): List<Location>
+ routeAreaRequest(location: Location): List<Location>
+ routeSpecificAreaRequest(citizen: Citizen, area: List<Location>): List<Location>
+ editArea(filledForm: InterventionForm): boolean
+ routeReportRequest(citizen: Citizen): List<Report>
+ routeReportRequest(authority: Authority): List<Report>
+ routeReportRequest(citizen: Citizen, location: Location): List<Report>
+ routeMarkUnsafeArea(area: List<Location>): boolean
<<interface>> HandleReports
+ routeReportsRequest(authority: Authority): List<Report>
+ routeReportRequest(report: Report): Report
+ routeReportEvaluation(report: Report, status: Status): boolean
+ routeTicketGeneration(): boolean
+ forwardReportGeneration(report: Report): boolean
+ forwardReportGeneration(accident: AccidentInfo): boolean
<<interface>> StatisticsService
+ routeStatisticsRequest(location: Location): List<Statistic>
<<interface>> AuthenticationService
+ authenticateNewAuthority(authority: Authority): boolean
+ authenticateExistingAuthority(authority: Authority): boolean
+ authenticateNewCitizen(citizen: Citizen): boolean
+ authenticateNewCitizen(citizen: Citizen): boolean
+ authenticateAdmin(admin: SystemManager): boolean
<<interface>> ViolationsCollector
+ buildStatistic(area: List<Location>): Statistic
<<interface>> AdministratorService
+ sendLogInRequest(admin: SystemManager): boolean
+ loadAreas(): List<Location>
+ getSpecificArea(area: List<Location>): List<Location>
+ addInterventionToArea(filledForm: InterventionForm): boolean
+ markUnsafeArea(area: List<Location>): boolean
<<interface>> CitizenWebService
+ sendSignUpRequest(citizen: Citizen): void
+ sendLogInRequest(citizen: Citizen): void
+ sendReportRequest(citizen: Citizen): List<Report>
+ sendUnsafeAreasRequest(citizen: Citizen): List<Location>
+ sendUnsafeAreasRequest(location: Location): List<Location>
+ sendHFVAreasRequest(citizen: Citizen): List<Location>
+ sendHFVAreasRequest(location: Location): List<Location>
+ getSpecificArea(citizen: Citizen, area: List<Location>): List<Location>
+ sendReportRequest(citizen: Citizen): List<Report>
+ sendReportRequest(citizen: Citizen, location: Location): List<Report>
+ sendStatisticsRequest(location: Location): List<Statistics>
<<interface>> CitizenMobileService
+ sendSignUpRequest(citizen: Citizen): boolean
+ sendLogInRequest(citizen: Citizen): boolean
+ getPosition(citizen: Citizen): Position
+ generateReport(report: Report): boolean
+ sendUnsafeAreasRequest(citizen: Citizen): List<Location>
+ sendUnsafeAreasRequest(location: Location): List<Location>
+ sendHFVAreasRequest(citizen: Citizen): List<Location>
+ sendHFVAreasRequest(location: Location): List<Location>
+ getSpecificArea(citizen: Citizen, area: List<Location>): List<Location>
+ sendReportRequest(citizen: Citizen): List<Report>
+ sendReportRequest(citizen: Citizen, location: Location): List<Report>
+ sendStatisticsRequest(location: Location): List<Statistics>
<<interface>> AdminOptions
+ addIntervention(filledForm: InterventionForm): boolean
+ markUnsafeArea(area: List<Location>): boolean
<<interface>> PictureRecognitionService
+ forwardPhotoForRecognition(photo: byte[]): boolean

Figure 2.9: Component interfaces diagram.

2.6 Selected architectural styles and patterns

In order to make sure every functional and non-functional requirement, as well as the imposed availability and flexibility constraints, could be fulfilled, we opted to utilize the following architectural styles and patterns for developing the system.

- **Object-oriented Design**

Developers should be faithful to the five basic concepts of Object Oriented Design, which are listed hereunder:

- *Object/Class*: the correspondence between data structures and the functions/methods that interact with the data;
- *Information hiding*: the ability to protect some components of the object from external entities;
- *Inheritance*: the ability of a class to extend or override functionalities of another class;
- *Interface*: the ability to defer the implementation of a method;
- *Polymorphism*: the ability to replace an object with its sub-objects;

- **Client/Server Architecture**

This architectural style was chosen to ensure that the logic layer of the system, which requires a considerably large amount of computational power, could be deployed on a rack of servers, leaving the client applications thinner and limited to a small number of functionalities.

- **Three-Tier Physical Architecture**

In Three-Tier Architectures, the *presentation*, *application* and *data* layers are split into just as many dedicated machines, meaning: end-user devices, a rack of intermediate servers (also called *middle tier*) and the DBMS. This kind of architecture guarantees a greater opportunity at reaching the desired levels of flexibility and scalability, as well as security, since the data access is separated from the clients by the middle tier.

- **RESTful Architecture**

As far as client and server communication goes, we chose to employ a RESTful API, as its principles apply well to the characteristics and needs of the system to be developed. As a matter of fact, REST:

- is a *client-server* architecture;
- is *stateless*: all the information needed to perform a request is always contained in each communication between server and client, and session state information is only needed on the client-side. Hence, if authentication is required in order to access a resource, the client needs to authenticate itself with every request;
- is *cacheable*: performance can be improved by caching some information on intermediary devices;
- provides a *uniform interface*: each device communicating with the API follows the same rules;
- is *layered*: each device can only see the components of the architecture which it's communicating with and not what lies beyond, meaning each component is independent from the others and can generally be easily replaced or extended.

2.7 Other design decisions

- **Picture-analysing process localized on clients**

In general, the configuration of the system is a thin client one, but we decided to localize the component for analysing pictures (and only this one) about incoming reports from citizens on the application itself, and not into the server, thus resulting in an "impure" thin client configuration.
This choice is due to multiple reasons:

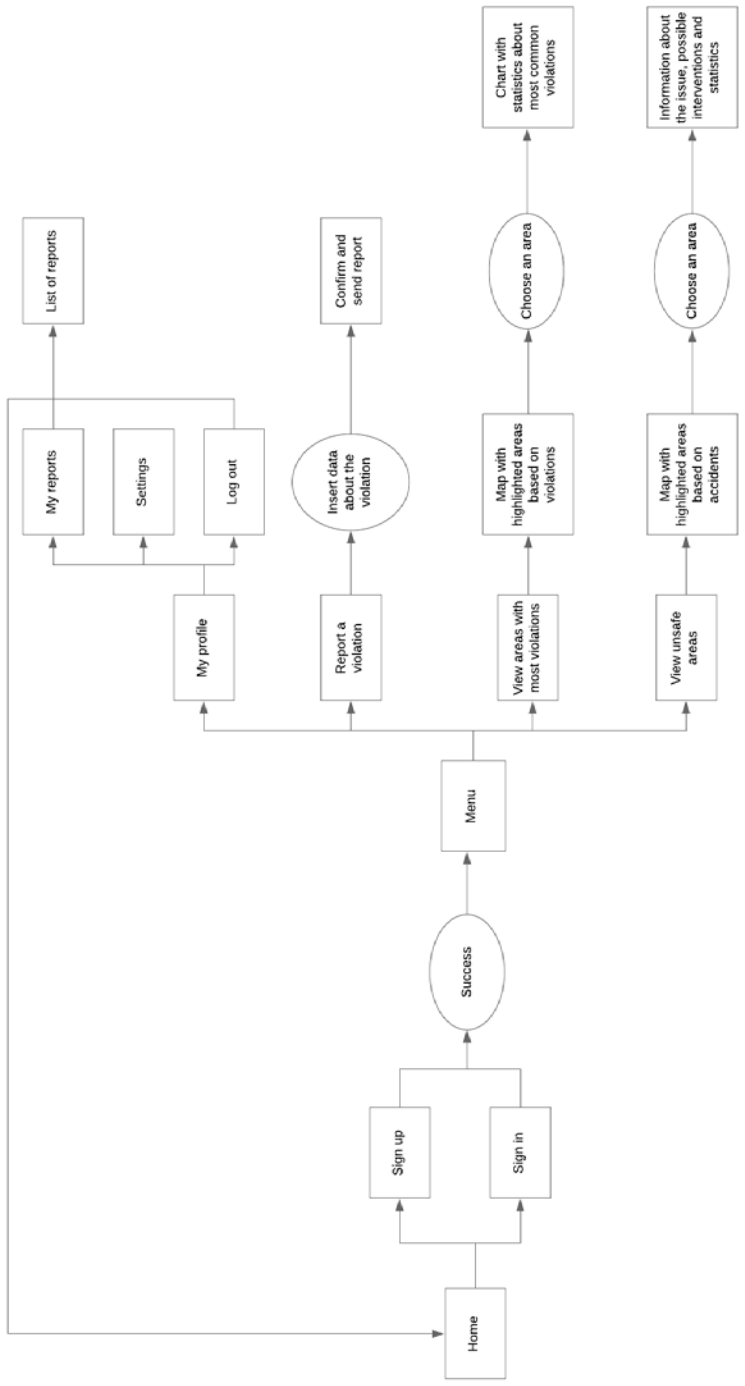
1. If the component would be localized on the server, each time the citizen who is trying to take a photo of a vehicle to send a report the application should, in order to validate it, send a request to the server every time a photo is taken, thus resulting in a sequence of invocations that, in a bad case, would return a fail check and ask the citizen to retry the operation. So, both for avoiding too many calls of this kind and to not rely too much on the Internet network for sending them to the server, we prefer to move the component directly in the client application so that, when a report is received by the server, it is surely ready to be stored without the need of any integrity check.
2. The component is devoted to satisfy only this operation, so localizing it on the server would result in a waste of computational resources while the task can be efficiently performed by a common smartphone. Moreover, the localization on the server would require it to possibly satisfy many requests from different users in parallel so, for the above reason and as the component doesn't require any interaction with other components or the DBMS, we prefer to just localize it on the client application.

- **Machine learning for analysing pictures**

The component devoted to analyse pictures will be implements with a machine learning algorithm. The algorithm takes inspiration from a face recognition algorithm, with license plates as the subject of recognition and not faces. The algorithm will be trained by providing it a series of various pictures representing valid (vehicles with readable license plate) and invalid (the license plate is not readable for any reason) ones. The algorithm will have to learn the shape of a license plate (which is rectangular) and to read it by comparing the symbols with Latin alphabet and Arabic numbers; for the test pictures which are valid the algorithm will compare the acquired result with the actual solution, which is given, while for invalid ones will have to return a fail.

3. User interface design

This section provides UX diagrams to represent the possible interactions and flow of events in the user interfaces of the various applications. The layouts for these can be found in section 3.1 of the RASD.



Home
Sign up
Sign in
Success
Menu
My profile
Report a violation
View areas with most violations
View unsafe areas
My reports
Settings
Log out
Insert data about the violation
Map with highlighted areas based on violations
Map with highlighted areas based on accidents
List of reports
Confirm and send report
Choose an area
Choose an area
Chart with statistics about most common violations
Information about the issue, possible interventions and statistics

Figure 3.1: CitizenMobileApp UX diagram.

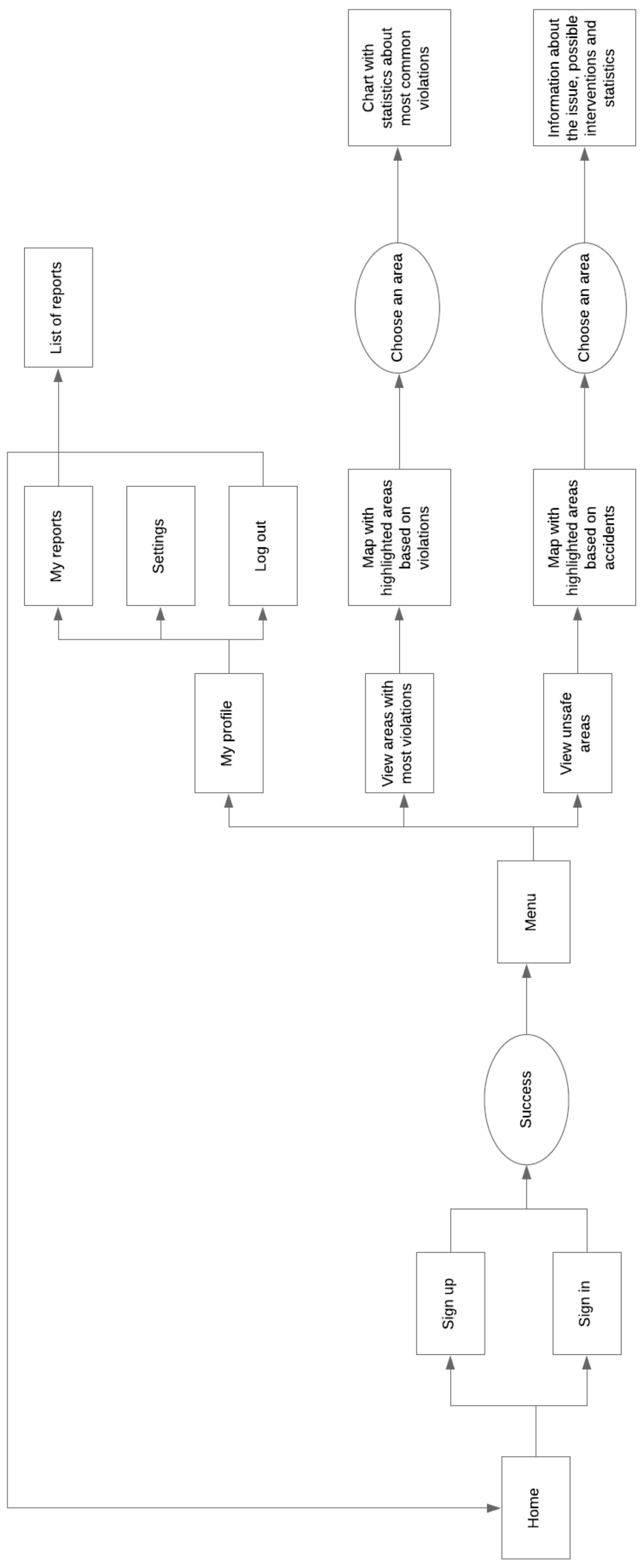

Home
Sign up
Sign in
Success
Menu
My profile
View areas with most violations
View unsafe areas
My reports
Settings
Log out
List of reports
Map with highlighted areas based on violations
Choose an area
Chart with statistics about most common violations
Map with highlighted areas based on accidents
Choose an area
Information about the issue, possible interventions and statistics

Figure 3.2: CitizenWebApp UX diagram.

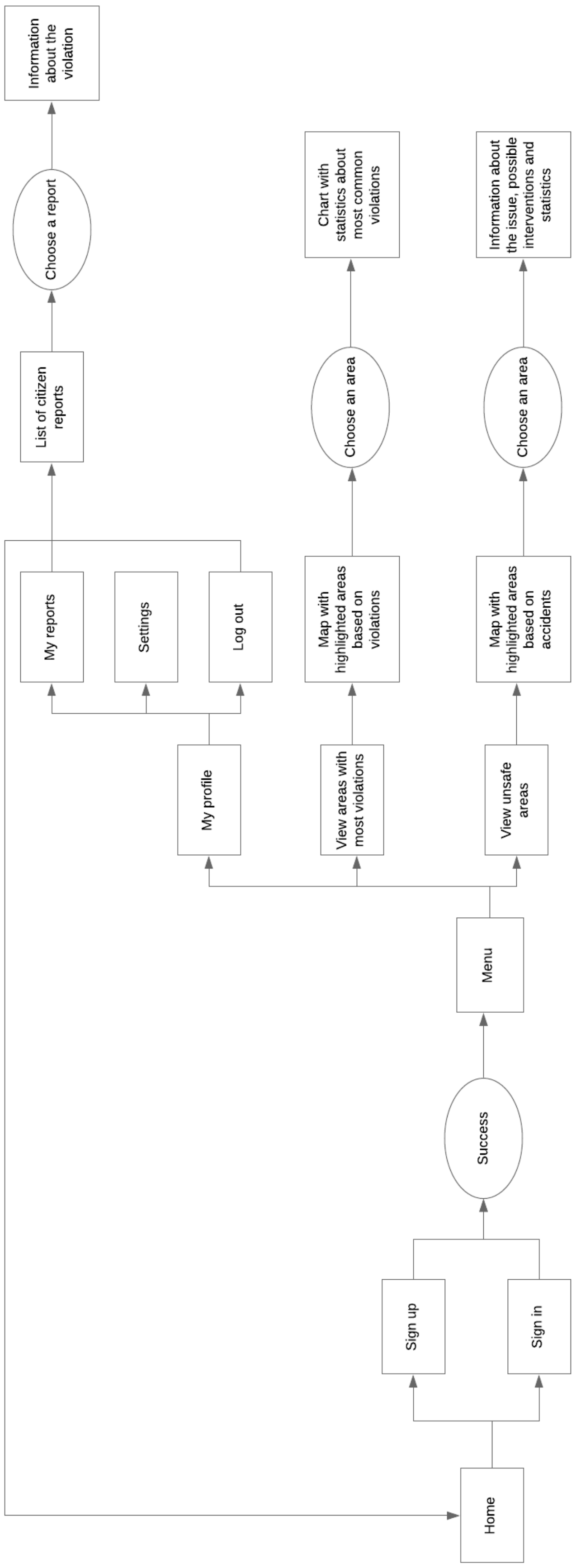
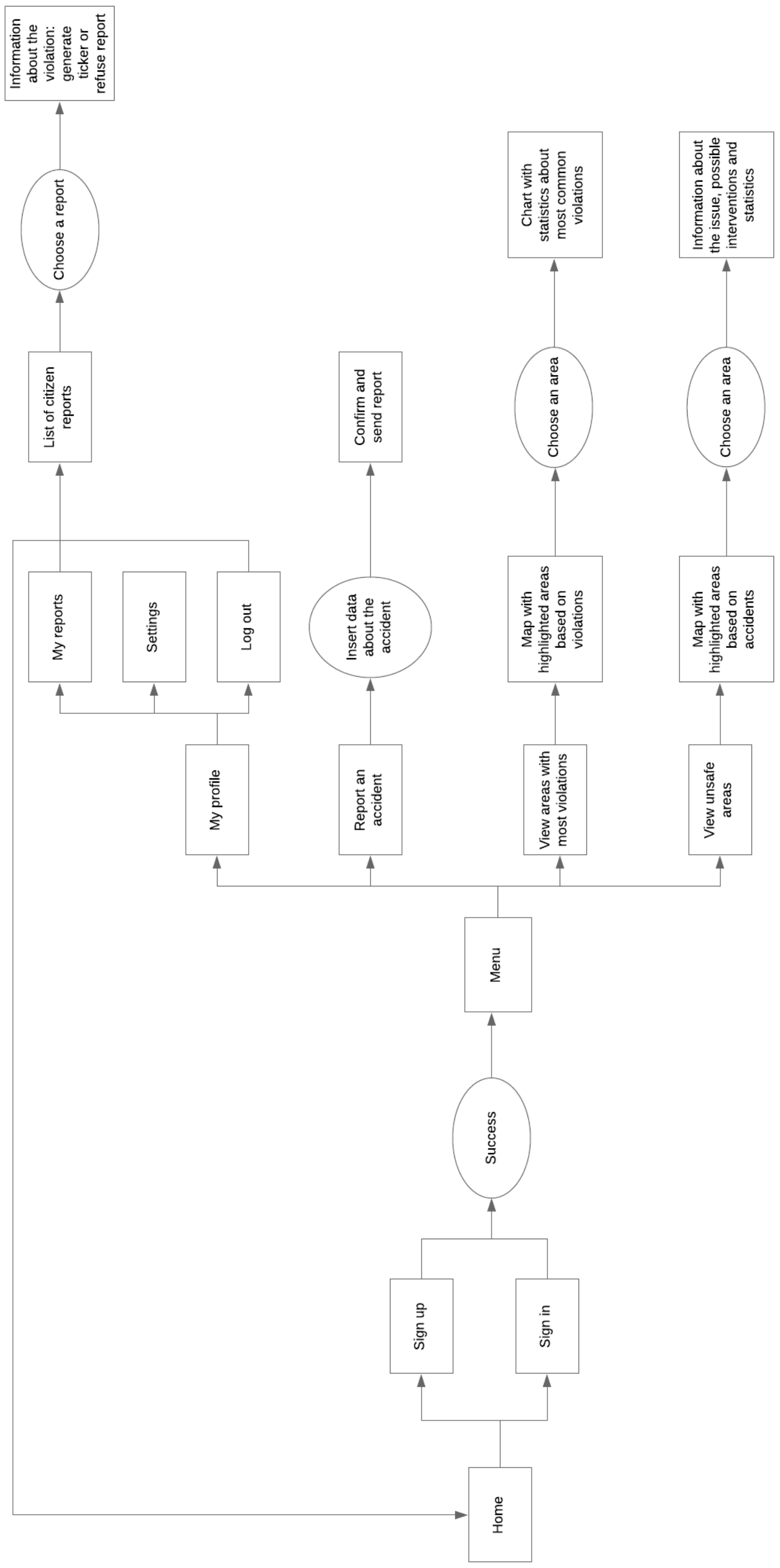

Home
Sign up
Sign in
Success
Menu
My profile
View areas with most violations
View unsafe areas
My reports
Settings
Log out
Map with highlighted areas based on violations
Map with highlighted areas based on accidents
List of citizen reports
Choose a report
Information about the violation
Choose an area
Chart with statistics about most common violations
Choose an area
Information about the issue, possible interventions and statistics

Figure 3.3: AuthorityMobileApp UX diagram.



Home
Sign up
Sign in
Success
Menu
My profile
Report an accident
View areas with most violations
View unsafe areas
My reports
Settings
Log out
Insert data about the accident
Map with highlighted areas based on violations
Map with highlighted areas based on accidents
List of citizen reports
Confirm and send report
Choose an area
Choose an area
Choose a report
Information about the violation: generate ticker or refuse report
Chart with statistics about most common violations
Information about the issue, possible interventions and statistics

Figure 3.4: AuthorityWebApp UX diagram.

4. Requirements traceability

In this section we map each one of the requirements identified in the RASD to one or more of the high level components described previously; in particular, each requirement is linked to the list of components which are bestowed with the task of fulfilling said requirement.

[R1] The system must allow new users to sign up by providing personal/mandatory information;

- AuthenticationManager

[R2] The system must be able to authenticate registered users;

- AuthenticationManager

[R3] The citizen must be able to insert relevant data about violations, such as type of violation, during the filling out process;

- ReportManager

[R4] The system must be able to analyze pictures that are being submitted in a report and recognize whether the license plate is readable/present or not;

- PictureAnalyzer

[R5] The system must inform the user whether their report has been stored successfully or not;

- ReportManager

[R6] The system must ask the user if they want to retry the submission process using the same data that failed being sent, or if they want to cancel it;

- ReportManager

[R7] The system must be able to distinguish every user unambiguously;

- AuthenticationManager

[R8] The system must store information with an association to the user who submitted it;

- ReportManager
- AuthenticationManager

[R9] The system must be able to retrieve stored information;

- ReportManager
- AuthenticationManager
- AreaManager
- SystemManagerInterface
- TrafficTicketGenerator

[R10] The system must allow reports to have only one status at a time (accepted, rejected, to be checked);

- ReportManager

[R11] The system must tell the user whether their report has been accepted, rejected or is still waiting to be checked;

- AuthenticationManager
- ReportManager

[R12] The system must be able to distinguish between authorities and citizens;

- AuthenticationManager

[R13] The system must be able to distinguish between submitted reports and reports that have been reviewed and accepted by the police;

- ReportManager

[R14] The system must anonymize data shown to regular users (citizens), that is hide information about the vehicles that were parked illegally and about who submitted a particular report; in other words, data about reports that is shown to users must only contain the type of violation, date, time and position;

- AuthenticationManager
- AreaManager
- StatisticsProvider

[R15] The system must show the full data about a report to authorities;

- AuthenticationManager
- ReportManager

[R16] An authority must be able to submit reports about accidents;

- AuthenticationManager
- ReportManager

[R17] An authority must be able to insert relevant information about the occurred accident, such as location and injured people;

- AuthenticationManager
- ReportManager

[R18] The system must be able to access map information;

- AreaManager

[R19] The system must show the user their local map information;

- AuthenticationManager
- AreaManager

[R20] The system must show the user possible solutions for unsafe areas, if there are any;

- AreaManager

[R21] The system must allow system managers to edit the status of an area as unsafe and the other way around;

- SystemManagerInterface

[R22] An authority must be able to generate a traffic ticket from a report;

- TrafficTicketGenerator
- ReportManager

[R23] The system must offer the possibility to generate a traffic ticket only to reports which has the status of accepted;

- ReportManager
- TrafficTicketGenerator

[R24] The system must allow Authorities to know which Citizen sent each report;

- ReportManager
- AuthenticationManager

[R25] The system must be able to compute meaningful statistics on reports about each kind of violation in which a traffic ticket has been generated;

- StatisticsProvider

[R26] The system must make data about statistics visible to all users;

- StatisticsProvider

[R27] The system must allow system managers to suggest interventions for unsafe areas;

- SystemManagerInterface

[R28] The system must make data about suggested interventions visible to all users;

- AreaManager

Traceability matrix

To provide a more compact view, for each defined component inside the application server (plus the PictureAnalyzer component which, even though it is located on the client tier, can be considered as part of the server) we highlight the requirements they are demanded to satisfy inside the following table.

Component	Involved Requirements
AuthenticationManager	[R1] [R2] [R7] [R8] [R9] [R11] [R12] [R14] [R15] [R16] [R17] [R19] [R24]
ReportManager	[R3] [R5] [R6] [R8] [R9] [R10] [R11] [R13] [R15] [R16] [R17] [R22] [R23] [R24]
AreaManager	[R9] [R14] [R18] [R19] [R20] [R28]
TrafficTicketGenerator	[R9] [R22] [R23]
StatisticsProvider	[R14] [R25] [R26]
PictureAnalyzer	[R4]
SystemManagerInterface	[R9] [R21] [R27]

5. Implementation, integration and test plan

The system is composed of different subsystems:

- CitizenApp
- AuthorityApp
- AdministratorApp
- SafeStreetsServer
- External systems: DBMS, GoogleMaps

Considering the architecture of the SafeStreets system, for the integration and testing we chose a bottom-up strategy. In the bottom-up approach, individual components are specified in detail and are then connected to each other, until the realization of the entire system is achieved. This strategy allows us to start the integration and its testing without having to wait for the complete implementation and the unit testing of each component in the system. As a matter of fact, bottom-up testing makes it so that components at a lower hierarchy are tested individually, and then the components that rely upon these components are also tested, despite being the most relevant. We chose this strategy as it is more convenient to introduce tested modules one by one, considering the frequent and short tests make localizing errors easier.

In this section we treat the SafeStreetsServer subsystem, and analyse the main features available for SafeStreets users, as well as their significance, explaining the reasons for our choices concerning the implementation, the testing and the integration. First of all, it's essential to underline that the external systems (DBMS, GoogleMaps) are commercial components that have already been developed and this is why they are available to be used. One other important consideration is that the development of the components and their functionalities proceed together with the unit testing on such components: in this way, when the components aren't completely developed, they have already been tested at unit level.

For what concerns the diagrams, the arrow starts from the component which is dependent on the one it's connected to.

The main features of the system are the following:

- **Report a violation**: this is a key feature of the application and, to realize it, the implementation of the ReportManager, one of the most significant and complex components, is necessary, and it also needs to be tested; as explained before, the ReportManager interacts with the database, in order to store and query data and it also processes submitted reports by users.

- **Visualize unsafe areas and areas with most violations**: this is another fundamental functionality guaranteed to users; AreaManager is responsible for the identification of critical areas and to provide this feature, it has to be implemented, unit tested and it also has to be integrated with the ReportManager.

- **Generate ticket**: the possibility of generating tickets is provided only for authorities; the component which deals with this core functionality is the TraffickTicketGenerator, that has to be fully implemented. It's important to integrate it with the ReportManager, because tickets are based on reports of violations.

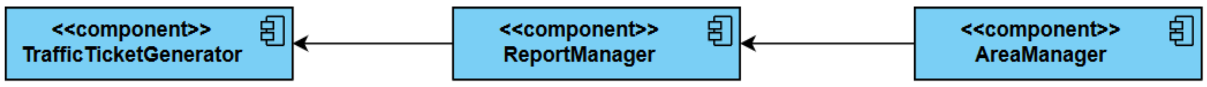


Figure 5.1.

Figure 5.1 shows that AreaManager and TrafficTicketGenerator, in order to provide their respective features, need the integration with ReportManager.

- **Consult statistics about most frequent violations or about accidents**: this feature allows users to view particular statistics concerning accidents and most common violations in a certain area; it's provided by the StatisticsProvider component, which has to be integrated with the AreaManager. By consequence, the StatisticsProvider, in order to guarantee this important functionality, needs the integration with the AreaManager, as shown in Figure 5.2, as it focuses on the most violated areas.

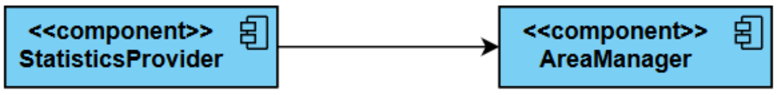


Figure 5.2.

- **Make a suggestion or mark an area as unsafe**: this possibility is reserved only for administrators from the AdministratorApp; it's provided by the SystemManagerInterface component, which allows administrators to make suggestions about a dangerous area and to mark areas as unsafe and, in the same way, unmark an area as unsafe once an intervention has been carried out. This component has to be implemented and unit tested at this point in time.

- **Sign up and login**: the sign up and login functionalities are managed by AuthenticationManager; the implementation of this component, although necessary for the proper functioning of the system, is not very difficult and it can be performed at this time.

The last component in SafeStreetsServer is the Router. It plays a key role because it has to route all users' requests to the right component of the server. It interacts with all server components mentioned above (apart from the SystemManagerInterface component), as shown in Figure 5.3; this is why its implementation and its testing are left for last.

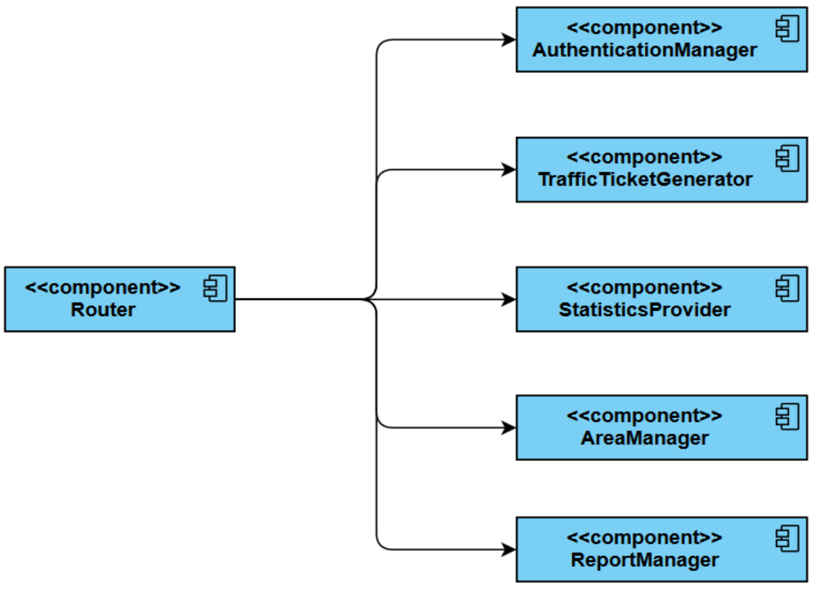


Figure 5.3.

Integration between the client and the application server:

At this point, it's necessary to integrate the client with the application server; it should be done once the components involved have been developed and tested.

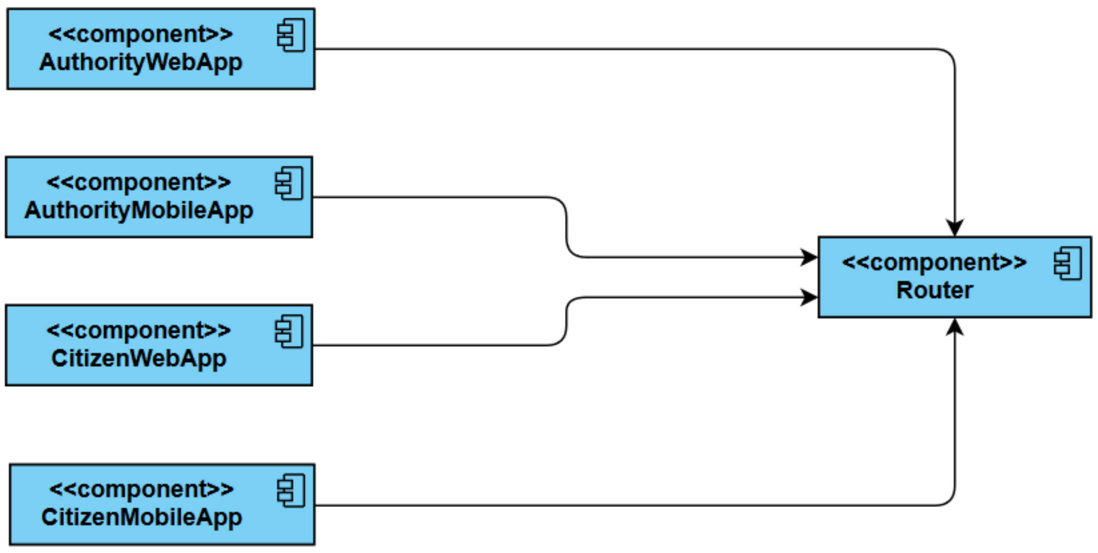


Figure 5.4.

It should be noted that the CitizenMobileApp needs to be integrated with the PictureAnalyzer, the component which deals with the reading of the license plates.

Integration between the administrator and the server:

AdministratorDataEditing and SystemManagerInterface have to be integrated (after their implementation) in order to ensure the proper functioning of the administrators' operations.

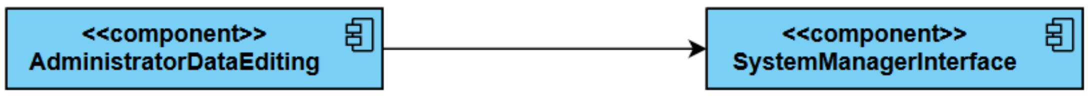
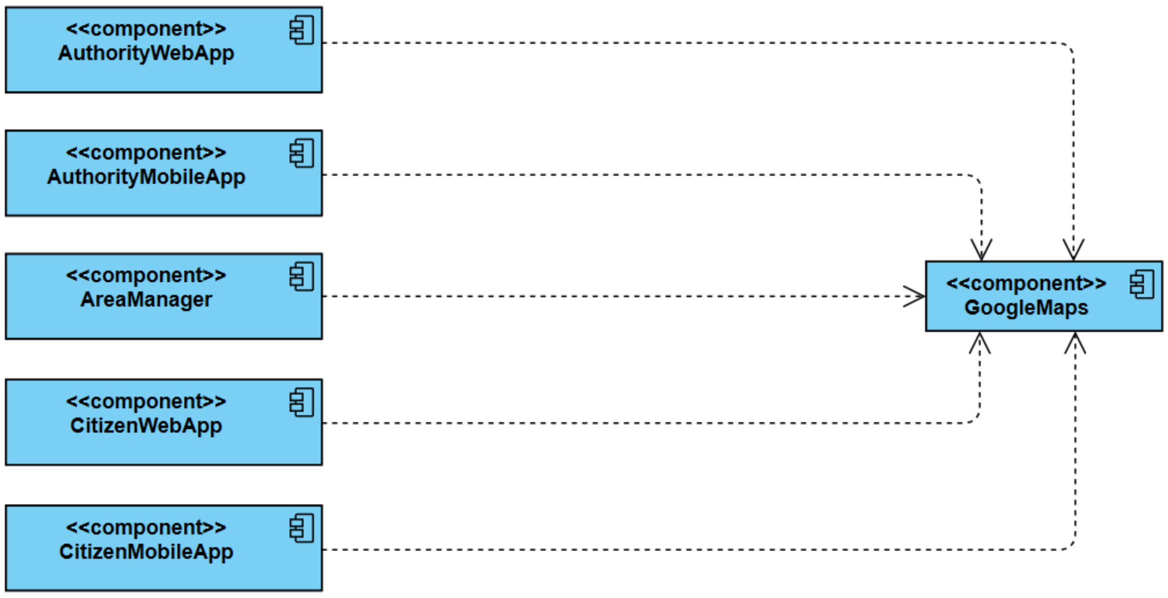


Figure 5.5.

Integration of components with GoogleMaps:

This is an important integration between the following components and the external system GoogleMaps, which is essential in order to provide some of the main functionalities of the application. Obviously, the components related to the integration have been implemented and tested.

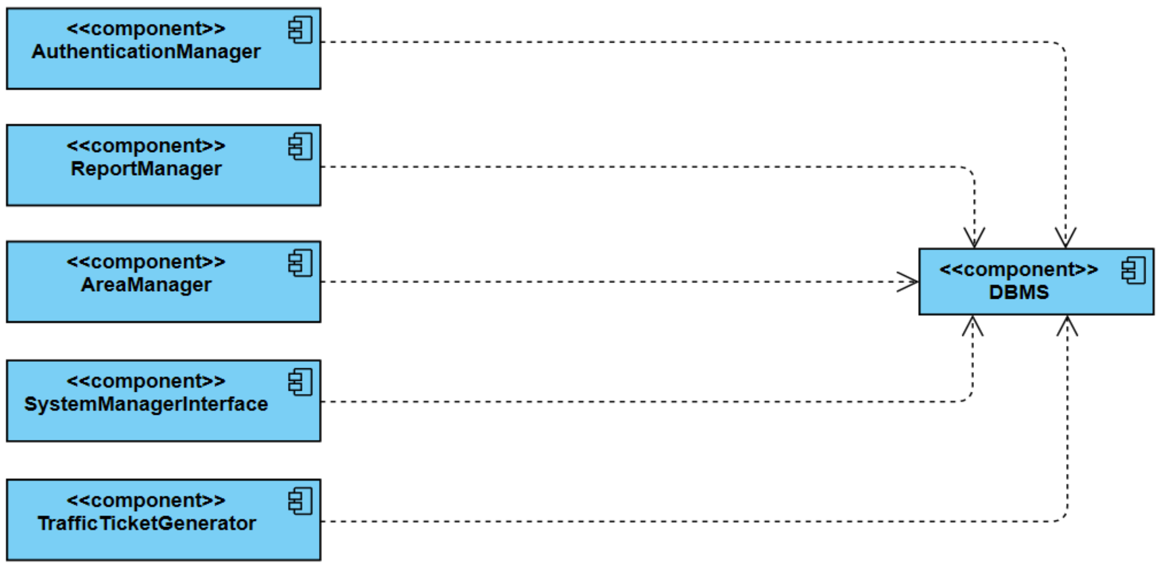


<<component>> AuthorityWebApp
<<component>> AuthorityMobileApp
<<component>> AreaManager
<<component>> CitizenWebApp
<<component>> CitizenMobileApp
<<component>> GoogleMaps

Figure 5.6.

Integration of components with the Database Management System:

The last integration is between the components shown here and the database; as a matter of fact, these components of the application server have to work with the DBMS in order to realize the main features of the system.



<<component>> AuthenticationManager
<<component>> ReportManager
<<component>> AreaManager
<<component>> SystemManagerInterface
<<component>> TrafficTicketGenerator
<<component>> DBMS

Figure 5.7.

6. Appendix

6.1 Effort spent

6.1.1 Pozzi Matteo

Date	***Topic***	***Effort [hrs]***
19/11/2019	Architecture overview + Component diagram	4
20/11/2019	Component diagram	2
24/11/2019	Component diagram + Runtime View	3
26/11/2019	Runtime View	3
27/11/2019	Runtime View	3
03/12/2019	Deployment	2
04/12/2019	Deployment + Runtime + Design Decisions	4
09/12/2019	Requirements traceability matrix + fixes	2
	Total effort	**23**

6.1.2 Ventura Andrea

Date	*Topic*	*Effort [hrs]*
14/11/2019	Ux diagrams	2
15/11/2019	Ux diagrams	2
17/11/2019	Ux diagrams	2
18/11/2019	Introduction + Overview	4
26/11/2019	Implementation and testing plan	3
27/11/2019	Implementation and testing plan	3
29/11/2019	Implementation and testing plan	4
03/12/2019	Various	5
07/12/2019	Implementation and testing plan	3
08/12/2019	Implementation and testing plan	4
	Total effort	**32**

6.1.3 Sacco Sara

Date	***Topic***	***Effort [hrs]***
17/11/2019	Introduction	1
19/11/2019	Overview + Components	4
20/11/2019	Overview + Requirements Traceability	1
22/11/2019	Architecture	1
24/11/2019	Architecture	1
25/11/2019	Fix requirements	2
28/11/2019	Components	1
30/11/2019	Component Interfaces	4
01/12/2019	Component Interfaces + Class diagram	2
03/12/2019	Various	5
04/12/2019	UX diagrams	1
08/12/2019	Architectural styles and patterns	5
09/12/2019	Various	2
	Total effort	**30**