POLITECNICO
MILANO 1863

M.Sc. Computer Science and Engineering
Software Engineering 2 Project

Design Document

SafeStreets

Ferrara Fabiana, Formicola Stefano, Guerra Leonardo

9 December 2019

GitHub Repository: https://github.com/ste7en/FerraraFormicolaGuerra

Version 1.0

Contents

Chapter 1

Introduction

1.1 Purpose

This document constitutes the Design Document (DD). It provides a more technical overview to the Requirement Analysis and Specification Document (RASD) of the system-to-be, describing the main architectural components, their communication interfaces and their interactions.
It will also present the implementation, integration and testing plan. This type of document is mainly addressed to developers, since that it provides a guide during the developement process through an accurate vision of all parts of the software-to-be.

1.2 Scope

Crowd-sourced applications have become more popular nowadays thanks to the massive diffusion of smart devices and the consequent interconnection between people so that they started feeling part of a community, where everyone can contribute concretely.

SafeStreets is a crowd-sourced application whose aim is, indeed, to provide a tool to allow registered citizens to help *Municipality*, reporting *Traffic violations* that occur in their cities. In particular, they can take pictures of parking violations, specifying their type. A *License plate recognition service* recognizes the license plate from the *User picture*. SafeStreets stores the generated *User reports* and send a *User report notification* to *Municipality*. Examples of parking which represent a violation are the ones on bike lanes, sidewalks, in front of vehicle entrances or on reserved parking lots.
Users and *Municipalities* can also retrieve analytics about data collected by the application in order to obtain information, for example, about violations in certain areas or to identify vehicles with the highest number of violations, respectively.

Moreover, given that *Municipality* provides access to its data about *Accidents*, the system-to-be can integrate those information with its own data and finally suggest *Municipality Possible interventions* to apply.
SafeStreets can also build statistics from data sent by *Municipality* which concern issued *Traffic tickets* generated by *User reports*, for example about the most egregious offenders, or to show the effectivness of the SafeStreets initiative.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

User: individual registered to SafeStreets who agreed to the acquisition and processing of the data he/she sends;

User identifier: email provided by the *User* during the registration phase; it represents a single *User*;

User report: report created by the *User* including the fields *User identifier*, *User picture*, *Timestamp*, *Recognized license plate* and type of *Traffic violation*;

Terms and conditions: a set of regulations which *Users* must agree to follow in order to use SafeStreets;

Privacy statement: describes why and how SafeStreets collects and uses personal data and provides information about *Users*' rights.

Reference code: unique code which represents a single *Municipality*;

User picture: picture of *Traffic violation* taken by the *User*;

Traffic violation: violation of parking laws;

User position: *Users*' GPS location;

License plate recognition service: third party service which provides the recognition of the license plate from a *User picture*;

Recognized license plate: text representing the license plate recognized by the *License plate recognition service*;

Timestamp: digital record of date and time of the day when it has been registered;

Public statistics: set of *User report* statistics provided by SafeStreets, available for all the registered *Users*;

Detailed statistics: set of *User report* statistics provided by SafeStreets, only available for the *Municipality*;

Traffic ticket: a notice issued to someone who violates a traffic regulation;

Accidents: data of *Municipality* Database, concerning previous *Traffic violations*;

Possible interventions: suggestions automatically generated by SafeStreets and forwarded to the *Municipality*;

Ticket feedback: information on whether a *User report* led to the generation of a *Traffic tickets* or not;

Municipality: district of a town or a city which has local government.

1.3.2 Acronyms

ACID: Atomicity, Consistency, Isolation, Durability;

API: Application Programming Interface;

DAO: Data Access Object;

DB: Database;

DBMS: Database Management System;

GPS: Global Positioning System;

GUI: Graphical User Interface;

IT: Information Technology;

MVC: Model View Controller;

REST: Representational State Transfer;

UX: User Experience.

1.3.3 Abbreviations

R_n: n^{th} Requirement;

U_n: n^{th} Use Case.

1.4 Revision History

1. Version 0.1 - 11th November 2019 - Start of the DD;
2. Version 1.0 - 9th December 2019 - First Release.

1.5 Reference Documents

- Agile Modeling. *An Introduction to Agile Modeling.* http://www.agilemodeling.com/essays/introductionToAM.htm
- Agile Modeling. *Introduction to the Diagrams of UML 2.X.* http://www.agilemodeling.com/essays/umlDiagrams.htm.

1.6 Document Structure

This document is structured as follows:

1. Introduction A general introduction of the system-to-be. It aims at giving general, but exhaustive, information about what this document is going to explain.

2. Architectural Design An overview about the high-level components and their interactions, with focus on both static and dynamic view, helped by diagrams.

3. User Interface Design A representation of how the User Interface will look like.

4. Requirements Traceability An explanation about how the requirements defined in the RASD map to the design elements defined in this document.

5. Implementation, Integration and Test Plan Identification of the order in which the sub-components of the system should be implemented, integrated and tested.

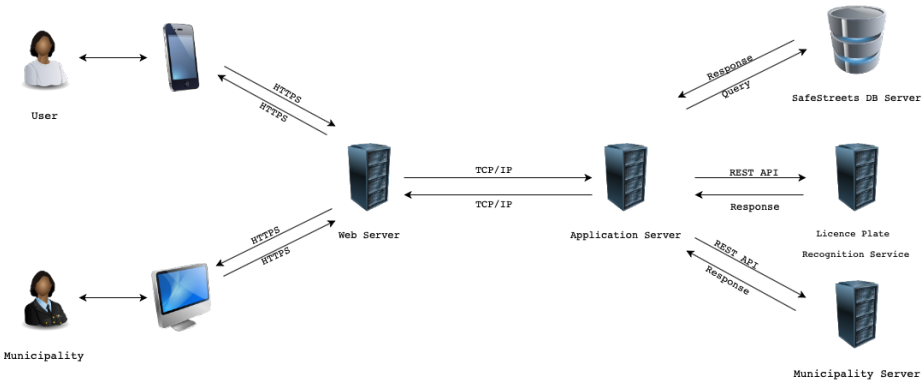
6. Effort spent Effort spent by all team members shown as the list of all the activities done during the realization of this document.

7. References References of documents upon which this project was developed.

Chapter 2

Architectural Design

2.1 Overview



Overview Diagram

The figure above represents an high-level description of the main components which constitute the system-to-be. They are organized in a 4-tier architecture, in particular:

Browser: is the Presentation tier for the *Municipality* Browser, it communicates with the Web Server;

Mobile application: is the Presentation tier for the *User* Mobile Application, it communicates with the Web Server;

Web Server: represents the Service tier, it communicates with the *Municipality* Browser and with the Mobile Application on one hand (i.e. the Presentation tier) and with the components of the Application Server on the other (i.e. the Business logic tier);

Application Server: represents the Business logic tier of the system, it communicates with the Web Server on one hand and with the Database tier on the other;

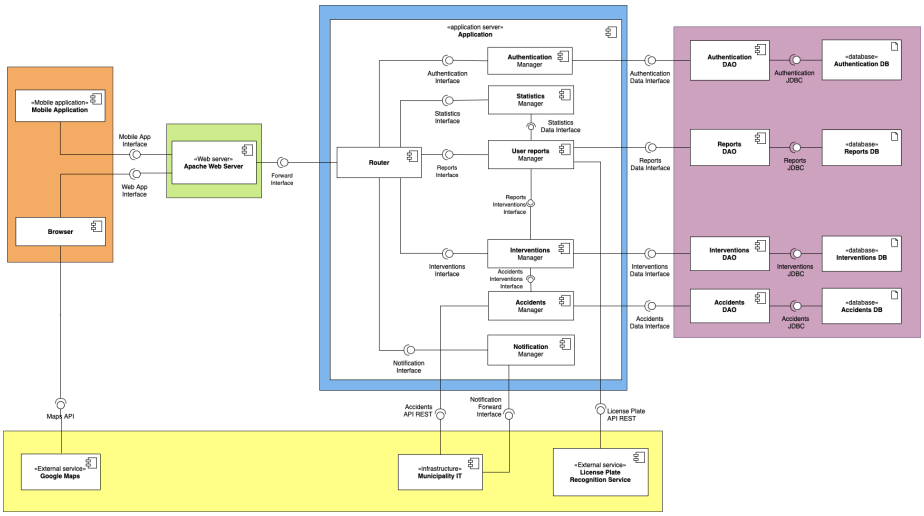
Data Access Objects and Database: represent the Database tier of the system; the Database communicates with the Business tier through each Data Access Object (DAO).

In particular, Web Server and Application Server have been separated mainly for security reasons. Further details about the architectural choices can be found in the sections 2.6 and 2.7.

2.2 Component View

In this section the system is presented in terms of its components with the help of a Component Diagram. Then, the functionalities of the components themselves are exploited and detailed.

The color in the background of the components represent the tier they belong to, in particular the Presentation tier is represented in orange, the Service tier is in green, the Business logic tier is in blue, the Database tier is in violet and the external services and infrastructures are in yellow.



SafeStreets Component Diagram

2.2.1 Presentation tier

Mobile Application: must communicate with the Apache Web Server through HTTPS connection; the Application UI must be designed user friendly and it has to follow the guidelines provided by the Android and iOS producer. The application itself must localize, through the GPS connection of *User*'s device, the position when reporting a violation. Then, it has to provide all *User report's* data to the system's Apache Web Server.

2.2.2 Service tier

Apache Web Server: must communicate with the *User*'s Mobile Application and the *Municipality* Browser through HTTPS connection; its primary function is to store, process and deliver web pages to requesting clients.

2.2.3 Business logic tier

Router: communicates with the Apache Web Server through TCP/IP protocol; its function is to forward requests and data coming from the Web Server to the right component in the Application Server. For simplicity reasons the Router component is not often mentioned in this document, although its role is crucial to the proper functioning of the whole system;

Authentication Manager: handles the registration process for each *User*, allows Registered Entities to be identified by their identifier (the email for the *User* and the *Reference code* for the *Municipality*) and stores their password in Authentication Database through Authentication DAO. Moreover, it guarantees a chain of custody matching *User reports*' data and *Users' identifiers* in a digital signature;

Statistics Manager: contains all the business logic in order to aggregate *User reports* data and generate *Public* and *Detailed Statistics* based on particular filters;

User reports Manager: contains all the business logic in order to send and receive *User report* data, loading and storing them in Reports Database through Reports DAO. It also handles the *Ticket Feedback* issuing confirmation sent by the *Municipality*, updating its status. Additionally, it manages the transfer of *User picture* to the *License plate recognition service*, eventually sending back to the client the *Recognized license plate* or a message suggesting to re-take the photo if it has not been possible to recognize the license plate. Finally, it allows specific *User reports'* information retrieval based on particular parameters;

Interventions Manager: contains the logic able to cross *User reports'* and *Accidents'* data in order to generate *Possible interventions* to suggest to *Municipality*;

Accidents Manager: contains all the business logic in order to send and receive periodically *Accidents* data from *Municipality*, loading and storing them in Accidents Database through its DAO;

Notification Manager: for each new violation reported by the *Users*, it is responsible of sending a notification to the concerned *Municipality*.

2.2.4 Database tier

Authentication, Reports, Accidents and Interventions DAO: the details about the Data Access Objects are shown in subsection 2.6.4;

Authentication, Reports, Accidents and Interventions Database: the details about the Relational Databases are presented in subsection 2.7.3.

2.2.5 External services and infrastructures

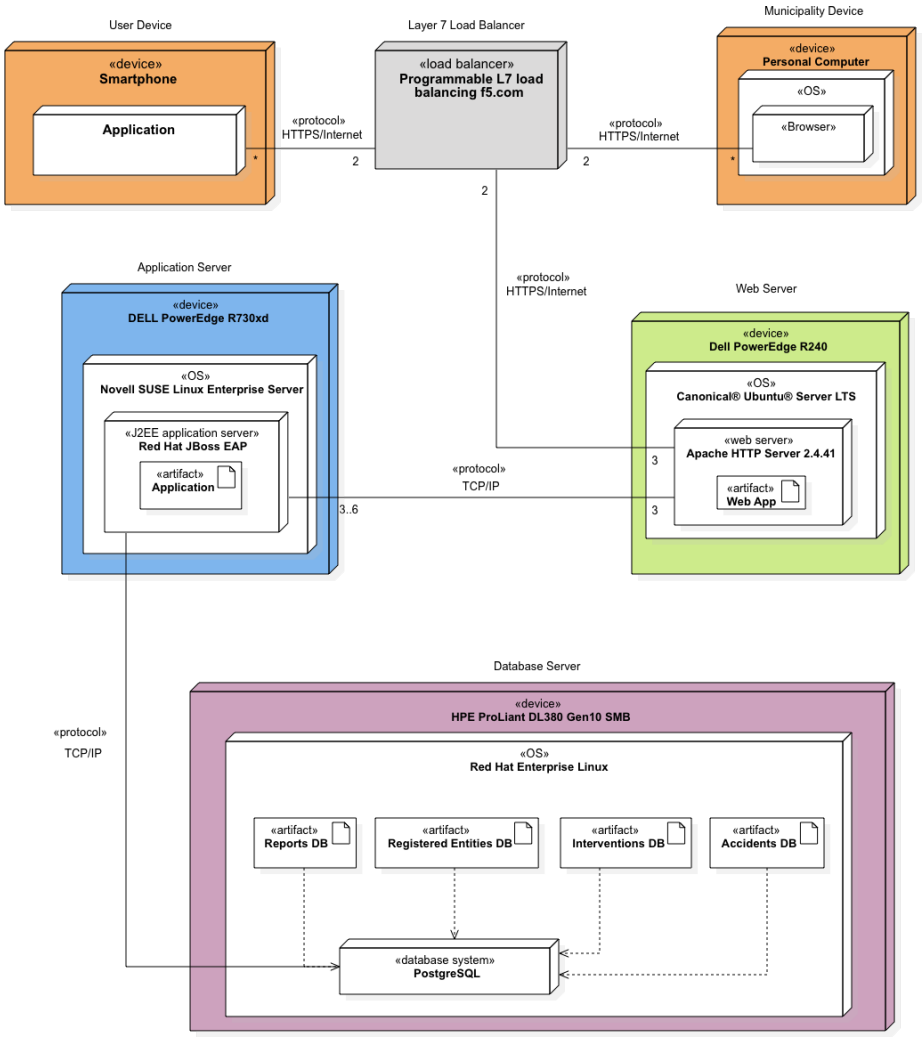
Google Maps : communicates with the Browser in order to provide the maps visualization service that allows to identify the area related to the suggested *Possible Interventions*. It uses APIs which guarantee also the integration with the service itself;

Municipality IT: communicates with the Application Server through RESTful APIs, both to receive notifications about new *User reports* (in particular from the Notification Manager) and to provide Accidents data from its own database (in particular answering the request of the Accidents manager);

License plate recognition service: communicates with the Application Server (in particular with the User reports Manager) through RESTful APIs, to receive a *User picture* and send back a text representing the license plate recognized in the picture (if one is recognized).

2.3 Deployment View

The following section aims at describing the deployment diagram of the system-to-be, from a physical point of view, with details on how both hardware and software will be implemented and delivered.



SafeStreets Deployment Diagram

A load balancer is at the system's core, to balance the calls from clients to servers. These calls use ISO/OSI layer 7 protocol HTTP to implement a RESTful web service between each *User*'s smartphone or *Municipality*'s device and the Application Servers.

From an organization internal perspective, instead, all servers communicate with each other and with the database through a TCP/IP protocol. Indeed the Application Server, which handles the business logic rules, communicates with the Web Server through public interfaces defined in the following paragraphs, while the communication between each J2EE Application instances and the Database Server is made possible through PostgreSQL public API of the database system.

To manage the situation of high workload and ensure both reliability and availability, the server architecture has been split in several machines: three Web Servers will handle the HTTP/REST calls and from a minimum of 3 to a maximum of 6 machines will host and replicate the Application Server.
In order to understand better the normal work load of the system, a load testing analysis should be performed once the implementation phase will be over, with a testing tool like Apache JMeter.
For availability purposes, also the load balancers have been duplicated with an IaaS management of internal and external IP addresses of the machines.

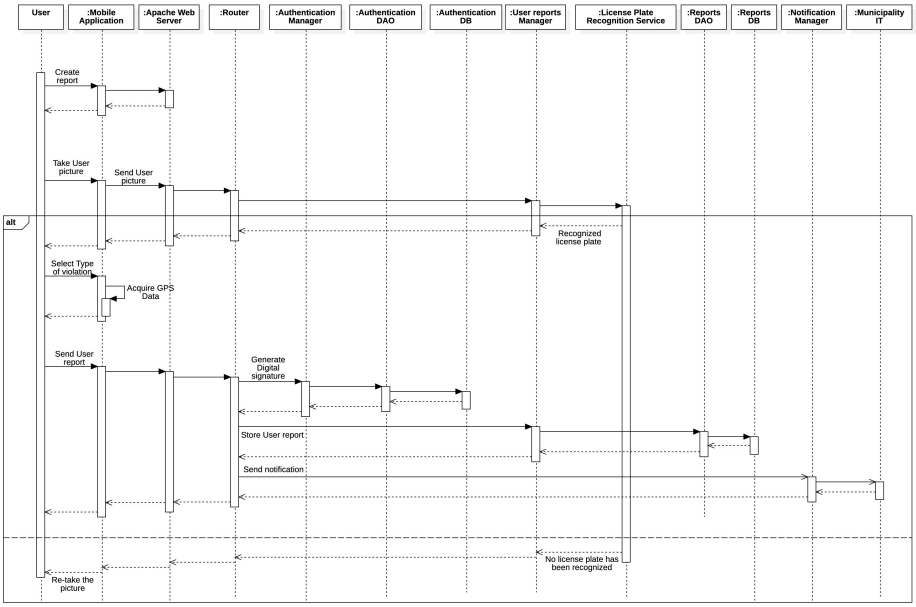
2.4 Runtime View

In this section the dynamic behaviour of the system is described, with the help of some Sequence Diagrams concerning the most relevant use cases.

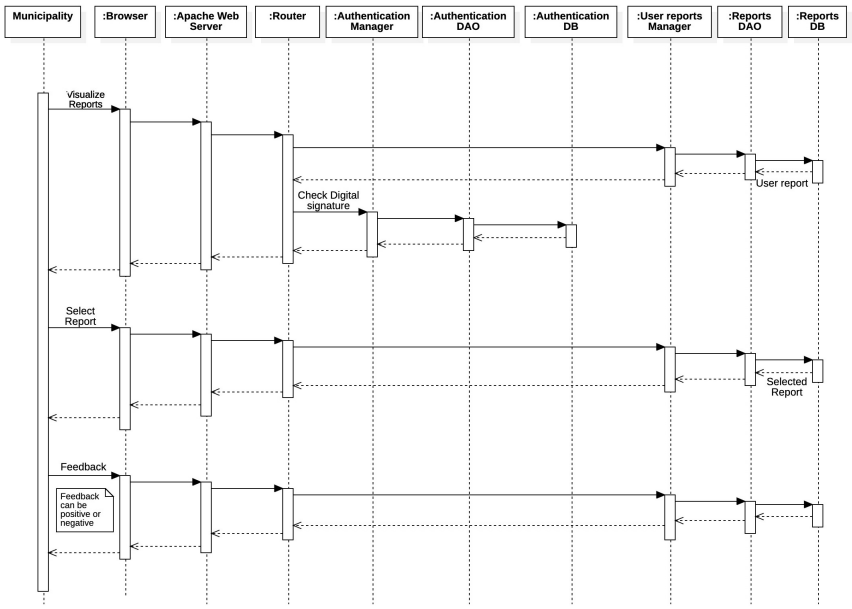
In the Sequence Diagram U_4, once the *User report* is sent by the *User*, its Digital signature is generated by the Authentication Manager using data from the Authentication DB, and then stored with the correspondent *User report* in the Reports DB; in the Sequence Diagram U_5, instead, before returnig to the *Municipality* the selected *User report*, the Digital signature is checked by the Authentication Manager, using data from the Authentication DB.
These processes are executed in order to guarantee the integrity of the *User report* chain of custody (subsection 2.7.2).

In the Sequence Diagram U_{8a}, the described process is periodically executed to update the Accidents in the Accidents DB, retrieving only the latest ones from the Municipality IT, in order to optimize the database management.

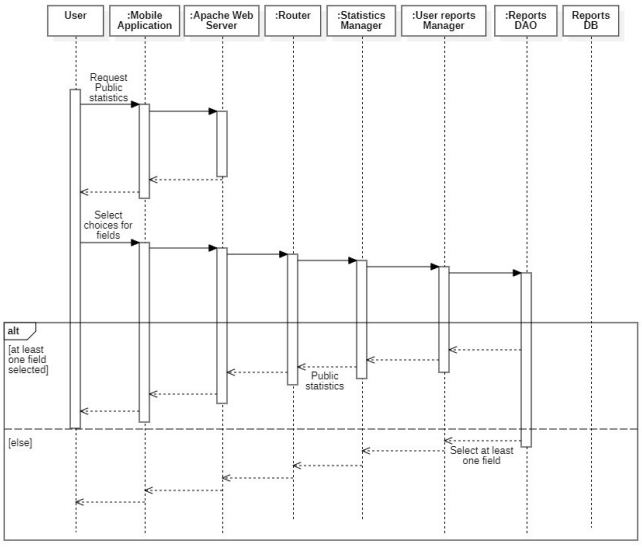
Furthermore, most of the use cases require, as first task, the authentication of *User* or *Municipality*; the only case in which the authentication phase is not required is the generation of the *Possible interventions* by SafeStreets, which is automatically performed periodically (more details in Sequence Diagram U_{8a}). For this reason, the authentication process is not shown in the following Diagrams.



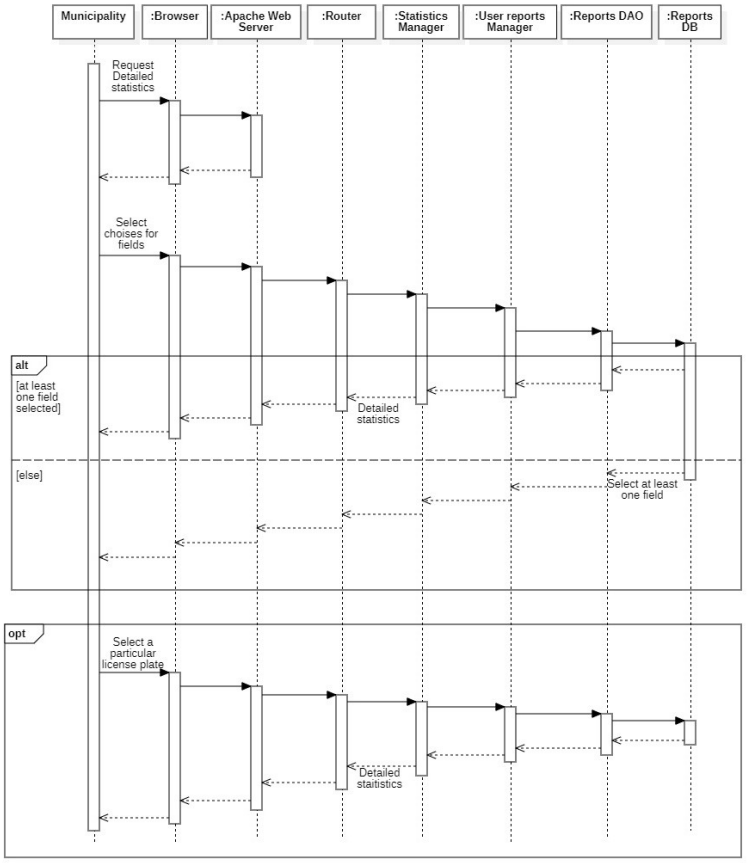
U4: User sends User report to SafeStreets Sequence Diagram



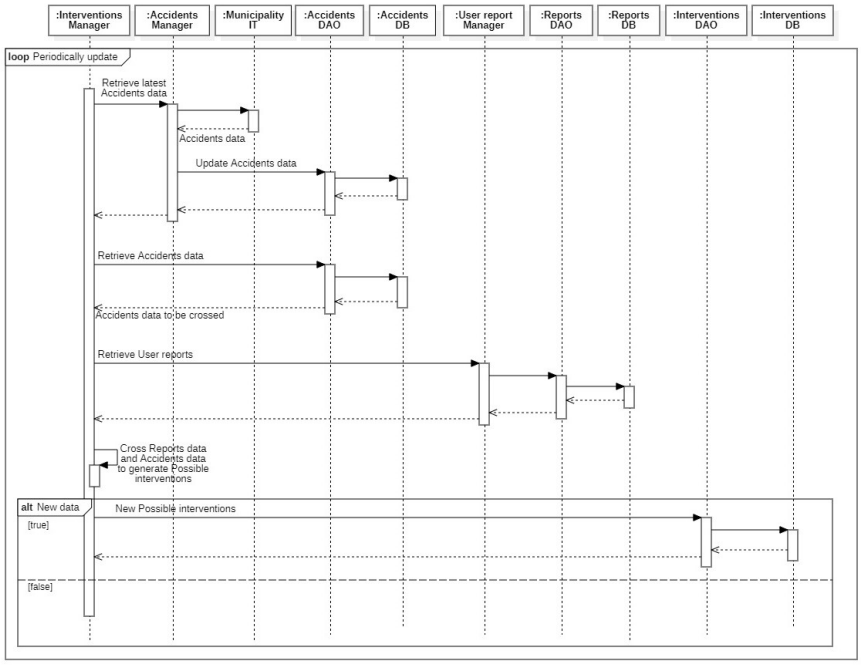
U5: Municipality answers to a User report Sequence Diagram



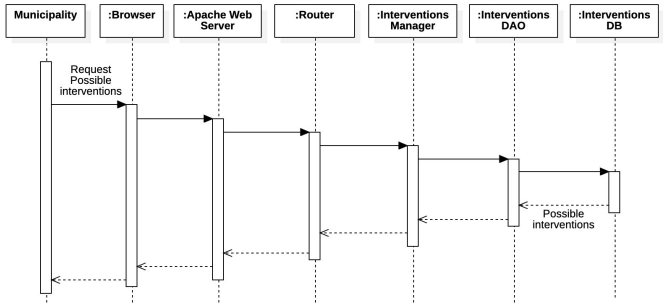
U6: User visualizes Public statistics Sequence Diagram



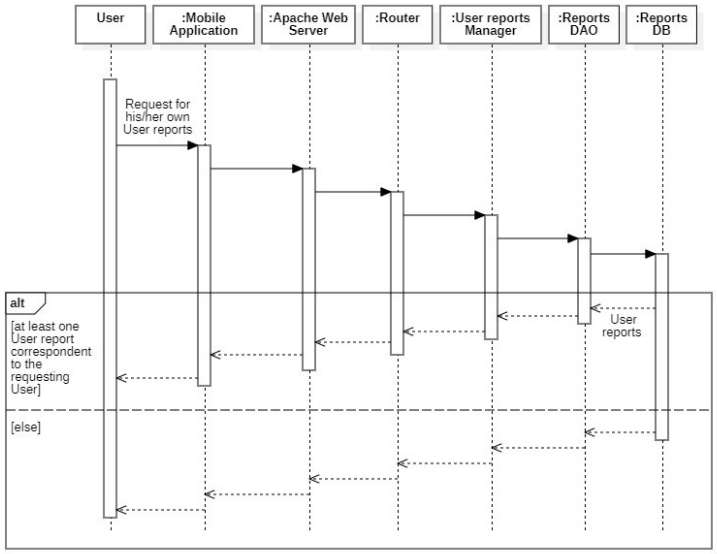
U7: Municipality visualizes Detailed statistics Sequence Diagram



U8a: SafeStreets creates Possible interventions for Municipality Sequence Diagram



U8b: Municipality visualizes Possible interventions



U9: User visualizes his/her contribution to SafeStreets Sequence Diagram

2.5 Component interfaces

This section shows the interfaces belonging to the system. We focused mainly on the internal interfaces of the Application Server and on those which are external but related to the Application Server itself.
Furthermore, the Forward Interface between the Web Server and the Router is reported without any method for semplicity reasons, as it contains all the methods belonging to the Interfaces between the Router component itself and all the other components of the Application Server.
Further details about the Interfaces are mentioned in the following commented code lines.

```
//        --- COMPONENT INTERFACES ---

//Interfaces between MobileApplication, Browser and Web Server: Web server manages the HTTPS communications with MobileApplication and Browser

//User
public interface MobileAppInterface{} //HTTPS request
//Municipality
public interface WebAppInterface{} //HTTPS request

//---

//Interface between Web Server and Application
//It contains the methods to forward the requests coming from the Web Application to the Router in the Application
public interface ForwardInterface{
  //...
}

//---

//Interfaces within Application
public interface ReportsInterface{

  //User
  boolean getMyReports(String email);
  boolean validateReportPicture(Picture picture); //picture from Mobile Application
  boolean sendUserReport(UserReport userReport);

  //Municipality
  boolean getReports();
  boolean sendFeedback(UserReport userReport);
}
public interface AuthenticationInterface{
  boolean signUpUser(String email);
  boolean loginUser(String email, String password);
  boolean loginMunicipality(String referenceCode, String password);
}
public interface StatisticsInterface{
```

```
38    boolean visualizePublicStatistics(Date startDate, Date endDate,
        String typeOfViolation, Position position);
39    boolean visualizeDetailedStatistics(Date startDate, Date endDate,
         String typeOfViolation, Position position, String
        licensePlateNumber);
40  }
41  public interface InterventionsInterface{
42    boolean visualizePossibleInterventions(Date time);
43  }
44  public interface StatisticsDataInterface{
45    Map<String,String> collectAsPublicStatistics(); //aggregates User
         reports data
46    Map<String,String> collectAsDetailedStatistics(); //aggregates
        User reports data
47  }
48  public interface ReportsInterventionsInterface{
49    List<UserReport> getUserReports(Position position, Date time);
50  }
51  public interface AccidentsInterventionsInterface{
52    List<Map<String,String>> getAccidents();
53  }
54  public interface NotificationInterface{
55    boolean createNotification(UserReport userReport);
56  }
57
58  //---
59
60  //Interfaces between Application and DataAccessObject
61  public interface AuthenticationDataInterface{
62    User getUser(String email);
63    Municipality getMunicipality(String referenceCode);
64    boolean insertUser(String email, String password);
65    boolean insertMunicipality(String referenceCode, String password)
        ;
66    boolean checkLoginUser(String email, String password);
67    boolean checkSignupUser(String email);
68    boolean checkLoginMunicipality(String referenceCode, String
        password);
69
70  }
71  public interface ReportsDataInterface{
72    boolean insertUserReport(UserReport userReport);
73    //to retrieve User reports from Reports DB for My Reports(User
        request) and Ticket feedback(Municipality request)
74    List<UserReport> getUserReports(String email, Date startDate,
        Date endDate, String typeOfViolation, Position position, String
         licensePlate);
75  }
76  public interface InterventionsDataInterface{
77    //to retrieve latest Interventions from Interventions DB
78    List<Map<String,String>> getInterventions(Date time);
79    //to add the latest Interventions to Interventions DB
80    boolean insertInterventions(List<Map<String,String>>
        interventions);
81  }
82  public interface AccidentsDataInterface{
83    //to add new Accidents to Accidents DB
```

```
84    boolean insertNewAccidents(List<Map<String,String>> accidents);
85    //to retrieve the quiered Accidents data from Accidents DB
86    List<Map<String,String>> getAccidents(Date startDate, Date
        endDate, Position position);
87  }
88
89  //---
90
91  //Interfaces between DataAccessObject and Database
92  //DataAccessObjects manage the communications with Databases
        throught JDBC APIs
93  public interface AuthenticationJDBC{}
94  public interface ReportsJDBC{}
95  public interface InterventionsJDBC{}
96  public interface AccidentsJDBC{}
97
98  //---
99
100 //Interfaces with External services
101 public interface NotificationForwardInterface{
102   boolean sendNotification(UserReport userReport);
103 }
104 public interface AccidentsAPIREST{
105   List<Map<String,String>> getAccidents(Date time);
106 }
107 public interface LicensePlateAPIREST{
108   String getRecognizedLicensePlate(Picture picture);
109 }
110 public interface MapsAPI{} //HTTPS request
```

2.6 Selected Architectural Styles and Patterns

2.6.1 4-Tier Architecture

For the system to be developed and its infrastructure the chosen architecture is the Four-Tier one. As can also be found in the Deployment Diagram in the section 2.3, four main tiers are needed by the system: one or more clients on different devices, a Web Server that will communicate with the Application Server and a Database. The main reason behind the segmentation between Web Server and Application Server relies on network access and on security purposes as well.
The Four-Tier model is architected to create a foundation for excellent performance, device-tailored experiences, and allows for integration of both internal services and applications as well as third-party services and APIs. Moreover, all parts of the system can be upgraded independently and even more scalability can be obtained through replication.

The chosen pattern for the whole architecture of the system is the remote presentation, where clients are not provided with any application logic but the GUI, with which *Users* and *Municipalities* interact with the Application Server, through the Web Server.
For example, a *User* can send a *User report* taking a *User picture* that will, then, be sent to the Application Server for its validation and license plate number recognition.

The Web Server represents the Service Layer, communicating with the Presentation Layer handled by clients' mobile or web applications. At the system's core is the Application Server, which implements the Business Logic tier through a JavaEE application. The Database, instead, representing the Database tier, will be queried by the Business logic tier. It is the place where all information are stored and there is no database on clients' side; consequently, all data will be downloaded through the network from the server.

2.6.2 Thin Client

The decision to adopt the thin client architecture has been taken for different purposes.
First of all, if a logic component changes, the application won't need to be updated but only the server will. This also allows the data processing to be only server side, without impacting client devices' performances.

Isolating the user interface from the other three tiers behind it in this way, gives frontend and user-experience designers much more control without disrupting backend processes or engineering. Similarly, it leaves backend designers free to adopt the best technology for their tier as long as the established protocols and encoding for information delivery to the client tier are maintained.

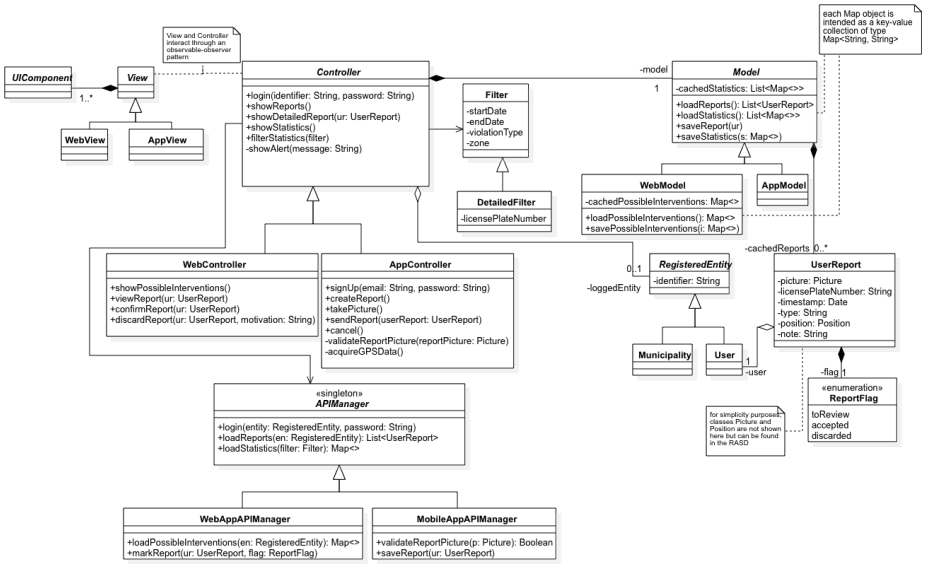
The communication between each thin client and the server happens through HTTPS protocol, which carries REST requests and responses formatted as JSON in their payloads.

2.6.3 Model-View-Controller

The chosen design pattern for mobile and web applications implementations is the Model-View-Controller (MVC), which separates an application into three interconnected elements, each one realizing a functionality.

While the Controller updates the View, the Model represents a thin data manager, where a small cache of data previously downloaded from the server are temporary stored, manipulated and updated by the Controller.

This pattern, from a Software Engineering point of view, guarantees the reusability of code and parallel development.



Client Applications Class Diagram

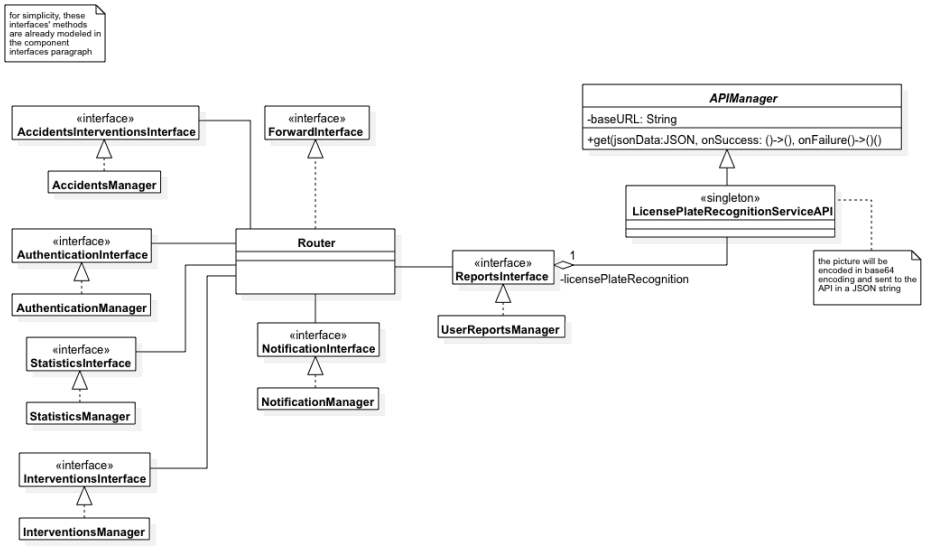
The picture above models the class diagram of clients' applications in our system, with a Controller that communicates with the Web Server through an API Manager class, responsible of handling asynchronous connections and data decoding.

2.6.4 Data Access Object

The Data Access Object (DAO) architectural pattern has been used in our system in order to separate the Business logic tier services from the Database. The DAOs implement the access mechanism required to deal with the data source, which in our case is a RDBMS (Relational DBMS): the Business logic tier uses the interface provided by the DAO, which aims to hide completely the details about the interaction with the data source and this happens because the DAO works as an adapter between the Business logic tier and the Database, as it doesn't change the provided interface if the data source implementation changes.

2.6.5 Application Logic

For completeness, here it is shown the class diagram describing the application running on the Application Server, that handles the entire Business logic of the application itself.



Application Logic Class Diagram

2.7 Other design decisions

2.7.1 User Authentication and Password Storing

In order to guarantee the confidentiality of *Users'* and *Municipalities'* data, the access to the system is guaranteed by entering a personal identifier and a password. For *Users*, the personal identifier coincides with the email address inserted during the registration phase.

For each registered entity, their access passwords are hashed using a SHA512 algorithm and then saved in the relative Database.

2.7.2 Chain of Custody

User reports constitutes sensitive data because they contain license plates' numbers, GPS data and, in general, information about *Traffic violations* that will be shown to a *Municipality*. To ensure that these information won't be altered, a digital signature mechanism has been implemented to realize the chain of custody.

In particular, when a *User* sends his/her *User report* to the Application Server, a hash function on its data and the *User* entity is computed, returning the digital signature corresponding to the specific report. This hashcode is, then, stored in the Authentication Database and accessed at retrieval time. When a *Municipality* client sends a request to access those *User reports*, the same hash function is computed and the result is checked with the stored one, in order to ensure that no alterations have occurred.

2.7.3 Relational Database

The data management of the system will be handled by a Relational DBMS. This choice has, from the organization point of view, both internal and external motivations: the first thing to consider is the effort and time spent by the development team, which is familiar with relational databases, while the usage of a NoSQL DB would require time for the study and design process of it. Moreover, a relational database is transactional: each access operation represents a real-world event of every enterprise and the so-called ACID properties are guaranteed:

Atomicity : no partial execution is allowed;

Consistency : every state is consistent after a transaction execution;

Isolation : transactions should be executed in isolation from other transactions;

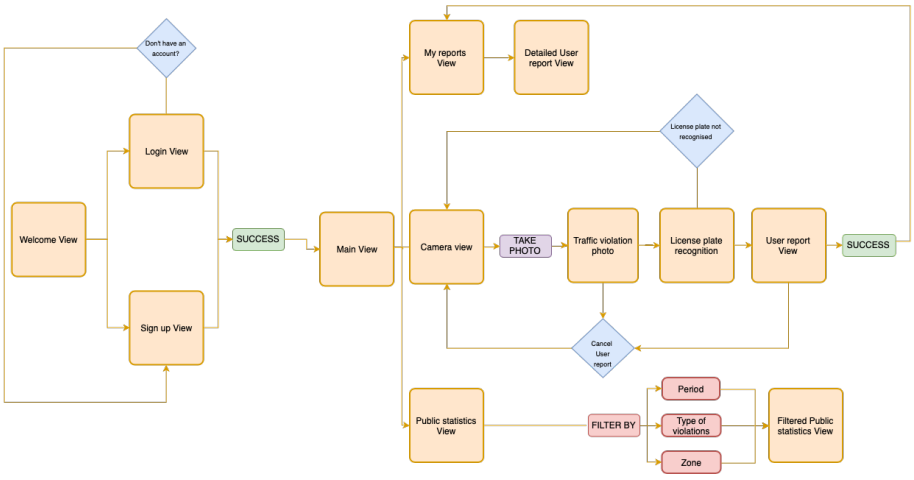
Durability : even in case of failures, changes in the database should persist.

Chapter 3

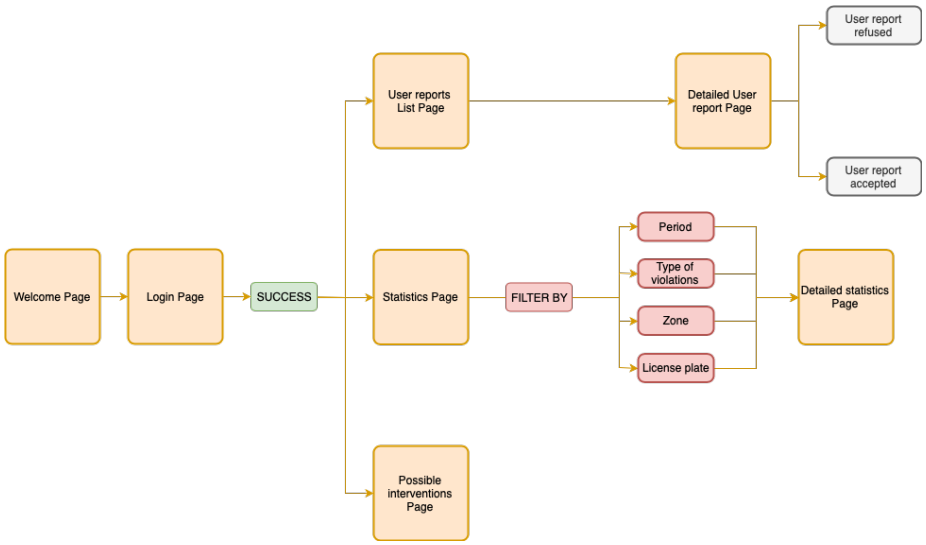
User Interface Design

3.1 UX Diagrams

In this section are presented the possible screens, the displayed information and the possible interaction that the *User* and the *Municipality* might find using the system.



Mobile Application UX Diagram



Web Application UX Diagram

3.2 User Interface Mockups

The mockups have been included in the section 3.1.1 of the Requirements Analysis and Specification Document (RASD).

Chapter 4

Requirements Traceability

In this section, the mapping between the DD components and RASD functional requirements is shown. The requirements are grouped by the components which are used to satisfy them, also highlighting the specific functionality of the components themselves.

In the provided mapping, the "Router" component is not mentioned for simplicity reasons, but clearly it is directly or indirectly connected to the fulfillment of the majority of the system functionalities, as it routes to the right component every message coming from the Web Application side.

R_1 At sign up, *User* must provide email and password.

R_2 At sign up, *User* must accept *Terms and conditions*, including the *Privacy statement*.

- **Authentication Manager:** handles the registration process for each *User*.

R_3 Identify a *User* by his/her *User identifier*.

R_{13} Query SafeStreets Database for a *User* by his/her *User identifier*.

R_{29} Identify *Municipality* by its *Reference code*.

- **Authentication Manager:** allows Registered Entities to be identified by their identifier and password that are stored in Authentication Database through Authentication DAO.

R_4 Receive *User picture*.

R_5 Receive *User* choice for the type of *Traffic violation*.

R_6 Receive *User position*.

R_9 Create *User report*.

R_{10} Store *User report* in SafeStreets Database.

R_{14} Retrieve specific *User reports* by querying SafeStreets Database.

R_{20} Allow *Users* to request all their *User reports* and the related *Ticket feedback* stored in SafeStreets Database at any time.

R_{21} Send to a specific *User* his/her stored *User reports*.

R_{22} Send to a specific *User* the *Ticket feedback* related to his/her *User report*.

- **User reports Manager:** contains all the business logic in order to send and receive *User report* data, loading and storing them in Reports Database through Reports DAO.

R_7 Send *User picture* to the *License plate recognition service*.

R_8 Receive *Recognized license plate* from the *License plate recognition service*.

R_{26} Allow *User* to re-take the *User picture*.

- **User reports Manager:** manages the transfer of *User picture* to the *License plate recognition service*, eventually sending back to the client the recognized license plate.

R_{12} Store *Ticket feedback* in SafeStreets Database.

R_{25} Receive *Ticket feedback* from *Municipality*.

- **User reports Manager:** handles the *Ticket Feedback* issuing confirmation sent by the *Municipality* and updates its status in Reports Database through Reports DAO.

R_{11} Send *User report notification* to *Municipality*.

- **Notification Manager:** for each new violation reported by *Users*, it is responsible of sending a notification to the concerned *Municipality*.

R_{15} Validate the data which constitutes the *User report*.

- **Authentication Manager:** guarantees a chain of custody matching *User reports'* data and *Users' identifiers* in a digital signature.

R_{16} Generate *Possible interventions* crossing *Municipality Accidents* data with SafeStreets Database data.

R_{17} Store generated *Possible interventions* for the *Municipality*.

- **User report Manager:** allows specific *User reports'* information retrieval based on report position and timestamp.
- **Accidents Manager:** periodically, it retrieves latest accidents data from *Municipality's* database, stores them in Accidents database and collects accidents basing on particular parameters.

- **Interventions Manager:** contains the logic able to cross *User reports'* and *Accidents'* data in order to generate *Possible interventions* to suggest to *Municipality*.

R_{18} Allow *Municipality* to set specific constraints to define *Detailed statistics*.

R_{19} Send *Detailed statistics* to the requesting *Municipality*.

R_{24} Send *Public statistics* to the requesting *User*.

R_{23} Allow *User* to choose among filters which define *Public statistics*.

- **Statistics Manager:** contains all the business logic in order to aggregate *User reports* data and generate *Public* and *Detailed Statistics* based on particular filters.

R_{27} Request *Accidents* data to *Municipality*.

R_{28} Receive *Accidents* data from *Municipality*.

- **Accidents Manager:** contains all the business logic in order to send and receive *Accidents* data from *Municipality* IT, loading and storing them in Accidents Database through Accidents DAO.

Chapter 5

Implementation, Integration and Test Plan

5.1 Development Process

The following table represents the main features of the system-to-be, along with the relative importance that it has for the customer and the difficulty of implementing it.

Feature	**Importance for the customer**	**Difficulty of implementation**
Sign up and login	Low	Low
User report generation	High	High
Own reports visualization	Medium	Low
Public statistics visualization	Medium	Medium
Ticket feedback generation	High	Low
Detailed statistics visualization	Medium	Medium
Possible interentions visualization	Medium	High

Of course, the Implementation, Integration and Testing process will be based on this evaluation.

5.2 Implementation

5.2.1 Implementation Plan

This subsection shows the list of the Application Server components and reports and explains the order of their implementation based on what is described in the previous table (Section 5.1).

User reports Manager : it is the first component to be implemented as it has an high importance for the customer, it has an high difficult of implementation and will be used by many other components;

Accidents Manager : it follows the implementation of the User report Manager component, since it is used by the Interventions Manager component to generate *Possible Interventions* (so it has to be implemented before that), but the correlated feature has a lower importance for the customer than the User report management;

Interventions Manager : it should be implemented after the User report Manager component and the Accidents Manager because it uses them in order to generate *Possible Interventions*;

Statistics Manager : it could be implemented in parallel with Accidents Manager, after User report Manager, as it uses this one to generate both *Public Statistics* and *Detailed Statistics*. Anyway, it is preferable to do it after Accidents Manager due to its lower importance for the customer;

Notification Manager : it could be implemented in parallel with Accidents Manager and Statistics Manager, after User report Manager, but it is done immediately after User report Manager as it is part of the same feature ("User report generation");

Authentication Manager : it follows the implementation of User report Manager and should be built in parallel with Notification Manager due to the fact that it is used to accomplish the "*User report* generation" feature (in particular, for the digital signature), despite it also manages the "Sign up and login" feature which is the one with the lowest importance for the customer;

Router : it is the last component that has to be implemented because it provides the communications redirection with the other components.

5.2.2 Implementation Choices

Mobile Application

The Mobile Application must be implemented in two different architecture respecting native languages, Swift 4 for iOS application and Java for Android ones. The communication with the device must be done using the default frameworks

of the respective system; moreover, the communication with the Apache Web Server must be performed with HTTPS protocol.

Browser

The Web Application must be implemented for different operating systems in order to be compatible with the most number of existing browsers such as Google Chrome, Mozilla Firefox, Microsoft Edge, Apple Safari. The communication with the Apache Web Server must be performed through the HTTPS protocol, while the communication with Google Maps service must be performed through Maps API.

Web Server

The implementation of this tier is realized through an Apache Web Server 2.4.41. A TCP/IP protocol is used to interface with the J2EE Application Server and the HTTPS protocol to interface with the clients and the load balancer (Section 2.3).

Application Server

The Application Server implementation will be executed on a machine with Red Hat JBoss EAP on it, using a J2EE software. A TCP/IP protocol is used to interface the software with every Database Server, while the interface with External Services like *Municipality* IT and *License plate recognition service* uses RESTful APIs.

Database

The choice that has been taken for the Database is PostgreSQL as Relational DBMS, already previously described and motivated in subsection 2.7.3.

5.3 Integration

In this Section the integration flow is shown. Each component is integrated in the component that receives the arrowhead.

5.3.1 Entry Criteria

For the success of the integration phase some precondition need to be satisfied:

- The RASD and the DD must be available to all the stakeholders;
- The DBMS should be completely operative in order to test all the components that use it;
- The Maps API should be fully operative and available.

5.3.2 Elements to be integrated

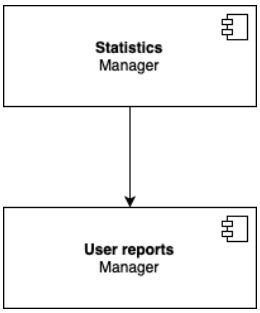
Considering the huge complexity of the system we have to distinguish the subsystems into four categories:

- Presentation tier components: Mobile application, Web application;
- Service tier components: Web Server;
- Business logic tier components: Application Server internal components;
- Database tier components: Data Access Objects, DB components;
- External components: Google Maps, License Plate Recognition Service, Municipality IT.

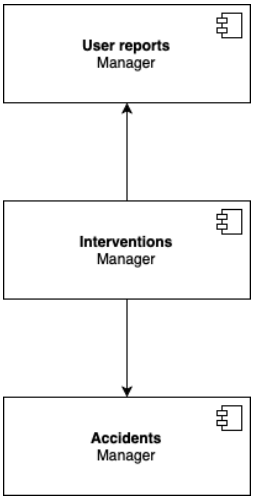
5.3.3 Integration Sequence

Integration of the internal components of the Business logic tier

Each internal component of the Application Server, that cointains the entire Business logic of the system, is integrated with the Router component. It provides to estabilish communications between the Web Server and the Application Server components and it has been omitted in this paragraph due to redundancy reasons.



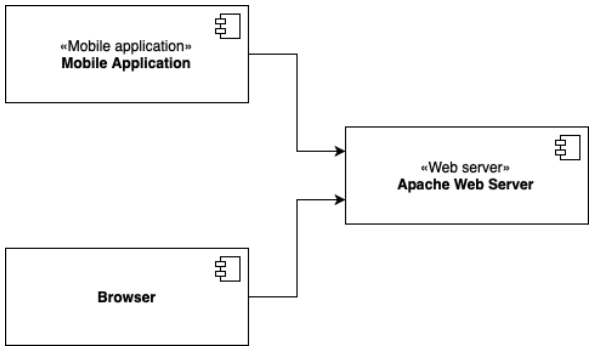
Statistics Manager Integration



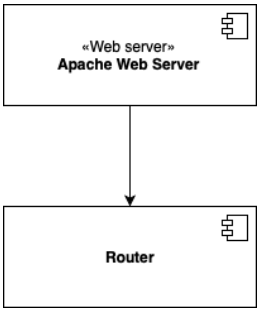
Interventions Manager Integration

Integration with the Web tier

The following diagrams represent the integration between the Presentation tier and the Service tier and then between the Web Server component and the Router component.



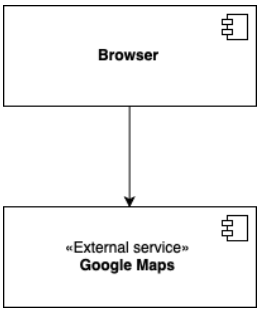
Web Server Integration



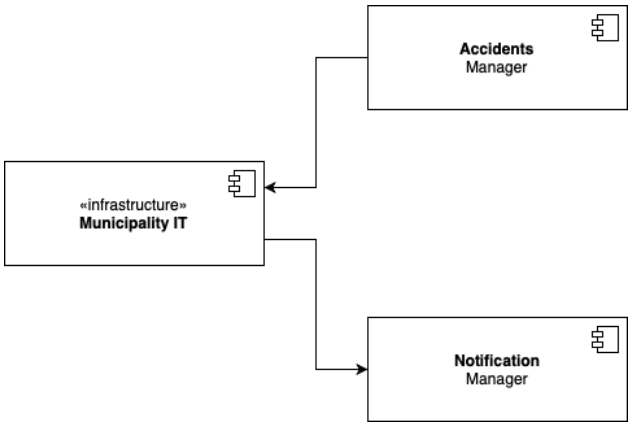
Router Integration

Integration with the external services

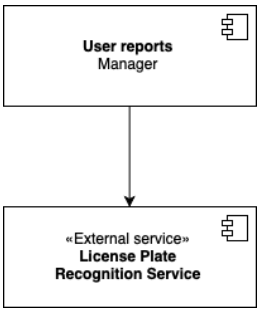
The following diagrams represent the integration of the components that are part of the Presentation tier and the Business logic tier with some External Service.



Google Maps Integration



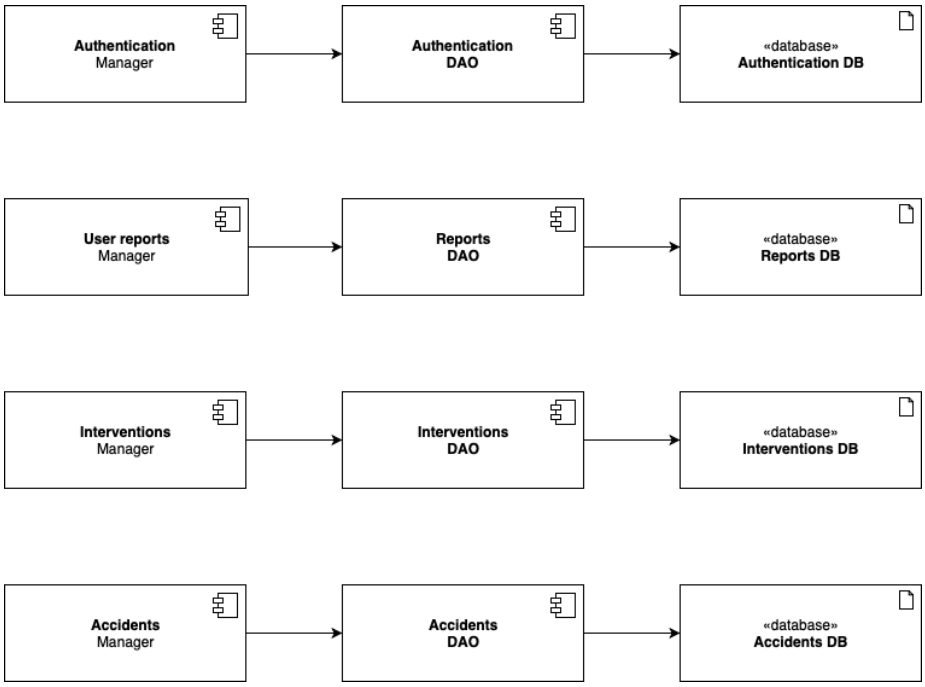
Municipality IT Integration



License Plate Recognize Service Integration

Integration with the Database tier

The DataAccessObject components essentially "are used" by manager components to access database components, as the arrowhead shows.



DataAccessObjects - Database Integration

5.4 Test Plan

Unit Testing

All the classes should be tested through Unit Tests checking their behaviour, in particular using tools like JUnit. The line coverage is expected to be of at least 90% (only the View part could not respect the line covarge constraint).

The purpose is to validate that each unit of the software works as designed. It is performed prior to Integration Testing. Every component have to be tested using a white-box testing unit-test after every component implementation.

Integration Testing

The approach adopted is incremental, so testing is done by joining two or more modules that are logically related. Then the other related modules are added and tested for the proper functioning.

The strategy, specifically, is bottom-up; each module at lower levels is tested with higher modules until all modules are tested successfully. In this way it should be easier to localize any eventual fault and no time is wasted waiting for all modules to be developed.

Chapter 6

Effort Spent

Ferrara Fabiana Total hours of work: 77h

- 1h General LaTeX setting
- 1h Industrial Tutoring
- 7h DD Review homework
- 2h Overview diagram
- 4h Component diagrams
- 4h Deployment diagrams
- 1h Meeting with Professor
- 4h Component diagrams revision
- 6h Sequence diagrams
- 2h Deployment diagrams revision
- 4h Component diagram revision
- 5h Component interfaces
- 5h Class diagrams
- 8h Architectural Styles and Patterns
- 3h User Interface Design
- 3h Requirements Traceability
- 3h UML Diagrams Revision
- 1h Meeting with Professor

- 1h Industrial Tutoring
- 4h Implementation, Integration and Testing
- 6h General Revision
- 2h Final Revision

Formicola Stefano Total hours of work: 77h

- 1h General LaTeX setting
- 1h Industrial Tutoring
- 7h DD Review homework
- 2h Overview diagram
- 4h Component diagrams
- 4h Deployment diagrams
- 1h Meeting with Professor
- 4h Component diagrams revision
- 6h Sequence diagrams
- 2h Deployment diagrams revision
- 4h Component diagram revision
- 5h Component interfaces
- 5h Class diagrams
- 8h Architectural Styles and Patterns
- 3h User Interface Design
- 3h Requirements Traceability
- 3h UML Diagrams Revision
- 1h Meeting with Professor
- 1h Industrial Tutoring
- 4h Implementation, Integration and Testing
- 6h General Revision
- 2h Final Revision

Guerra Leonardo Total hours of work: 77h

- 1h General LaTeX setting
- 1h Industrial Tutoring
- 4h DD Review homework
- 2h Overview diagram
- 4h Component diagrams
- 4h Deployment diagrams
- 1h Meeting with Professor
- 4h Component diagrams revision
- 6h Sequence diagrams
- 2h Deployment diagrams revision
- 4h Component diagram revision
- 5h Component interfaces
- 5h Class diagrams
- 8h Architectural Styles and Patterns
- 3h User Interface Design
- 3h Requirements Traceability
- 3h UML Diagrams Revision
- 1h Meeting with Professor
- 1h Industrial Tutoring
- 4h Implementation, Integration and Testing
- 6h General Revision
- 2h Final Revision

Chapter 7

References

1 E. Di Nitto. *Lecture Slides*. Politecnico di Milano.

2 E. Di Nitto. *Mandatory Project Assignment AY 2019-2020*. Politecnico di Milano.