Politecnico di Milano

Software Engineering II project

SafeStreets

Design Document

Authors:
Mattia Calabrese
Federico Capaccio
Amedeo Cavallo

Professor:
Elisabetta Di Nitto

December 15, 2019

version 1.1

Contents

List of Figures

List of Tables

1 Introduction

1.1 Purpose of this document

In this document we are going to describe software design and architecture of the SafeStreets system.
The software architecture of a system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.[1]

1.2 Scope

SafeStreets is a crowd-sourced application that intends to provide users with the possibility to notify authorities when traffic violations occur, and in particular parking violations.

The system allows users to send pictures of violations, including suitable metadata, to authorities. Examples of violations are vehicles parked in the middle of bike lanes or in places reserved for people with disabilities, double parking, and so on. In addition, the system allows users to mine the previously stored information, for example by highlighting the streets (and the areas) with the highest frequency of violations, or by showing different types of statistics built upon the collected data.

The system will also provide a communication interface to the municipality's provided service to create a secure bridge for data transfer. This connection will enable SafeStreets to cross its data with municipality's to make analysis and build different types of statistics. Moreover the system will offer back to the municipality the possibility to retrieve information about the violations in order to generate traffic tickets from it and receive suggestions on possible interventions.[2]

1.3 Glossary

The *SafeStreets: Requirements Analysis and Specification Document*[3] should be referenced for terms not defined in this section.

1.3.1 Definitions

1.3.2 Acronyms

RASD: Requirements Analysis and Specification Document

DD: Design Document

API: Application Programming Interface

GPS: Global Position System

DB: DataBase

DBMS: DataBase Management System

GIS: Geographic Information System

ER: Entity Relationship Model

REST API: REpresentational State Transfer API

JAX-RS: JAVA API for REST Web Services

VPN: Virtual Private Network

HTTPS: HyperText Transfer Protocol over Secure Socket Layer

1.3.3 Abbreviations

m: meters (with multiples and submultiples)

w.r.t.: with respect to

i.d.: id est

i.f.f.: if and only if

e.g.: exempli gratia

etc.: et cetera

1.4 Reference documents

Context, domain assumptions, goals, requirements and system interfaces are all described in the *SafeStreets: Requirements Analysis and Specification Document.* [3]

1.5 Document overview

This document is structured as

1. **Introduction**: it provides an overview of the entire document
2. **Architectural design**: it describes different views of components and their interactions
3. **User interface design**: it provides an overview on how the user interfaces of our system will look like
4. **Requirements traceability**: it explains how the requirements we have defined in the RASD map to the design elements that we have defined in this document.
5. **Implementation, integration and test plan**: it focuses on the implementation and testing strategies that will be adopted to build the system

2 Architectural design

2.1 Overview

2.1.1 Context viewpoint

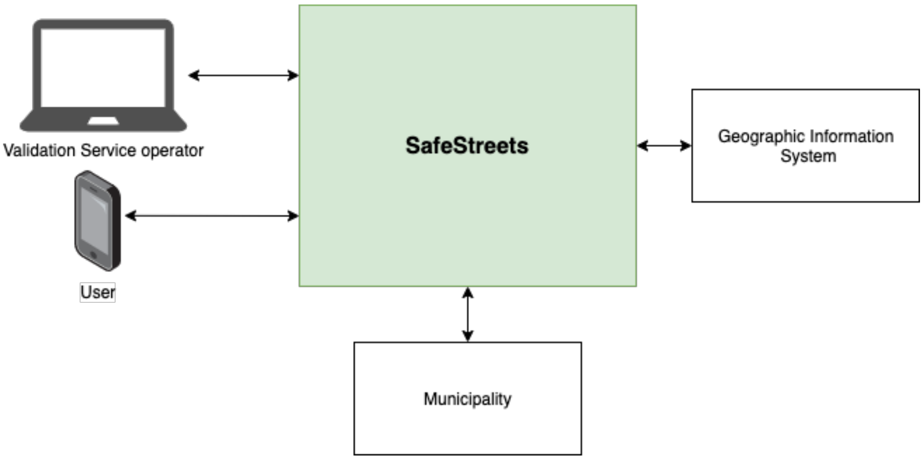

Figure 1: Context viewpoint

The system will interact with three actors: users, the Municipality and the Geographic Information System. Moreover we recognize that in most of the interactions the system is providing a service to agents so, after taking in consideration different alternatives, we decided to use a client-server architectural approach.

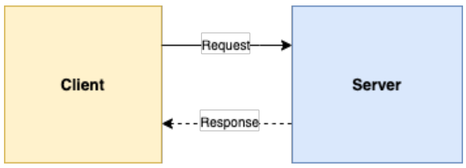

Figure 2: Client Server architecture

2.1.2 Composition viewpoint

Going deeper in the analysis of our system composition, we are able to identify some of the modules that will be required in order to provide the functionalities specified in the Requirement Analysis and Specification Document.

Communication Interfaces Since our system interacts with many external agents, it needs to have different *Communication Interfaces* in order to communicate with them.

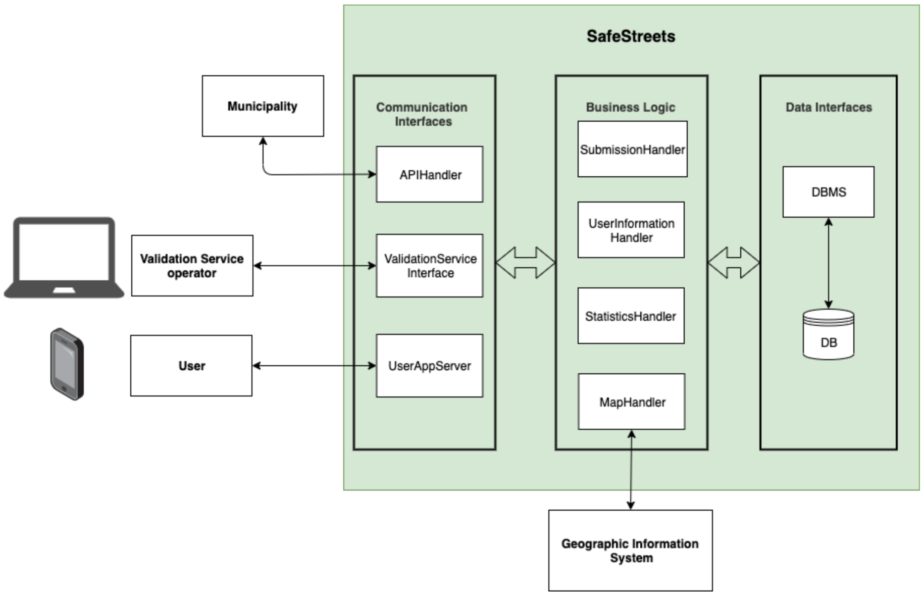

Figure 3: Composition viewpoint

- An API is needed to ensure that the *Municipality System* will be able to share information with our system
- A software module is needed in order to support the mobile app that will be offered to the users in order to exploit the functionalities of the system
- A software module is needed to provide the *Customer Care* the functionalities it needs

Business Logic The actual application logic of our system needs to manage the users information, reports and violations information; for each of these purposes several software modules are necessary; they will use communication interfaces to communicate with the agents and they will be able to retrieve data from the data interfaces.

Data Interface Our system needs a way to access and store the data it produces or retrieves from external resources, that is why *Data Interface* modules are needed. These modules allows interaction between the *Business Logic* modules and the System Databases; moreover they provide an interface to communicate with the GIS in order to allow the *Business Logic* modules to access its functionalities.

2.2 Component view

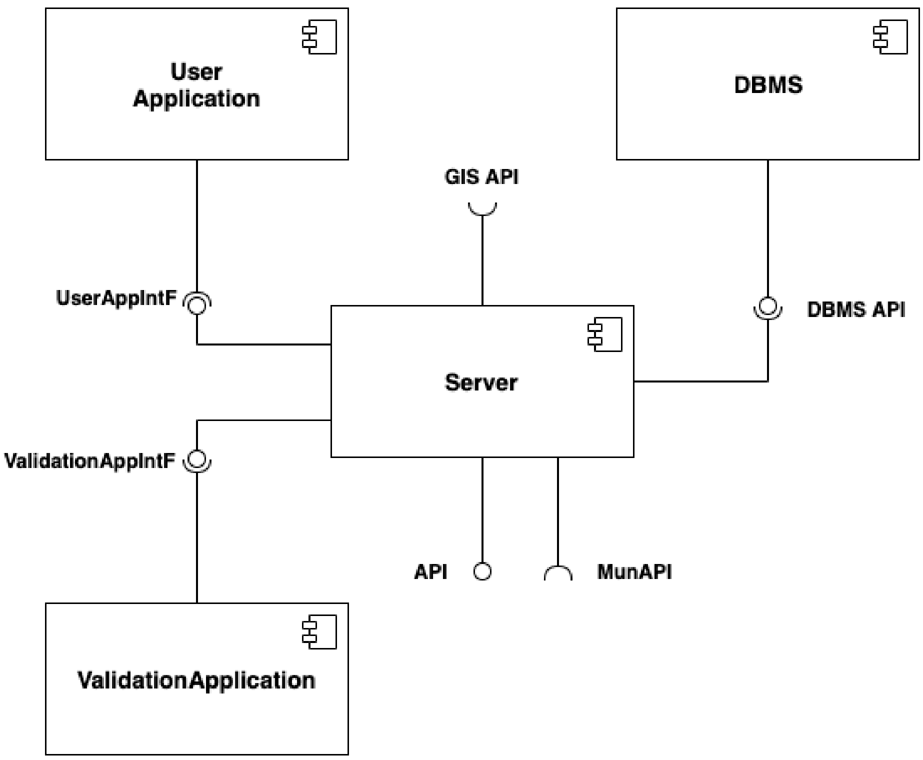

Figure 4: High-level components

Considering all the previous graphs, we have identified the following high level components and interfaces, shown in Figure 4, implementing the functionalities defined in the RASD [3]:

- **User Application**
  - Register
  - Login
  - Submit a new violation
  - View the map with user's own previous submissions
  - View the map with all the violations by street
  - View the map with safe and unsafe areas
  - Consult statistics about violations
  - View and edit personal information

- **SafeStreets Operator Application**
  - View each user profile, including personal information, submissions history and violations history
  - Mark and unmark users as banned
  - Analyse and approve (or reject) violation reports
- **DBMS**
  - Store and retrieve data
- **GIS API**
  - Retrieve a reference to an up-to-date map
  - Enrich the map with the requested violations
  - Enrich the map with graphics representing safe and unsafe areas
- **Municipality API**
  - Retrieve information about accidents
- **API**
  - Expose information about the submitted violations
  - Suggest possible solutions to avoid violations

2.2.1 DB component

ER model In Figure 5 is represented the ER model of the system's database.

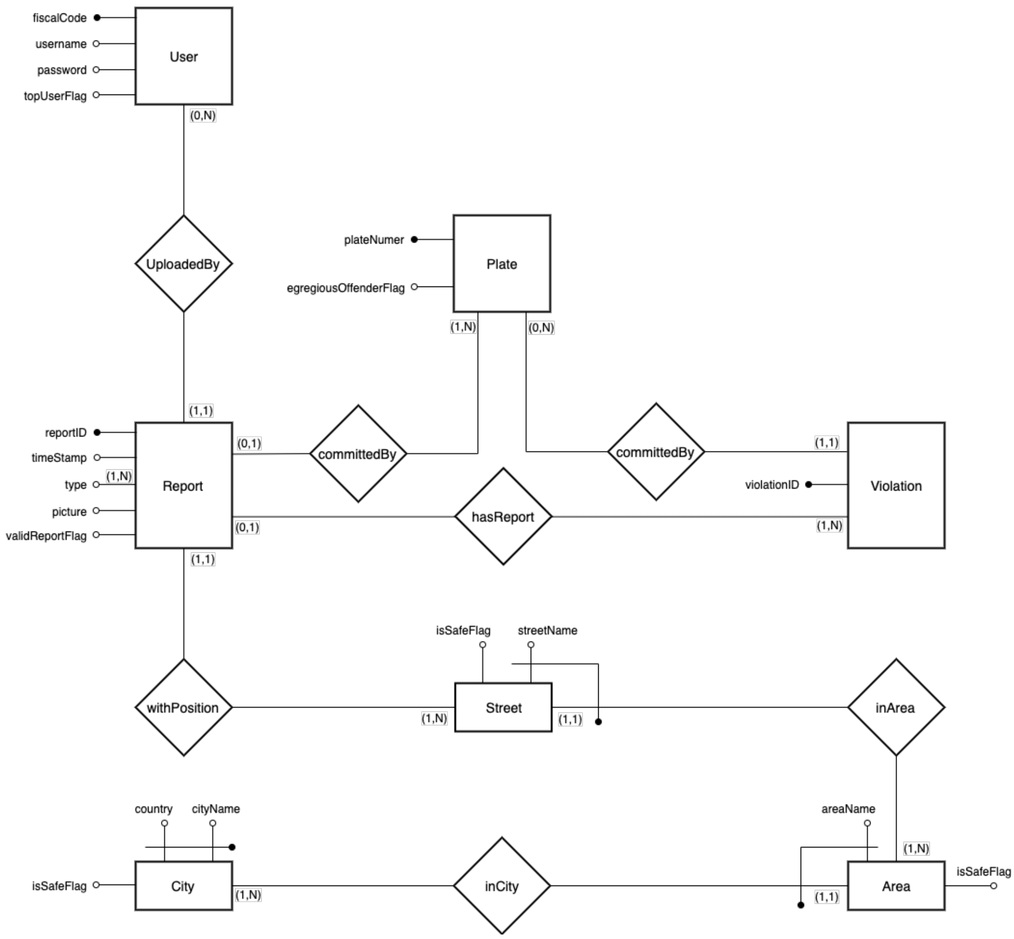

Figure 5: ER model

Users Beyond the primary key, the *username* of each registered user must be unique.

Reports Reports are identified by a unique *reportID*, but also the couple *(uploadedBy,timeStamp)* must be unique. Every report can have one or more types; the committedBy Plate represents the manual inserted plate number, so it's an optional value. The *hasViolation* field can be empty until the report is validated and approved, so it's used in pair with the *validReportFlag*, which is also optional in order to distinguish between validated and to be validated reports.

Violations Beyond the primary key, the *hasReport* attribute must be unique. The *committedBy* field is set to the recognised and validated Plate.

Plates Every plate is identified by its number; the *egregiousOffenderFlag* is used to indicate whether the owner of the vehicle having that plate often commits violations.

Streets Every street is identified by the couple *(streetName,area)* which must be unique. The *isSafeFlag* represents whether there are enough violations that took place in that street in order to mark it as unsafe.

Areas An area is identified by the couple *(areaName,city)* which must be unique. The *isSafeFlag* represents whether there are enough violations that took place in that area in order to mark it as unsafe.

Cities A city is identified by the couple *(cityName,country)* which must be unique. The *isSafeFlag* represents whether there are enough violations that took place in that city in order to mark it as unsafe.

2.2.2 Server component

In this section, we illustrate the *Server* structure and its main components and interactions, in order to explain how interfaces, communication with external components and system functionalities are performed and managed.

The white box representation (Figure 6) shows the parts composing the *Server* component and their interactions by means of lollipop-socket notation. When designing this component's internal structure, several concerns and requirements were taken into account:

- All of its required and provided interfaces had to be delegated to some part of its internal structure
- All of the functionalities related to violation submission and report validation had to be addressed and provided by some parts of this component
- Interface specific parts had to be designed in order to communicate in different ways with different external components
- Associations between internal components needed to be clarified

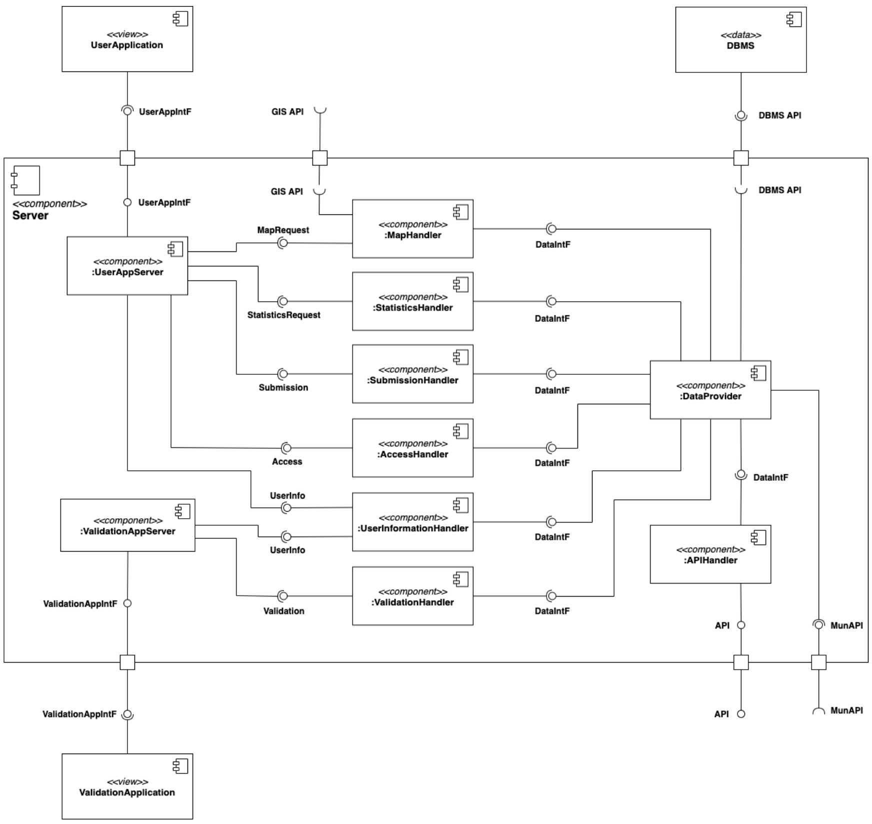

Figure 6: Server component

UserAppServer This component acts as an interface between the user and the application logic of the system: it receives users' violation submission requests or information consultation requests and routes them to the specific handler. This component solves the concerns related to the handling of the requests sent by the users without work loading the rest of the application logic. It also allows the decoupling of the application logic from the presentation logic, which has to be realised in a four tier client-server architecture.

ValidationAppServer This component provides the *ValidationApplication* client with the information that the operator needs in order to validate the reports submitted by the users and it finally registers the approval or rejection in the *DBMS*. This component was decoupled from the *UserAppServer* component for its structure, but it offers different functionalities and it is only used and accessible by a specific operator via the client application.

MapHandler This component processes the map visualisation requests performed by the users. It is supported by an external *GIS Service* to retrieve and enrich the map with the information requested. Each of its modules is in charge of retrieve the information to visualise and contact the map service, providing the necessary data. This approach allows the decoupling of the presentation logic from the application logic.

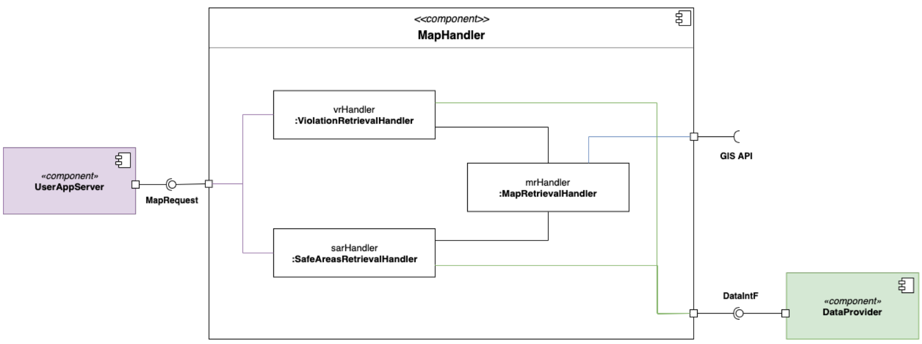

Figure 7: *MapHandler* object diagram

StatisticsHandler This component processes the statistics requests performed by the users, retrieving data from the *DBMS*, analysing it and turning it into the requested information, which can regard a general or more specific scope. Having a component for this specific purpose enhances decoupling in the system and provides a security layer for the integrity of the data.

SubmissionHandler This component processes the violation data submitted by the users and the *ImageAnalyser* module is involved to perform the recognition of the license plate in the picture. Finally, the *PlateValidator* module checks whether the recognised plate and possibly the manually inserted plate match and are valid. This data is stored as a report and later validated by the operator to eventually turn it into a violation.

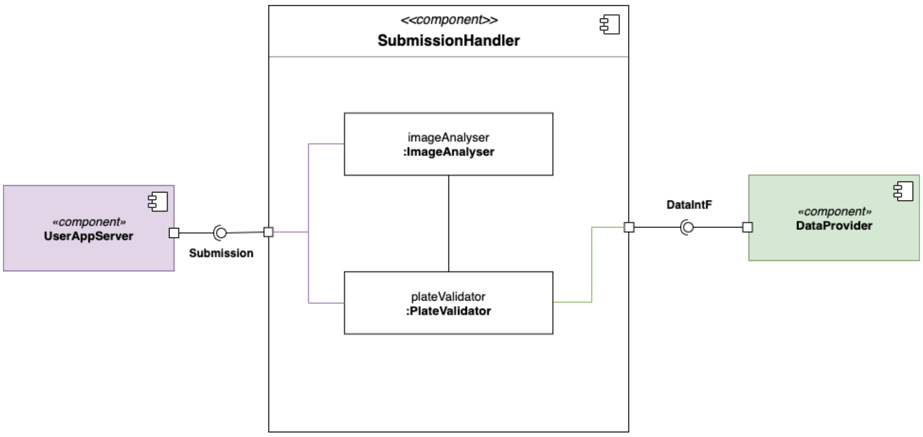

Figure 8: *SubmissionHandler* object diagram

AccessHandler This component manages the registration and login phase of the users, encapsulating the logic required to verify the credentials or register new user's information in the *DBMS*. Having a component for this specific purpose enhances decoupling in the system and ensures that these functionalities will not be generate any changes in other components of the system.

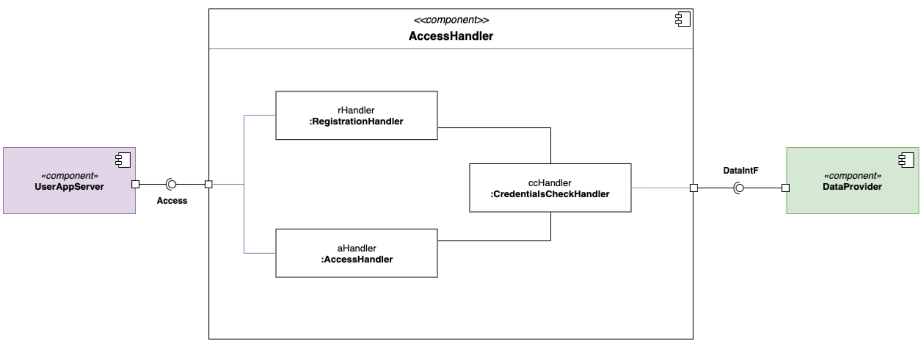

Figure 9: *AccessHandler* object diagram

UserInformationHandler This component manages the user information present in the system and provides *UserAppServer* and *ValidationAppServer* the data accessible by the user, given his roles' permissions. The main purpose of this component is to secure users' data, avoiding unauthorised users to consult sensible information.

ValidationHandler This component is used to provide the reports stored in the *DBMS* that still have to be validated to the *ValidationApplication*, in order to allow the *ValidationOperator* to evaluate the submission and decide to approve or reject it. The decision is then stored and the violations information are updated. This component is required to avoid possible malicious or involuntary wrong usages of the system by the customers, trying to submit an unreal violation.

DataProvider This component is in charge of providing a unified interface to access the different data sources of the System, both the *DBMS* and the files contained in the File System of the OS hosting the application logic. This component allows a better decoupling between the application logic components and the underlying data layer.

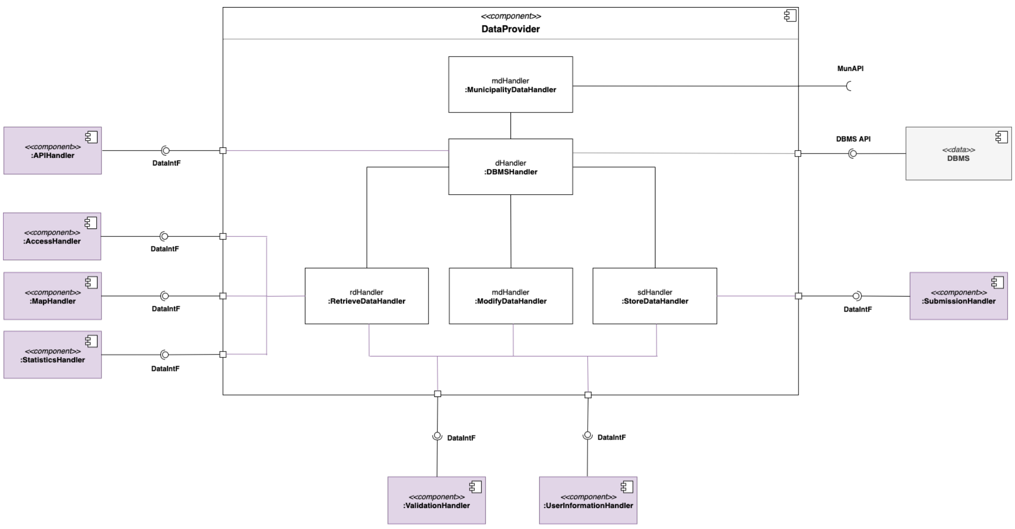

Figure 10: *Data Provider* object diagram

APIHandler This component is used to manage the authorities requests of accessing informations about submitted violations. The *SecurityChecker* module guarantees that only authorised municipalities is allowed to retrieve that data. Having a component for this specific purpose enhances decoupling between the presentation layer and the data layer, ensuring data reservation and protection. Figure 11 shows the internal structure of this component.

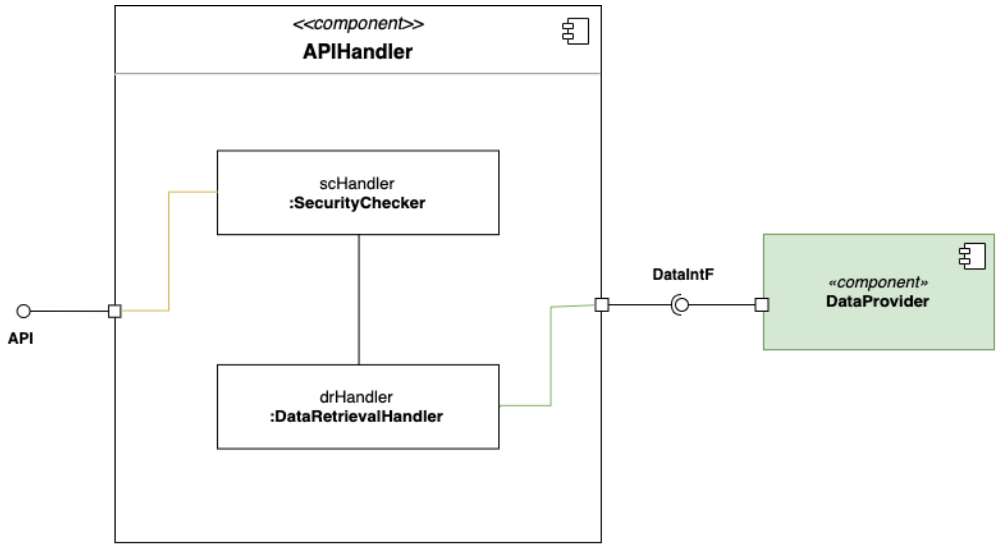

Figure 11: *APIHandler* object diagram

2.3 Deployment view

2.3.1 Four tier architecture

Taking into account that:

- the *Composition viewpoint* diagram shows the need of database decoupling from the actual system
- in the *Server component view* we can clearly distinguish modules who take care of presentation and communication with the client
- in the *Server component view* we can clearly distinguish modules who take care of the specific application logic

we decided to design the system on a four tier architecture pattern.

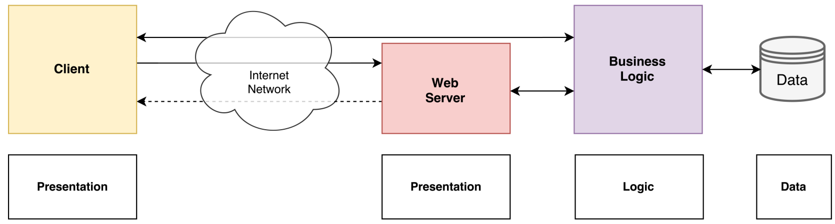

Figure 12: Four tier architecture with internet layer

Mapping of Server components on architecture: to clarify at a finer level how the Server components are mapped in the four tier layered architecture the following diagram represents the Server components highlighted in the same color of the tier in the previous diagram:

- Client: components used by users in order to access the functionalities offered by the system
  - UserApplication
  - ValidationApplication
- Web Server: components which provides interfaces to clients in order to allow them to use functionalities offered by the system
  - ValidationAppServer
- Business Logic: components which realizes the functionalities offered by the system
  - UserAppServer
  - MapHandler
  - StatisticsHandler

  - SubmissionHandler
  - AccessHandler
  - UserInformationHandler
  - ValidationHandler
  - DataProvider
  - APIHandler

- Data: components which store and manage the access to the data produced and needed by the Business Logic
  - DBMS

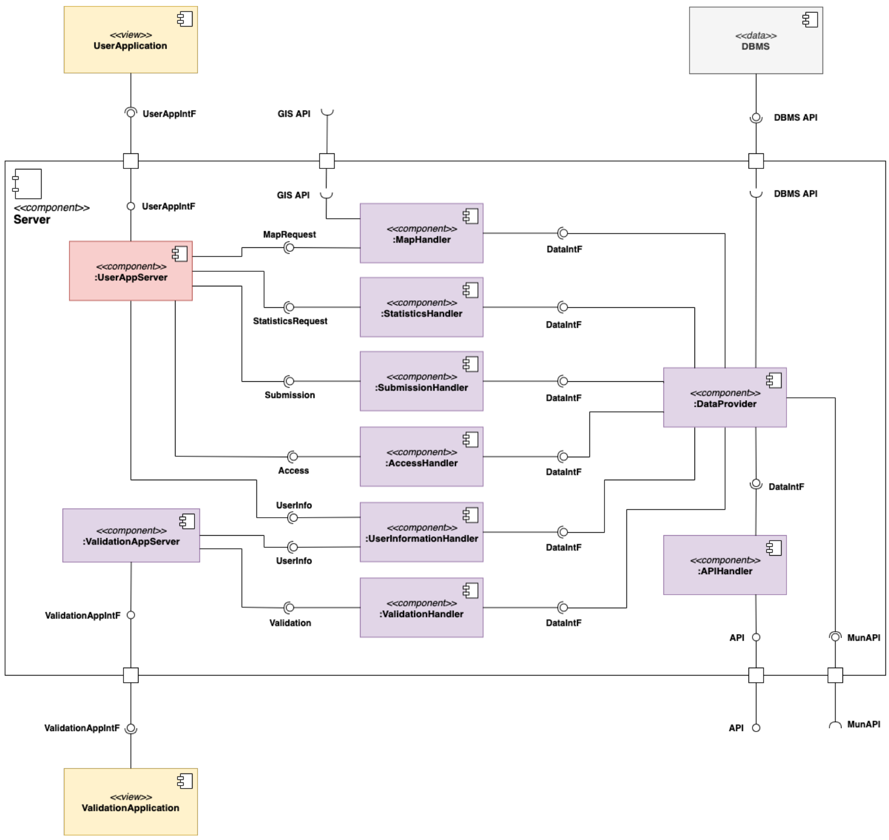

Figure 13: Mapping server component on architecture

2.3.2 Implementation choices

The following technologies were chosen for the implementation of the SafeStreets system. J2EE was the main choice we made because it offers a wide range of functionalities that makes the development of scalable multi tiered web based application much easier than with other technologies.

Web Pages: JSP was chosen over Servlets because the content of our web pages has few dynamic parts so writing small chunks of Java code in html pages is a more suitable solution

Mobile App: Java was chosen for server compatibility reasons and in order to deploy the app both on Android and on iOS

Application Logic: EJB were used to implement the Application Logic components since our application is developed using J2EE

Application Server and Web Server: GlassFish 5.1 was chosen since it offers containers both for the EJB and the JSP pages

Provided APIs: all the API provided by this system are compliant with the REST paradigm so JAX-RS was used

Data - Application Logic Communication: JPA over JDBC was chosen as a mean to enable communication between the Application Logic and the DBMS.

Web Server - Application Logic Communication: JNDI was chosen as a naming and directory service to allow the web servers to access the functionalities offered by the Application Logic

DBMS: MySQL was chosen as DBMS since it is the most popular and widespread and it has a lot of documentation and a big community of users; in combination with it we chose InnoDB which allows the usage of foreign keys

2.3.3 Decision Tree

Figure 14 represents a summary of the main design and implementation decisions that were made during the development of the architecture of the SafeStreets system.

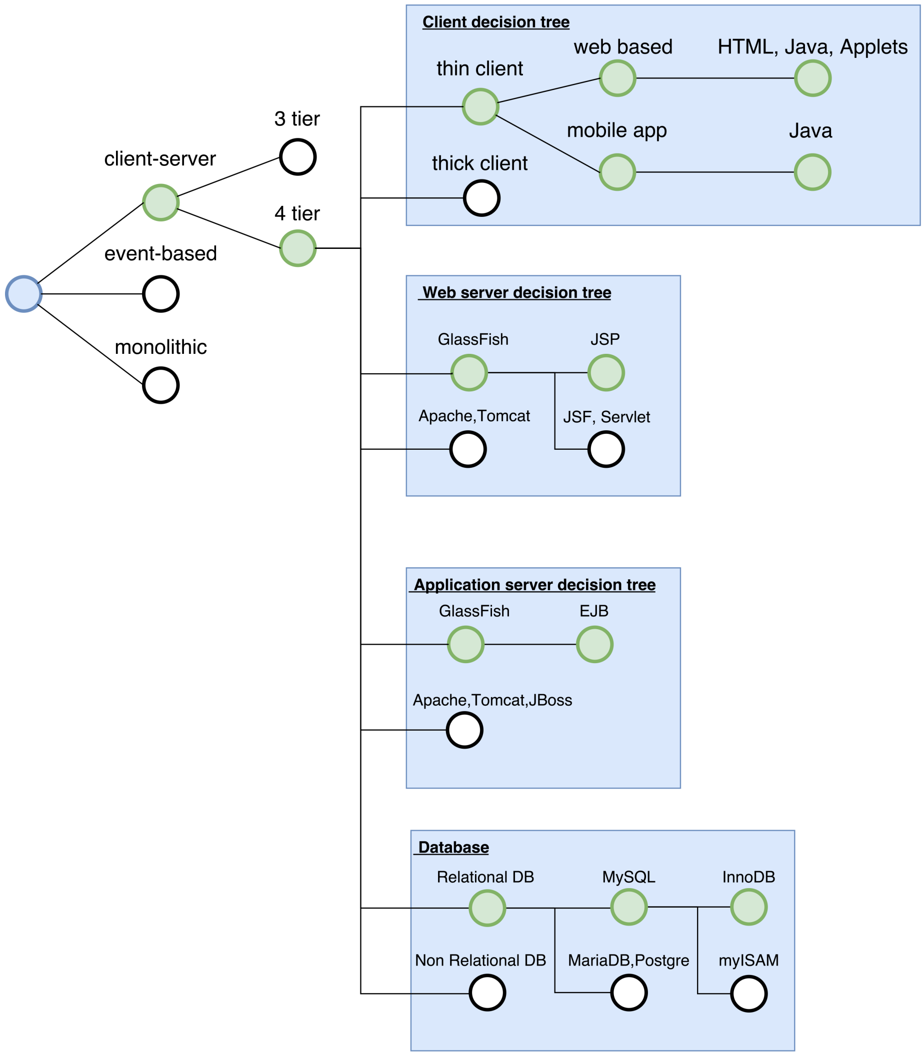

Figure 14: Decision Tree

2.3.4 Deployment diagram

The diagram in Figure 15 represents the mapping of the software components depicted in Figure 13 and Figure 4 on the devices that will run them. Many design concerns were considered while developing this solution.

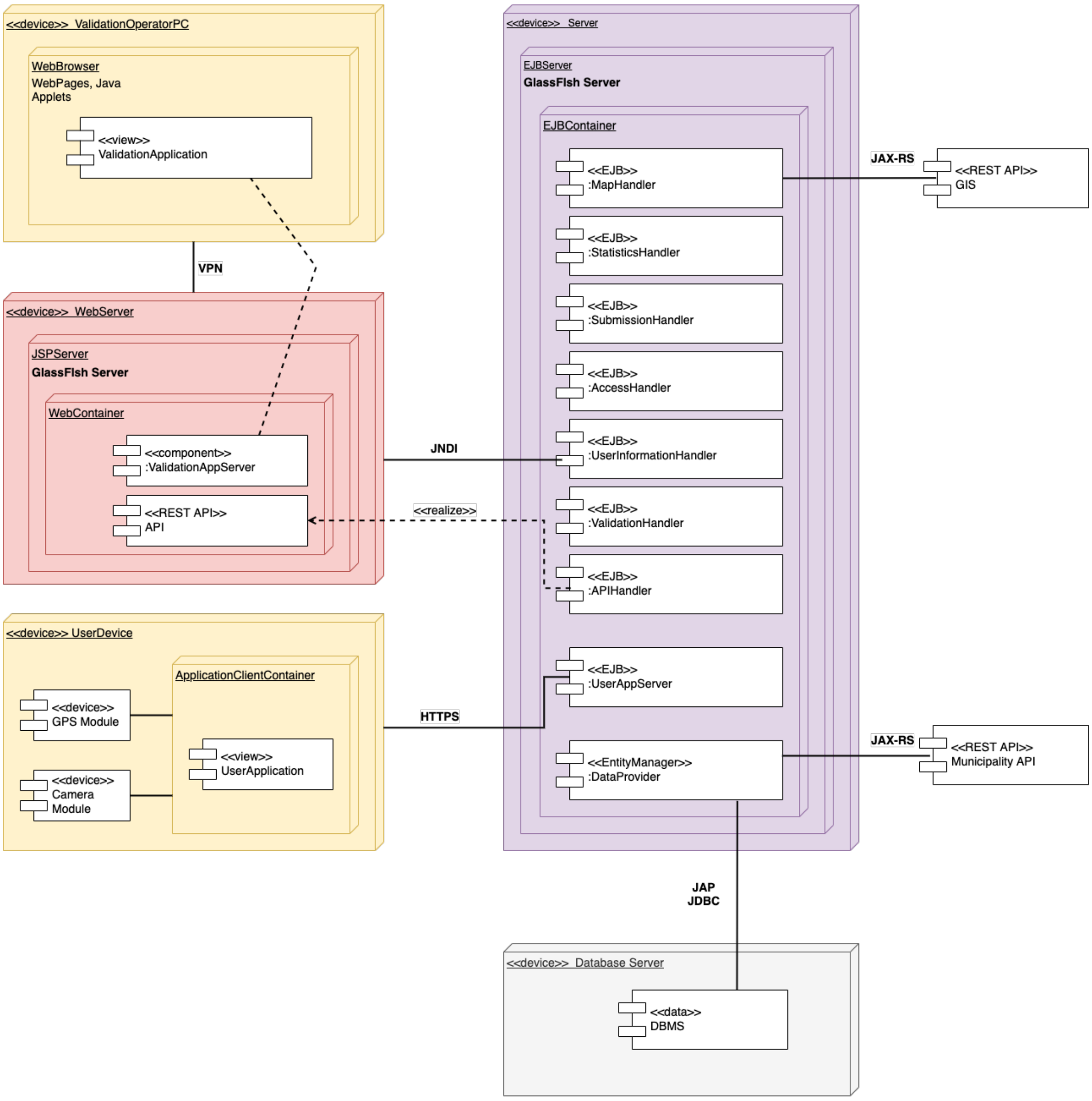

Figure 15: Deployment diagram

Security: security is ensured in different points of the architecture, in particular the *UserAppServer* uses HTTPS as communication protocol to communicate with the users; the devices running the *ValidationApplication* component are in a VPN with the device running the *ValidationAppServer*.

Scalability: this model of deployment is scalable in the sense that the system administrators will be able to add more devices and deploy more instances of the needed components when and where performance issues will arise, in order to maintain a minimum level of performance even with loads increase.

Decoupling: decoupling in this architecture is present at different levels; in the deployment diagram it is clear that the each of the four tier runs on different devices, moreover the *UserAppServer* component runs on a different device than the *ValidationAppServer*.

Redundancy: in this iteration of the architecture no redundancy of components or devices is present, but it is allowed in prevision of future expansions of the system's infrastructure.

Fault Tolerance: deploying different components on different machines allows the system to be easier to recover in case of a problem on one of the machines; for an example the Web Server running the *ValidationAppServer* component goes down, it can be replaced with another machine and in the mean time the *ApplicationServer* would still be up and running and still be able to provide the *UserApplication* with its functionalities.

2.4 Runtime view

In this section we represent some runtime views of the supposed interactions between the components of the *SafeStreets* system.

Notes to read the diagram When an attribute is modified in a JPA object the changes are reflected into the *DataProviderComponent* in order to keep the database updated. The *Login Phase* is represented only in the first diagram in order to show how it will be performed but will no be repeated in every diagram in order to improve readability.

2.4.1 Upload

The sequence diagram in Figure 16 shows how the upload of a report is handled in the Server component of the system.

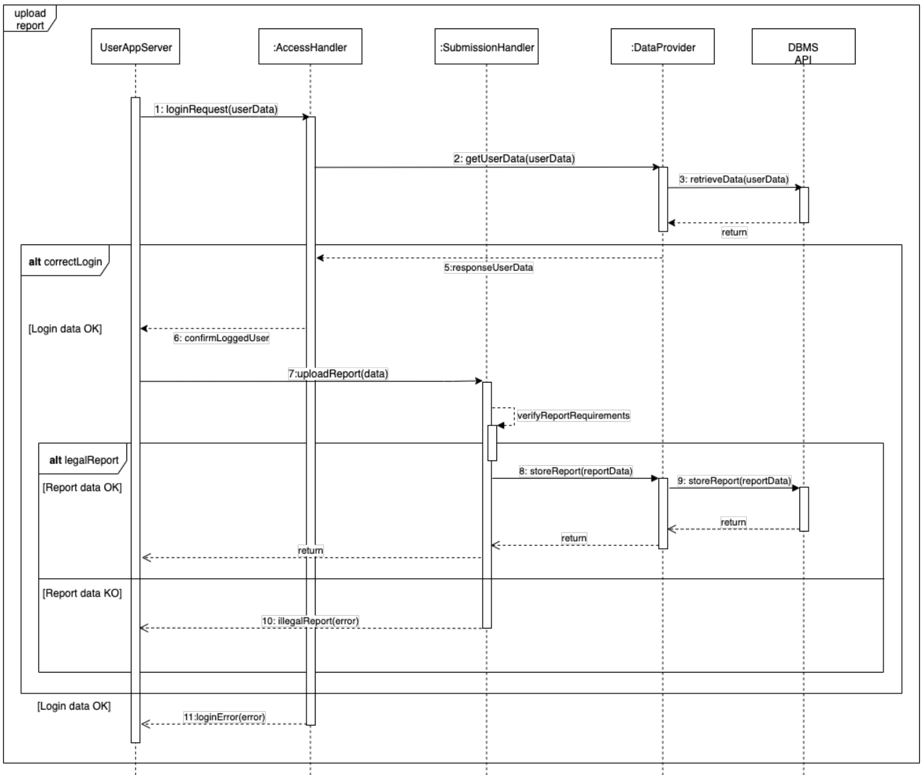

Figure 16: *Upload submission* sequence diagram

Note As already stated in the following diagrams we will assume that the user has already logged in correctly in order to focus on the purpose of the diagram.

2.4.2 Map request

The sequence diagram in Figure 17 shows the map request process.

Interactions not represented

- The *MapHandler* needs an interaction with the GIS while calling its "processUserDataForMap()" method to process latitude and longitude information needed to generate the right "userData" to query the DBMS.
- To improve redeability of the diagram we omitted the interaction between the *DataProvider* and the *Municipality API* since the first checks if the latter has new data regarding the interested area of the map requested before returning data

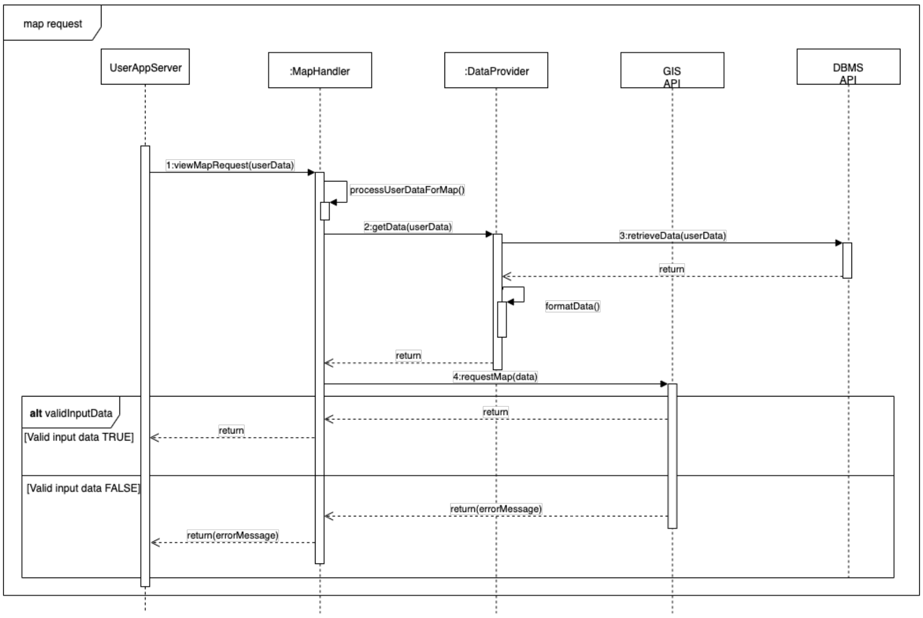

Figure 17: *Map request* sequence diagram

2.4.3 Show statistic

The sequence diagram in Figure 18 shows the interactions between the modules involved in the process of retrieving data for a certain statistic and show it to the user.

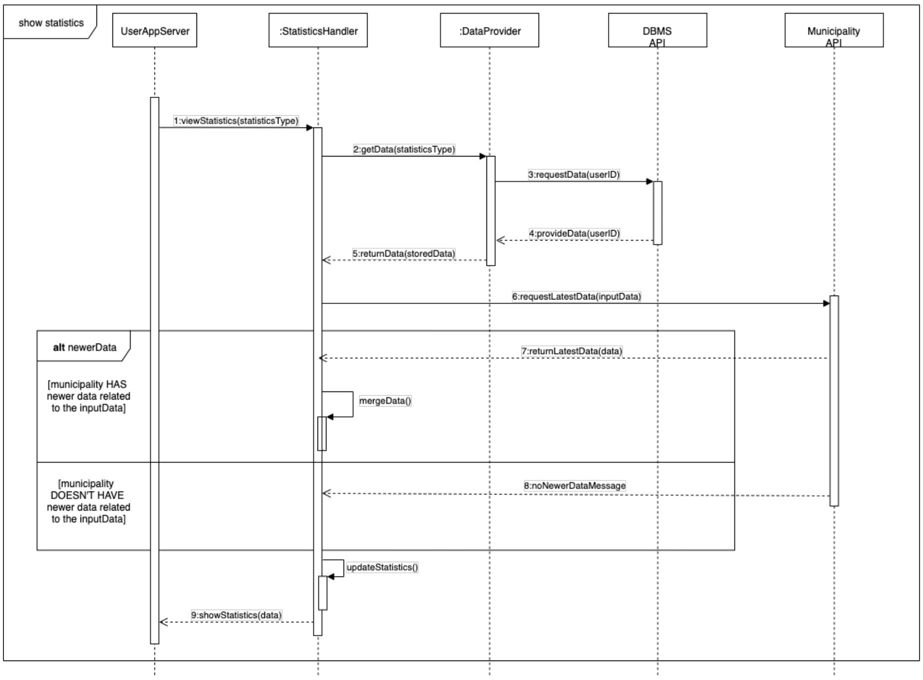

Figure 18: *Show statistic* sequence diagram

2.4.4 See user info

The sequence diagram in Figure 19 shows the interactions between the modules involved in the process of retrieving data of a certain user and show it to the *ValidationApplication*.

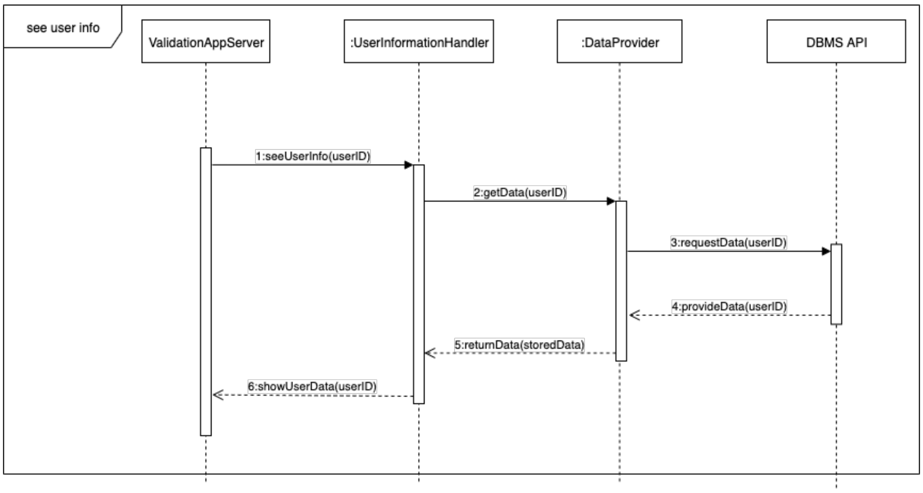

Figure 19: *See user info* sequence diagram

2.4.5 Validate reports

The sequence diagram in Figure 20 shows the interactions between the modules involved in the process of validating the submitted reports, performed by the *ValidationApplication*.

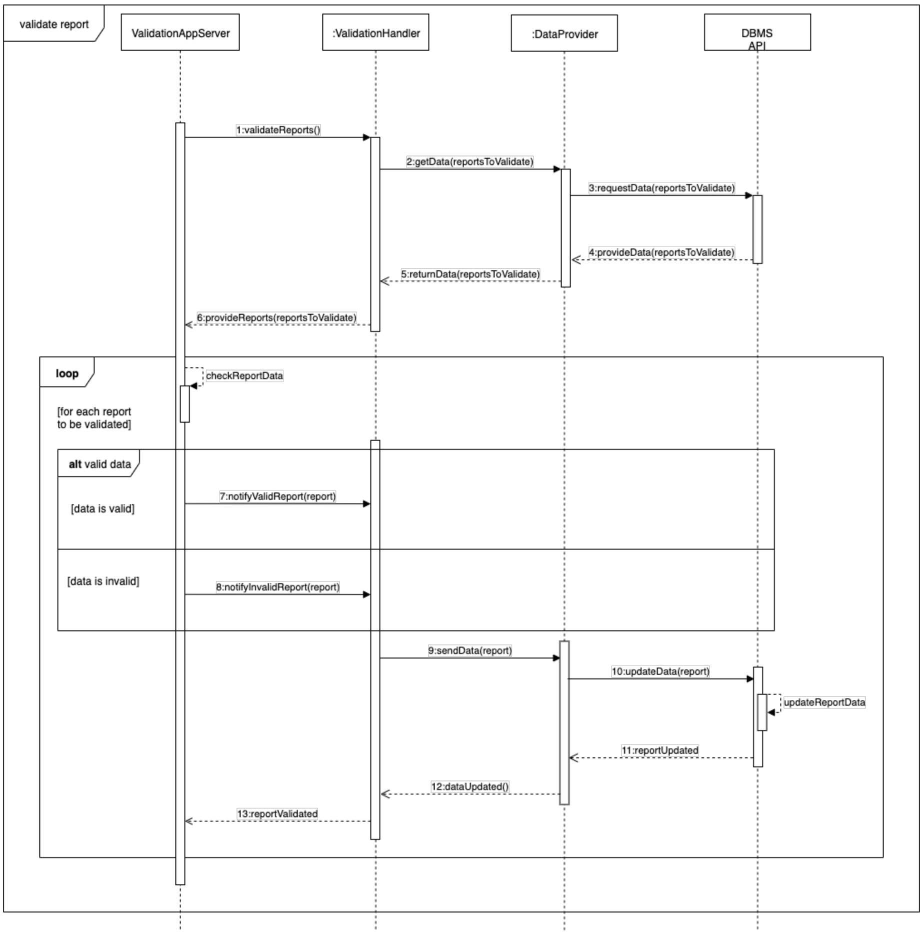

Figure 20: *Validate reports* sequence diagram

2.4.6 Send data to municipality

The sequence diagram in Figure 22 shows the interactions between the modules involved in the process of providing violations data to the municipality.

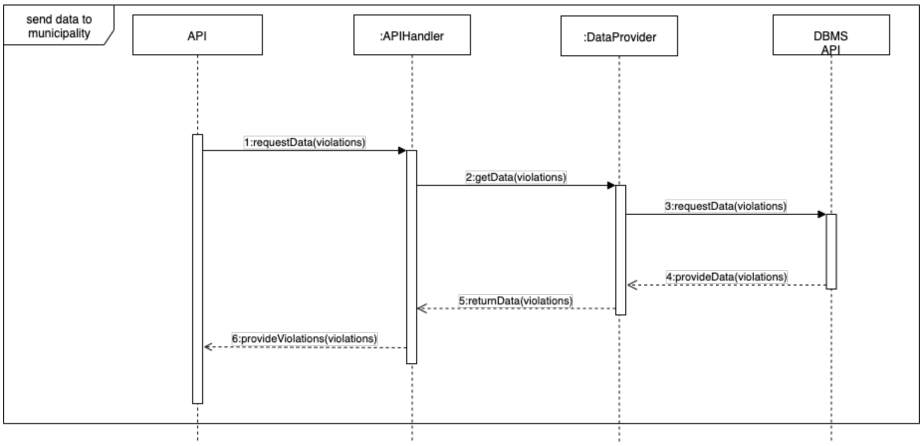

Figure 21: *Send data to municipality* sequence diagram

2.5 Component Interfaces

The component interfaces offered by the application server are represented in the following component interfaces diagram. The arrows represents the dependencies relations. All the data parameters and return values are classes that implements the JsonValue interface, in order to be optimized for serialization.

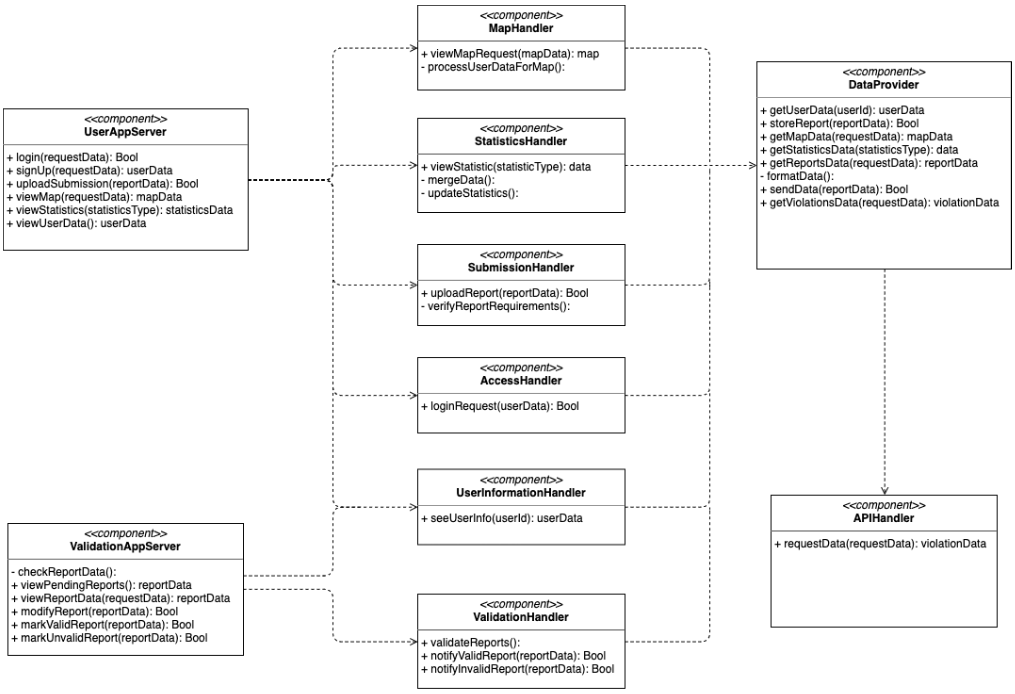

Figure 22: *Component interfaces* diagram

2.5.1 User Application interface

The User Application interface is responsible for communications between the user application and the user application server. Because of the nature of exchanged data (password and privacy-related information), the protocol to be used is HTTPS.

2.5.2 Validation Service interface

Via the Validation Service interface is responsible for communications between the validation service application and the validation application server.

All computers used by the validation service are connected in the same dedicated subnet as the validation application server and this web server is accessible only from the aforementioned subnet.

A VPN is used to achieve what mentioned above. Because of the sensitive data exchange, the protocol to be used is HTTPS

2.5.3 Data Provider interface

Trough the Data Provider API the system can retrieve and write data into the database. Note that the Data Provider component is the only one that access directly the DBMS API component.

2.5.4 API Handler interface

The API Handler Interface must be realized using the REST paradigm. The external Municipality system should query the API Handler in order to retrieve an updated list of violations and statistics.

3 User interface design

In this section are presented or referenced some mockups of the main features and related user interfaces the system is supposed to offer to the user and to the validation service through the proper web-based application.

Mockups show how the user interface is supposed to offer to the user the possibility to interact and make request to the system (obviously such user's interactions will result in a client-server communication of the user/validation service app view with the related server component based on the protocol chosen for that communication).

The main goal of our mockups design process is to build an interface that clearly distinguish functionalities offered by the system taking into account the architectural decoupling offered by the taken design choices.

3.1 User app

The user app mockups have been discussed and presented in the User Interfaces section of the *SafeStreets: Requirements Analysis and Specification Document* [3]. For further implementation details refer to the Implementation Choices section of this document.

In addition we present the UX diagram to show the flow which the user will follow to navigate inside the application, in accordance with the mockups specified in the RASD:

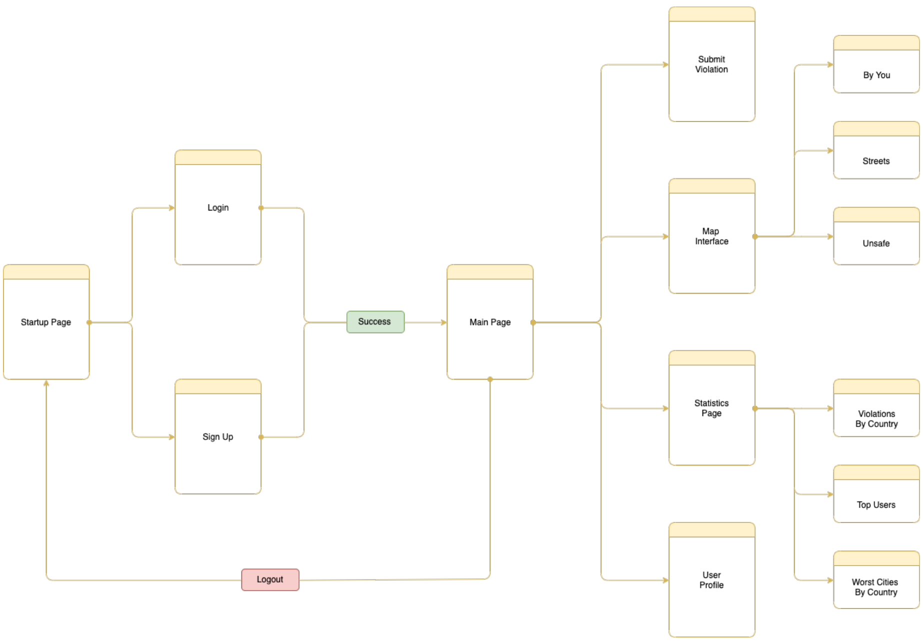

Figure 23: *UX Diagram* - User

3.2 Validation Service app

The validation service app has a simple interface in order to provide features to validation service operators in a clear and simple way. This interface must be optimised for a desktop monitor size.

The UX diagram presented below shows the flow which the validation operator will follow to navigate in the website, in order to manage the reports submitted.

In accordance to it, the web pages mockups are specified in the following page.

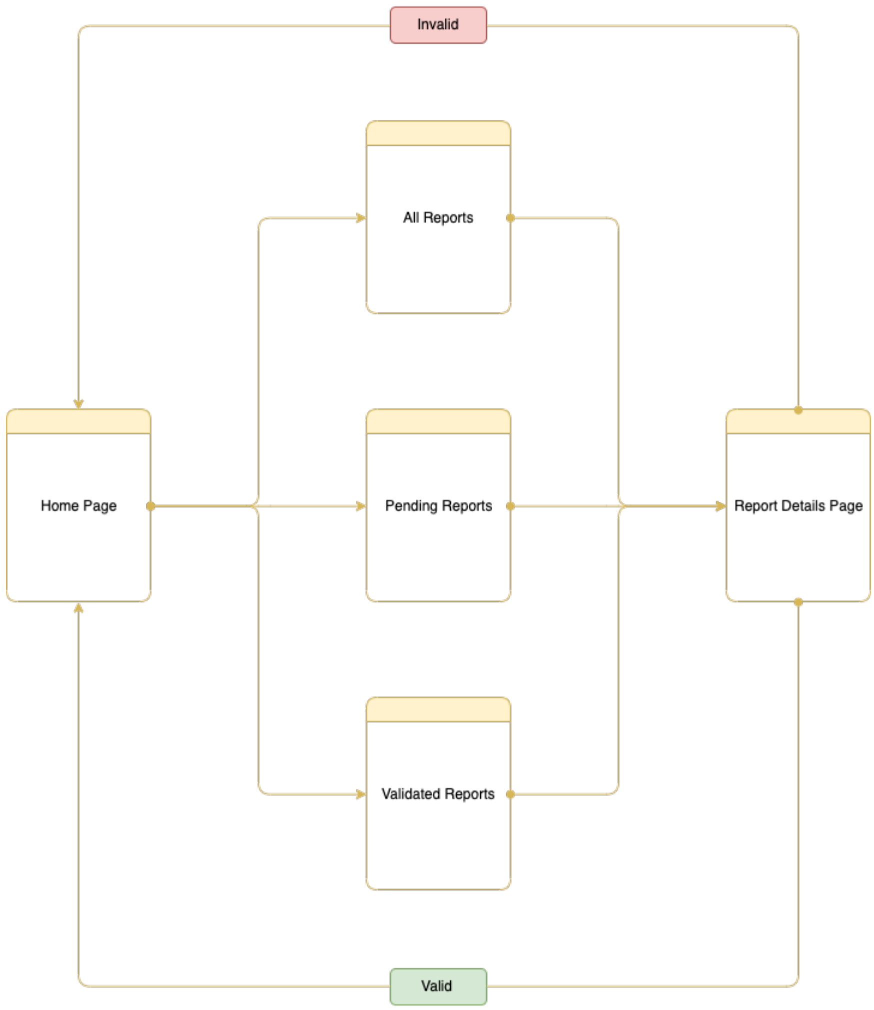

Figure 24: *UX Diagram* - Validation Service

3.2.1 Home page

All main features are accessible directly from the home page to provide a rapid and intuitive access to them.
From the home page the operator can:

- Visualise all the submitted reports
- Filter by the pending reports of possibly occurred accidents
- Filter by the already validated reports
- Delete single or multiple reports
- Select a specific report to visualise all the stored data regarding it, accessing the *report details page*

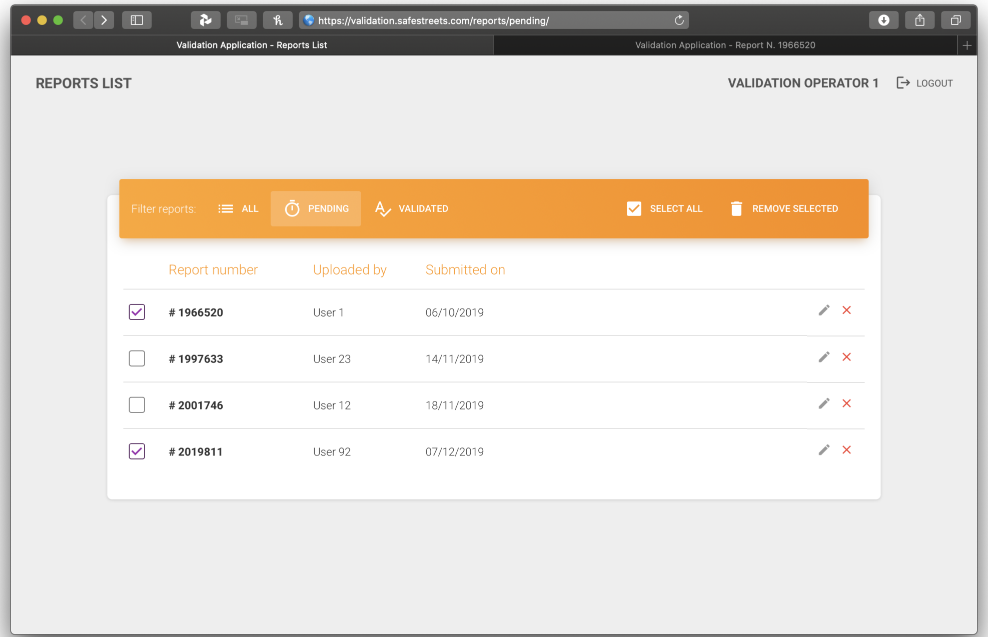

Figure 25: *Validation service home page* mockup

3.2.2 List of Reports and Violations

From the report details page, the operator can:

- Visualise all the detailed information about the selected report
- Visualise the result of the image recognition algorithms associated to the selected report
- Manually type the report license plate if not present, wrongly typed by the user, or totally not recognised
- Mark and unmark reports as valid in order to transform it into a violation

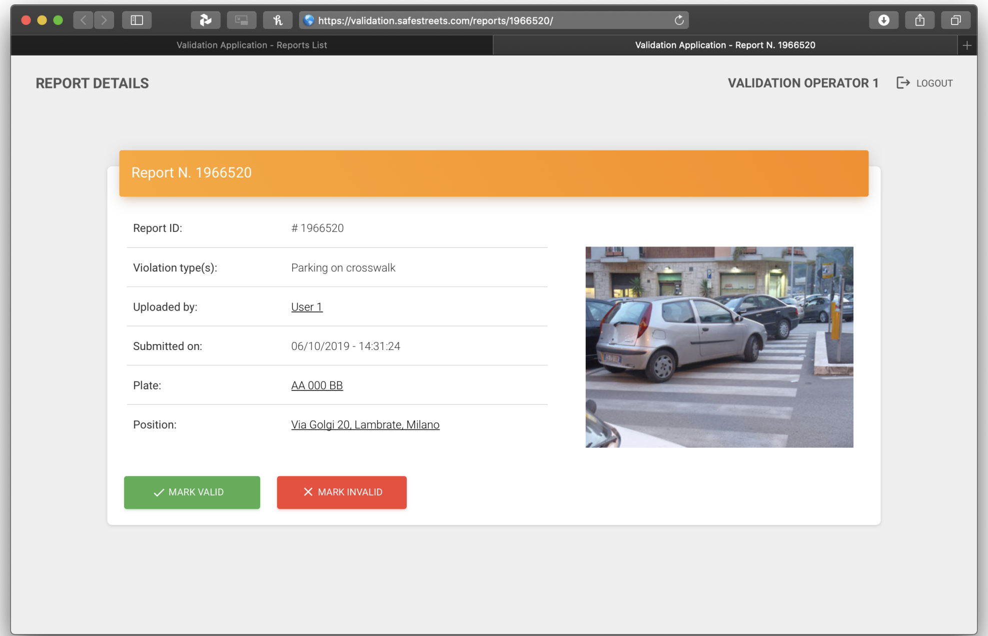

Figure 26: *Validation service reports and violations list* mockup

4 Requirements traceability

4.1 Functional requirements

In the Table 1 is presented a mapping correspondence between the requirements defined in the RASD related to each goal and the components identified in the server component diagram.

Requirements of goal	Components
G1 Allow guest users to register to the system	UserAppServer AccessHandler DataProvider
G2 Allow registered users to authenticate to the system	UserAppServer AccessHandler DataProvider
G3 Allow users to transfer data to the system describing occurred violations, including the suitable metadata to describe the submitted violation	UserAppServer SubmissionHandler DataProvider
G4 Ensure that the chain of custody of the information provided by the users is never broken, and the information is never altered or manipulated	UserAppServer SubmissionHandler DataProvider APIHandler
G5 Allow the system to retrieve data about the accidents that occur on the territory and data about issued tickets via the municipality provided service	APIHandler DataProvider
G6 Allow the system to cross the information submitted by the users and the information retrieved from the municipality to build and provide statistics	StatisticsHandler DataProvider APIHandlers
G7 Allow users to consult a map highlighting the streets (and the areas) with the highest frequency of violations, the identified potentially unsafe areas and view statistics about previously stored violations	UserAppServer MapHandler DataProvider StatisticsHandler
G8 Allow municipality to consult the system data and receive suggestions on possible interventions via a restrict access API	APIHandler DataProvider
G9 Provide validation service the ability to validate user's submissions of data describing occurred violations	ValidationAppServer ValidationHandler DataProvider

Table 1: Mapping goals on components

4.2 Non functional requirements

Security Security protocols are used for transmission and storage of sensitive data. Validation Service Server is accessed over a VPN to ensure security.

Performance requirements To guarantee a short response time we have tried to decouple components in order to enable an instance pooling management of components by the EJB container and so an as much as possible concurrent management of requests. During all the design process we also have kept in mind the scalability of the software.

Availability To ensure availability requirements server will be running 24 hours for day. The architecture is designed with the purpose of enabling a redundancy architecture if availability constraints would make it necessary.

Portability Users access the system through a web-based application that enables the portability and a cross-operative system compatibility.

5 Implementation, integration, test plan

We now want to present how we will implement our system by explaining the strategy we will follow for the Server and how the features we want to offer will be implement and integrated.

5.1 Server

The Server will be implemented following a bottom-up strategy since it will be built on various modules that will cooperate in order to ensure that the goals and standards stated in the *RASD* [3] will be met.
The Server relies on the GIS and the Municipality API, which will be considered reliable and accurate services and will be tested only "server side" (i.d. checking that our system can exploit them properly).
For what it concerns the the Server, each submodule will be implemented and tested with formal methods before being integrated with the other modules that rely on it, since integrating a malfunctioning module with other can only lead to bigger problems and the effort and time needed to fix them could increase exponentially.
To better understand the decisions we made about implementation, testing and integration here are two tables that list the features we want to offer linked with the relevance we gave them considering the difficulty of the implementation and the importance for the user.

5.2 UserApplication

Feature	**Importance**	**Difficulty of implementation**
Sign up and login	Low	Low
Consult personal data	Low	Low
Upload a picture	High	Medium
Consult the map	Medium	High
See statistics	Medium	High

Table 2: Users features

- **Sign up and login:** sign up and login are features that are fundamental to identify users and allow them to properly upload reports. The modules that will support the login phase will be *AccessHandler*, that will be implemented right before the *UserInformationHandler* since their functionalities are linked. Moreover they will be tested following the "relies on" relationship. They both rely on *DataProvider* module, so their testing procedure will be subjected to the one of the *DataProvider* and the same stands for the integration procedure

- **Consult personal data:** the module *UserInformationHandler* is the one responsible of making this feature available. It will be integrated with the

DataProvider, and will be implemtented after the *AccessHandler* since a user needs to sign up to the system before accessing his information

- **Upload a picture:** this feature is the core of the whole system, and several modules are involved in its realization. The *SubmissionHandler* is responsible of coordinating all the operations and method calls necessary to perform the action. It will call the *DataProvider* and will interact with the *GIS*, for these reasons it will be integrated with them.
- **Consult the map:** this feature will concerne the *DataProvider*, the *MapHandler* and the *GIS API*. The first one will be responsible of retrieving the proper information from the database, while the *MapHandler* will format the data sent from the *userAppServer* in order to forward it to the *GIS API* to allow it to offer the rendered map
- **See statistics:** the *StatisticsHandler* will trigger the *DataProvider* to retrieve updated data from the *Municipality API* and the *DBMS* with the objective of updating all the possible statistics that the system will offer to the user and the Municipality

5.3 Municipality

Feature	Importance	Difficulty of implementation
Authenticate	Low	Low
Access the API	High	Low
Retrieve violations' data	High	Medium
Retrieve statistics	Medium	Medium
Retrieve tickets's data	High	High
Share data	Medium	Medium

Table 3: Municipality features

All these features will be made available through the *API*. Since it will be of fundamental importance for the system's interface with the Municipality it will have to be implemented on a strong and reliable backend (all the "logic" modules) and for this reason it will be implemented after the whole Server will be fully tested.

5.4 Integration

In this section we want to show how we want to integrate modules with one another. The tail of the arrow starts from the module that uses the one pointed by the head.
Even with the most rigorous testing procedure the integration can generate unpredictable side effects, so right after two modules are integrated they are tested again, and so on until the whole Server of the system will be fully implemented.

5.4.1 Integration of the internal components of the Server

For better readability we separated the pictures of the integration of the *DataProvider* with the other modules of the backend.

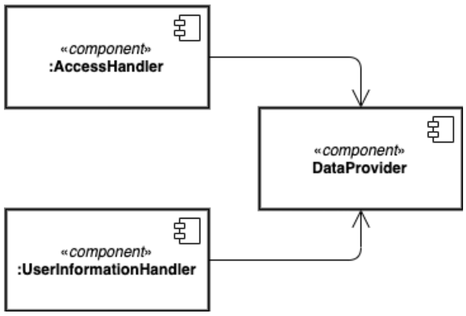

Figure 27: User data integration

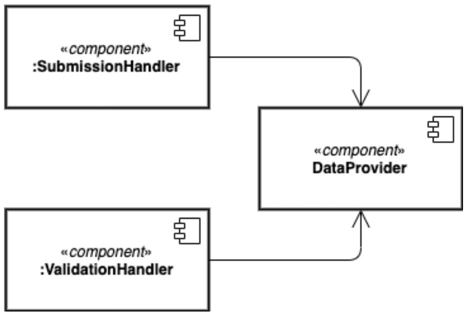

Figure 28: Submissions and violations data integration

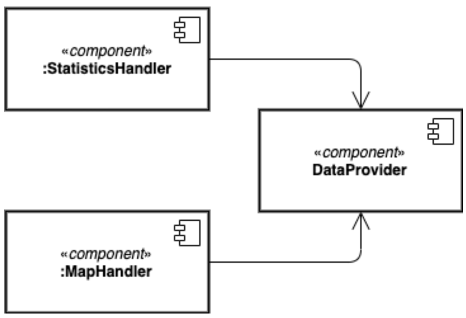

Figure 29: Statistics and map data integration

5.4.2 Integration of the frontend with the backend

The *UserApplication* and the *CustomerCareApplication* are considered to be frontend modules since they are the connection with the external world. These

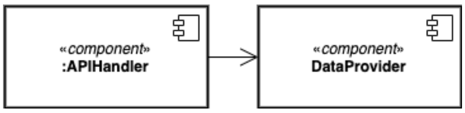

Figure 30: Municipality API data integration

modules will be the last to be integrated because, as already stated, we want to be sure to expose well implemented interfaces before doing so.

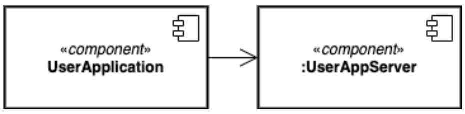

Figure 31: User app integration

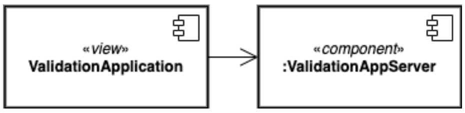

Figure 32: Validation Service integration

5.4.3 Integration with the external services

The *GIS* and the *Municipality* are the external services with which our sistem interfaces, and will be integrated at different stages of implementation.
The *GIS* will be integrated with the *:MapHandler* component since the latter relies on it, while the *Municipality API* will be integrated as soon as the backend will be properly implemented.

Appendices

A Software and tools used

For the development of this document we used

- LaTeX as document preparation system
- GitHub as version control system
- Draw.io for graphs
- Balsamiq Mockups for user interface mockups

B Hours of work

This is the amount of time spent to redact this document:

- Mattia Calabrese: ~ 24
- Federico Capaccio: ~ 29
- Amedeo Cavallo: ~ 24

C Changelog

- **v1.0** December 9, 2019
- **v1.1** December 15, 2019

References

[1] Bass, Clements, Kazman, *Software Architecture in Practice*, SEI Series in Software Engineering, Addison-Wesley, 2003

[2] E. Di Nitto, M. Rossi, *Software Engineering 2 Assignment*, AA 2019-2020

[3] M. Calabrese, F. Capaccio, A. Cavallo, *SafeStreets: Requirements Analysis and Specification Document*, Politecnico di Milano - Software Engineering II Project, 2019