POLITECNICO
MILANO 1863

Software Engineering 2

Design Document

Amorosini Francesco
Casali Alice
Fioravanti Tommaso

8 December 2019

Contents

Introduction

1.1 Purpose

While the RASD presented a general view of the SafeStreets application and its features, this document aims to further analyze the system's design and architecture, as well as describing for each of its components their runtime behaviour, integration, interfaces, implementation and testing plans. This document is mainly intended to be used by the test and development teams as a guidance in the development process, but also to prevent structural degradation during maintainance and extension phases. Nonetheless, the document is also addressed to all the stakeholders who are interested in supervising the development process.

1.2 Scope

SafeStreets is a mobile application that aims to improve the safety of urban areas by giving its users the possibility to report traffic violations to authorities. Users are logged in either as citizen, those who report the violations, or authorities, those who are notified about newly reported violations and are supposed to take action on them.
The system is in charge of collecting all the reports, storing them, and notifying the authorities about them. The stored reports are then used to build statistics, find unsafe areas, and compute suggestions on how to improve the safety of such areas. The system may also interact with local Municipalities' Systems in order to retrieve information about accidents and iussued traffic tickets: in this case some of the above functions are enhanced, and some new functions are enabled (g.e computing statistics on traffic tickets).

Further information about the scope of the application can be found in the Chapter 1 of the RASD.

1.3 Definitions, Acronyms and Abbreviations

1.3.1 Definitions

- *Client*: a piece of computer hardware or software that accesses a resource or a service made available by a Server.
- *Server*: a device or a computer program that provides resources or functionalities to other programs or devices.
- *Firewall*: a network security systems that monitors incoming and outcoming network traffic, applying predefined security rules.
- *Microservice*: a basic function of the application exectuted independently from the others.

1.3.2 Acronyms

- **RASD**: *Requirement Analysis and Specification Document*, the document in which all the requirements and goals of the application are throughly described.
- **API**: *Application Programming Interface*, interface, or communication protocol between Client and Server intendend to simplify the building of the Client-side software.
- **GPS**: *Global Positioning system*, technology widely used to get the user's position.
- **DBMS**: *Data Base Management system*, software that provides organized space memory to store information.
- **OCR**: *Optical Character Recognition*, software dedicated to the detection of characters contained in a document and to their transfer to digital text that can be read by a machine. In this context, OCR will be used to read license plates.
- **UML**: *Unified Modeling Language*, a standard visual modeling language intended to be used for analysis, design, and implementation of software-based systems.
- **MVC**: *Model View Controller*, a software design pattern commonly used to provide user interfaces.
- **CRUD**: *Create, Read, Update, Delete*. These are the basic four operations which allow users to interact with a resource in a relational database (`INSERT`, `SELECT`, `UPDATE`, `DELETE`).
- **UX**: *User eXperience*, a diagram used to describe the interfaces provided by the application.

1.3.3 Abbreviations

- [R_i]: i-th requirement.

1.4 Reference Documents

- Specification document: *SafeStreets Mandatory Project Assignment.pdf.*
- Requirement Analysis and Specification Document: *RASD.pdf.*

1.5 Document Structure

This document is presented as it follows:

1. **Introduction** presents a general overview, the scope and the purpose of the document.
2. **Architectural Design** shows the main components of the system and their relationships. This section will also discuss the architectural choices of the design process.
3. **User Interface Design** provides some further details to the user interface defined in the RASD.
4. **Requirement Traceability** maps all the functional requirements defined in the RASD over the components that will accomplish them.
5. **Implementation, Integration and Testing Plans** shows the order in which the implementation and the integration of the components will occur, and how the testing phase will be carried out.
6. **Effort spent** displays the time spent writing this document by each member of the team.

Architectural Design

2.1 Overview

In this chapter the architectural structure of the system will be discussed at multiple levels of abstraction. A high-level view of the components and their interactions is represented in Figure 2.1. The details will be explained in the following sections.

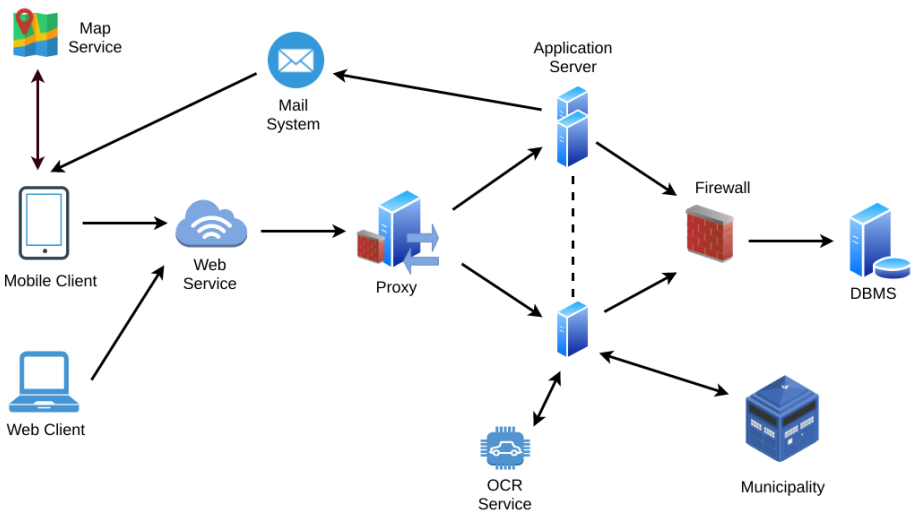

Figure 2.1: High-level overview of the system.

2.2 Component view

The UML component diagram in Figure 2.2 aims at capturing the internal modular structure of the components, showing how they are connected together in order to form larger components. Components are wired together by using an assembly connector to connect the required interface of one component with the provided interface of another component.

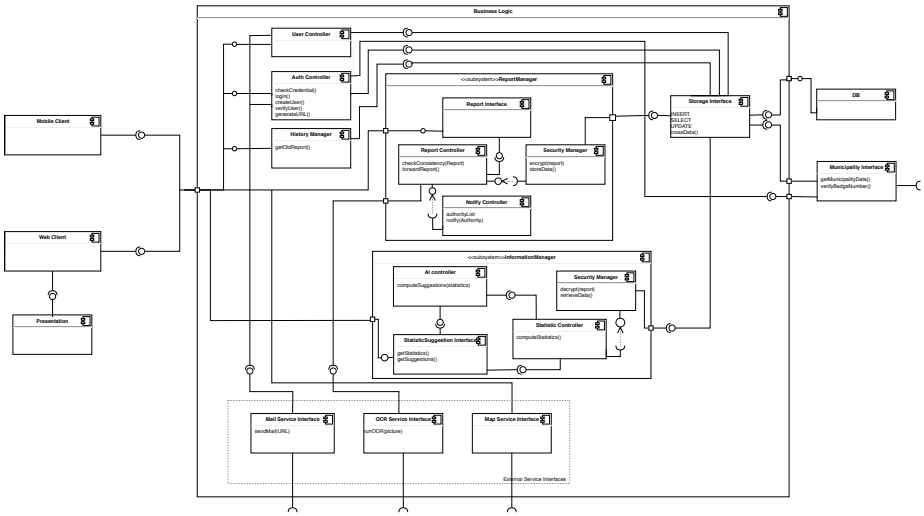

Figure 2.2: Component diagram of the system.

- **Mobile Client** and **Web Client**: the client device that accesses to the functionalities of the entire system. It will be implemented using the thin clien approach.
- **Presentation**: a Web Client, accessing through the browser, needs this component in order to display the web pages of the application. This layer only provides the structure of the user interface without accessing any data or application logic.
- **User Controller**: it takes care of all the operations related to the user's data. It exposes methods to change account credentials and it stores the user's data in the database using the Storage Interface.

- **Auth Controller**: it takes care of all the authentication-related operations: it is responsible both for the login and the registration process, and it also uses the *Storage Interface* to retrieve and store new users' credentials in the databse. During the registration process, the Auth Controller uses the *Mail Service Interface* to communicate with the Mailing Service in order to send registration emails.

- **History Manager**: its functionality is to retrieve and show to the Client his/her report's history.

- **Report Manager**: it's a *subsystem* component in charge of all the processing functions on the newly recieved reports:

  - **Report Interface**: it exposes to the User an interface in the form of a blank report to be compiled.
  - **Report Controller**: it aims to check the validity of the sent reports. To do this, it communicates with the Report Interface to check them as soon as they are received.
  - **Notify Controller**: the role of this component is to select the correct authorities to notify based on their municipality, using his *AuthorityList*. It depends on the Report Controller, in the sense that it will notify the authorities only if the check of the report is successful and the type of the report is *Traffic Violation*.
  - **Security Manager**: it takes care of the security side: as specified in the Security section of the RASD, before being stored, the report is encrypted so that no one can access it besides some specific system components. This component also depends on the Report Controller verification process, due to the fact that if the report doesn't pass it, it will be discarded and therefore never encrypted.

- **Storage Interface**: its main purpose is to retrive the reports from the database, decoupling the other components of the system from the specific data sources. This component is also in charge of authenticating the system with local municipalities' systems, and eventually crossing their data with SafeStreets' own information. Also, since the most recent version of the statistics and the suggestions are computed whithin the AI Controller and the Statistics Controller, this component is also in charge of keeping the stored version of the data up to date, eventually resolving data conflicts due to the replication of the mentioned components.

- **DB**: this component represents the DBMS, which provides an interface to read and store data. User credentials and submitted reports are stored in the database.

- **Information Manager**: a *subsystem* in charge of building statistics, compute safety suggestions and deploy them to the users who requested them.

  - **StatisticsSuggestions Interface**: it exposes to the users the latest statistics and safety suggestions. As aready mentioned in the RASD, the exposed data may be different according to the level of visibility granted to different users.
  - **Statistics Controller**: it analyzes the stored reports to build statistics such as unsafe urban areas, most frequent violations, and offenders. Depending on the agreements with local municipalities, this component can also compute statistics on iussued traffic tickets or refine its statistics about accidents.
  - **AI Controller**: its role is to analyze previously built statistics and compute safety suggestions. These suggestions are addressed to the authorities who use the SafeStreets application.
  - **Security Manager**: similarly to the one described in the Report Manager subsystem, this component is in charge of decrypting stored reports so that they can be analyzed. To ensure that no report is ever altered, this component can perform read-only operations, whereas the one in the Report Manager is the ony one which disposes of write permissions.

- **Municipality Interface**: this component represents the point of access to the municipality's system. Its main tasks are to interact with the Auth Controller to confirm the validity of the badge number provided by users, to grant or refuse the access of the Storage Interface to the municipality's database, and eventually to mediate the retrieving data process.

- **External Services Interfaces**: it's a set of components whose main task is to make an API call to the corresponding third party service.

  - **Mail service Interface**: it's an interface towards the mailing service, which is in charge of sending confirmation mails at the end of the registration process.
  - **OCR service Interface**: it calls a service used by the Report Manager to read license plates from pictures submitted in the reports by users.
  - **Map service Interface**: it provides the users with the functionality of visualizing the statistics (and their own position) on a geographic area using an interactive map.

2.3 Deployment view

In this section it is described the deployment view of the components inside the system. The deployment diagram shows the distribution of hardware components.
The system is composed of a multitier architecture and each specific role is clarified below.

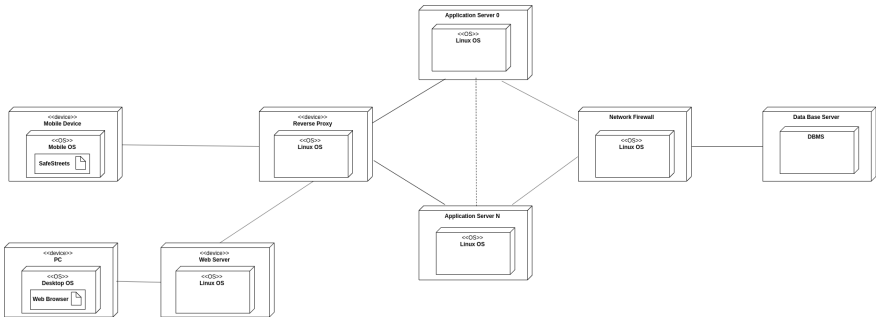

Figure 2.3: Deployment diagram of the system.

- **Client**
  The first tier contains the mobile clients which installed the SafeStreets application and the desktop clients which accessed its functionalities via web browser.

- **Web Server**
  A web server is used to store, process and deliver web pages to desktop clients. The communication between client and server takes place using the Hypertext Transfer Protocol (HTTP).

- **Reverse Proxy**
  The second tier contains a reverse proxy to implement the load balancing of the several requests towards the application servers. Furthermore, a reverse proxy works as a single point of access which improves the security and enables a caching function which can speed-up the most frequent requests. Linux machine have been chosen for safety and simplicity.

- **Application servers**
  This is the logic level of the application where all the computations happen, which corresponds to the second tier of the architecture. The servers are distributed to increase the scalability, reliability and availability of the network.

- **Firewall**
  The access to the Database is protected from a firewall to avoid unauthorized accesses to sensible data.

- **Database Server**
  This is the last layer of the architecture. All the data are stored here structured in a relational DBMS.

2.4 Runtime view

Registration Runtime View

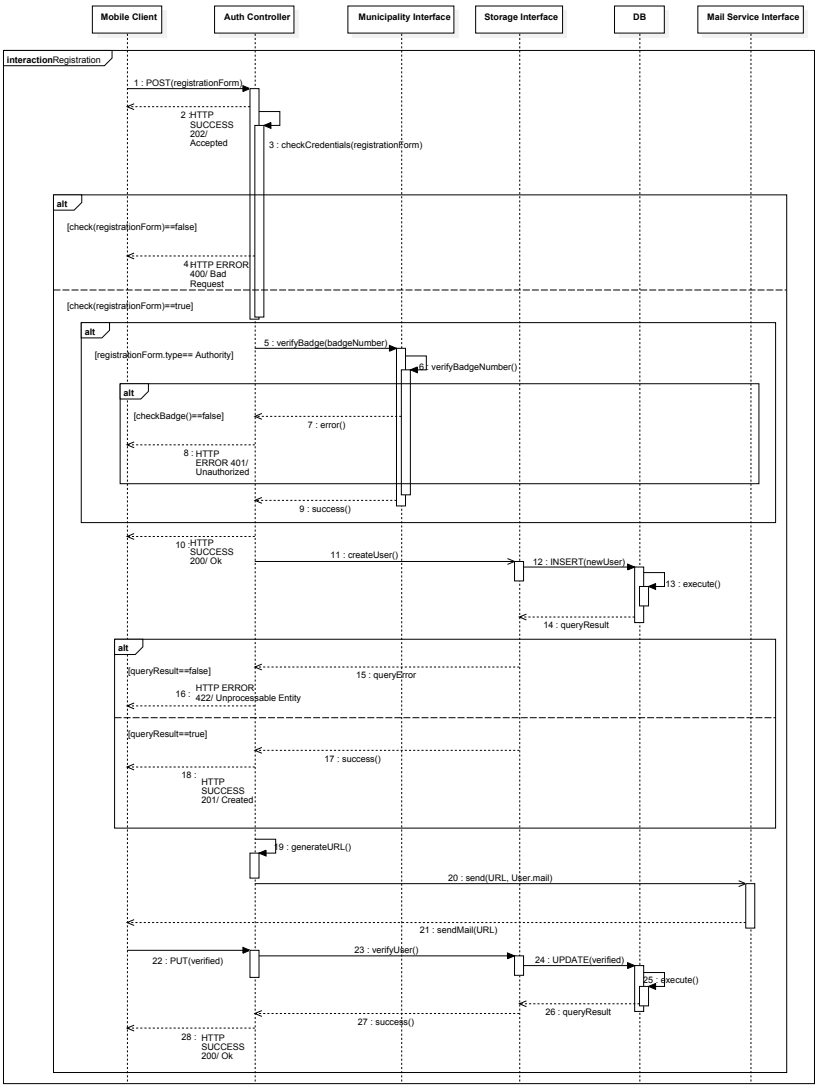

Figure 2.4: Registration Runtime View.

Every guest has to register before accessing the functionalities of the application. The guest fills a form containing with his/her authentication credentials, which is sent to the Application Server through an HTTP `POST`. The *Authentication Controller* handles the request and verifies that the entered data are valid and complete. If the form is valid, a new user is created into the database and an email containing a confirmation URL is forwarded by an external mailing system to the guest. Once the guest confirms the registration clicking on the URL link, the *Storage Interface* updates the newly created user to a *verified* status, so from this moment on the guest is able to authenticate and to use the functionalities of the application. In case the guest is an authority, once the authority has filled and sent the form, the *Authentication Manager* contacts the *Municipality Interface* to ensure that the entered badge number corresponds to an authority working at the provided municipality. If the Municipality Interface replies that the badge number is correct, the registering procedure is the same as before; otherwise an HTTP error code is sent back to the guest.

Login Runtime View

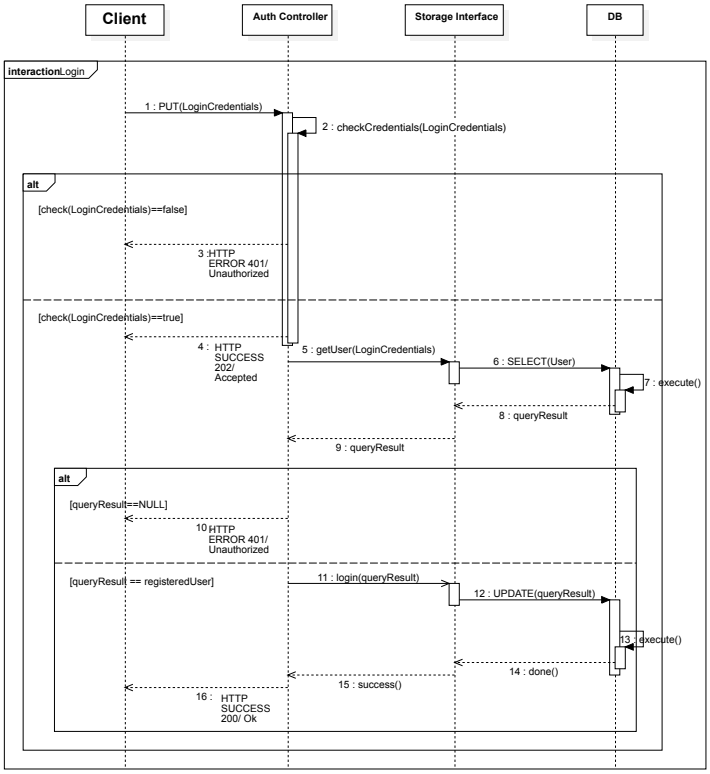

Figure 2.5: Login Runtime View.

Once a guest has undergone the registration procedure, he/she can log in to access the application's functionalities. To do that, he/she fills the login form with the credentials provided during the registration phase. The entered data are sent to the Application Server through an HTTP `PUT`. The *Authentication Controller* checks if the data provided are valid: it asks the *Storage Interface* to run a `SELECT` query on the database to verify that the data provided belong to a registered user (this action may involve an HTTP `GET`). If the query returns a non-null result, then the *Authentication Controller* requests to `UPDATE` the guest's status to *logged* and it returns a success message, otherwise an HTTP error is sent back to the guest which has not been able to log in successfully.

Send Report Runtime View

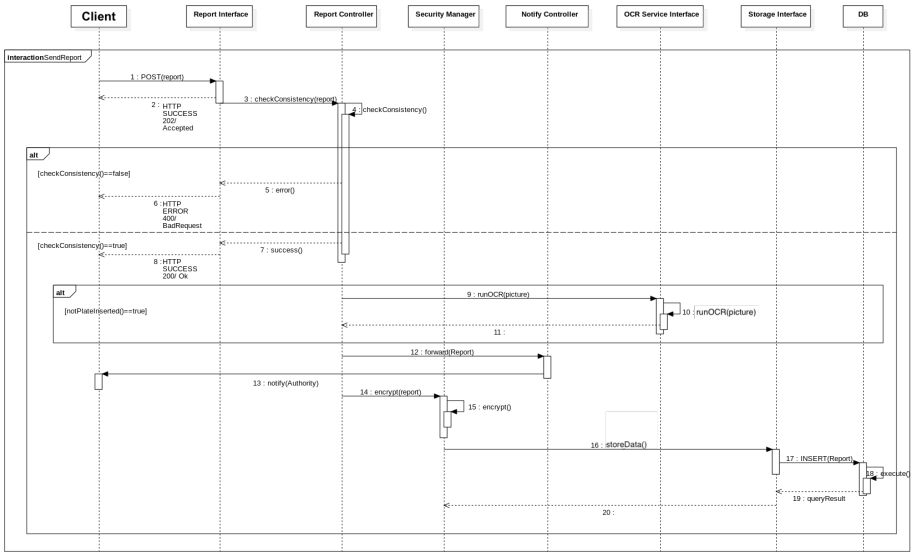

Figure 2.6: Send Report Runtime View.

A citizen is logged in and opens the menu to send a new report. The *Report Interface* provides the report form that has to be filled. Once the report is compiled and sent by the user, the *Report Interface* forwards it to the *Report Controller* in order to to check for its validity. If the citizen did not insert any license plate in the report, the *Report Controller* uses an external OCR service which executes an OCR algorithm to retrieve the license plate from the picture provided by the citizen. At this point, when all the data are avaiable, the report is shared with the *Notify Controller* which will check its authority list to choose which ones have to be notified regarding the recieved report. Finally, the *Security Manager* encrypts the report and stores it in the database using the *Storage Interface*, that runs an `INSERT` query.

Build Statistics Runtime View

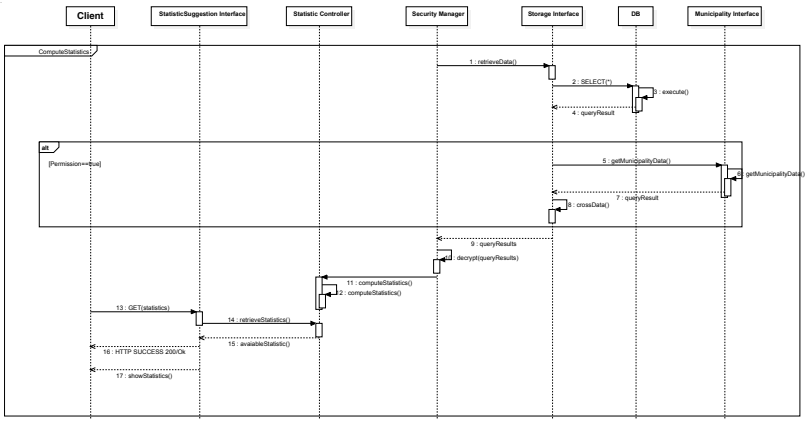

Figure 2.7: Build Statistics Runtime View.

The procedure to compute the statistics is performed by the *Information Manager* subsystem. Firstly, the *Security Manager* uses the *Storage Interface* to retrieve all the reports needed and decrypts them. After that, the decrypted data is sent the *Statistic Controller* to compute the statistics on them. If the municipality has given the permission to access their database, the Storage Interface can also retrieve their data contacting the *Municipality Interface*, and cross them with SafeStreets's. When an user taps on the statistics button, a `GET` request is sent to the StatisticsSuggestions Interface, which retrieves the latest avaiable statistics from the *Statistic Controller* and shows them to the user.

Compute Suggestions Runtime View

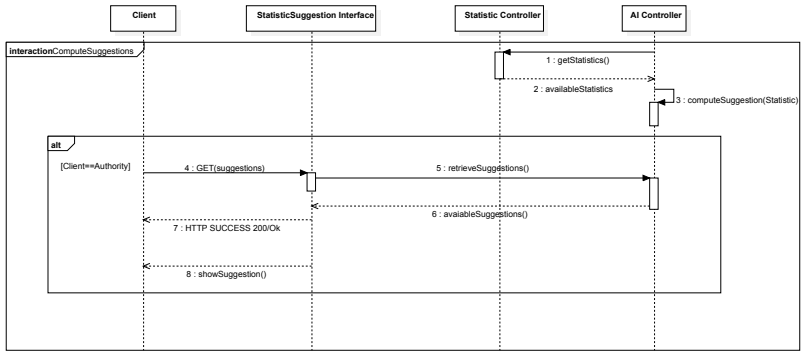

Figure 2.8: Compute Suggestions Runtime View.

The procedure to compute the suggestions is performed after the latest statistics are computed and avaiable. When available, the *Statistic Controller* sends the statistics it built to the *AI Controller*, which runs an artificial intelligence in order to compute some suggestions to improve the safety of urban areas. Safety suggestions are only addressed to the authorities, which are the only users enabled to send a `GET` request to the StatisticsSuggestions Interface and retrieve the latest suggestions available.

Report History Runtime View

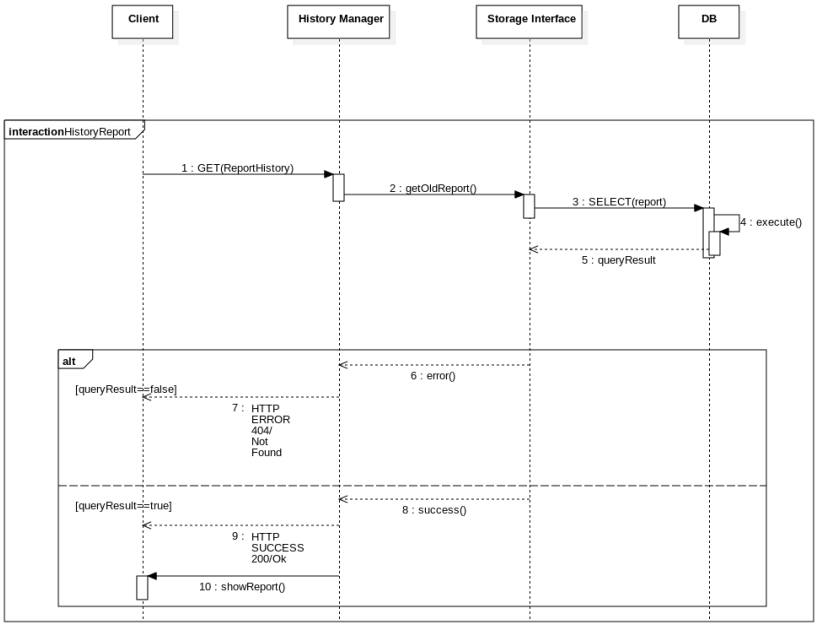

Figure 2.9: Report's History Runtime View.

A logged in user wants to retrieve all the reports he/she has sent up to that moment with a `GET` request. The *History Manager* handles this functionality communicating with the *Storage Interface*, which is able to retrieve all the reports relative to a specific user. This is done by running a `SELECT` query on the database and, if the query result is non-null, an HTTP Success message is returned to the user and the Storage Manager shows the reports; otherwise an HTTP Error is send back to the User. In case the user is an authority, he/she can retrieve all the submitted reports in a given area or period of time, so the procedure is the same with the only difference that the `SELECT` query filters on different attributes.

2.5 Component Interfaces

2.5.1 API REST

Authentication Controller

The *Authentication Controller* provides methods for login and registration. The Authentication Controller exposes the following methods:

Registration

Endpoint	*/auth/register
Method	POST
Data Params	username: [string] email: [string] password: [string] If the user is an authority: municipality: [string]
Success Response	code: 200 body: message: "Registration successful" code:202 body: message: "Accepted request" code:201 body: message: "New user created"
Error Response	code: 400 body: error: "Failed check" code: 401 body: error: "Unauthorized access" code: 422 body: error: "Unprocessable entity"
Notes	Allows the user to request the registration as citizen or authority.

Activate Account

Endpoint	*/auth/activate
Method	PUT
Data Params	username: [string] password: [string] verified: [bool]
Success Response	code: 200 content: message: "Account activated"
Error Response	code: 404 NOT FOUND content: error: "Incorrect activation code" code: 401 body: "error": "Account not activated" code: 401 body: error: "Wrong password"
Notes	Allows a Client to activate the account.

Login

Endpoint	*/auth/login
Method	PUT
Data Params	username: [string] password: [string]
Success Response	code: 200 content: accessToken: [string] code:202 body: message: "Accepted credentials"
Error Response	code: 401 body: error: "Unauthorized access"
Notes	Allows the Client to obtain an authentication.

Report Controller

The *Report Controller* handles the submission of the User, in this case the Citizen. The use of the send method is condi- tional on the Citizen authentication which is handled by the *Authentication Controller*.

Send Report

Endpoint	*/reports
Method	POST
Data Params	date: [string] time: [string] license plate: [string] pictureID: [string]
Success Response	code: 200 content: message: "Report received successfully" code:202 body: message: "Accepted request"
Error Response	code: 400 BAD REQUEST body: error: "Incorrect data"
Notes	Allows the Citizen to send a report to the system.

StatisticsSuggestions Interface

The *StatisticsSuggestions Interface* handles the GET requests of the Users by giving them statistics and suggestions (only in case that the User is an Authority).

Get Statistics

Endpoint	*/stats
Method	GET
Data Params	
Success Response	code: 200 content: message: "Request satisfied successfully"
Error Response	
Notes	Allows the User to retrieves up to date statistics.

Get Suggestions

Endpoint	*/suggs
Method	GET
Data Params	
Success Response	code: 200 content: message: "Request satisfied successfully"
Error Response	
Notes	Allows the Authority to retrieves up to date suggestions.

History Manager

The *History Manager* handles the request of the User providing an old report stored in the database. When a date is selected, the corrisponding old report is requested and delivered to the User.

Get old report

Endpoint	*/reports
Method	GET
Data Params	old date: [date]
Success Response	code: 200 content: message: "Report received successfully"
Error Response	code: 404 body: error: "Report not found"
Notes	Allows the User to get an old report.

2.6 Selected architectural styles and patterns

The following architectural patterns are used to build the structure of the system in order to provide all the services of SafeStreets application.

Client-Server Architecture

Client-Server architecture is a computing model that features two roles: a Server that hosts, delivers and manages most of the resources and services, and a Client which exploits them.

Motivations

This structure provides several advantages:

- Scalability and Mantainability: it is possible to repair or add more resources to the architecture without significative service interruptions.
- Security: the server is able to manage what levels of access each user can have with respect to specific resources.

Reverse Proxy Design Pattern

A simple proxy acts as an interface to refer an object in another machine. The reverse proxy offers a single point of access (with HTTP) to multiple Clients who want to access to several application servers. In practice, it is a wrapper for an object behind the scenes. Proxies also provides extra functionalities such as data caching (g.e. store the newest statistics available) and security (g.e. client-side firewall).

Motivations

It is useful to organize the requests of multiple clients and to save frequent requested informations. It also protects the application servers from external attack and slightly improve the overall performances.

Three-tiered Architecture

This type of architecture is a kind of Client-Server paradigm where three tiers are phisically separated:

- The *presentation tier* is the top-most level of the architecture, which provides an interface that users can use to directly access the application. It is the top-most tier and the only one accessible from the Client.

- The *application tier* runs the business logic of the application and executes functions that elaborate data. A Reverse Proxy is needed to handle the Client requests and to balance the workload, the requests are forwarded to the application serves in order to provide the right data.
- The *database tier* includes the data persistence mechanisms and the data access layer that encapsulates the persistence mechanisms and exposes the data.

Motivations

A multi-tier application architecture provides a model with several advantages: in this way developers can create flexible and reusable applications that can be modified, enhanced or maintained just by operating on a specific layer, instead of reworking the entire application. Furthermore, a general multitier architecture can also help improve the development efficiency by allowing teams to focus on their core competencies.

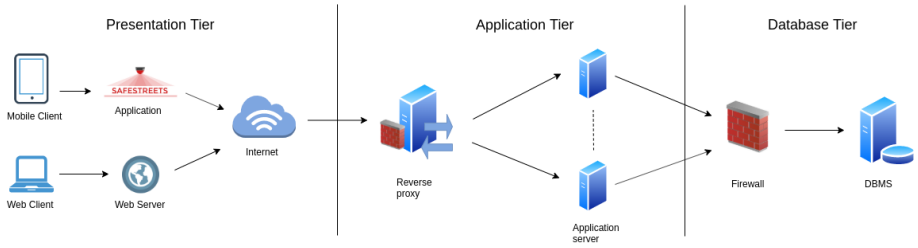

Figure 2.10: Three-tier architecture schema.

MVC Design Pattern

Model-View-Controller is a software design pattern commonly used to develop user interfaces. Indeed, this pattern is used for the front-end implementation of the application. It divides the program logic in three interconnected elements: *model* that directly manages data and rules of the application, *view* which handles any representation of data, *controller* that accepts input and converts it to commands for the model or view. This mechanism is used to separate internal representation of information from the ways information is presented to and accepted from the user.

Motivations

MVC allows full encapsulation of objects. This means that each component can be changed without creating issues to other components. Furthermore, MVC provides decopuling of its components, which means that developers are able to work in parallel on different components of the pattern without interfering with each other.

REpresentational State Transfer

REST is software architectural style that has become a standard in the creation of Web API. It describes those types of interfaces capable of exchanging data and enabling CRUD operations through HTTP requests, without using auxiliary technologies like cookies or other protocols. This architecture makes the communication client-server totally stateless (every request is independent from the others), and also allows for different representations of a resource (g.e statistics) based on who is accessing it (g.e. user from a certain location).

Motivations

We decide to use REST to simplify the design and the implementation of the application. Indeed, the developers do not need to take in account the traceability of states. REST also enables a design approach based on microservices.

Relational DBMS

The Data tier consists in a relational database where all the informations are strored in a structured way. There are a lot of table structures that can be used, in Figure 2.6 we show an example of how data can be organised in a Entity Relation Schema.

Motivations

Relational databases are a good choice for those applications that involve several transactions at a time. They allow for the use of foreign keys, which are used to uniquely identify atomic pieces of data within a table, but nonetheless they are equipped with mechanisms to guarantee referencial integrity and reliability. Moreover, most relational databases support SQL, which is an easy and human-readable language that enables CRUD operations for the storage.

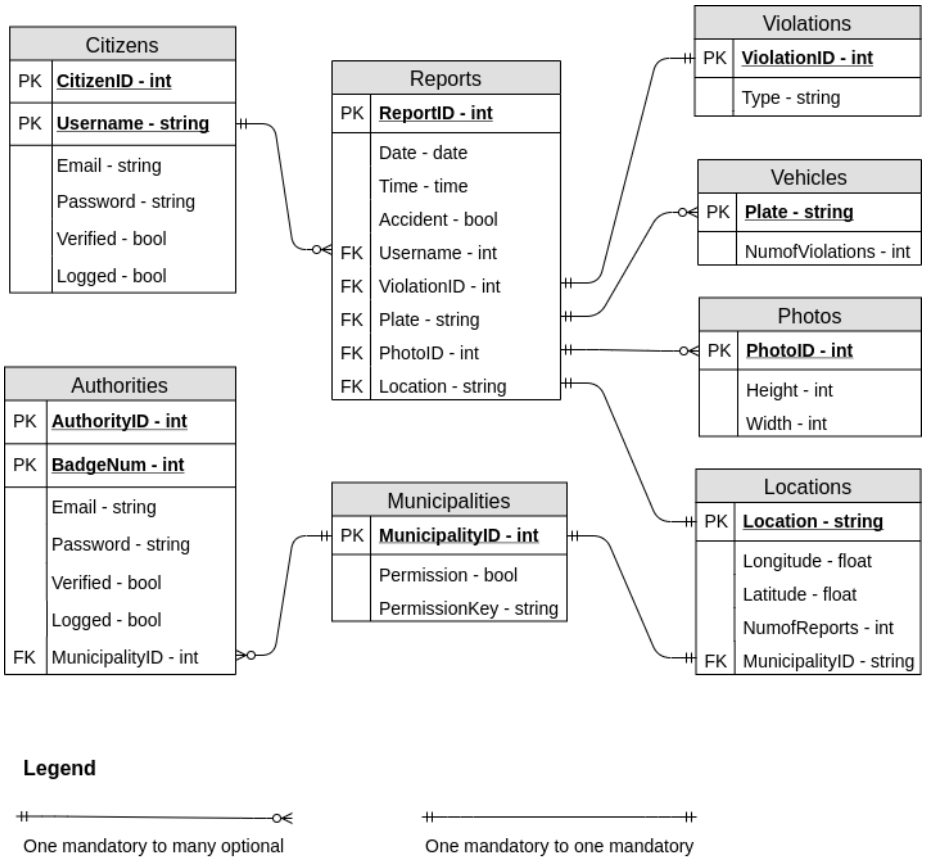

Figure 2.11: Example of entity relation schema.

2.7 Other Design Decisions

Thin Client

A thin Client is a lightweight computer which does most of the computation in a remote Server. In this paradigm in fact, since the Servers take care of several duties such as storage of data and performing calculations, the Client does not need to have a large memory or powerful computing capabilities to run the application.

Motivations

The application is thought mostly for mobile phones which do not have great computing power or very large memories. Thanks to a thin client a internet

connection is virtually the only requirements to use the application. Furthermore, this architecture simplify the front-end implementation by shifting most of the execution to the servers.

Microservices

Microservices is a software development technique that is based on developing a single application as a suite of smaller services. Each service runs its own process and is independently deployable from the other, and its interactions with other microservices are orchestrated using a RESTful approach based on communicating with lightweight mechanisms (HTTP).

Motivations

A microservices approach favors a decentralized management of the system's functionalities by loosely copuling its subsystems. This approach promotes information hiding between different services, gives rise to high resilience to failures (which are isolated whithin the subsystem involved), and allows to replicate each service according to its usage rather than duplicating the whole application.

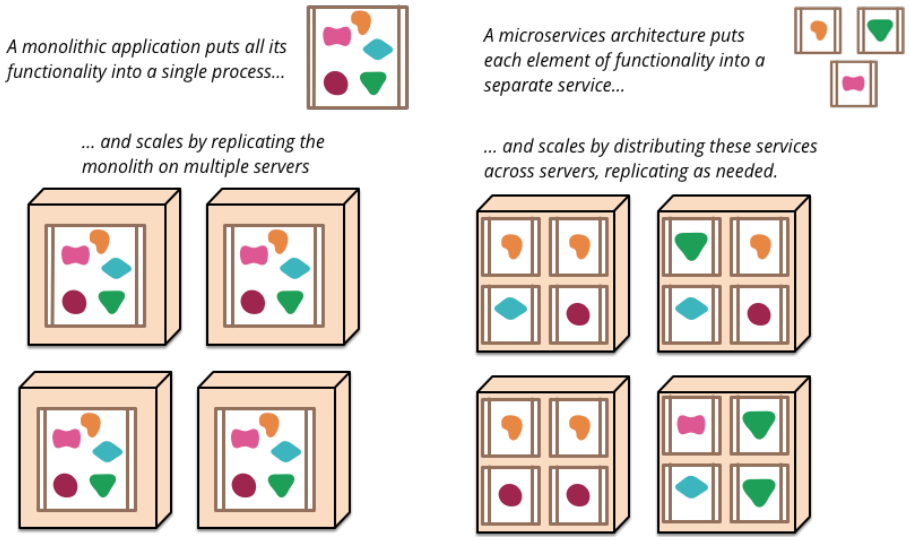

Figure 2.12: Replication schema for a microservices approach.

User Interface Design

3.1 Mockup

The mockups of the mobile application can be found in the RASD in the chapter *Specific Requirements*. For completeness purposes, we present a possible view of the web application.

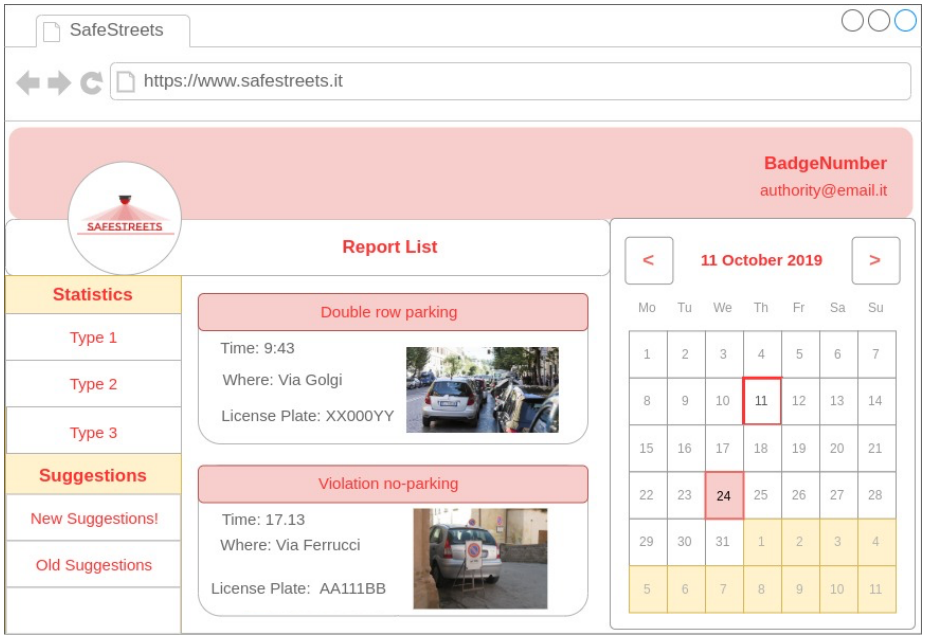

Figure 3.1: Web page graphics.

3.2 User Experience Diagram

The User Experience diagram (UX) diagrams reported below provide additional information about the User Interface that the application provides. The diagrams show the basic core actions that can be performed by the user through the mobile and web application.

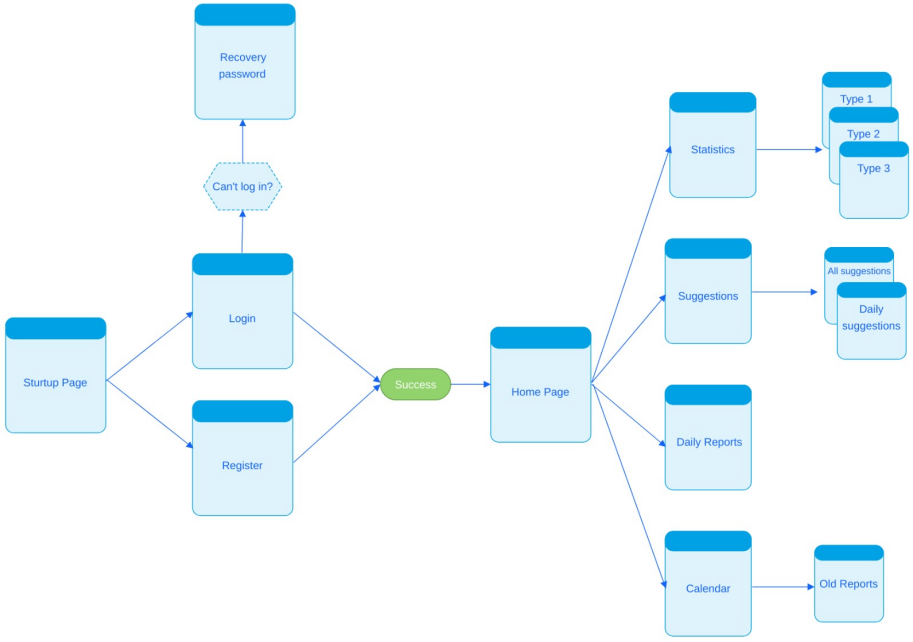

Figure 3.2: UX flow of an authority user.

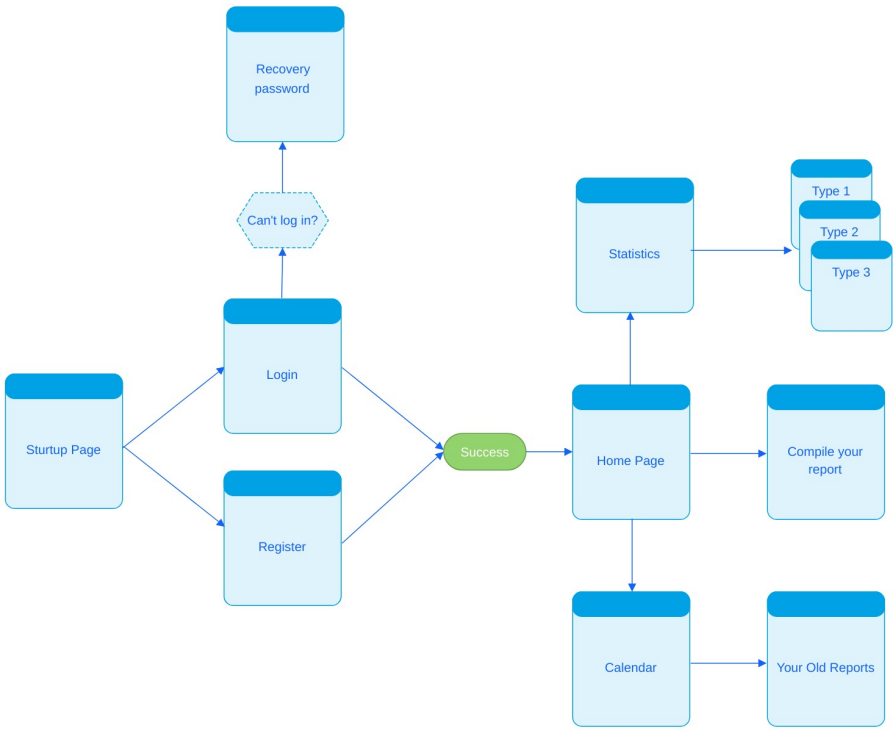

Figure 3.3: UX flow of a citizen user.

Requirements Traceability

All the decisions and the architectural choices taken in this document aim to achieve all the requirements defined in chapter 3 of the RASD. In this chapter it will be shown, for each requirement, which component or part of the system will accomplish them.

4.1 Functional Requirements

In the following table it is shown which components of the system are in charge of carrying out the functional requirements defined in section 3.2 of the RASD.

Component(DD)	**Requirement(RASD)**
Auth Controller	• [R.1] A visitor must be able to register. During the registration the System will ask to provide some credentials. • [R.2] Check if the guest credentials are valid • [R.3] If credentials are valid, the System sends a confirmation email • [R.4] Allow to log in using personal credentials. • [R.13] Provide a recognition system of badge number. • [R.20] Use third party services to enable some functions (Mail Service).

User Controller	• [R.5] Allow to change username, only if the new username is not already in use by another User, email, only if the new email is in a correct format and password, only if the new password is different from the precedent and respects the minimum length. • [R.6] Send a confirmation email if username, email or password is changed (similarly to the registration process). • [R.7] Allow to change password if it has been forgotten, through the personal email. • [R.20] Use third party services to enable some functions (Mail Service).
Report Interface	• [R.8] The Citizen must be allowed to create reports.
Report Controller	• [R.9] Check if the Report created by the Citizen is valid. • [R.20] Use third party services to enable some functions (OCR Service).
Notify Controller	• [R.10] Notify the Authorities whenever a new Report is submitted. • [R.16] All traffic violations must be notified to Authorities according to their type and municipality.

Map Service Interface	• [R.15] Allow Users to see the their position in a map. • [R.20] Use third party services to enable some functions (Map Service).
History Manager	• [R.14] Allow the Citizen to consult the history of its personal reports. • [R.17] Provide the Authorities with a list of the most recent reports submitted. • [R.18] Allow Authorities to also retrieve older Reports not shown in the application.
Storage Interface	• [R.19] Interact with the Municipality's System to get the permission for using their data. • [R.24] Cross the data coming from external DBMSs with the System's own data.
Security Manager	• [R.22] Provide protection of data using encryption. • [R.28] After its validation, no one should be able to modify a submitted Report.
StatisticsSuggestions Interface	• [R.11] Provide two different level of visibility regarding the statistics. Only the Authorities can consult statistics with private data. • [R.12] SafeStreets' application must show the most recent version of the suggestions and statistics.

Statistic Controller	• [R.23] Acknowledge whether or not multiple istances of Reports are referred to the same infraction, even if protracted over time. • [R.25] Cross data from different Reports in order to establish their truthfulness. • [R.21] Statistics and suggestions have to be periodically recomputed
AI Controller	• [R.26] Unsafe areas are detected by means of an artificial intelligence. • [R.27] Safety Suggestions are computed by means of an artificial intelligence.

Table 4.2: Functional requirements mapping on system's component

4.2 Non-Functional Requirements

The non-functional requirements defined in Section 3.3 and 3.5 of the RASD are mainly a matter of concern for the development team. However, design choices such as the three-tier architecture and the microservices approach aim for increased performances and in general to improve the software quality attributes.

Implementation, Integration and Test Plan

As previously illustrated in the Component Diagram, the System is made of many components that can be divided into the following categories:

- **Frontend components**: Client application.
- **Backend components**: User controller, Authentication controller, Report controller, History Manager, Notify controller, Security Manager, AI controller, Statistic controller, StatisticSuggestion Interface and Storage Interface.
- **External components**: DB, Mail service interface, OCR service interface, Map service Interface and Municipality Interface.

In order to implement, integrate and test the system, a bottom-up approach will be used, considering also the importance of the various functionalities. Thus, since some components are grouped into subsystems as illustrated in the Figure 2.2, the components in the same subsystem will be implemented, integrated and tested first and then the subsystems will be integrated and tested together in order to verify the correct behaviour of the system. As specified in the RASD, the System is not responsible for failures of third party services, so the components of the external subsystems don't need to be tested and are assumed to be reliable. Therefore the implementation and integration process will be performed in two phases:

1. Implementation and integration of each component of the same subsystem;
2. Integration of different subsystems;

The following table lists the main features avaiable to the customer. Their importance and implementation difficulty is shown to better understand the decisions about implementation, testing and integration that will be taken in the rest of this section.

Functionality	Importance for the customer	Difficulty of implementation
Sign up and login	*Low*	*Low*
Consult older reports	*Medium*	*Low*
Use external map services	*Medium*	*Low*
Report a violation	*Medium*	*Medium*
Visualize statistics	*High*	*High*
Visualize suggestions	*Medium*	*High*

Table 5.1: Main functionalities with their importance and implementation difficulty

Regarding the first phase, components integration of the same subsystems will be applied only for backend components. More specifically, the components to be integrated and tested together are:

- User Controller and Storage Interface, forming the *User subsystem*;
- Authentication Controller and Storage Interface, forming the *Authentication subsystem*;
- History Controller and Storage Interface, forming the *HistoryReport subsystem*
- Report Interface, Report Controller, Notify Controller and Security Manager, forming the *Report Manager subsystem.*;
- AI Controller, StatisticSuggestion Interface, Statistic Controller and Security Manager, forming the *Information Manager subsystem.*

Also the Report Manager and Information Manager, as well as the first three subsystems listed above, will use the Storage Interface to interact with the DB. Moreover, the *Mail Service Interface* will be used both by the User and the Authentication subsystem; the latter one also interacts with the *Municipality Interface*. Finally the Report Manager could use the *OCR Service Interface*. Thus in the implementation and integration process the mentioned components must also be taken into account. Considering that the Storage interface is used by many components and therefore also by many subsystems, it is useful to implement and test it first and then try to integrate it with the other components/subsystems.

For the second step all higher-level subsystems will be integrated. In particular, this integration activity involves the Business Logic and the Client subsystem. It is important that the verification and validation processes starts as soon as the development of the system begins in order to find errors as quickly as possible. As specified above, an incremental approach is used for the integration process, so that bug tracking is more easier. Moreover, also scaffolding techniques must

be used when needed: in particular due to the fact that the frontend and the external subsystems are independent, the external services will be used in place of stubs to test the interfaces.
At the end, when the integration system is completed, the whole system is tested: for this purpose a fundamental type of test is the performance test that is useful to identify bottlenecks affecting the response time, the utilization and the throughput. It identifies also inefficient algorithms, Hardware/Network issues and Query optimimzation possibilities and this is very important since in this system there are a lot of interactions with the database.

5.1 Sequence of Component Integration

The following diagrams describe the process of implementation, integration and testing. The arrow goes from the used component/subsystem to the component/subsystem that uses it.

5.1.1 Integration of the backend subsystems

All the components are first implemented and unit tested. Then some of the components are integrated into subsystem and integration tests will be performed. The integrations are the following:

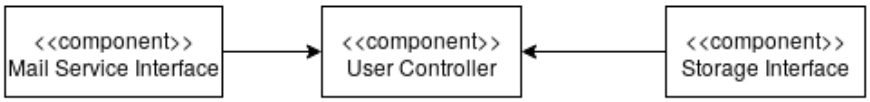

Figure 5.1: User subsystem.

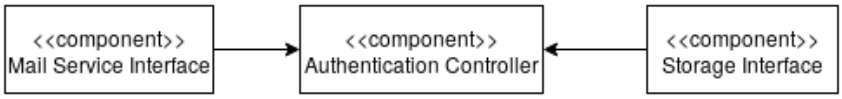

Figure 5.2: Authentication subsystem.

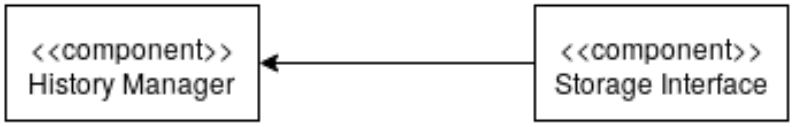

Figure 5.3: HistoryReport subsystem.

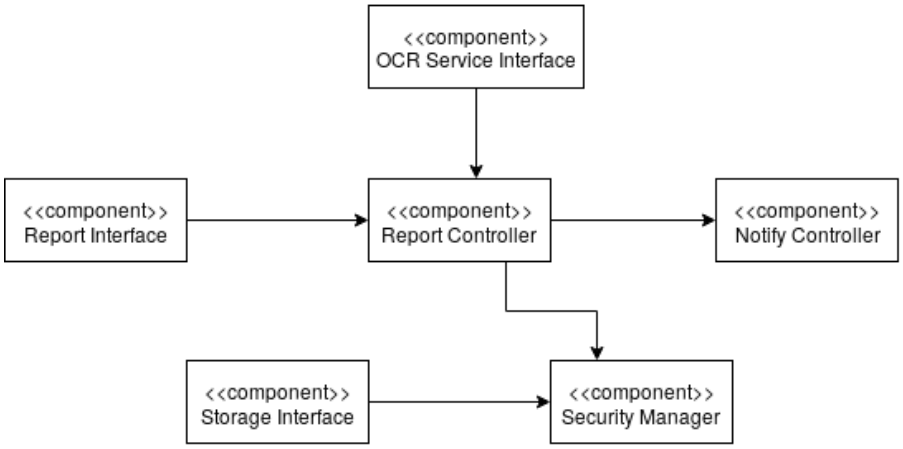

Figure 5.4: Report Manager subsystem.

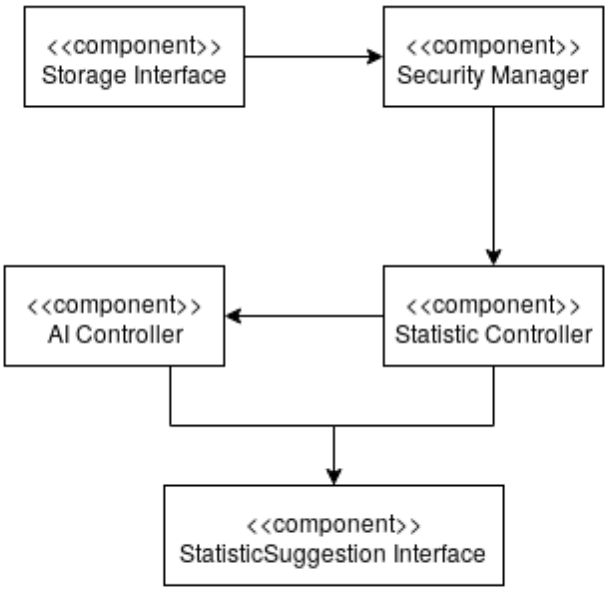

Figure 5.5: Information Manager subsystem.

5.1.2 Integration of backend and frontend

When all the components of the backend are fully implemented, integrated and tested, the frontend will be integrated and tested with the backend: in particular it will be integrated the Client subsystem with the Report Manager subsystem, Information Manager subsystem, HistoryReport subsystem, User subsystem and Authentication subsystem.

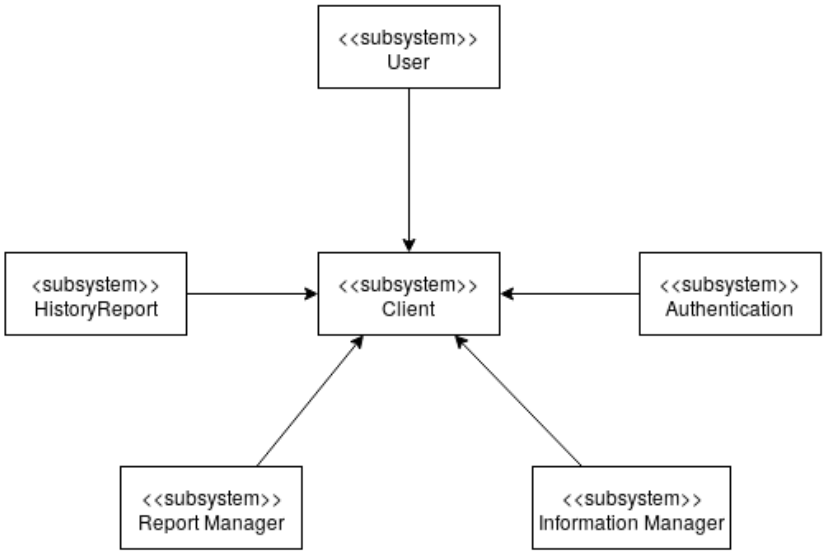

Figure 5.6: Client subsystem.

5.1.3 System integration

At the end of the previuous procedures the *frontend*, *backend* and *External Components/Subsystems* are integrated and tested together.

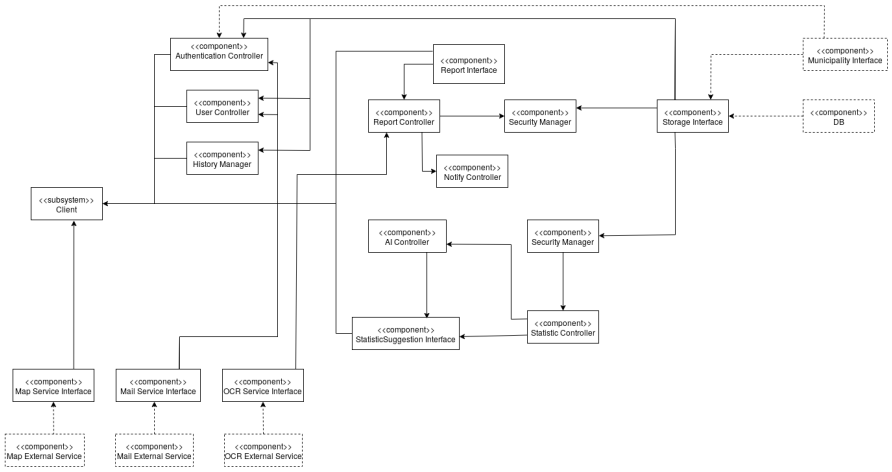

Figure 5.7: Integration of the whole System.

Effort Spent

Francesco Amorosini

Task	Hours
Purpose and Scope	3
Runtime view review	2.5
REST API Table review	3
REST and microservices approach	3.5
Relational database	2
Requirement mapping on components	4.5
Recap Call: • Discussion on document consistency • Schedule of missing sections	 1 1
Priority table for component integration	2
Image polishing	2.5
English check and impagination	5
Group review	2
Consistency fixes in diagrams	3
	Total: 35

Alice Casali

Task	Hours
Discussion about architectural design	3
High-level architecture diagram	3
Architectural Structures	4
MVC explanation	2
Deployment View	3
Added REST architecture	2
Fixed Deployment View	1
API REST table for Computation Interfaces	4
User interface mockup	2
User interface UX diagrams	3
Recap Call: • Discussion on document consistency • Schedule of missing sections	 1 1
Added new API REST table	2
Relational database	2
Sequence Diagram review	2
Completed API REST table	2
Relational database schema	3
Group review	3
	Total: 43

Tommaso Fioravanti

Task	Hours
Component View	5
Runtime View	5
Description of Sequence diagrams	2
Review of Component Interface	1
Requirement mapping on components review	1.5
Implementation, Integration and Test plan	5
Diagrams of component integration	2
Recap Call: • Discussion on document consistency • Schedule of missing sections	 1 1
Component diagram review	1
Runtime diagrams review	2
Review of priority table in component integration	1
Further refinements	1
Group review	3
	Total: 31.5