POLITECNICO
MILANO 1863

Software Engineering 2

Design Document

SafeStreets

Version 1.0 - 08/12/2019

Authors:

Andrea Falanti

Andrea Huang

Professor:

Prof. Elisabetta Di Nitto

Contents

Introduction

1.1 Purpose

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 application allows users to send pictures of violations, including their date, time, and position, 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.

SafeStreets stores the violation reports provided by users, that can be made from the official mobile application or from the SafeStreets website, so that users and authorities can visualize and analyze the data received by the system, for example by highlighting the streets (or the areas) with the highest frequency of violations, or the vehicles that commit the most violations. The data can be accessed with different levels of visibility, where most sensitive data can only be mined by authorities.

SafeStreets also provide a service aimed at helping municipalities to identify potentially unsafe areas and suggest possible interventions. To improve the accuracy of these features, a municipality can also offer a service for retrieving info about accidents in their territory, so that Safestreets is able to cross the municipality's data with its own stored data and also analyze and suggest possible improvements to the areas with most reports.

Goals

G1. Report certified traffic violations to authorities of city where the violations have happened.

G2. Store data about violations and cross it with municipality's one when an API to its data is provided.

G3. Allow municipalities to generate a list of possible reasonable interventions to resolve the violations detected in most unsafe areas.

G4. Recognize public institutions automatically during registration to setup correctly accounts of municipalities and authorities.

G5. Provide a synthesis of all the violations' data to users, based on their authorization level.

   G5.1. Normal users can access only to aggregated data, without seeing any private info about a reported violation, like the license plate or the personal info of the user who reported the violation.

   G5.2. Authenticated authorities can access to all violations' data, as aggregated data or singular reports.

G6. Allow users to send a report of a possible traffic violation, identifying clearly the transgressor.

G7. Allow users to check their personal info and the list of violations they reported.

1.2 Scope

The SafeStreets service is offered to common users to report traffic violations that hinder the normal flow of the traffic. It is thought to offer an aide to the public officers in detecting violations and thus provide a more regulated traffic system to the citizens. The service stands in the middle between the common citizen and the authorities, providing a real-time update of the situation on the streets.

The software will be distributed in Italy and will provide to any Italian citizen the possibility to report traffic violations by taking a photo of the transgressor's vehicle with its license plate visible and an appropriate view of the situation. The system will locate automatically the location of the violation, assuming the device of the user has the GPS service enabled. When the report is received by the system, it's stored into a database and authorities that have jurisdiction over that city can access it to consult its details and validate or invalidate it. The user will be able to track the status of their reports from the mobile app. SafeStreets, besides receiving reports from users, will also provide the possibility to consult its stored data with different levels of authorization for privacy safety reasons. A common citizen will be able to find the most frequent violations by location or time, but no private information of the violations will be provided. Whereas, the registered authorities will be able to perform the same queries and access to all the private information of the transgressors, for example, they will be able to find the most frequent transgressor in a certain area.

To help municipalities in identifying potentially unsafe areas, SafeStreets retrieves data about traffic accidents in their territories from their specific service. It crosses this data with its own stored data and computes possible interventions to areas most affected by violations, for example suggesting to add a barrier between the bike lane and the part of the road for motorized vehicles to prevent unsafe parking.

In the following we list the different kinds of events that can occur in the possible scenarios of interaction with SafeStreets. World events are phenomena occurring in the world that are not observable and not controllable by the SafeStreets' system. Shared events are phenomena that are either controlled by the world and observed by the SafeStreets's system or vice versa. Machine events are phenomena that are observed and controlled inside the SafeStreets' system, so they are not known to the world.

World events:

- Traffic violations
- Violation detection
- Street interventions and improvements
- Authorities interventions

Shared events:

- Violation report codification
- Data visualization and analysis
- Filtered data request

- Validated violations notification to authorities
- Intervention suggestion to municipality

Machine events:

- Database queries
- Possible interventions computation
- License plate recognition
- Meta-data completion

Phenomenon	Shared	Controlled by
Traffic violations	N	W
Violation detection	N	W
Street interventions and improvements	N	W
Authorities interventions	N	W
Violation report codification	Y	W
Data visualization and analysis	Y	M
Filtered data request	Y	W
Validated violations notification to authorities	Y	M
Intervention suggestion to municipality	Y	M
Database queries	N	M
Possible interventions computation	N	M
License plate recognition	N	M
Meta-data completion	N	M

Legend:

- Y := Yes, N := No.
- W := World, M := Machine.

1.3 Definitions, Acronyms, Abbreviations

Definitions

- **Citizen or common user**: a common citizen, without any public office, who uses the application to report traffic violations;
- **Authorities**: the public officials who certify the violations reported on the application;
- **Municipality**: the public institution that provides traffic violation data to the application, and may consult and analyze intervention suggestions from it;
- **Violation**: an event that does not conform with the traffic laws and that can be reported to authorities.
- **Intervention**: a possible public intervention produced by the application aimed at solving frequent violations in some areas of the city.
- **(Report) Supervisor**: authority that have validated or invalidated a report.

Acronyms

- GPS = Global Positioning System
- S2B = Software to Be
- RASD = Requirement Analysis and Verification Document
- PEC = Posta Elettronica Certificata
- API = Application Programming Interface
- AJAX = Asynchronous JavaScript and XML
- GDPR = General Data Protection Regulation
- OCR = Optical Character Recognition
- DD = Design Document
- CPU = Central Processing Unit
- DBMS = DataBase Management system

- REST = Representational State Transfer
- SQL = Structured Query Language
- IO = Input Output
- UI = User Interface

Abbreviations

- Rn = n-th requirement (defined in RASD).

1.4 Revision history

- Initial version 1.0 (08/12/2019)

1.5 Reference Documents

- Specification document: "SafeStreets Mandatory Project Assignment"
- IEEE Std 830-1998 IEEE Recommended Practice for Software Requirements Specifications

1.6 Document Structure

The Design Document is composed by seven chapters:

Introduction: first chapter of the document, it describes informally the main goals of the system and what problems they address. Note that first two sections are borrowed from RASD document, as they provide a general introduction to the system.

Architectural design: second chapter, it provides information about the system architecture, how it is physically deployed and which technologies are used to fulfill its functional requirements. It also describes the patterns and design decision used to implement the project and why they are useful in achieving the goals of the system.

User interface design: third chapter, it contains the mockup of the screens visualized by clients who interact with the system, already defined in RASD and reported here for simplicity of reading.

Requirement traceability: fourth chapter, it describes explicitly how each requirement of the system is fulfilled and which components are involved for its realization.

Implementation, integration and testing: fifth chapter, it provides information about how the system will be implemented, how all its components will be integrated to build the definitive system and how testing will be performed. To achieve this, a preliminary analysis is performed to identify the importance and difficulty of implementation of the various functionalities and annexed components. The results of the analysis are used to establish the order of implementation of the components and how much testing will be performed on them.

Effort spent: sixth chapter of the document, shows how much effort each member of the group has spent on the various chapters of the document.

References: final chapter, contains links to material, information or documentation related to the content discussed in the document.

Architectural design

2.1 Overview

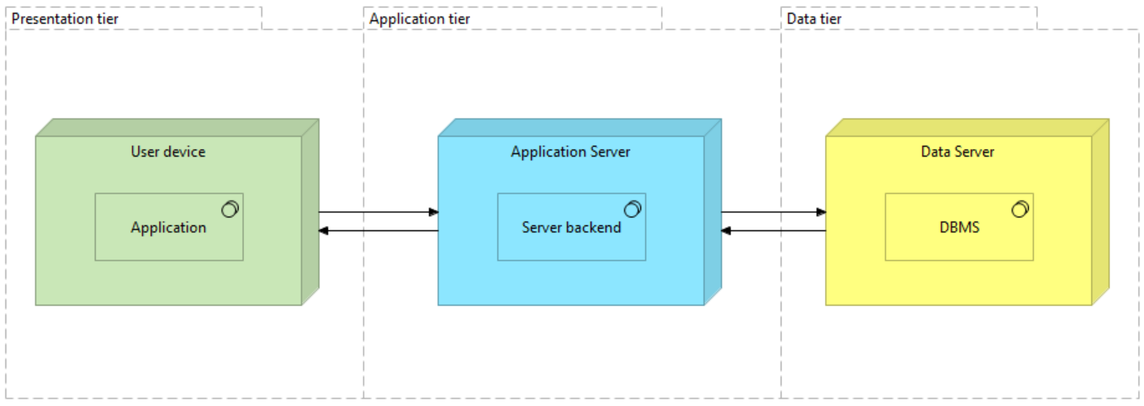


Figure 2.1: Three-Tier architecture.

SafeStreets system is organized as a three-tier architecture, a standard design architecture used to provide a good level of security and performance thanks to subdivision of the tasks between the three tiers. Each tier runs on a different machine (or a group of machines with same structure), so that every tier is responsible of a specific task and the computational work is splitted between different machines, optimizing the performances of the overall system. This also improve the security of the system, because user devices need to pass through the application server to access the data and have no way to directly access it.

The presentation tier runs directly on customers devices, which can be both smartphones running the official mobile app or PCs using the web app, and its main goal is to handle the data and assets received by the application server to display the information on the device screen. User can then see the information requested and also interact with the presentation layer by using classical view widgets (like button, textfield, sliders, etc...) to require additional data based on their inputs. To summarize, the main goals of presentation layer are to allow communication with the application server API to require data based on user input and to visualize the results on the user's screen.

The application tier is composed by a replicated array of application servers, to scale horizontally the system using machines with the same structure and functionalities. The array of servers is managed by a load balancer, that distributes the clients' requests (and so the computational cost) equally between all the machines. Because application servers need to receive multiple requests from different clients in an efficient way, the load balancer integrates a router module that actively listens for any request made on specific server ports. The router has an handler for any endpoint of SafeStreets API that calls the manager related to the endpoint functionality, passing any input received in the request so that the manager can correctly consult or manipulate the database and produce an output, that is then returned by the router to the client that made that request. The list of managers that fulfill the functionalities is reported in a later section, in which every manager is also defined in detail. It's important to state that the application server also integrate the functionalities of a web server, so that can return static assets to clients that requires them.

The data tier's task is to provide access to data to the application tier, so that the manager modules can interrogate the database and also store or modify data in a reliable way. Data tier is composed by a single server, with cloned databases that share memory disks. This structure helps in maintaining high performances while keeping the data safe from possible hardware issues or data loss.

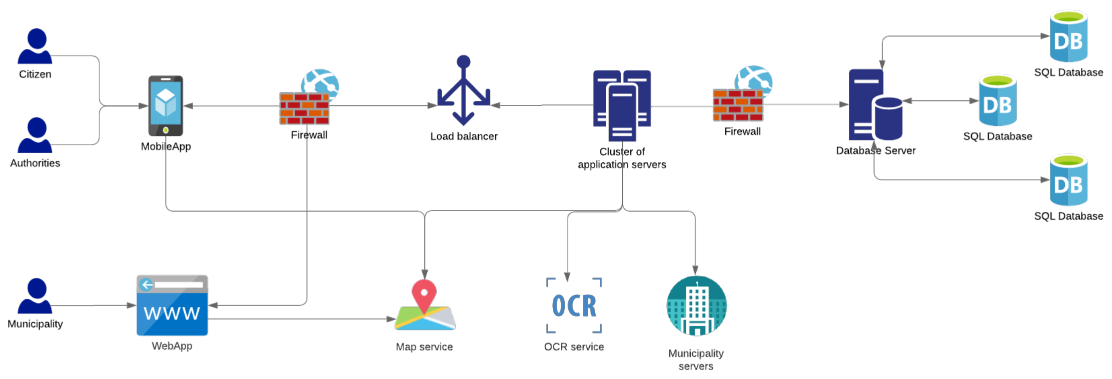


Figure 2.2: System architecture.

The three tiers are separated by firewalls to improve the security of the system. Firewall between application server and client is a must to provide a layer of protection from fraudulent users and hackers, because the application server could be accessed by anyone external to the system. The firewall between application server and database server isn't strictly necessary but can be used to negate all accesses that don't originate from the cluster of application servers, providing an additional layer of protection from fraudulent accesses.

The above system architecture diagram also shows the services external to SafeStreets system. A map service will be interrogated by all clients that need data visualization on a map, and the application server can call the API of a generic OCR service to recognize license plates of vehicles involved in reports from their photos, call the map service for metadata completion when registering a report, and can interact also with municipalities' API to integrate their data before generating the interventions.

2.2 Component view

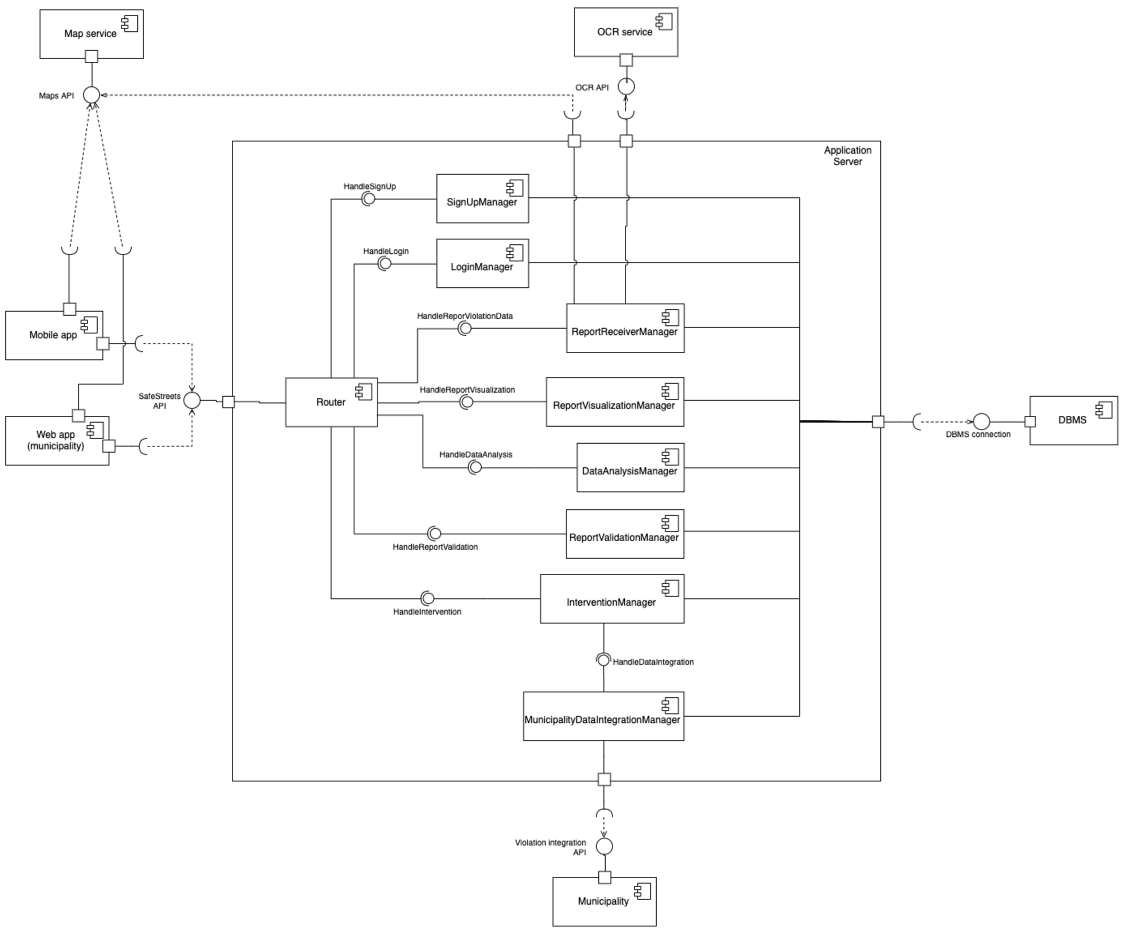


Figure 2.3: Components diagram.

Customers app subsystems (clients):

- MobileApp (citizen/authorities)
- WebApp (municipality)

Interfaces summary

In the following, a list of all the system's interfaces subdivided into relevant categories is provided:

External interfaces:

- DBMS
- Maps API
- OCR API
- Violation integration API

Router Interfaces accessible from mobile app:

- HandleSignUp
- HandleLogin
- HandleReportViolationData
- HandleReportVisualization
- HandleDataAnalysis
- HandleReportValidation

Router interfaces accessible from municipality web app:

- HandleSignUp
- HandleLogin
- HandleDataAnalysis
- HandleIntervention

Component descriptions

In this section, every internal component of the application server is analyzed and described in detail:

- **Router:** it listens for all requests made by clients on the SafeStreets API endpoint and provides handlers for each type of request. The handlers call various managers internal to the application server that are pertinent to the request made by the client, so that the router can receive from them the result of the operation and send back to the user a response.

- **SignUpManager:** module with the goal of handling registration requests of all user types. It checks that no data is missing and its correctness, by checking if the email has already been registered and if the password satisfy the given rules (e.g. at least 8 characters, at least 1 alphabetical symbol and 1 number, ecc...) and it's correctly repeated in the "repeat password" field. If data provided by the registration is valid, then the manager registers the account by inserting it into the main database and returns a confirmation to the router, else it returns an error.

- **LoginManager:** module that handles login requests made by clients. It checks that email and password provided by the user in the login form is actually associated to an account registered in the database. If login is successful, a session package installed in the server will register the session, so that other components can check easily which account made the request and discriminate services and accessible data for each account type. This module returns a confirmation to the router module if login is successful, otherwise it returns an error.

- **ReportReceiverManager:** module that allows the correct registration of the reports data generated by the citizens and also the communication with the OCR service, used to recognize correctly the license plates from the first photo of the report. When a citizen starts the report generation procedure, it will be asked to take pictures of the violations, in which is stated that the first one should have the license plate of the vehicle involved in the violation clearly visible. After the citizen takes the pictures, the first one is sent to the server and then is passed to this manager responsible for sending a request to the OCR service to recognize the license plate of the vehicle. After receiving the result, the manager forwards it to the router and it's sent back to the user for confirmation. The managers also handles the whole report data when the citizen submits it to the application server. If the report is incomplete because some data is missing, the manager reports the error to the router that then sends back it to the client, otherwise the manager completes it with some meta data like the timestamp, address obtained by reverse geocoding the GPS coordinates with the Maps service, the userID of the citizen, and 'pending' status. Then it stores the report in the database, returning an "ok" message.

- **ReportVisualizationManager:** module associated to single reports data requests or list of reports requests. It provides the possibility to query database for searching reports associated to a specific city, or to a specific user, or a single report given its id. This covers all the requirements regarding specific report visualization that the users can make with the mobile apps. In fact, authorities can access a list of all reports that are related to their city and a citizen instead can access the list of all reports made by himself; both type of users can then select a report from the list and send a request to the router module with the report id attached. It's important that the ReportVisualizationManager checks what type of account made the request from the session, so that it can check whether the request is valid or fraudulent in order to avoid the possibility of not authorized data access.
- **ReportValidationManager:** module that permits a report validation or invalidation made by authorities. It associates the authority ID to the report and changes the status of the report according to the one selected by the supervisor. Obviously the changes are made directly on the database, updating the record with the ID of the report present in the request. Also here is checked that the report that is going to be modified is actually accessible from the authority that made the request (it's situated in the city in which the authority has jurisdiction, that is retrievable from the database).
- **DataAnalysisManager:** module that handles the requests for data analysis, that involve the representation of data from multiple reports that match the filters specified by the client. Every request received by this component contains temporal filters (from what date and to which date), spatial filters (the interested city) and violation type filter, that are then used by the manager to query the database and return to client all the validated reports data that satisfy the filters. Note that like ReportVisualizationManager, also this component need to check which type of account made the request and filter some data based on its permissions. If the request is made by a citizen account, the license plates, the submitter data and the supervisor (authority that validated the report) are omitted from reports' data, to avoid common citizen to visualize sensible data that is also not relevant for their analysis; instead if the request is made by authorities and municipalities' accounts, they can visualize all data associated to the reports. Another consideration is that the server only provides the data

to clients but doesn't define or bind the method to visualize this data in any way.

- **InterventionManager:** module that handles municipalities' requests for generating new lists of possible interventions, targeted to provide possible solutions to the frequent violations in the most unsafe areas of their cities. Before generating the interventions, this module calls the MunicipalityDataIntegrationManager to integrate municipality's data in the SafeStreets' database, if possible. After receiving a response from the MunicipalityDataIntegrationManager, the list of interventions will be generated. An algorithm consults all the reports present in SafeStreets database that are located in the city of the municipality that made the request, and checks for the areas with the highest density of violations. After computing the most unsafe areas of the city, the algorithm checks the type of violations recurring for each single area and from that it computes all possible interventions that can solve the problem. When all the process is finished, the result is returned to the router, in combination of a possible error raised from the MunicipalityDataIntegrationManager so that the user is aware of the fact that data integration was not possible and he could investigate the problem.
- **MunicipalityDataIntegrationManager:** module that allows the integration of violation data from any municipality API structured according to SafeStreets reference API model. It is accessed by the InterventionManager when a municipality requests the generation of a list of possible interventions to do in its territory. The module checks if the municipality that made the request has provided an API for accessing its data, if present in the database it requests all violation data that have a date later than the one indicated in last intervention generation timestamp (gets all data if no intervention generation was made by the municipality in the past), and integrate it in SafeStreets database. When the process is done, the manager notifies the InterventionManager, that is waiting a response from it before initiating the intervention generation. Note that if no API is associated to the municipality the integration will be immediately aborted, instead if the municipality's API endpoints or its provided data have not the structure expected (not compliant with the reference model) the MunicipalityDataIntegrationManager will return an error to the InterventionManager, that will continue without the municipality's data integration, using

only data contained in SafeStreets' database (the error is reported to the user in this case).

Entity-relationship diagram

The entity-relationship diagram below describes the conceptual model of the database.

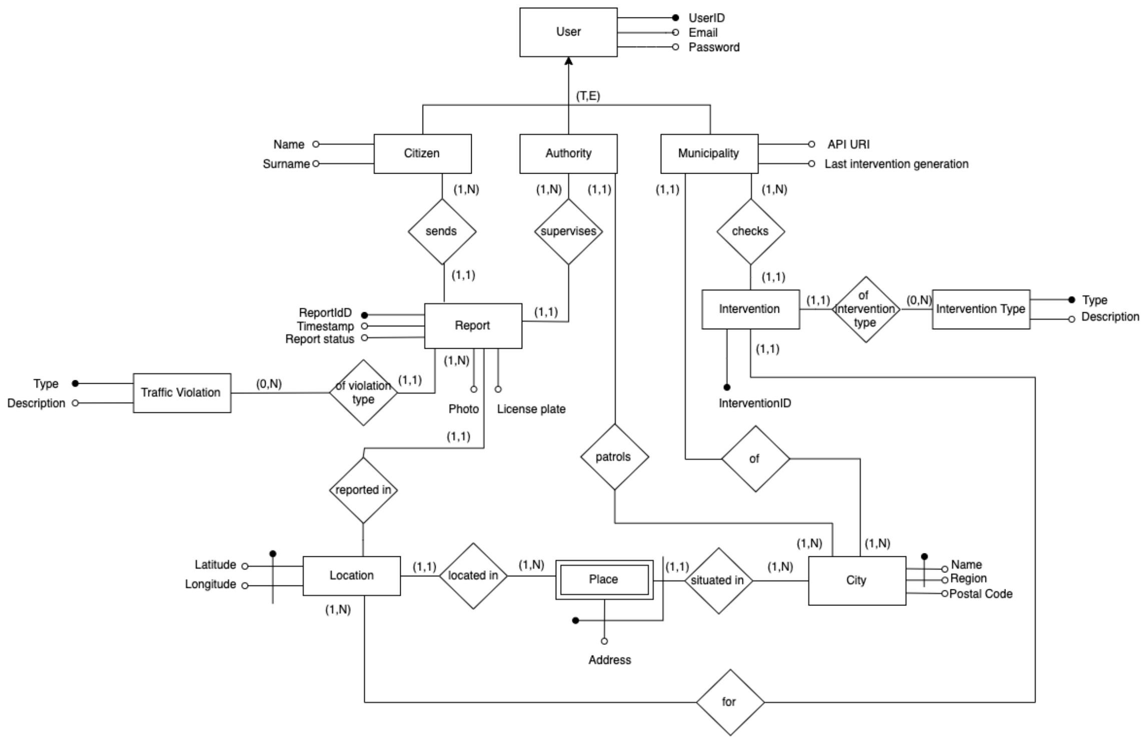
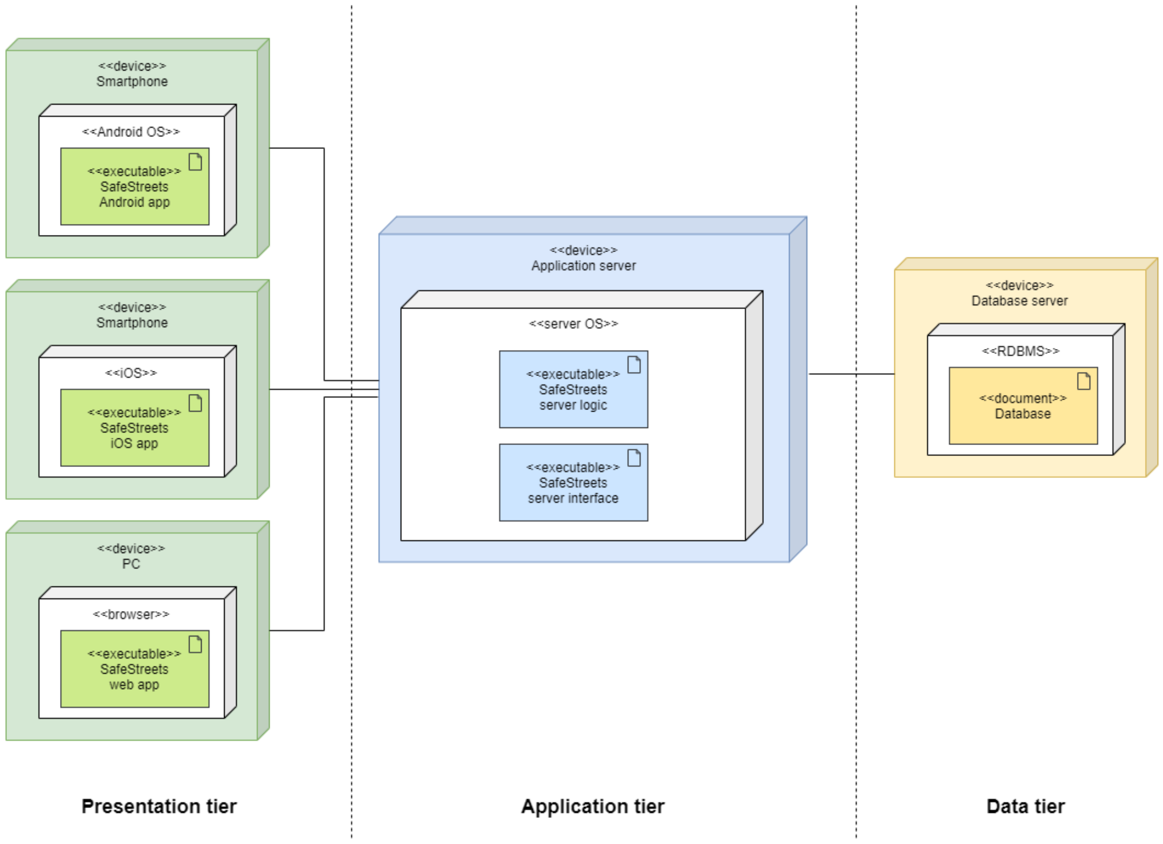


Figure 2.4: Entity-relationship diagram.

In comparison with the class diagram presented in RASD document, the main entities user, report and intervention have UserID, ReportID, and InterventionID primary key attributes, respectively, that are used for a faster lookup on the database. In fact using strings or multiple fields primary keys are not advisable in case performances are concerned, because indexing them is more difficulty and require more database block lookups compared on indexing on a single int number. Also, the IDs guarantees the uniqueness of the primary key in case that a table doesn't have any relevant unique identifier among its fields.

2.3 Deployment view

This section focuses on explaining how the system will be deployed, describing the machines involved in the system and what components they will run or contain.



<<device>> Smartphone
<<Android OS>>
<<executable>> SafeStreets Android app
<<device>> Smartphone
<<iOS>>
<<executable>> SafeStreets iOS app
<<device>> PC
<<browser>>
<<executable>> SafeStreets web app
<<device>> Application server
<<server OS>>
<<executable>> SafeStreets server logic
<<executable>> SafeStreets server interface
<<device>> Database server
<<RDBMS>>
<<document>> Database
Presentation tier
Application tier
Data tier

Figure 2.5: Deployment diagram.

In the presentation tier, clients devices will run mobile apps on both Android and iOS smartphones, instead personal computers can run the web application if the user has access to a municipality's account. We decided to implement mobile apps for smartphones to cover citizens and authorities needs of interacting with SafeStreets system in an easy and fast way, because both categories of users will use the apps during their normal routines and it's nowadays common to have a smartphone with you all the time. For municipality instead we choose to implement a web application because a municipality member will probably use the software in his workplace, so using a computer is optimal for the task and having a larger layout with a better level of details when retrieving data is surely an appreciated feature. A web application is preferred to a website because municipalities will be interested in data research and possible improvements of unsafe areas, so the possibility to filter content and dynamically display and retrieve data is a must. To access the web application, the only requirements is to have a web browser that supports AJAX, so no restrictive requirements are necessary for running it, making it really easy to integrate the use of the software in workflows of every municipality.

In the application tier, the application server structure will be developed internally but deployed on an external hosting service, because internalizing the management of an array of servers is costly and time consuming, while hosting services are nowadays quite cheap, provide assistance and externalize interventions and maintenance. The server needs to provide a REST API to clients and an appropriate backend logic for handling each type of request correctly. Components used by the server are listed and explained widely in section 2.2, so we focus now on server technology. The server structure will be replicated on an array of machines, handled by a load manager for distributing the workload equally and avoid system congestion.

In the data tier, the database is deployed along with its DBMS. The database will be distributed in various instances to improve performances and improve the reliability of the storage. The database server, like the application one, will be hosted on an external service because a specialized service can perform maintenance and provide higher level of support, avoiding SafeStreets team to worry about these topics. A relational DBMS is preferred for the system, because the data is structured and query performance is important for data analysis. The team has also more acquaintance with the SQL technology, while have never work with a noSQL DBMS.

2.4 Runtime view

The following sequence diagrams show the most important use cases of SafeStreets business. It's important to note that the application server keeps track of the sessions with each client and each session saves the ID of the user so that it can be used in later operations on the database. For instance, UserID is used to retrieve the city of the authorities and municipality when they want to visualize the violations and interventions, respectively, in their jurisdiction area.

2.4.1 Report a violation

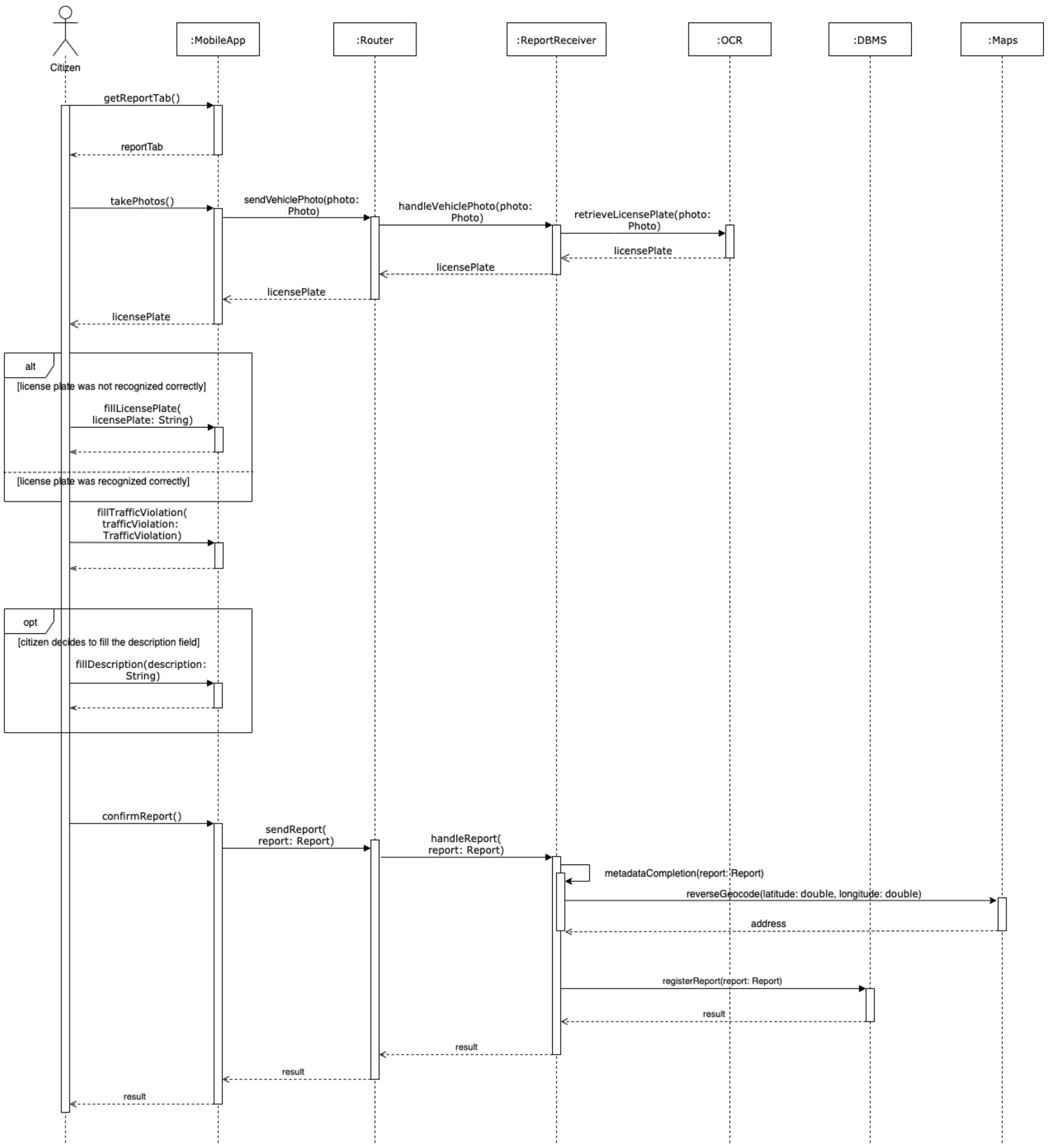

Citizen
:MobileApp
:Router
:ReportReceiver
:OCR
:DBMS
:Maps
getReportTab()
reportTab
takePhotos()
sendVehiclePhoto(photo: Photo)
handleVehiclePhoto(photo: Photo)
retrieveLicensePlate(photo: Photo)
licensePlate
licensePlate
licensePlate
licensePlate
alt
[license plate was not recognized correctly]
fillLicensePlate(licensePlate: String)
[license plate was recognized correctly]
fillTrafficViolation(trafficViolation: TrafficViolation)
opt
[citizen decides to fill the description field]
fillDescription(description: String)
confirmReport()
sendReport(report: Report)
handleReport(report: Report)
metadataCompletion(report: Report)
reverseGeocode(latitude: double, longitude: double)
address
registerReport(report: Report)
result
result
result
result

Figure 2.6: Report a violation sequence diagram.

In this sequence diagram the use case in which a citizen reports a traffic violation is shown. After detecting a violation in the streets, the citizen selects the report tab from the main tab of the SafeStreets mobile app. In this case, the report tab has no information to retrieve from the SafeStreets server, which is the case of other services, so the app opens the tab immediately. At this point, the citizen uses the app's internal photo camera service to take photos of the violation, making sure that the first photo contains the transgressor's vehicle with its license plate. After the citizen has completed this first part of the report, the app sends the first photo to the 'Router' component, which forwards it to the 'ReportReceiverManager'. This component uses the external OCR service to recognize the license plate of the vehicle from the photo it received. Then 'ReportReceiverManager' returns the license plate to the 'Router', and this sends the license plate back to the mobile app that updates the license plate field. At this point the citizen checks whether the license plate was recognized correctly, if not he inserts it manually. Then he chooses the traffic violation type, and he optionally fills the description field. The citizen confirms the report, and the mobile app sends the report with GPS coordinate to Router, which forwards it to 'ReportReceiverManager'. This component completes the report with metadata such as timestamp, GPS coordinates and address retrieved from the coordinates, userID, and a 'pending' status. Finally, it registers the report in the system's database.

2.4.2 Analyze aggregate data

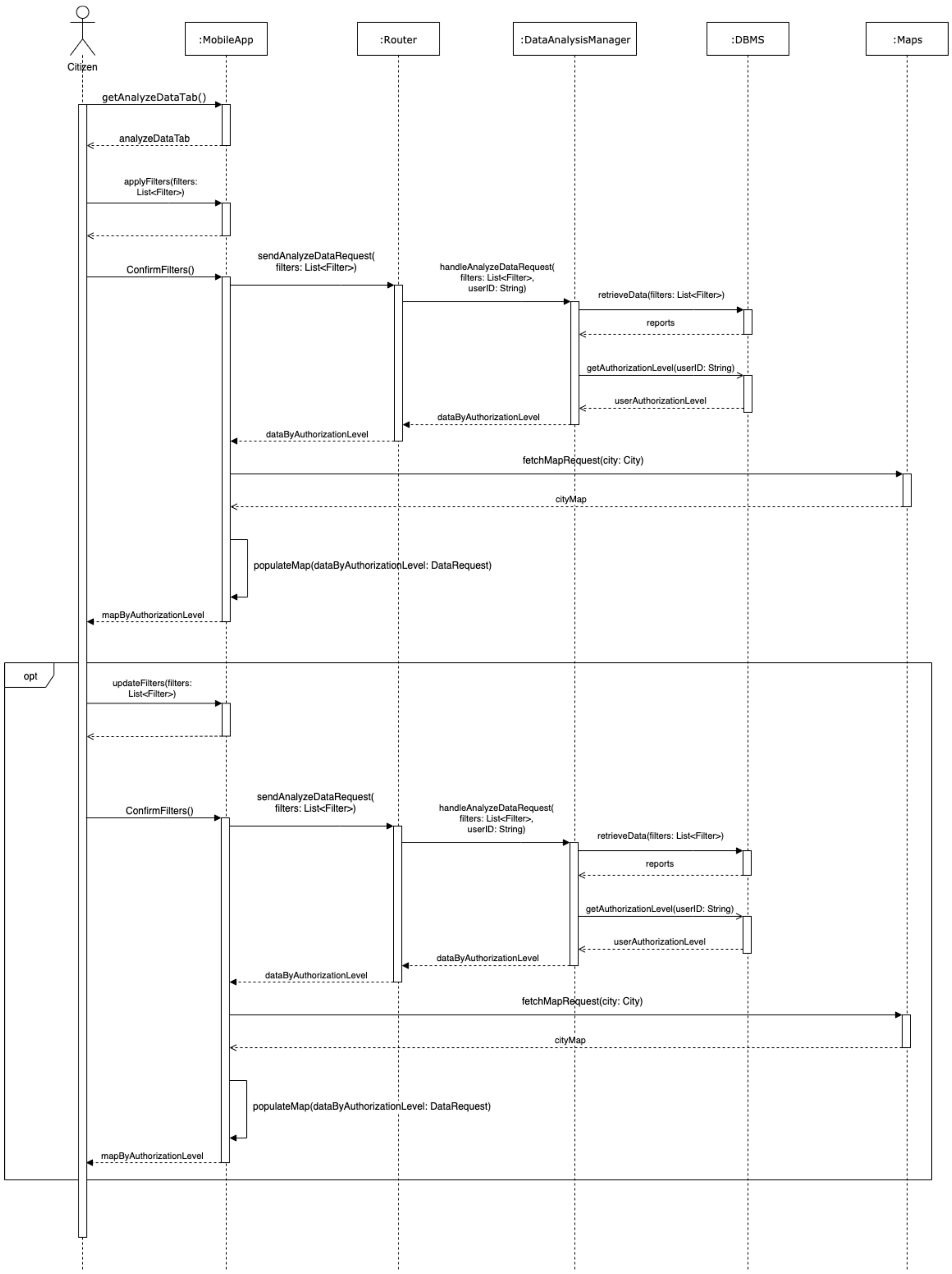

Citizen
:MobileApp
:Router
:DataAnalysisManager
:DBMS
:Maps
getAnalyzeDataTab()
analyzeDataTab
applyFilters(filters: List<Filter>)
ConfirmFilters()
sendAnalyzeDataRequest(filters: List<Filter>)
handleAnalyzeDataRequest(filters: List<Filter>, userID: String)
retrieveData(filters: List<Filter>)
reports
getAuthorizationLevel(userID: String)
userAuthorizationLevel
dataByAuthorizationLevel
dataByAuthorizationLevel
fetchMapRequest(city: City)
cityMap
populateMap(dataByAuthorizationLevel: DataRequest)
mapByAuthorizationLevel
opt
updateFilters(filters: List<Filter>)
ConfirmFilters()
sendAnalyzeDataRequest(filters: List<Filter>)
handleAnalyzeDataRequest(filters: List<Filter>, userID: String)
retrieveData(filters: List<Filter>)
reports
getAuthorizationLevel(userID: String)
userAuthorizationLevel
dataByAuthorizationLevel
dataByAuthorizationLevel
fetchMapRequest(city: City)
cityMap
populateMap(dataByAuthorizationLevel: DataRequest)
mapByAuthorizationLevel

Figure 2.7: Analyze aggregate data sequence diagram.

In this sequence diagram the use case in which a citizen wants to analyze aggregate data is presented. First of all, the citizen selects the analyze aggregate tab from the mobile app main tab. In this tab the citizen selects the filters that are useful for his analysis; examples of filters are the timestamp interval, the location and type of the violation. After the citizen has confirmed the filters for his analysis, the mobile app sends an analyze data request with the filters to 'Router'. This component forwards the request to 'DataAnalysisManager', which retrieves data according to the filters from the system's database. Then 'DataAnalysisManager' retrieves the authorization level of the citizen from the database using the userID retrieved from the session and sends back to 'Router' appropriate data according to the authorization level of the user. If the user is a citizen, like in this case, license plate, report submitter info and supervisor info are not sent back to the client, whereas if the user is an authority, all data of the reports are sent back so that the authorities can check the data more accurately, e.g. check the license plate of the most frequent transgressor. At this point, 'Router' forwards the data back to the mobile app, that fetches the map of the city the citizen is interested in and populates the map for visualization. After visualizing the map, the user can update the filters whenever he wants to, and the retrieval of the new map with the new filters is the same as in the above description for the first retrieval of the map. The update of the filters can be done zero or more times depending on the user's preference.

2.4.3 Violation validation

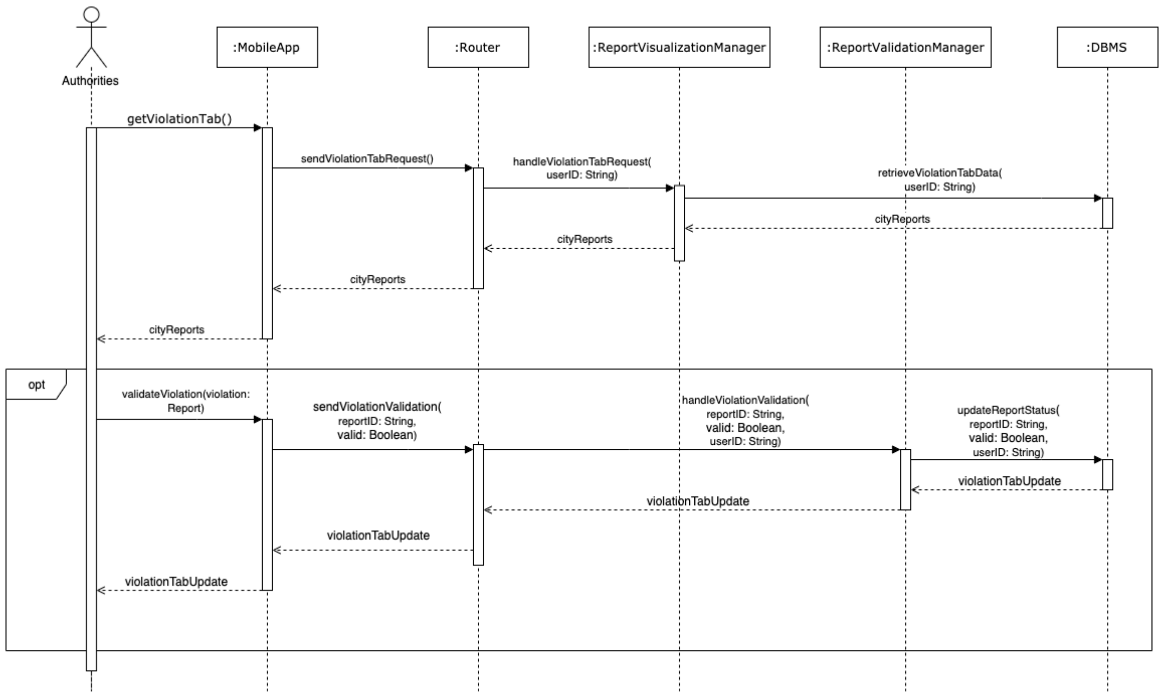
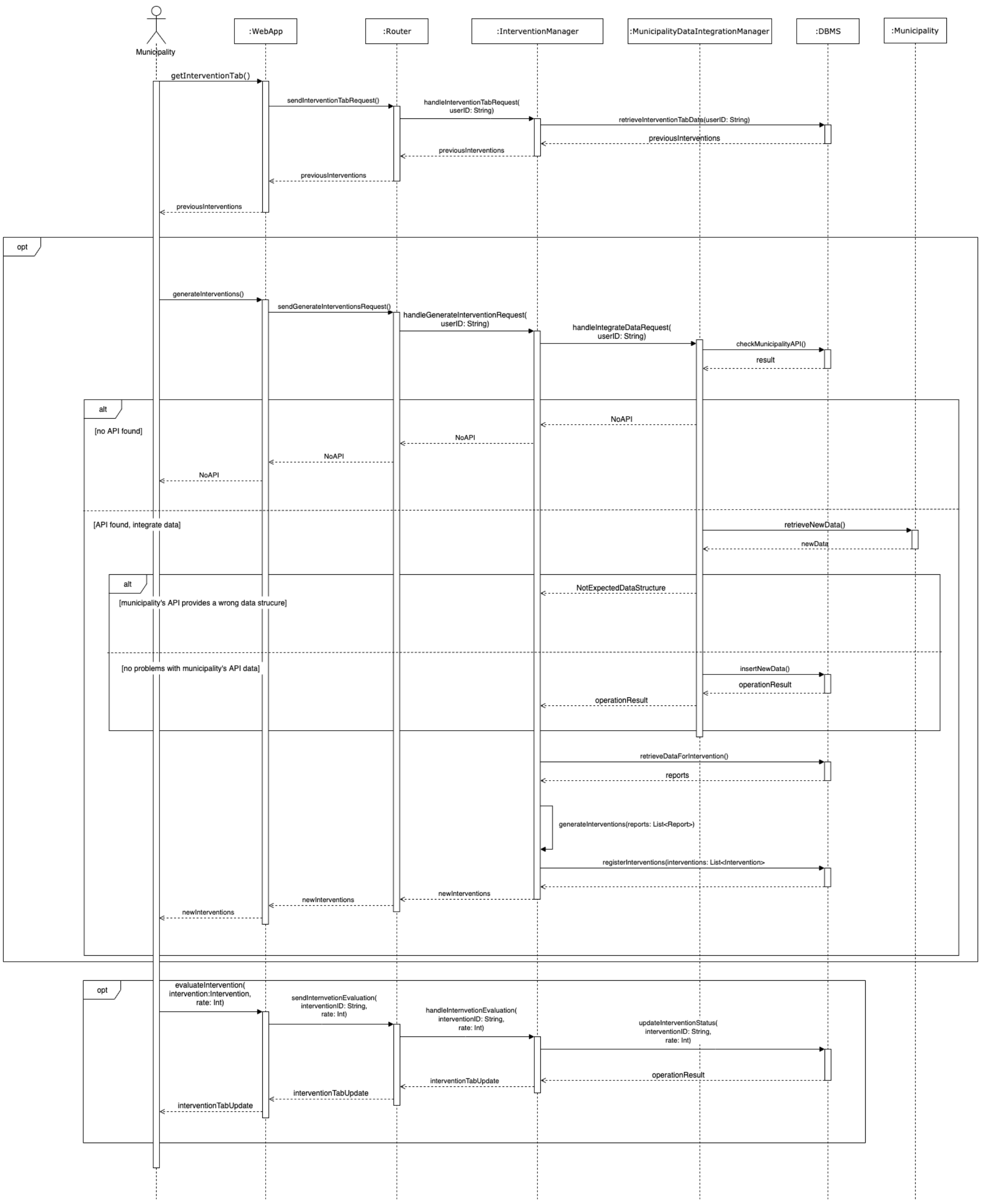

Authorities
:MobileApp
:Router
:ReportVisualizationManager
:ReportValidationManager
:DBMS
getViolationTab()
sendViolationTabRequest()
handleViolationTabRequest(userID: String)
retrieveViolationTabData(userID: String)
cityReports
cityReports
cityReports
cityReports
opt
validateViolation(violation: Report)
sendViolationValidation(reportID: String, valid: Boolean)
handleViolationValidation(reportID: String, valid: Boolean, userID: String)
updateReportStatus(reportID: String, valid: Boolean, userID: String)
violationTabUpdate
violationTabUpdate
violationTabUpdate
violationTabUpdate

Figure 2.8: Violation validation sequence diagram.

In this sequence diagram the use case in which an authority validates a violation report is shown. After the authority has selected the violation tab, the mobile app sends a request to retrieve data for the violation tab to ’Router’, which forwards the request to the ’ReportVisualizationManager’ component. This component verifies that the client that requested the violation tab is an authority (this passage is not shown in the diagram for simplicity, anyway in this case the actor sending the request is an authority, so the system continue with the process) and after that it retrieves the reports related to the city in the jurisdiction of the requesting authority, and finally sends back the reports of that city back to ’Router’, which forwards it back to the mobile app. Now the authority can visualize and select a report to validate, and this part can be done zero or more times depending on the number of reports to be verified. After validating/invalidating a report, the mobile app sends a violation validation message with the reportID and the validity of the report to ’Router’, which forwards it to ’ReportValidationManager’. The last component updates the status of the report, by changing the ’pending’ status to ’validated’/’invalidated’ and adding the authority’s userID to the supervisor field in the database, and returns the updated report as a violation tab update message to 'Router' (the diagram above assumes the success of the operation). Finally, 'Router' forwards the message back to the mobile app, which updates the changed report for visualization.

2.4.4 Suggestion evaluation

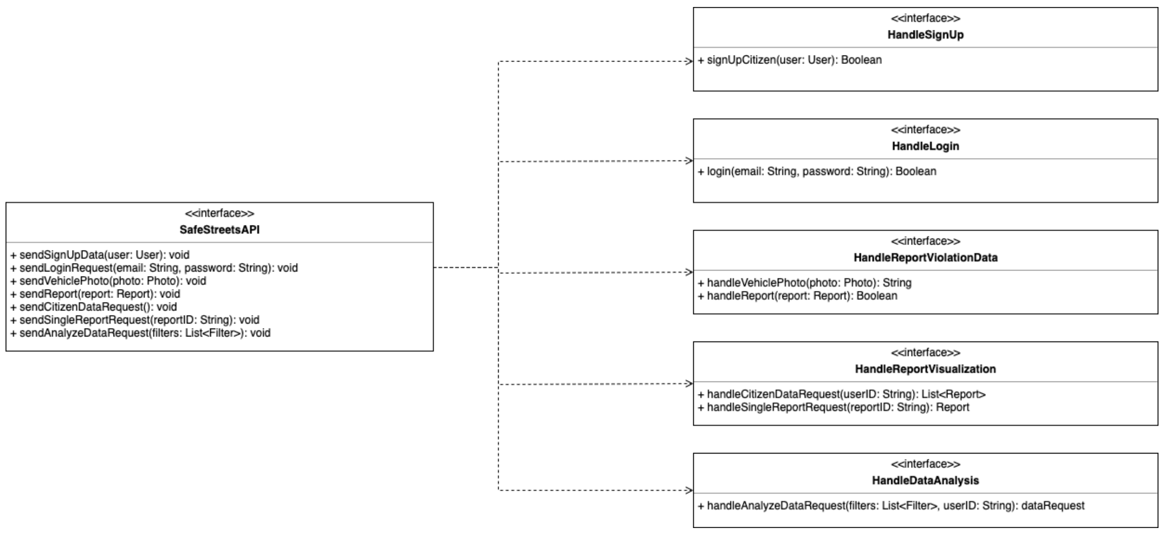


Municipality
:WebApp
:Router
:InterventionManager
:MunicipalityDataIntegrationManager
:DBMS
:Municipality
getInterventionTab()
sendInterventionTabRequest()
handleInterventionTabRequest(userID: String)
retrieveInterventionTabData(userID: String)
previousInterventions
previousInterventions
previousInterventions
previousInterventions
opt
generateInterventions()
sendGenerateInterventionsRequest()
handleGenerateInterventionRequest(userID: String)
handleIntegrateDataRequest(userID: String)
checkMunicipalityAPI()
result
alt
[no API found]
NoAPI
NoAPI
NoAPI
NoAPI
[API found, integrate data]
retrieveNewData()
newData
alt
[municipality's API provides a wrong data strucure]
NotExpectedDataStructure
[no problems with municipality's API data]
insertNewData()
operationResult
operationResult
retrieveDataForIntervention()
reports
generateInterventions(reports: List<Report>)
registerInterventions(interventions: List<Intervention>
newInterventions
newInterventions
newInterventions
opt
evaluateIntervention(intervention:Intervention, rate: Int)
sendInternvetionEvaluation(interventionID: String, rate: Int)
handleInternvetionEvaluation(interventionID: String, rate: Int)
updateInterventionStatus(interventionID: String, rate: Int)
operationResult
interventionTabUpdate
interventionTabUpdate
interventionTabUpdate

Figure 2.9: Suggestion evaluation sequence diagram.

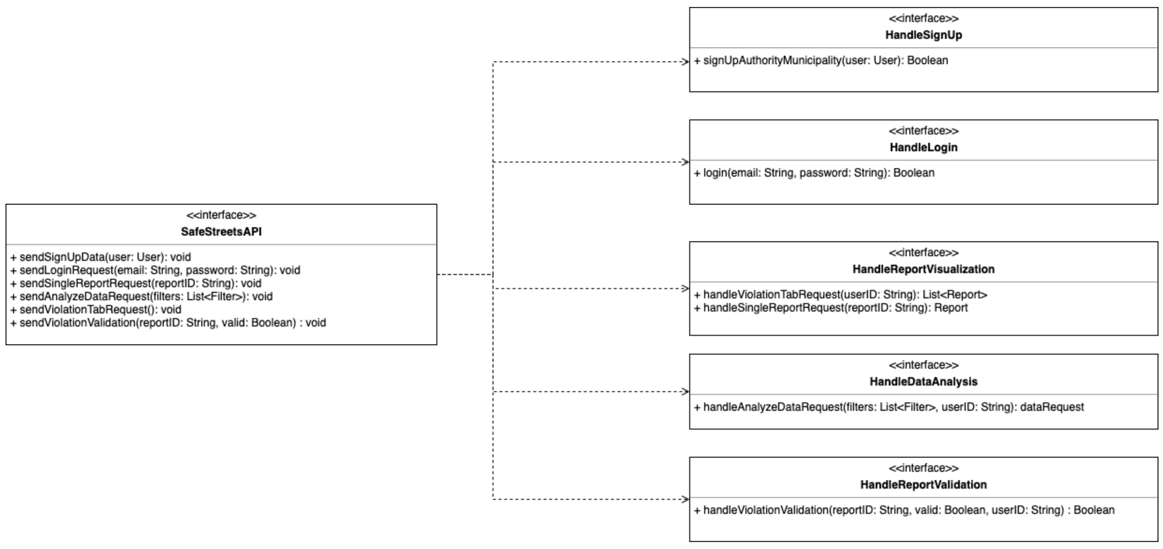
In this sequence diagram the use case in which a municipality generates and evaluates the interventions is shown. After the municipality has selected the intervention tab, the webapp of SafeStreets sends an intervention tab request to 'Router'. Then 'Router' forwards the request to the 'InterventionManager', which retrieves the previously generated interventions from the system's database after checking from session who the user is (so that can retrieve the municipality's city) and sends back to the webapp through the 'Router' component. At this point, the municipality can decide whether to generate new reports or not. If it decides not to generate new interventions, it can just visualize the previously generated internvetions, if any present, or even evaluate them. Whereas if the municipality decides to generate new interventions, the system sends a generate intervention request to 'Router', which forwards it to 'InterventionManager'. This component sends an integrate data request to 'MunicipalityDataIntegrationManager' to check whether the municipality provided an API or not. If no API is present then this data integration process is aborted, otherwise it tries to retrieve new data from the municipality. If the data provided by the municipality does not conform with SafeStreets API an error is generated, and 'InterventionManager' proceeds to generate interventions without any data integration, otherwise the new data is inserted into SafeStreets' database. Finally, 'InterventionMangaer' retrieves all relevant data from the database and generates a possible list of interventions that is stored in the database and sent back to the client via 'Router'. Now the municipality can inspect the newly generated interventions and decide whether to rate them or not. If it decides to rate them, it needs to give a score (e.g. between 0 and 5) as a measure of the quality of the suggestions. The score with the ID of the intervention is sent to 'Router', which forwards it to 'InterventionManager' to update the database with the new information. An update is then sent back to the client to keep the data between the client and the server synchronized.

2.5 Component interfaces



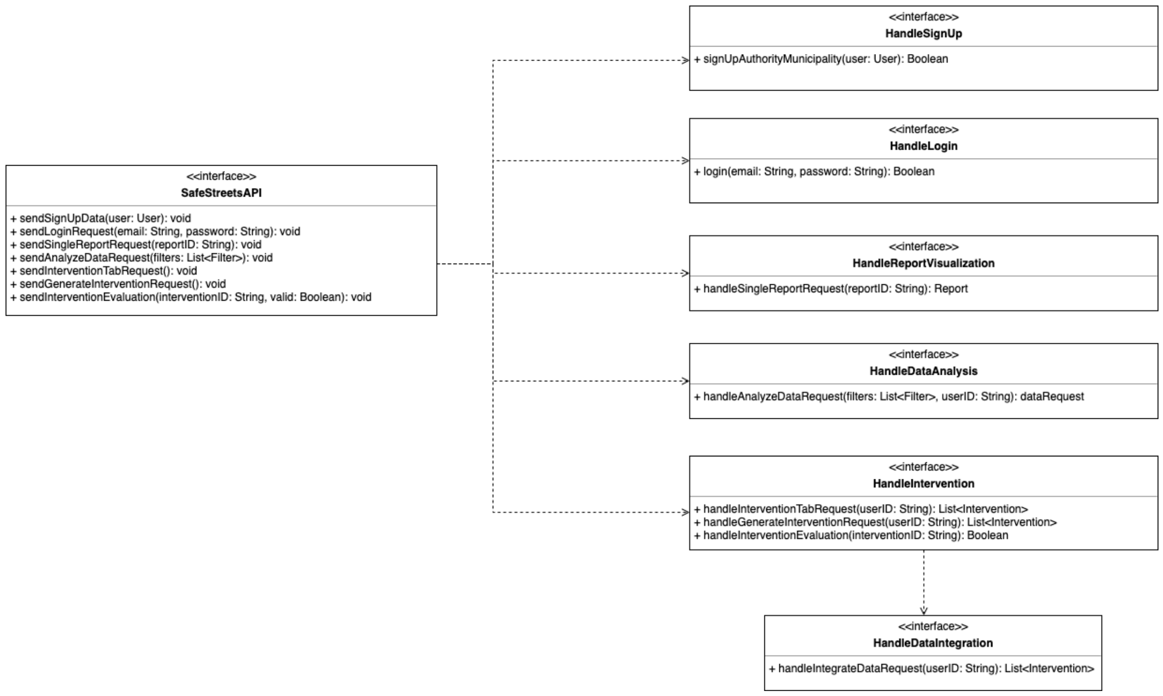
<<interface>>
HandleSignUp
+ signUpCitizen(user: User): Boolean
<<interface>>
HandleLogin
+ login(email: String, password: String): Boolean
<<interface>>
SafeStreetsAPI
+ sendSignUpData(user: User): void
+ sendLoginRequest(email: String, password: String): void
+ sendVehiclePhoto(photo: Photo): void
+ sendReport(report: Report): void
+ sendCitizenDataRequest(): void
+ sendSingleReportRequest(reportID: String): void
+ sendAnalyzeDataRequest(filters: List<Filter>): void
<<interface>>
HandleReportViolationData
+ handleVehiclePhoto(photo: Photo): String
+ handleReport(report: Report): Boolean
<<interface>>
HandleReportVisualization
+ handleCitizenDataRequest(userID: String): List<Report>
+ handleSingleReportRequest(reportID: String): Report
<<interface>>
HandleDataAnalysis
+ handleAnalyzeDataRequest(filters: List<Filter>, userID: String): dataRequest

Figure 2.10: Citizen component interfaces.



<<interface>>
HandleSignUp
+ signUpAuthorityMunicipality(user: User): Boolean
<<interface>>
HandleLogin
+ login(email: String, password: String): Boolean
<<interface>>
SafeStreetsAPI
+ sendSignUpData(user: User): void
+ sendLoginRequest(email: String, password: String): void
+ sendSingleReportRequest(reportID: String): void
+ sendAnalyzeDataRequest(filters: List<Filter>): void
+ sendViolationTabRequest(): void
+ sendViolationValidation(reportID: String, valid: Boolean) : void
<<interface>>
HandleReportVisualization
+ handleViolationTabRequest(userID: String): List<Report>
+ handleSingleReportRequest(reportID: String): Report
<<interface>>
HandleDataAnalysis
+ handleAnalyzeDataRequest(filters: List<Filter>, userID: String): dataRequest
<<interface>>
HandleReportValidation
+ handleViolationValidation(reportID: String, valid: Boolean, userID: String) : Boolean

Figure 2.11: Authority component interfaces.



<<interface>>
SafeStreetsAPI
+ sendSignUpData(user: User): void
+ sendLoginRequest(email: String, password: String): void
+ sendSingleReportRequest(reportID: String): void
+ sendAnalyzeDataRequest(filters: List<Filter>): void
+ sendInterventionTabRequest(): void
+ sendGenerateInterventionRequest(): void
+ sendInterventionEvaluation(interventionID: String, valid: Boolean): void
<<interface>>
HandleSignUp
+ signUpAuthorityMunicipality(user: User): Boolean
<<interface>>
HandleLogin
+ login(email: String, password: String): Boolean
<<interface>>
HandleReportVisualization
+ handleSingleReportRequest(reportID: String): Report
<<interface>>
HandleDataAnalysis
+ handleAnalyzeDataRequest(filters: List<Filter>, userID: String): dataRequest
<<interface>>
HandleIntervention
+ handleInterventionTabRequest(userID: String): List<Intervention>
+ handleGenerateInterventionRequest(userID: String): List<Intervention>
+ handleInterventionEvaluation(interventionID: String): Boolean
<<interface>>
HandleDataIntegration
+ handleIntegrateDataRequest(userID: String): List<Intervention>

Figure 2.12: Municipality component interfaces.

The above figures show the component interfaces of the application server that are actually used for each kind of user. The SafeStreetsAPI interface provides all the methods needed to access the services offered by the application server. Each method of this interface is associated by the 'Router' component with an internal handler interface provided by the component responsible of the requested service. Each internal interface handles the incoming requests and communicates with the DBMS for database modification. In the following, a description for each one of these interfaces is provided:

- HandleSignUp: this interface is responsible for the sign up of any kind of user; in particular if the user is an authority or a municipality, it verifies its identity by checking their PEC address into the Italian official PEC register.
- HandleLogin: this interface is responsible for authenticating the access of a user and establishing a connection session.
- HandleReportViolationData: this interface is responsible for recognizing the license plate of incoming vehicle photos, using an external OCR API; it also receives reports from the clients, completes them with metadata and stores them in the databases.

- HandleReportVisualization: this interface is responsible for sending all data regarding the reports of a specific citizen user, reports in the jurisdiction of the requesting authority for the violation tab or just one single report.
- HandleDataAnalysis: this interface is responsible for retrieving data that satisfies the filters of the incoming data analysis requests; for each kind of requesting user it sends back different amount of data for privacy security reasons, in fact both kinds of users can retrieve the location, time and type of violation of the reported transgressor, but only the authorities can retrieve the license plates and submitters' info, whereas citizen cannot.
- HandleReportValidation: this interface is responsible for receiving violation validations from authority users and update the database accordingly.
- HandleIntervention: this interface is responsible for sending the previously generated interventions list and generating new interventions when requested by the municipality, and it also receives intervention evaluations to update the database accordingly.
- HandleDataIntegretaion: this interface is responsible for retrieving data from the municipality's server and integrate it with data collected by SafeStreets, by inserting it in the main database.

2.6 Selected architectural styles and patterns

Three tier architecture

The system is developed following a classical three tier architecture, composed by presentation layer, application layer and data layer. This type of architecture is a common standard nowadays for medium size applications, because it allows to distribute work load between three tier of machines, decoupling the tiers neatly without compromising the functionalities. Every tier can be modified internally in a flexible way, without compromising the functionalities of other tiers (only the changes to interfaces between tiers are critical). Another important quality of this architecture style is the fact that data tier is accessible only from application server, so the system is more secure because clients are unable to communicate directly with the database.

RESTful API

The clients interact and send requests to the SafeStreets servers to a REST API, that provides a web interface common between all the clients. Using a REST API allows to have a de-coupling between clients and servers functionalities. The clients only need to call the correct endpoint for retrieving data or perform an action, expecting a certain output structure but without knowing the computations that happen behind the curtains. This also helps the maintainability of both servers and clients, because server can be updated without affecting directly the clients if the API endpoints are not modified (but can be expanded).

To achieve these goals, a REST API need to follow these properties:

- **Uniform interface:** API URLs have similar structure and query attributes, so that it's easier to understand how to use the API interface and what each endpoint does.
- **Stateless:** API returned data by a same endpoint has always the same structure, the server is not aware of the specific state scope of the call or the state of the client, it just provides data that can be used by the client as they please.
- **Client-Server model:** client and server evolve in a separate way and communicate only through API endpoints.
- **Caching:** a client-side or server-side cache system can be used to improve performances and speed up interactions, especially in case of multiple same requests in a very short period of time.
- **Layered architecture:** the system uses a standard 3-tier architecture. Clients communicate with the servers through API calls, where specific backend code handle each request. These handlers call the managers that have the functions used to interrogate and manipulate the data layer, to fetch the data that clients require to display or perform specific actions, like login.

2.7 Other design decisions

Relational database

A relational database is chosen to store the data managed by the system, as the data can be structured with a fixed schema without the necessity of "optional" field that can be frequently null and clutter the database. Another reason to choose a SQL database instead of a noSQL one is that SQL is a mature and optimized query language that the team is acquainted with and knows how to optimize big queries required for data analysis functionality; using a relational database with proper indexing should guarantee fast and reactive performance for every operation. One disadvantage of the usage of a relational database is that horizontal scaling is really hard to perform and so vertical scaling is the primary choice in case the system need more power for handling its data, while noSQL can scale horizontally in an easy way and have less upgrading costs.

Asynchronous programming over synchronous multi-threaded approach

Application servers should prioritize an asynchronous programming model over a multi-threaded approach. Asynchronous approach maximize throughput and scalability of the web server, because allows multiple I/O operation on a single thread, so that there is no thread context switching overhead. In this type of architecture, the system can perform all operations without blocking the thread, because it can pause tasks and perform callbacks when a task required for another one is terminated, performing other operations in the meantime. Note that asynchronous programming could also work on multiple threads to optimize the CPU utilization of the machine that host the server, but the various task are combined in an efficient way on each thread and multiple client requests can be executed on each one thanks to the non blocking approach, instead in a more classical synchronous multi-thread each task that performs I/O need a thread, degenerating quickly in a huge amount of threads in case of multiple request in a small time.

User interface design

3.1 User interface mockups

In this section a basic idea of how the user interface should look like when shown. It presents the basic login and signup tabs, the main tab showing the usable services, and a specific tab for each distinct service. The mockups are very similar for different kinds of users, only the services provided differ. For instance, 3.5 provide the services that only the citizens can use, whereas 3.10 shows the additional services that the authorities and municipality users can use.

Mobile app common interfaces

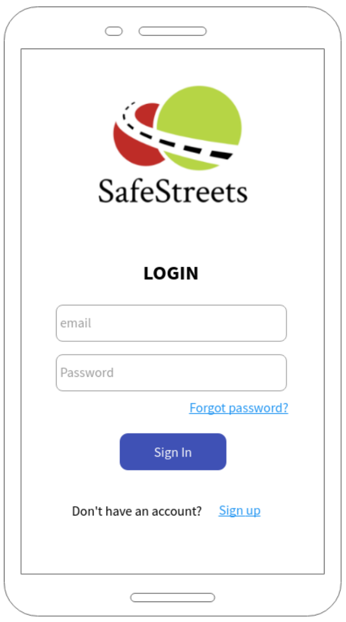
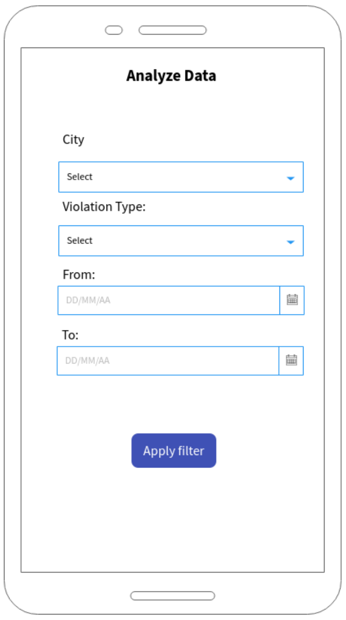


Figure 3.1: Login tab.



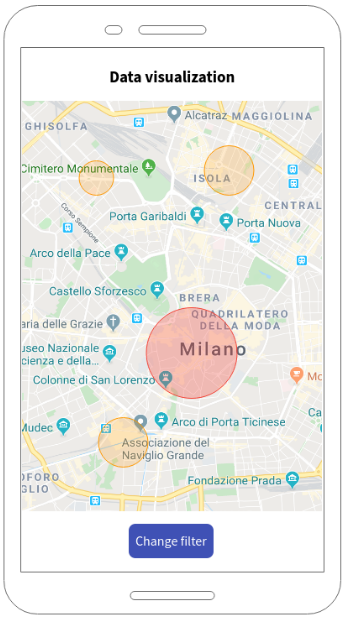
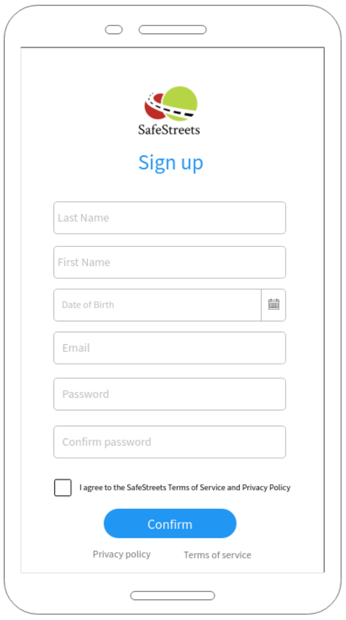


Figure 3.2: Analyze data. Figure 3.3: Data visualization.

Citizen interfaces



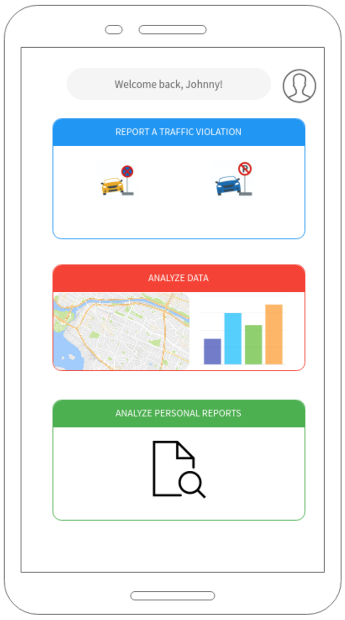


Figure 3.4: Citizen signup tab. Figure 3.5: Citizen main tab.

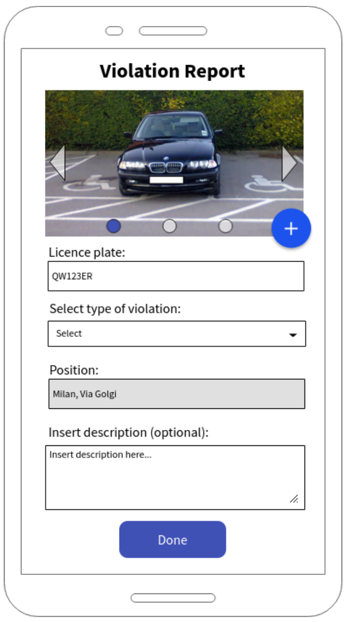

Violation Report
Licence plate:
QW123ER
Select type of violation:
Select
Position:
Milan, Via Golgi
Insert description (optional):
Insert description here...
Done

Figure 3.6: Report tab.

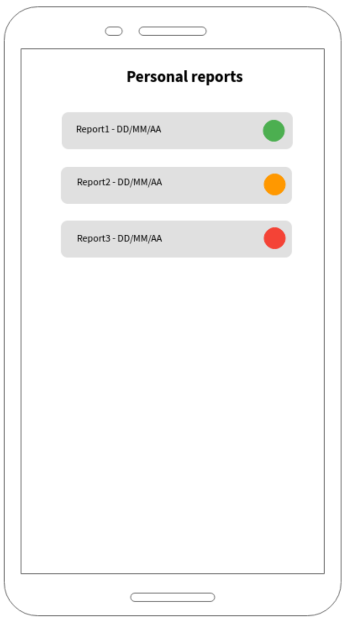


Figure 3.7: Personal reports.

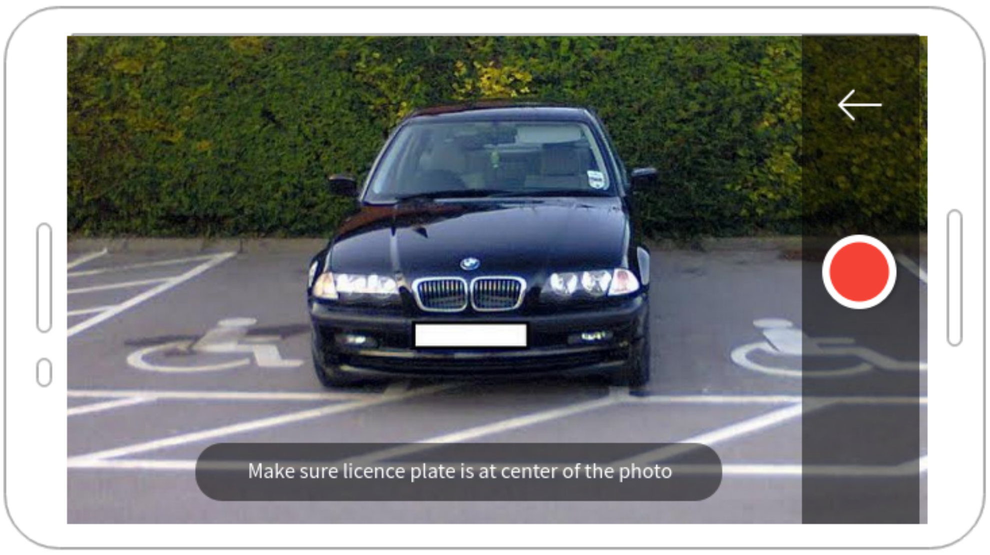


Figure 3.8: Violation photo.

Authorities interfaces

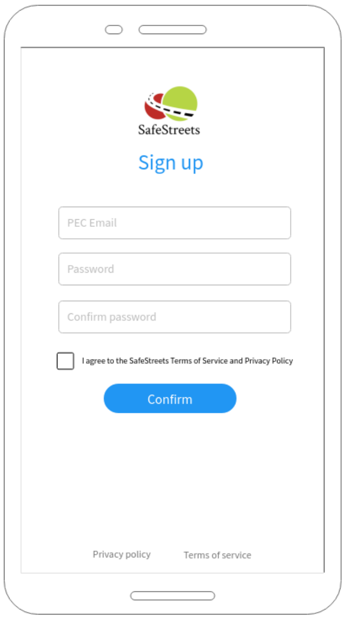


Figure 3.9: Authorities signup tab.

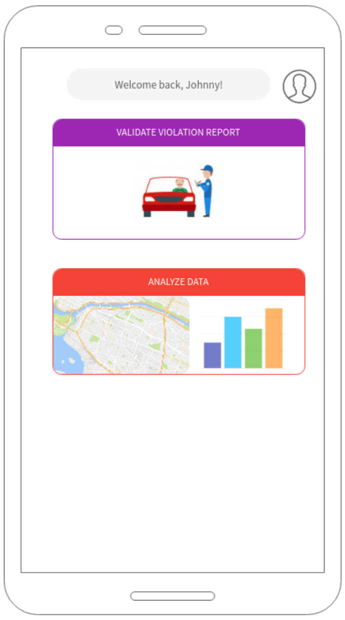


Figure 3.10: Authorities main tab.

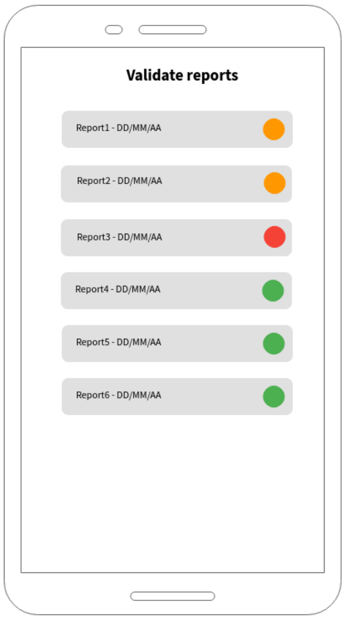


Figure 3.11: Validate reports.

Municipality interfaces (Web app)

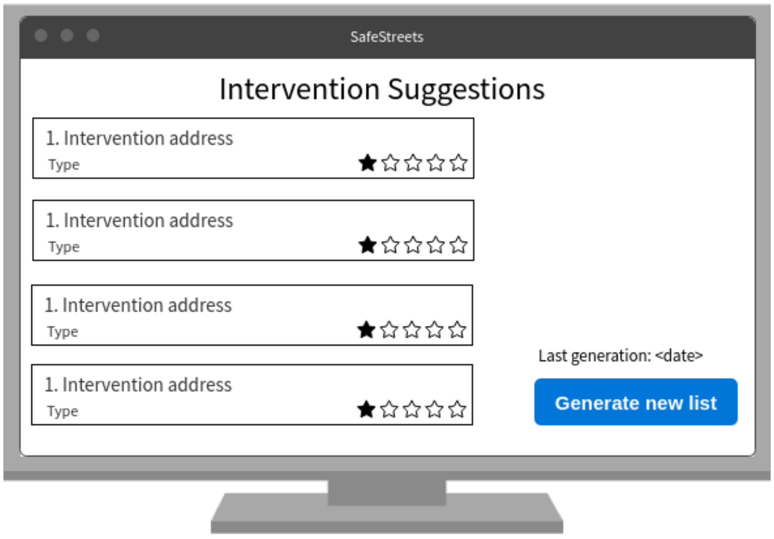


Figure 3.12: Intervention suggestions.

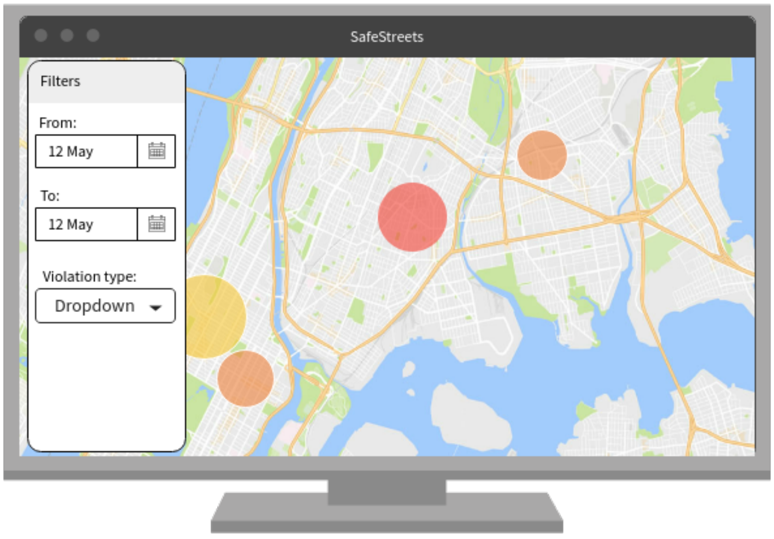


Figure 3.13: Municipality data analysis.

Requirement traceability

To fulfill the goal of the software to be, it's important that the requirements of the system (defined in RASD) are completely satisfied by the designed components described in component view section. Therefore, in this chapter is reported accurately which components are involved in each requirements and how they participate in their realization with a discursive description. For a better description on how the components actually work and interact which each other in detail, refer to component view section.

It is important to note that the router component is used in all interactions between clients and application server because it handles all requests received through the API endpoints, and calls the components responsible for the functionality related to the request and also return a response to the client. Its behaviour is consistent for each requirement and because of this it will be omitted from the analysis.

Also LoginManager and RegistrationManager are not listed in the traceability analysis. Even if essential for all users to access the functionalities of the system, requirements are focused on the actual services that SafeStreets provides to its customers, so login and registration only participate in them indirectly. In fact, all users need to register and then login to access the system, and all active accesses are then stored in a session database and the various managers can retrieve from it who made the request and which type of authorizations the client has.

R1. When a traffic violation is reported, the system must allow the authorities of the city where the violation happened to verify it.

- **ReportValidationManager:** allow authorities to validate or invalidate the reports.

R2. Reports about traffic violations need to be stored in a database, also associating them to the user profile who sent them.

- **ReportReceiverManager:** after the validation and completion of the report is done, this component calls an insert on the database, storing it permanently in the database server storage.

R3. The system needs to integrate reported traffic violations with data from the municipality when provided and analyze all the data with an algorithm to find possible interventions to address the detected main cause of the violations.

- **InterventionManager:** when the municipality has an API inserted associated it calls the MunicipalityDataIntegrationManager to integrate the data. At the end, the manager computes the possible interventions, pass them to the client and store them in the database for further consultations.
- **MunicipalityDataIntegrationManager:** when notified by the InterventionManager, it connects to the municipality's API to download all new data that is not already integrated, then notifies the InterventionManager to continue the operation.

R4. The system must be able to generate a list of possible interventions when a municipality requires it for evaluation, based on stored reports.

- **InterventionManager:** this component allow every municipality registered and logged in the system to generate a list of interventions based on the reports stored in SafeStreets database. The reports stored in the database when the algorithm is executed include the one provided from the municipality through its API, if present.

R5. Traffic violations data must be available for public consultation with different levels of visibility, according to user's level of authorization.

- **DataAnalysisManager:** handles the requests about aggregated data, which can be then visualized by the client in the most appropriate way. The returned data is different between the different kind of accounts, so the manager contains logic for discriminating the user that make the request and accordingly omits some details if the user hasn't the permission to see such data. To obtain the data requested by the client, it performs a query on the database constructed from the filters chosen by the user.

- **ReportVisualizationManager:** handles requests for single reports visualization and request for lists of interventions useful for the user. For doing this, it checks which user made the request and, based on the account info and permissions, return only the report that the user can actually access.

R5.1. The system should allow a normal user to access only to aggregated data, without the license plate of the reported vehicle or the personal info of the user who reported the violation.

- **DataAnalysisManager:** omits from citizens requests the data about license plates, to preserve privacy of transgressors and avoid potential social issues.
- **ReportVisualizationManager:** allows citizens to only check the report which themselves have generated, all others reports are completely inaccessible to preserve privacy of other users and avoid social issues.

R5.2. The system should allow the authorities to access the data without any limitation.

- **DataAnalysisManager:** allows authorities to analyze freely all data about reports, so that they have all the necessary data to improve their efficiency and punish quickly the transgressor.
- **ReportVisualizationManager:** allows authorities to retrieve a list about all reports of violations happened in their city (bound on account info when registered to the server). From that list they are able to check all these violations in detail.

R6. The system must ascertain the identity of a public institution that want to register to the system from its PEC, completing the account with the necessary data.

- **RegistrationManager:** contains a function for communicating with an external API of the public register of PEC addresses, passing to it the PEC address sent in the registration form by the client. It then wait the response and analyze the data received to identify the institution from the PEC domain, associating it to the right type of account.

R7. The system must receive correctly the reports about the various traffic violations from the users, discarding reports with incomplete data.

- **ReportReceiverManager:** this component manages the validation of the data inserted in the report, its completion with metadata like the reference to the user who submitted it and the timestamp. When a report sent to the server is incomplete, returns an error to be displayed on user device.

R8. The system must be able to recognize the license plate of the vehicle involved in the violation from the first photo of the report.

- **ReportReceiverManager:** when a citizen finishes to take pictures about the violation, the first photo is sent to the server and handled by this manager, that communicate with the OCR service and wait for the output, that is then redirected to the user for confirmation.

Implementation, integration and test plan

5.1 Introduction

The system is subdivided in various subsystems that are connected together through API interfaces but that are fairly independent between each other. In fact the mobile apps for both Android and iOS and the web app can be developed in parallel with the application server, because they will communicate without knowing the state of each other (stateless architecture). The only important aspect is to define clearly the API and the retrieved data before starting the implementation of the subsystems, so that the devices will communicate easily with the server when an integration test is performed.

The various external services used by the system are supposed to be completely functional and always available for the correct functionality of the application, so testing will performed mainly on the output of the manipulation of the data returned from these services.

As a preliminary step for choosing the implementation order and testing importance of each module we performed a qualitative analysis to estimate the complexity of each module and their relevance for SafeStreets' customers.

Functionality	Relevance for customers	Implementation complexity	
		server	client
Login and registration	low	medium	low
Report submission	high	medium	medium
Report detail visualization	high	low	low
Aggregated data visualization	medium	medium	high
Report validation	medium	low	low
Intervention list generation	high	high	low

5.2 Implementation plan

From the previous analysis it emerges that the components that are more difficult to implement are the ones relative to the aggregated data visualization and intervention list generation functionalities. These components would require the higher amount of testing in order to assure that the system will work flawlessly with the minimal amount of bugs.

As first step of the implementation of SafeStreets system, the underlying database must be created. A small database populated with some dummy data should also be developed, following the same structure of the production one. This database will allow the team to test easily the API of the server, because the output of a query will be always known if after each test any modification done on this database is reversed, so that it remains untouched.

After database implementation, the next step is to develop the server API. Defining clearly the API endpoints, the input they receive and the output they produce will allow to start building the various components that handle the logic behind the API calls, based on the received parameters. Also clients could be developed in parallel as they only display data and call the defined endpoints, they only need to satisfy the required input defined in API endpoints and working on manipulating the data that is expected back, which structure is also defined by the API definition (along with possible errors). Because of these considerations, clients and server can be developed and tested in parallel by two separate teams, that have competence with the respective technologies of these subsystems.

Concerning the application server, based on the relevance and complexity of the functionality, the following order of implementation should be followed to minimize the effort and build a cohesive system. For each functionality, an high-level implementation guideline is also provided to give a general idea of the system to the developers' team and interested stakeholders:

- **Login and registration:** standard in all types of web applications, this functionality should be pretty fast to implement because of already existing plugins and libraries. The only difficulty in the implementation is caused by the automatic recognition of municipalities and authorities performed by the components, done by querying the Italian public PEC register (https://www.indicepa.gov.it/documentale/index.php). The easier way we found at the time of writing is to download periodically the list of all registered PEC addresses (url: https://www.indicepa.gov.it/public-services/opendata-read-service.php?dstype=FS&filename=pec.txt), that can be used as a .csv file using tab as separator, store it in the server and check if the PEC domains given by authorities and municipalities exist and are associated to "Comuni e loro Consorzi e Associazioni" in the "Tipologia Istat" column. Implementing this component will allow to exploit the session, registered in the server after each client signs in, and this is important in about all functionalities because accessible data and services differs based on account type.

- **Report submission:** this functionality is essential for the whole system, without the reports sent by the user all others functionalities would be useless because of the absence of data. Considering this fact, this functionality needs to be implemented first in the server and exhaustively tested to check that valid report are received, completed and stored in the database correctly, while invalid ones are refused returning a proper error, following the API definition. This component is also tied to the external OCR service, so it's important to choose a quality service with an easily accessible interface for performing the license plate recognition. Also, when receiving the report it communicates with the Map service in order to complete the metadata associated with the report.

- **Report detail visualization:** this functionality is pretty simple to implement server-side and not time consuming. The component related to this functionality handles two kinds of requests: the first one is about the list of reports accessible by the requesting user and the second one is about the request of a single report. This can be done respectively by checking what type of account made the request, by checking the active sessions database, then adapt the query to perform on the main database based on the account info for retrieving the accessible list of reports, instead for single report access the component just needs to check with some predicates if the client can retrieve that report. Moreover, this functionality is essential for authorities registered to the system, as their main focus is to check the reports and their validity to efficiently patrol the city and punish the transgressors, so it's advisable to implement it early.

- **Intervention generation:** one of the most complex functionality of the system, shouldn't be implemented too late because it's the target service for a whole type of stakeholders (municipalities) and because of this needs proper testing before software release. Both InterventionManager and MunicipalityDataIntegrationManager components are involved in this functionality realization and need to coordinate together to reach the goal. Firstly, MunicipalityDataIntegrationManager must be implemented, because it's used by InterventionManager before starting the main computation of the interventions. Data integration is performed through the municipality's API that made the request for interventions, so the account info must be retrieved through checks on session and main databases. If the API is consistent with the guidelines defined by SafeStreets, the component will be able to retrieve the data and then integrate it in the database with an insert operation, otherwise it will return an error. In either cases, the InterventionManager starts the algorithm for finding the interventions relative to the municipality's area. The discriminating factors for defining consistent and valid interventions are the density of the violations in a relative small area, then the presence of a certain percentage of a violation type or a combination of them, these two pieces of info allows the possibility to define sensible intervention for the analyzed situation.
- **Report validation:** this feature is quite simple to implement, as the only thing that the relative component should do is to update the affected report tuple in the database and add the authority's id to the supervisor field (returned by checking session and main databases like other components). This functionality is not really important customer side, because it only allows the clients to better organize the visualization of the report lists as the validation

in the system holds no legal value and the authorities need to actually check the violations in place, but it's really important to provide accurate results in aggregate data visualization, as only validated reports and the one imported by the municipalities' API will be used for the statistics.

- **Aggregated data visualization:** complex feature both client-side and server-side. It's implemented as last functionality because it's not strictly necessary for fulfilling the main service of the system of allowing the violation reporting flow from citizens to authorities, but will provide interesting data for both citizen, authorities and municipalities, that can be used to have a better understanding and awareness of the situation in their city. It's also implemented as last because it indirectly depends from other services, because it queries the reports sent by citizens that are only validated by an authority, and discriminate the user account type based on the active session to provide only the data that can be visualized by that client. To implement this functionality, the DataAnalysisManager must be able to build a query command from the filters passed by client and also filter the results to include only the validated reports. All the reports are then sent to the client that made the request, that contains the logic for displaying in a practical way the structured data received, with the possibility to also display the data on a map, through interaction with the maps external service API.

5.3 Integration plan

In this section is described in detail the constraints on the integration of the modules and their optimal integration plan. The integration process can be grouped and ordered in the following way:

1. integration of components with the DBMS
2. integration of components with (other) external services
3. integration of components of the application server
4. integration of the front-end applications with the back-end server of SafeStreets

The design was studied to avoid coupling of the components, so that it's easier to test, modify and integrate the components. After the implementation of the components is complete, the integration plan can be pretty flexible and summarized as follows:

Integration of components with the DBMS
In this first phase of the integration plan every component that retrieves, inserts, updates data from the database is integrated with the chosen DBMS. The components involved in integration of this group are the following:

- SignUpManager
- LoginManager
- ReportReceiverManager
- ReportVisualizationManager
- ReportValidationManager
- DataAnalysisManager
- InterventionManager
- MunicipalityDataIntegrationManager

Integration of components with (other) external services
This group of components need to be integrated with additional external services, different from the DBMS, to completely fulfill their functionality:

- ReportReceiverManager $- >$ OCR service
- MunicipalityDataIntegrationManager $- >$ municipality service (API interface for retrieving data)

[Note: $- >$ means ”uses” in above relations]

Integration of components of the application server
In this group, integration between internal components of the application server are carried out, so that the complete application server can be built from the single modules and could be accessed by the clients in all its functionalities:

- SignUpManager $- >$ Router
- LoginManager $- >$ Router
- ReportReceiverManager $- >$ Router

- ReportVisualizationManager $->$ Router
- ReportValidationManager $->$ Router
- DataAnalysisManager $->$ Router
- InterventionManager $->$ Router
- MunicipalityDataIntegrationManager $->$ InterventionManager

[Note: $->$ means ”integrated into” in above relations]

Integration of the front-end application with the back-end server of SafeStreets
At the end of the process, server and clients can be test integrated to check if all the system processes are working correctly. The client applications make requests to the SafeStreets server API, so the two subsystems need to satisfy the API schema defined as first step of the implementation to be able to communicate without errors.

5.4 Test plan

During the implementation, given the relatively small components we apply a bottom-up test strategy. Following the bottom-up definition, we will start by unit testing the components of the application server that don't need stabs for their functionalities, so that we can use implement only the drivers for simulating interactions with these components. When the components pass their relative tests, it's possible to start the implementation of the modules that use that components so that we can substitute them to the drivers for integration tests. Therefore system's components must be progressively tested and integrated together, but first of all each component must be unit tested, so that it's possible to recognize if a single component is bugged or faulty. Integration tests will be performed after unit tests, providing the certainty that the flow of messages through the interfaces and the final output of the operations are consistent with the expected results.

As mentioned before, a small database with fictitious data should be implemented in first steps of implementation, so that it's possible to test the components more easily and to discover as soon as possible any issue. It's important that tests revert the operations performed on this test database after the test is completed, otherwise the modifications will alter the results of successive runs of test suites.

In the application server, the most important tests are relative to transactions done on the database and the data retrieved from database with the queries, these tests assure that from the data passed from the API request is possible to retrieve or modify the data correctly with the implemented query command. After these tests, it's also important to test the final output of the operations done by each component, to check if the output is consistent with the returned data schema of the API and that the computation output reflects the results expected by the requirements of the system.

In the clients a similar approach will be followed. It will be tested that from the UI interface the data inserted by the user is consistent with the data displayed on screen (its input) and that it's correctly encoded following the target API endpoint schema. Other test suites instead will assure that when receiving some data reflecting the API output schema, the client will be able to display this data correctly on the user screen.

Once the whole system is integrated, a final system test must be performed to verify the functional and non-functional requirements of the system. In particular, given the possibility of huge requests on the servers when the system is deployed, it is important to carry out a performance testing to identify bottlenecks affecting response time, utilization and throughput.

Effort Spent

Andrea Falanti:

Document section	Hours
Introduction	1.25
Architectural design	13
User interface design	0
Requirement traceability	2.75
Implementation, integration and test plan	5.75
Total	22.75

Andrea Huang:

Document section	Hours
Introduction	0.25
Architectural design	15.25
User interface design	0
Requirement traceability	0.75
Implementation, integration and test plan	2
Total	18.25

Note: User interface design effort time was added in RASD, because we share the same structure between the two documents and we have updated the effort there for RASD version 1.1.

References

- A quick introduction about advantages of an asynchronous programming in web context: https://codewala.net/2015/07/29/concurrency-vs-multi-threading-vs-asynchronous-programming-explained/.
- Italy public register of PEC addresses: https://www.indicepa.gov.it/documentale/index.php.