



POLITECNICO DI MILANO
SCHOOL OF INDUSTRIAL AND INFORMATION ENGINEERING
Master of Science in Computer Science and Engineering
Software Engineering 2 Mandatory Project

Safe Streets.

Design Document

Authors:

Rosetti Nicola
Sartoni Simone
Torri Vittorio

Academic Year 2019/2020

Milano, 09/12/2019
Version 1.0

Contents

1	Introduction	4
1.1	Purpose	4
1.2	Scope	4
1.3	Definitions,Acronyms,Abbreviations	4
1.4	Reference Documents	5
1.5	Document Structure	5
2	Architectural Design	6
2.1	Overview: High-level components and their interaction	6
2.2	Component view	6
2.3	Deployment view	11
2.4	Runtime view	14
2.5	Component interfaces	21
2.6	Selected architectural styles and patterns	25
2.6.1	Model-View-Controller Pattern	25
2.6.2	Facade Pattern	25
2.6.3	Publish-Subscribe Pattern	25
2.6.4	Thin client	25
2.6.5	RESTful architecture	25
2.6.6	Databases	26
2.7	Other design decisions	26
2.7.1	Database design	26
2.7.2	Consistency and update strategies among replicas	27
3	User interface design	28
4	Requirements Traceability	38
5	Implementation, integration and test plan	43
5.1	Implementation and unit testing	43
5.2	Integration testing	45
5.3	System testing	45
6	Effort spent	50
7	References	53

List of Figures

2.1	UML Component Diagram	7
2.2	UML Component Diagram for <i>Router</i> component	8
2.3	UML Component Diagram for <i>ReportManager</i> component	9
2.4	UML Component Diagram for <i>AccessManager</i> component	9
2.5	UML Component Diagram for <i>StatisticsManager</i> component	10
2.6	UML Component Diagram for <i>DataCollectionManager</i> component	10
2.7	UML Deployment Diagram	12
2.8	Informal architectural view of the system	13
2.9	UML Architectural Sequence Diagram for <i>SendReport</i> use case	15
2.10	UML Architectural Sequence Diagram for <i>UserLogin</i> use case	16
2.11	UML Architectural Sequence Diagram for <i>AgentCheckReport</i> use case	17
2.12	UML Architectural Sequence Diagram for the <i>CheckStatistics</i> use case	18
2.13	UML Architectural Sequence Diagram for <i>User Registration</i>	19
2.14	UML Architectural Sequence Diagram for the <i>Suggestions</i>	20
2.15	UML Interfaces Diagram	22
2.16	UML Class Diagram	24
2.17	Entity-Relationship diagram for the operational database	26
2.18	Dimensional Fact Model diagram for the data warehouse	27
3.1	Mobile App mockup for initial page	29
3.2	Mobile App mockup for main page	29
3.3	Mobile App mockup for the registration	29
3.4	Mobile App mockup for the registration confirm	29
3.5	Mobile App mockup for the login screen	30
3.6	Mobile App mockup for the report page	30
3.7	Mobile App mockup for statistics	30
3.8	Web mockup for reports page	31
3.9	Web mockup for suggestions page	32
3.10	Web mockup for the report analysis page	33
3.11	Web mockup for registration page	34
3.12	UML Activity Diagram for the User Mobile App navigation	35
3.13	UML Activity Diagram for the MunicipalitySupervisor Web App navigation	36
3.14	UML Activity Diagram for the MunicipalityAgent Web App navigation	37
5.1	Gantt chart for the implementation process	44
5.2	Integration diagram for suggestions	46
5.3	Integration diagram for statistics	46
5.4	Integration diagram for access, data collection, report management and presentation	47
5.5	Integration diagram for dispatching and notification	48

List of Tables

4.1	Table for mapping requirements to components	38
5.1	Classification of components based on customer importance and implementation difficulty	43
6.1	Nicola Rosetti's effort table	50
6.2	Simone Sartoni's effort table	51
6.3	Vittorio Torri's effort table	52

1 Introduction

1.1 Purpose

The purpose of this document is to underline and explain in detail what are the design choices for the deployment of the SafeStreet Application. The analysis has been made at different levels to underline different views of the same system. Different levels are identified through the document; here they are just mentioned:

- the high level architecture;
- the components that actually builds the system;
- the deployment view of the system;
- the messages exchanged among the components and the interfaces they offer;
- the pattern and technologies exploited for the system;
- the User Interfaces provided by the system.

1.2 Scope

Here the scope of the system is briefly summarized.

SafeStreets is an application system with the aim to support municipalities in recording parking violations, providing also additional functions for traffic tickets emission, statistics and suggestions.

The citizen can use the mobile app to send reports about parking violations, after having been registered, and to consult statistics about violations and accidents.

Municipality agents can use the web application to receive the reports by the users, analyze them and possibly emit traffic tickets. They can also consult the previously mentioned statistics.

Municipality supervisors can also receive the *suggestions* elaborated by the system to improve the safety on the streets and can also consult reserved statistics, containing personal data about the most egregious offenders.

The system is supported by APIs provided by the municipalities and by APIs provided by a map service, as explained in the RASD and in the following sections of this document.

1.3 Definitions, Acronyms, Abbreviations

- *GPS*: Global Positioning System

- *HTTPS*: HyperText Transmission Protocol over SSL
- *API*: Application Programming Interface
- *MVC*: Model-View-controller pattern
- *DBMS*: DataBase Management System
- *DW*: DataWarehouse

1.4 Reference Documents

- Assignment Document "SafeStreets Mandatory Project Assignment.pdf"
- RASD Document "RASD1.1.pdf"

1.5 Document Structure

The successive part of th document is divided into 6 chapters:

- *Chapter 2*: in this chapter all the design choices made during the developmenr of the appli-
cation presented at different levels of abstraction: from the more general and abstract one
concerning tiers and system architecture, to the more technical one, concerning protocols
and technologies chosen to deploy the system.
- *Chapter 3*: in this chapter the user interfaces are presented, using mockups to represent
how they should actually be once implemented: different mockups are presented to cover
all possible situations worth to be shown.
- *Chapter 4*: in this chapter the requirements traceability is done. Each requirement de-
scribed in the RASD is provided by one or more component presented in Chapter 2; in
this chapter it's described which components provide what requirements and how this
requirement are actually provided.
- *Chapter 5*: this chapter takes into account how the system is going to be implemented,
how the components will be integrated and what strategies of testing is going to be used.
- *Chapter 6*: this chapter is only meant to show for each component of the group the effort
spent, with associated date and hours.
- *Chapter 7*: in this last chapter are mentioned the references to documents or sites about
techologies, pattern and architectures used in this document.

2 Architectural Design

2.1 Overview: High-level components and their interaction

The application will be developed using the client-server paradigm on a three-tiered architecture. The three layers of the application (Presentation, Application and Data) are divided into clusters of machines (i.e. tiers) that actually cooperate to provide a specific functionality. In this case we have three tiers and each tier is responsible for one of the three layers. The client tier is responsible (only) for the **Presentation** layer; therefore, in this architecture, the thin-client has been adopted considering the fact that the required client-side functionalities are limited. The UIs provided are just meant to show results and to allow clients to choose what they want. In particular there are two types of clients: a **mobile app** for the citizen which use SafeStreets to report violations and check statistics information and a **web app** for the police agents and supervisors.

The Application tier takes care of the application layer encapsulating all is needed concerning the application logic. It receives the requests from the clients and handles them. It's also responsible for sending asynchronous notifications to the presentation layer when certain conditions are met. It can also filter requests from the clients, deleting invalid ones.

The **Application** tier communicates with the **Data** tier, responsible for the Data Access layer, that is the layer responsible for accessing the DataBases and performing queries on them.

To provide the required functionalities the system exploits also datawarehousing. The DataWarehouse is a component in the Data tier able to deal with historical data and aggregate data taken from the operational databases in a more efficient way than a traditional dbms.

2.2 Component view

In the figure 2.1 the *Component Diagram* for the *SafeStreets* system is reported. This is a high level view in which only the main components are immediately shown, while some components will be better detailed afterwards.

In this diagram when two components can communicate using different interfaces a single interface link is reported, for the sake of readability. The various components are now described and detailed:

- **Router**: it has the role of dispatching the requests coming from the users applications. Before doing this it has also the important role of verifying the user authentication, checking the token which is sent with all requests performed by an authenticated user. In figure 2.2 a more detailed view is provided, putting in evidence the various interfaces for the mobile and for the web app.
- **SuggestionsManager**: this components is the one which elaborates and provides the

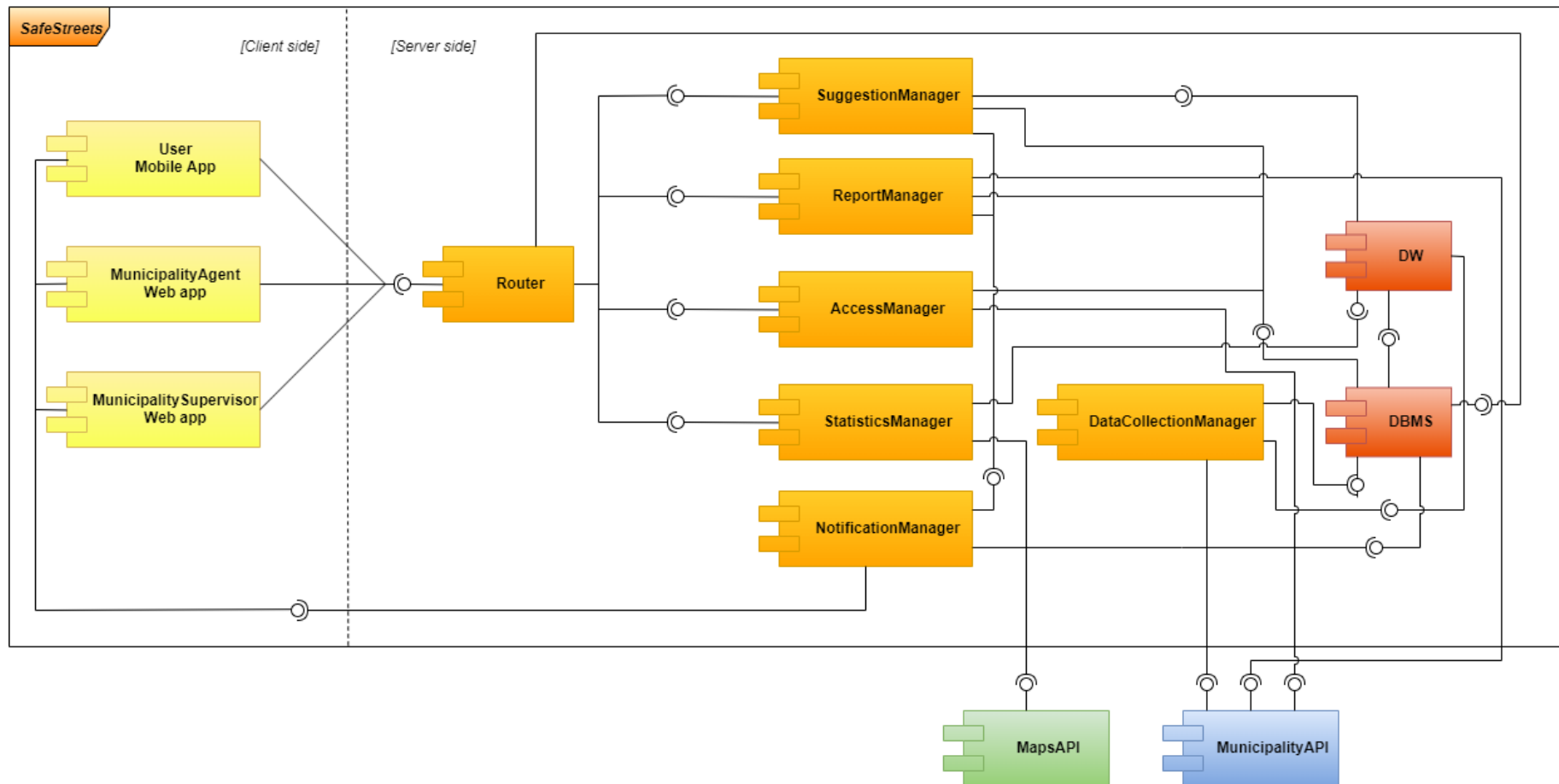


Figure 2.1: UML Component Diagram

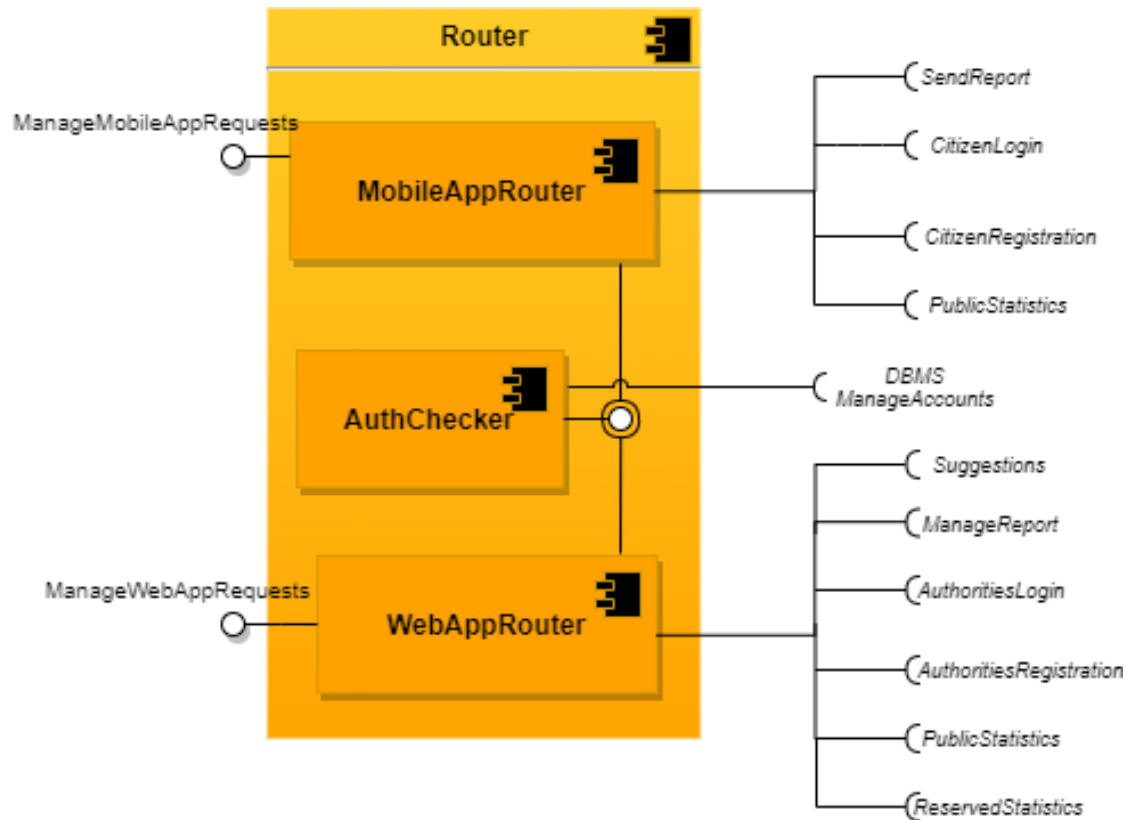


Figure 2.2: UML Component Diagram for *Router* component

Suggestions for the municipalities. It analyzes the data exploiting the *DataWarehouse* component for the aggregated queries about reports, accidents and tickets and the *DBMS Component* to retrieve non-aggregated information about the streets. It can provide the actual available suggestions for a municipality as an answer to a request coming from the *Router*, but it also notifies the *Municipality Supervisor web app* through the *Notification-Manager* component when it discovers a new suggestion (the suggestion discover task is periodically executed, once a month).

- **ReportManager:** this component is in charge of all concern the management of the reports sent by the users. It receives them by the mobile application and execute the automatic analysis to identify possible fake reports, it stores them through the *DBMS Component* and it provides the reports to the *Municipality Agents web app*, answering to their requests of confirm/enqueue/discard and allowing them to emit a ticket, exploiting the *Municipality APIs* which provide an interface for this service. Its detailed diagram is shown in figure 2.3.
- **AccessManager:** it manages everything about registration and login of the users. For the citizens registration it calls the municipality service for the verification of identity cards, while for the agents and supervisors registration it calls the municipality service for their identity verification. He stores the accounts data through the *DBMS* component and calls it also to verify the correctness of the login credentials. Once a login has been correctly performed it generates a token, stored in the database and sent back to the user, necessary to perform authenticated requests. This component is detailed in figure 2.4.
- **StatisticsManager:** it answers to the statistics requests, both public and reserved, and in doing this it calls the *DataWarehouse* component. It is shown in the diagram in figure 2.5.
- **NotificationManager:** this component is the one able to send push notifications to the user clients, both mobile apps and web apps. It is exploited by the *SuggestionManager*

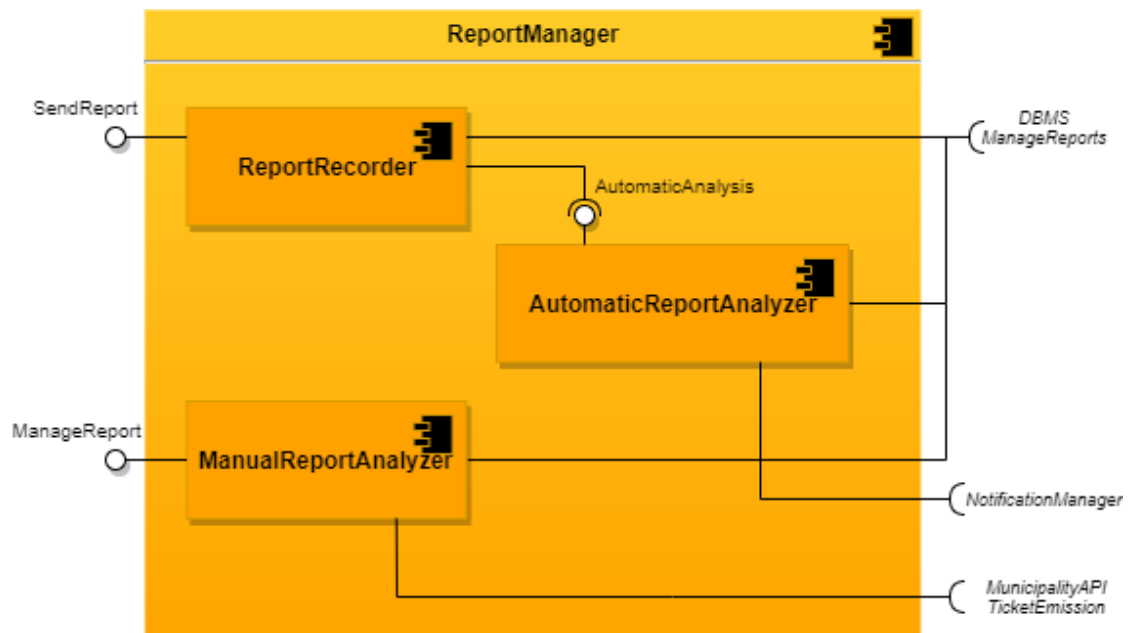


Figure 2.3: UML Component Diagram for *ReportManager* component

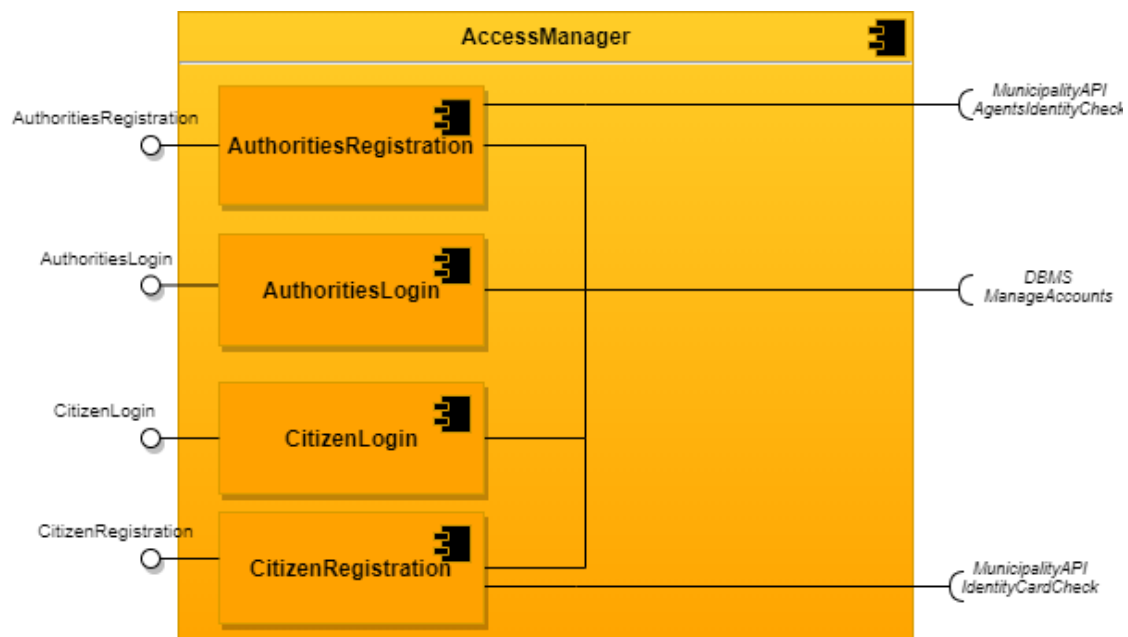


Figure 2.4: UML Component Diagram for *AccessManager* component

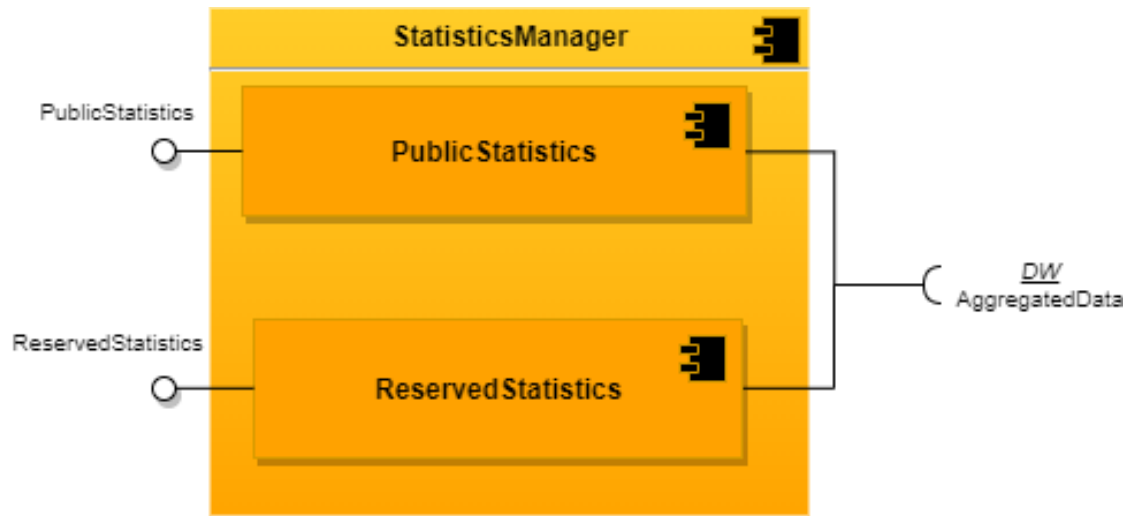


Figure 2.5: UML Component Diagram for *StatisticsManager* component

and by the *ReportManager*.

- **DataCollectionManager:** this component periodically retrieves the data made available by the municipality about accidents, tickets and street characteristics. These latter are stored in the operational database, while the others are retrieved as aggregated data and so are directly stored in the datawarehouse.

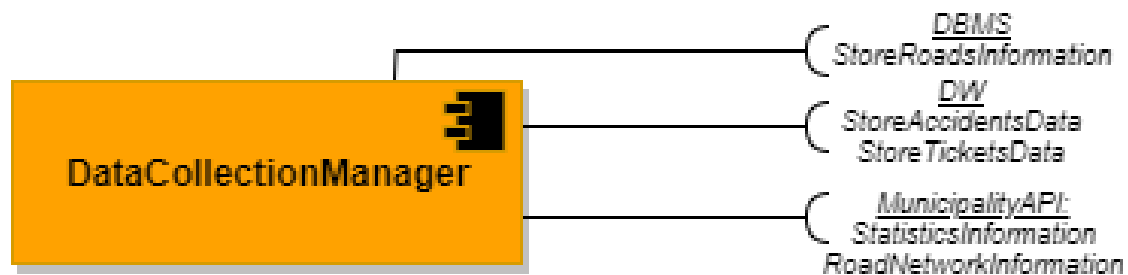


Figure 2.6: UML Component Diagram for *DataCollectionManager* component

- **DBMS:** this component manages the operational database, the main base for the functions of SafeStreets, being in charge of reports, users and streets data.
- **DataWarehouse:** this component manages the datawarehouse, the secondary database for the system, used to answer aggregate queries for statistics and suggestions. It is periodically alimented with the dbms data and with the already aggregated data coming from the municipality through the *DataCollectionManager* component.
- **MunicipalityAPI:** this is an external component, managed by the municipalities (an instance for each of them which supports the system), whose interface is standardized as described in the *Components Interfaces* section. It is necessary to retrieve the data for building statistics and suggestions, to verify the users identity and to have the possibility of traffic ticket emission through SafeStreets.

- **MapsAPI:** this is an external component which is exploited to obtain maps, which will be shown with the associated statistics data.
- **UserMobileApp:** this component entirely resides in the *Presentation layer* and it's just an interface to allow the users to use SafeStreets on their mobile devices, sending reports and retrieving statistics information. It performs only minimal controls on the forms before sending them (ex: they are not missing mandatory field) and attaches the user GPS position.
- **AgentWebApp:** this component is the *Presentation layer* of the web app which allows agents to see the reports, analyze them, emit tickets and retrieve users data.
- **SupervisorWebApp:** this component is the *Presentation layer* of the web app which allows supervisors to see the reports, possibly analyze them, retrieve all type of statistics and receive the suggestions elaborated by the system.

2.3 Deployment view

So far, only an abstract and general view of the system has been provided. In this section a better detailed view of the system shows how the component previously described are actually deployed in different machines and how each tier is organized. Here in figures 2.7 and 2.8 follow two diagrams: the first one is an UML Deployment Diagram which shows the allocation of the software components in the physical tiers of the system, while the second one is an informal diagram which shows more in detailed the network configuration of the system, with hardware devices such as firewalls and load balancers which are not related to the general architecture but are involved in assuring the proper working of the system.

W.r.t to the component diagram (figure 2.1), the different colors indicate the allocation of components. For the presentation tier clearly the *smartphones* contain the *UserMobileApp* while the *personal computers* and the *web server* contain the *Municipality Agent WebApp* and the *Municipality Supervisor Web App*.

All application logic components (orange in the *Component Diagram*) are deployed on the *Application Server*, while the *DBMS* component is deployed on the *DataBase server* and the *DataWarehouse* component is deployed on the *DataWarehouse server*.

In the *Mobile App* case the client contains all the *presentation layer* while in the *WebApp* case the layer is splitted between the *Web App* and the *WebServer*; the *WebServer* is responsible for contacting the application server and forward the client requests and responses.

All the servers are multiplied, cooperating together to improve performance, scalability, fault tolerance and availability. An elastic component (i.e. load balancer) is used to rule the accesses to different applications, dinamically balancing the load among all the Servers. The same is for the data servers. Each data server is associated with a single replica of the data and exploits the DBMS technology to access the *Database*. The *Database* is fully replicated in different nodes. Techniques and protocols are used to ensure consistency among replicas: they will be full explained in *Other design decisions* section.

To ensure and improve security firewalls are installed before and after the application servers to filter accesses from external and unsafe networks. By the creation of a *DMZ* (demilitarized zone) external entities can only have access to the exposed services. They will be properly configured to allow only legit requests coming from the Application Servers to reach the Data Servers. All the communications will exploit the HTTPS protocol, while unsafe communications will be refused by the firewalls. Security is crucial because the application works mainly with sensible information.

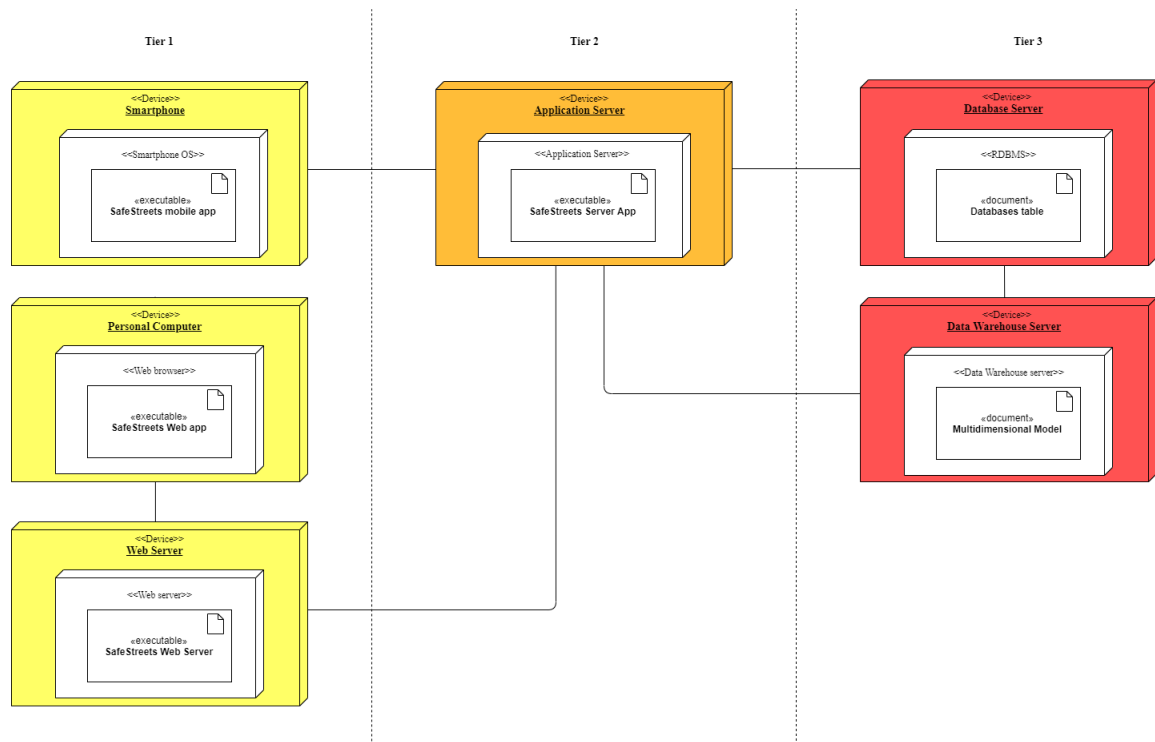


Figure 2.7: UML Deployment Diagram

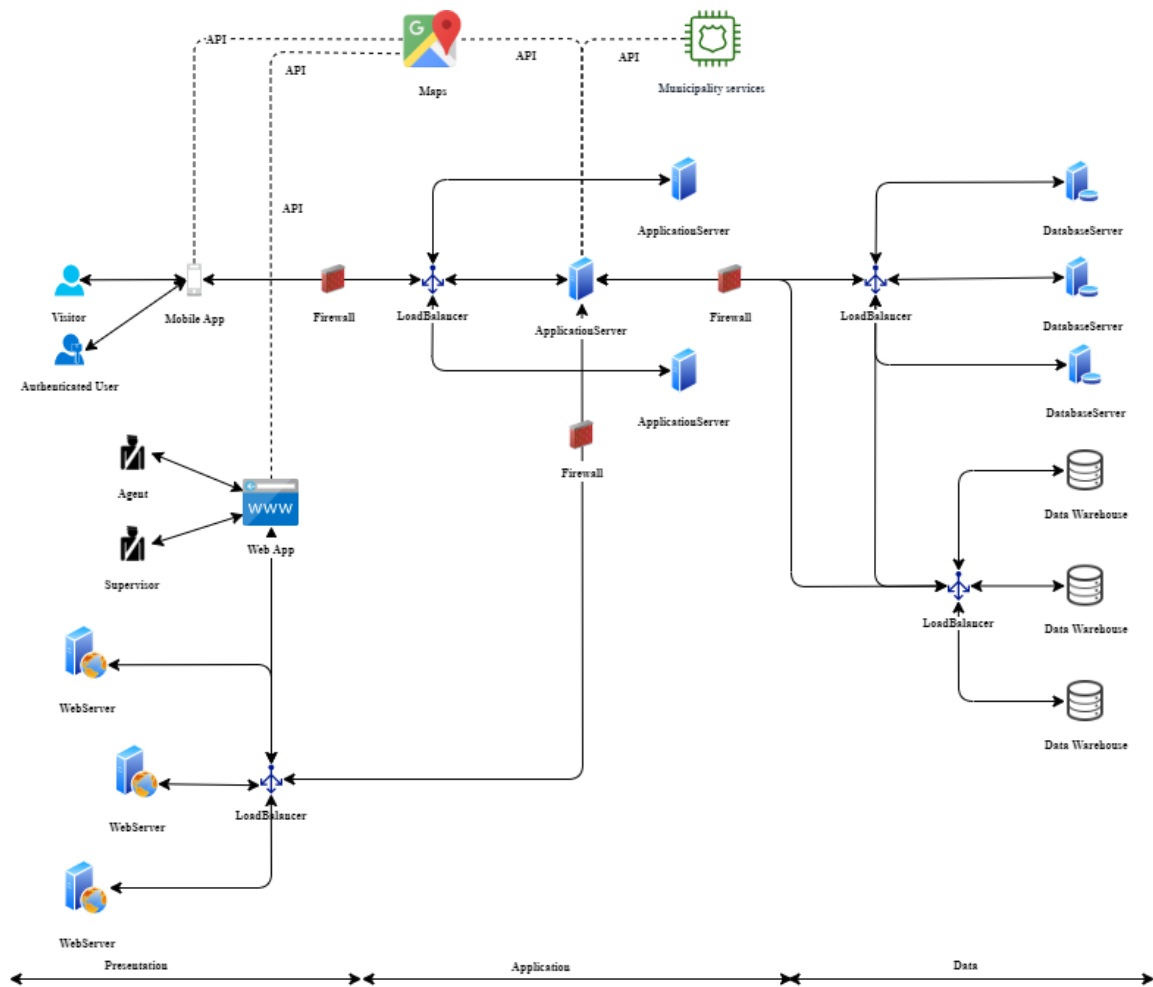


Figure 2.8: Informal architectural view of the system

2.4 Runtime view

In the figures 2.9, 2.10, 2.11, 2.12, 2.13 and 2.14 some architectural sequence diagrams are provided, showing the interactions among the different components in the execution of various functions of the systems.

The *NotificationManager* always requires an interaction with an external server (*Google Firebase Cloud Manager* for Android and Chrome, *Apple Push Notification Service* for iOS), but in the following diagrams it will never be represented, assuming it as internal to the component, for the sake of simplicity and readability.

In the sequence diagram of figure 2.9 is shown what messages are actually exchanged among components to process correctly a report send by a User. The User, through the **Mobile App**, fills the report form and sends it to the server for a confirm. The request is handled at first by the **Router** that checks if the User is correctly authorized (i.e. verifies if the corresponding token has the rights needed): if the check is negative, then the request is discarded, otherwise the **ReportManager** calculate the municipality in charge for a specific report comparing the GPS position of the report with the ones associated to municipalities borders in the database. After this, the **ReportManager** checks internally if the pictures in the report have been modified or not. If so, the report is stored as *Refused* and the **Notification Manager** will send a notification about the refused report to the involved **MunicipalityAgent Web App**. If the pictures are not fake then the report is stored as *New report* and the **NotificationManager** will be in charge of notifying the **Municipality Web App** about the new pending report. It has been assumed that the *getCityByPos* request to the **DBMS** returns a city for which the *SafeStreets service* is active. In effect the mobile app should not allow the user to send a report for a city in which the system is not active.

In the sequence diagram of figure 2.10 are described the components involved in the user login and their interaction. The User fills the spaces in the **Mobile App** inserting his username and password. The login request is then sent to the **Router** that forwards the message to the **AccessManager**. In particular the subcomponent **CitizenLogin** handles the login, asking the **DB** the password corresponding to the specified username. After having received the password, it internally checks if the two passwords (the one coming from the user and the one stored) coincide: if so, the user is successfully logged in and this is notified back to the client, showing him the main page of the App; if not, a notification is sent back to the client, showing him an error message and asking him to insert again username and password. Notice that everytime a new login is performed a new token is generated associated to the specific logged client.

In the sequence diagram of figure 2.11 is shown the managing of a *Report* by an *Agent*. The request of showing the first report to be analyzed comes from the **Agent WebApp** and is received by the **Router** component, which verifies the authentication and forwards it to the **ReportManager** (in particular the **Manual Report Analyzer** subcomponent). The latter returns the report data to the **WebApp** and then the *Agent* decides what to do. If he wants to issue a ticket the request is sent to the **MunicipalityAPI** and consequently the information is recorded by the **DBMS**.

If he decides to put the report on hold, this is simply recorded in the database. Otherwise if he discards the report this is recorded in the database and he also has the possibility to see the report author's data, always stored in the database.

The control for the user authentication is shown for every request, but only the first one presents the *Alternative* choice, for the others, marked with *, it has been assumed a positive answer,

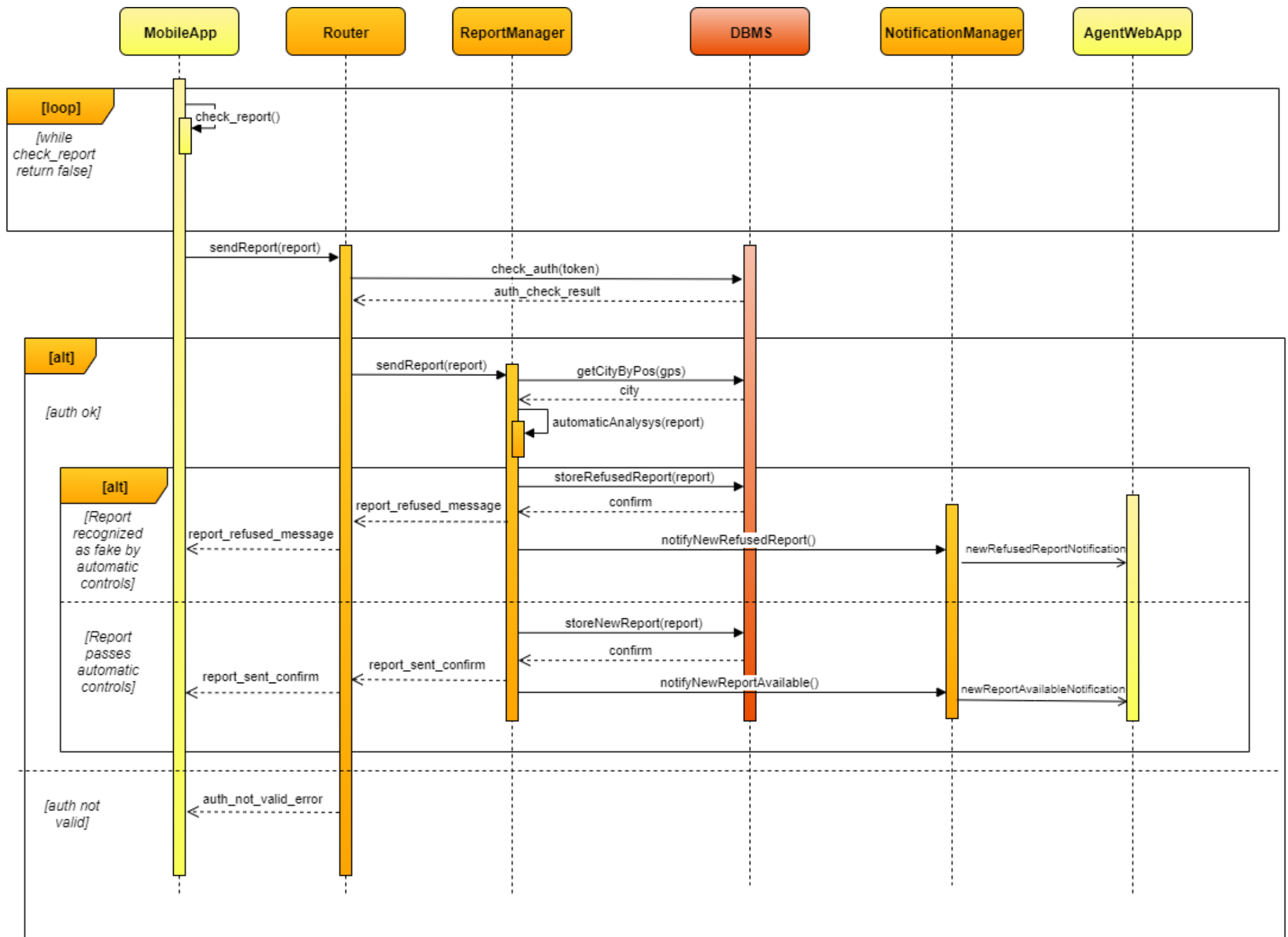
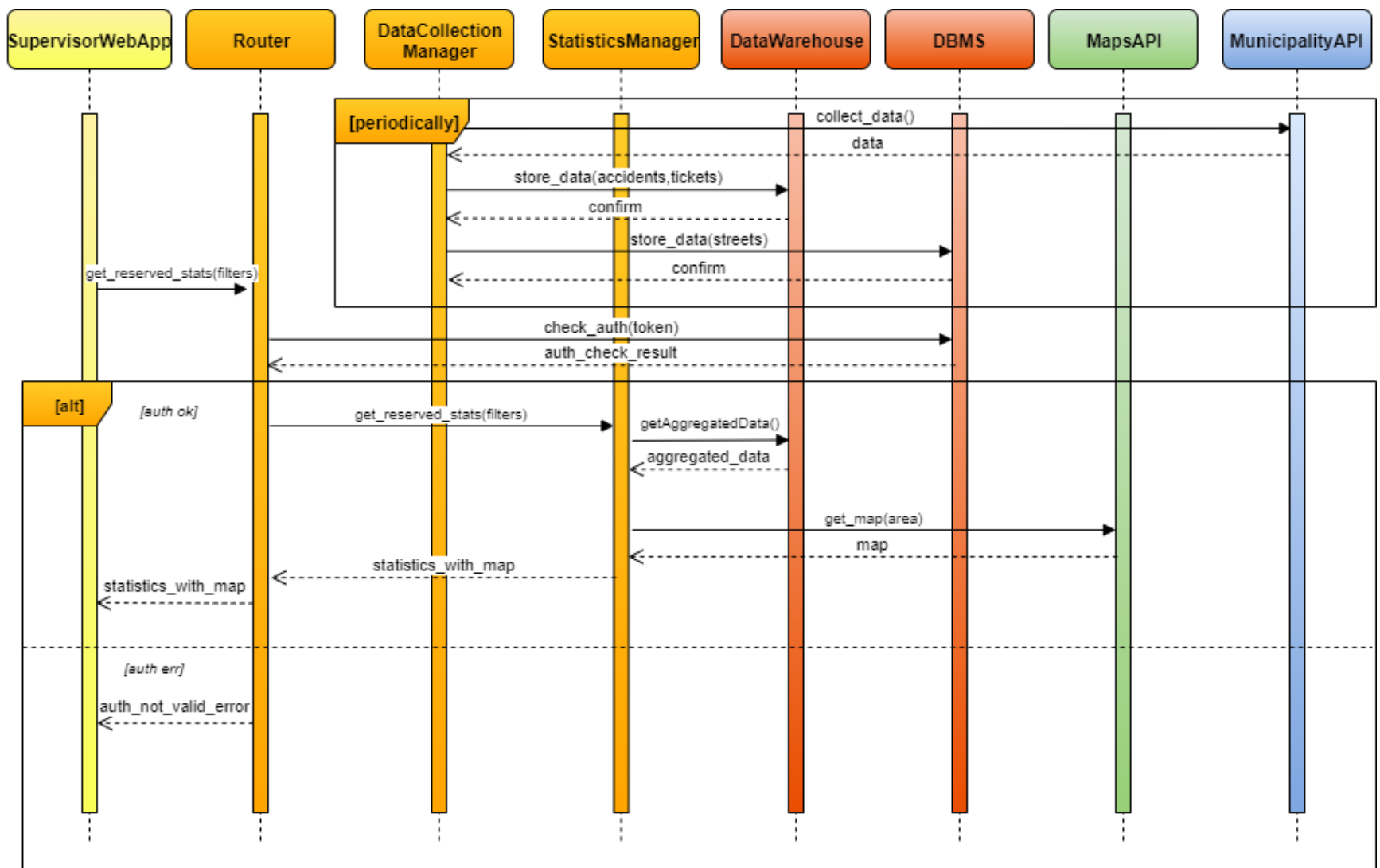
Figure 2.9: UML Architectural Sequence Diagram for *SendReport* use case

Figure 2.10: UML Architectural Sequence Diagram for *UserLogin* use case



Figure 2.12: UML Architectural Sequence Diagram for the *CheckStatistics* use case

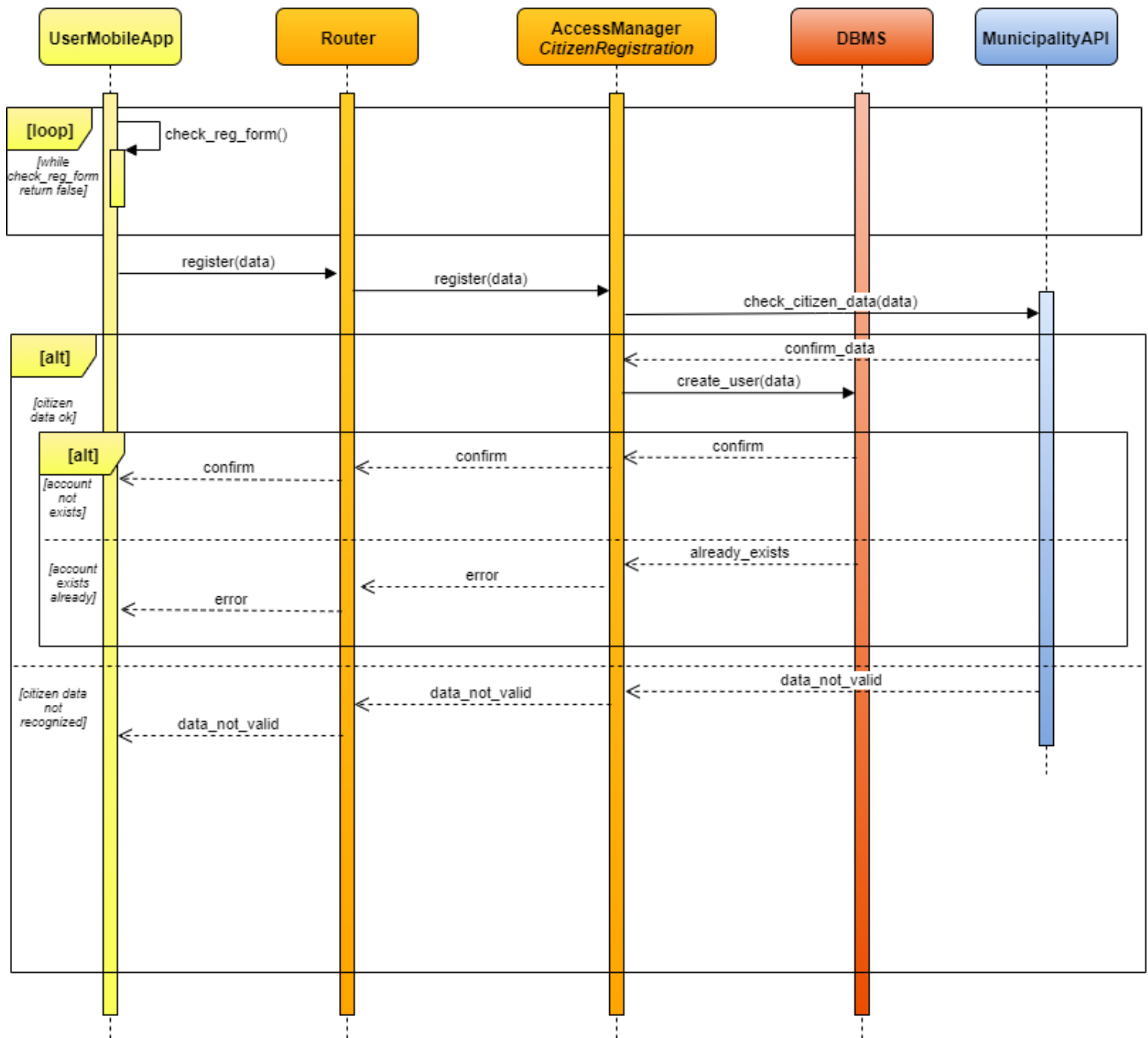
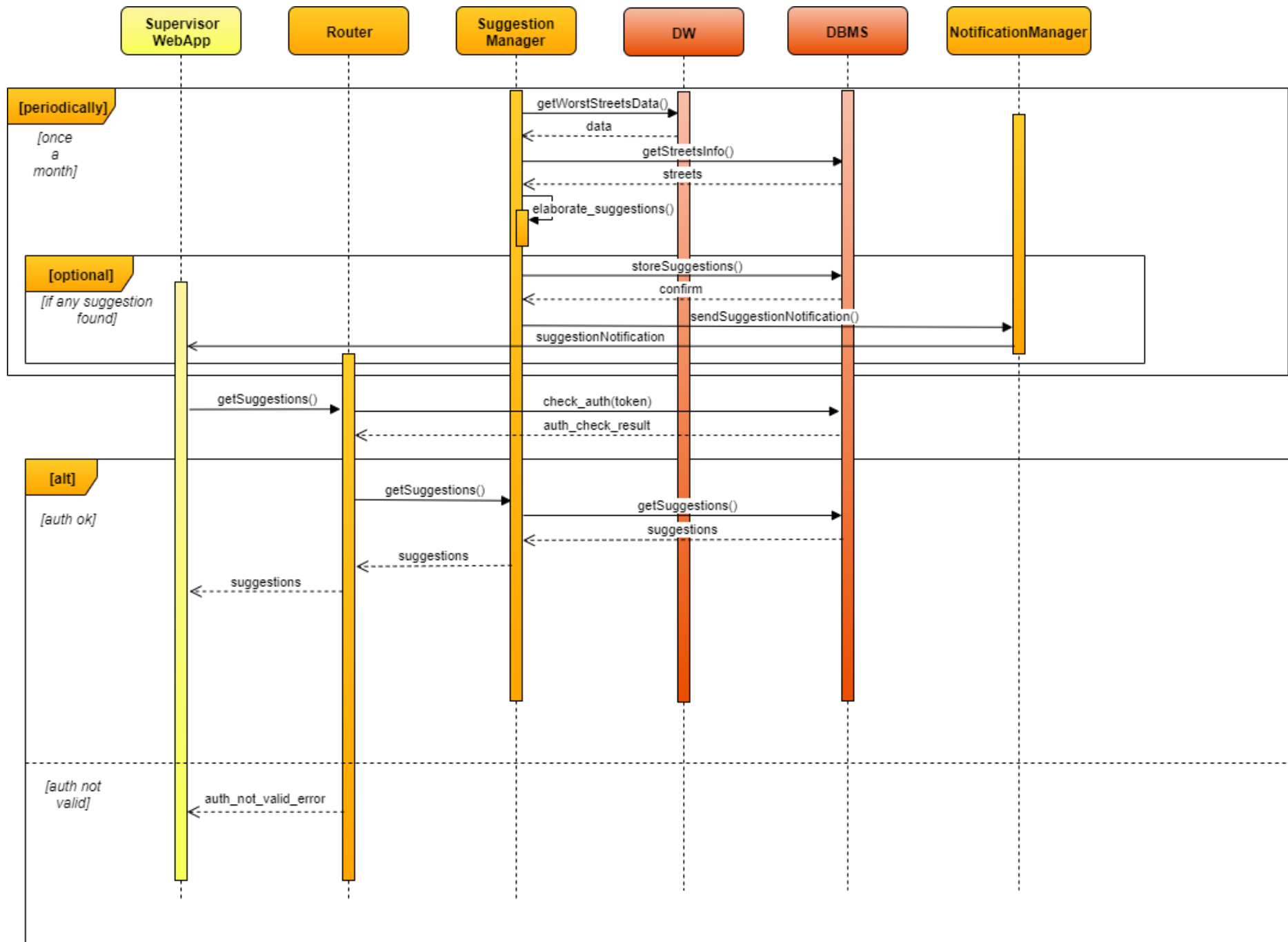


Figure 2.13: UML Architectural Sequence Diagram for *User Registration*

Figure 2.14: UML Architectural Sequence Diagram for the *Suggestions*

for the sake of readability, avoiding to repeat the *Alt* with the error message returned in case of authentication failure.

A similar assumption has been made for the municipality answer to the ticket emission request, it has been avoided to report the negative answer, for which the agent would be required to retry the request.

In the sequence diagram of figure 2.12 are described the interactions concerning the *Statistics*. Initially is shown how the **DataCollectionManager** component periodically collect data from the **MunicipalityAPI** and stores them.

Then is shown the request of *ReservedStatistics* by a **Supervisor WebApp**, with the **Router** which forwards the request to the **StatisticsManager** and this last component that builds the statistics consulting the **DataWarehouse** and the **MapsAPI** to retrieve a map of the interested area.

In the sequence diagram of figure 2.13 is shown the *Registration* process for a *User*. He compiles the form on the app, which already verifies that all fields have been properly filled, and the app sends the data to the **Router** which forwards the request to the **AccessManager** component (in particular the **CitizenRegistration** subcomponent). This component verifies the correctness of the provided data through the **MunicipalityAPI** and in case of a positive answer store the registration on the **Database**, otherwise it returns an error message.

In the sequence diagram of figure 2.14 are shown the interactions involved by the *Suggestions* function. For first is shown how the **SuggestionManager** periodically retrieve data from the **DataWarehouse** and from the **DBMS** and tries to elaborate new suggestions. If it succeeds in this it stores the new *Suggestion* and sends a notification to the **MunicipalitySupervisor WebApps** through the **NotificationManager**.

Then is shown how when a *Supervisor* wants to check the last suggestions, in any moment, he can access them by the corresponding page of his application.

2.5 Component interfaces

In the figure 2.15 the component interfaces, with reference to what was shown in the Component diagram, are represented. The arrows represent a dependency relation. Each component is described by its invocable methods:

- Each of invocable methods of the Router are responsible of forwarding a certain request to the associated component; for example ForwardCitizenRegistrationData has one single parameter of User type, which must be forwarded to the AccessManagerComponent, invoking its "citizenRegistration" method. For all methods which require an authentication the Router checks the authentication of the request with the AuthCheck component, even if its methods are not described in the Diagram because they are internal methods.
- ReportManager methods can be invoked by the Router when a new Report is sent ("sendReport" method) or an already existing Report must be managed by the responsible agent (ManageReport). In the first case the component will invoke the NotificationManager "pushReports" method, while in the second one it will invoke different MunicipalityAPI methods. In both case it will interact with the DBMS.
- The methods of the Access Manager are invoked by the Router when it receives a Registration or Login request for a citizen or an Authority. As in the previous component, other methods will be invoked, belonging to the DBMS and the MunicipalityAPI.
- The methods of the Statistics Manager are invoked by the Router when a request of public or reserved statistics is asked through the associated method of the Router. The StatisticsManager asks to the Data Warehouse component for tickets, accidents and reports and uses the MapsAPI to provide a map of the interested area.
- SuggestionManager is only invoked by the Router through its method "getSuggestions", which can return zero, one or more suggestions.

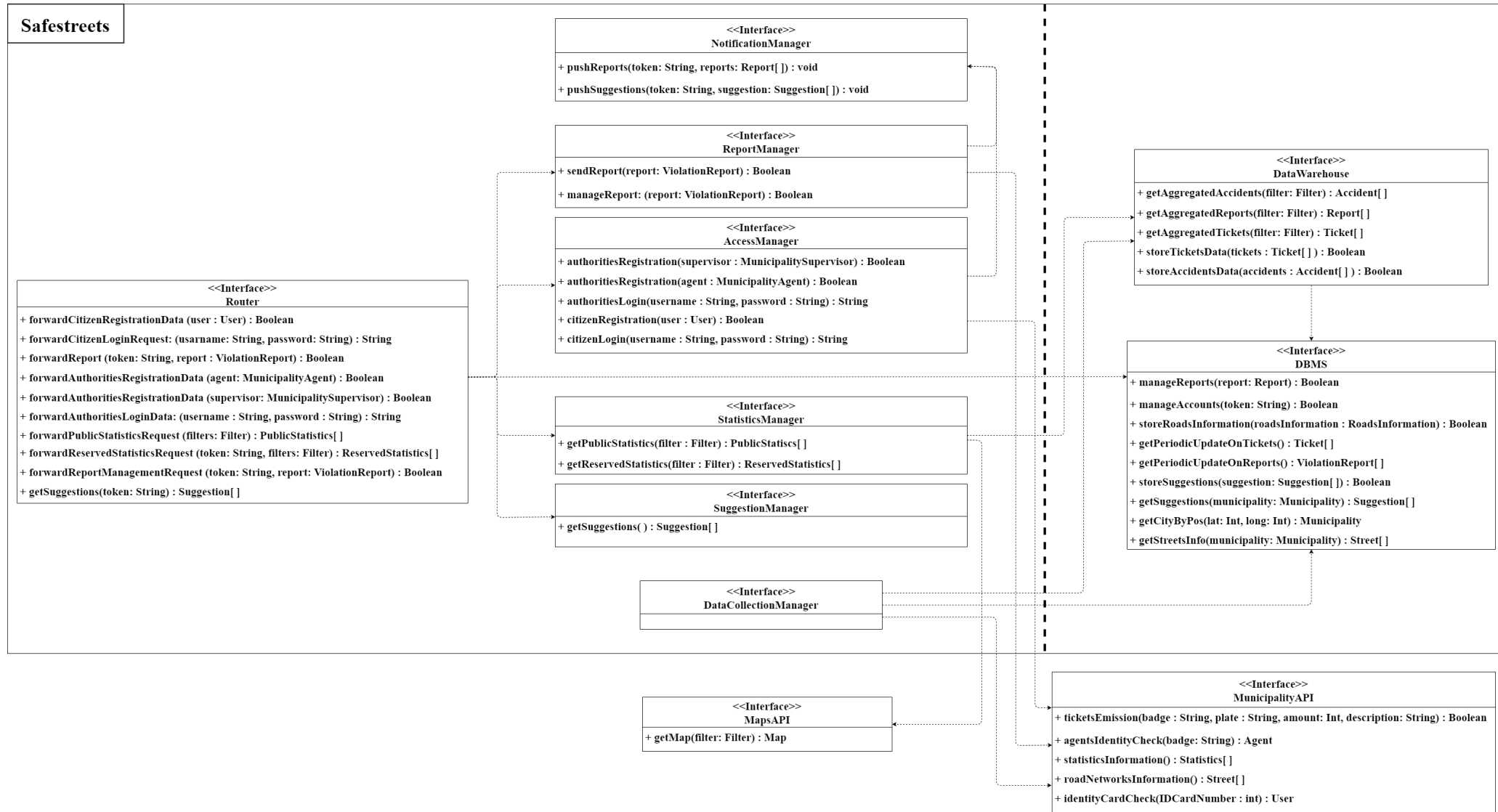


Figure 2.15: UML Interfaces Diagram

- NotificationManager provides methods for the AccessManager and the Router and it will invoke the external notification servers methods, not shown here.
- DataCollectionManager is never invoked by any other component, so its interface does not require any method, infact it is only periodically executed to update data. It obtains data through the MunicipalityAPI with the methods "statisticsInformation" and "roadNetwork-Information" and saves them both in the DataWarehouse, invoking methods "storeTickets-Data" and "storeAccidentsData", and in the Database, calling "storeRoadsInformation" method of the DBMS.
- Data Warehouse provides methods to obtain aggregated data about accidents, reports and tickets, invoked by the StatisticsManager; at the same time data about accidents are updated by the DataCollectionManager and about reports and tickets are updated through two different methods of the DBMS.
- DBMS provides methods to manage reports and accounts and to store data about roads from the DataCollectionManager. "getPeriodicUpdate" method is invoked by the DataWare-house to obtain updated data from DBMS.
- MunicipalityAPI provides methods to emit tickets, to identify both the registrating agent from its badge number and the registrating user from his Identity Card number and to obtain information about statistics and road networks.
- MapsAPI provides maps according to the filters given as parameter; return value of the API method is "Map" class, which generically denotes the class type for maps and depends on which API has been choosen.

In figure 2.16 is reported a *Class diagram* which is derived from the class diagram already reported in the RASD with some small modifications. It's still an high level diagram, not necessarily exactly corresponding to the implementation, but it can be a good starting point for the classes to be implemented.

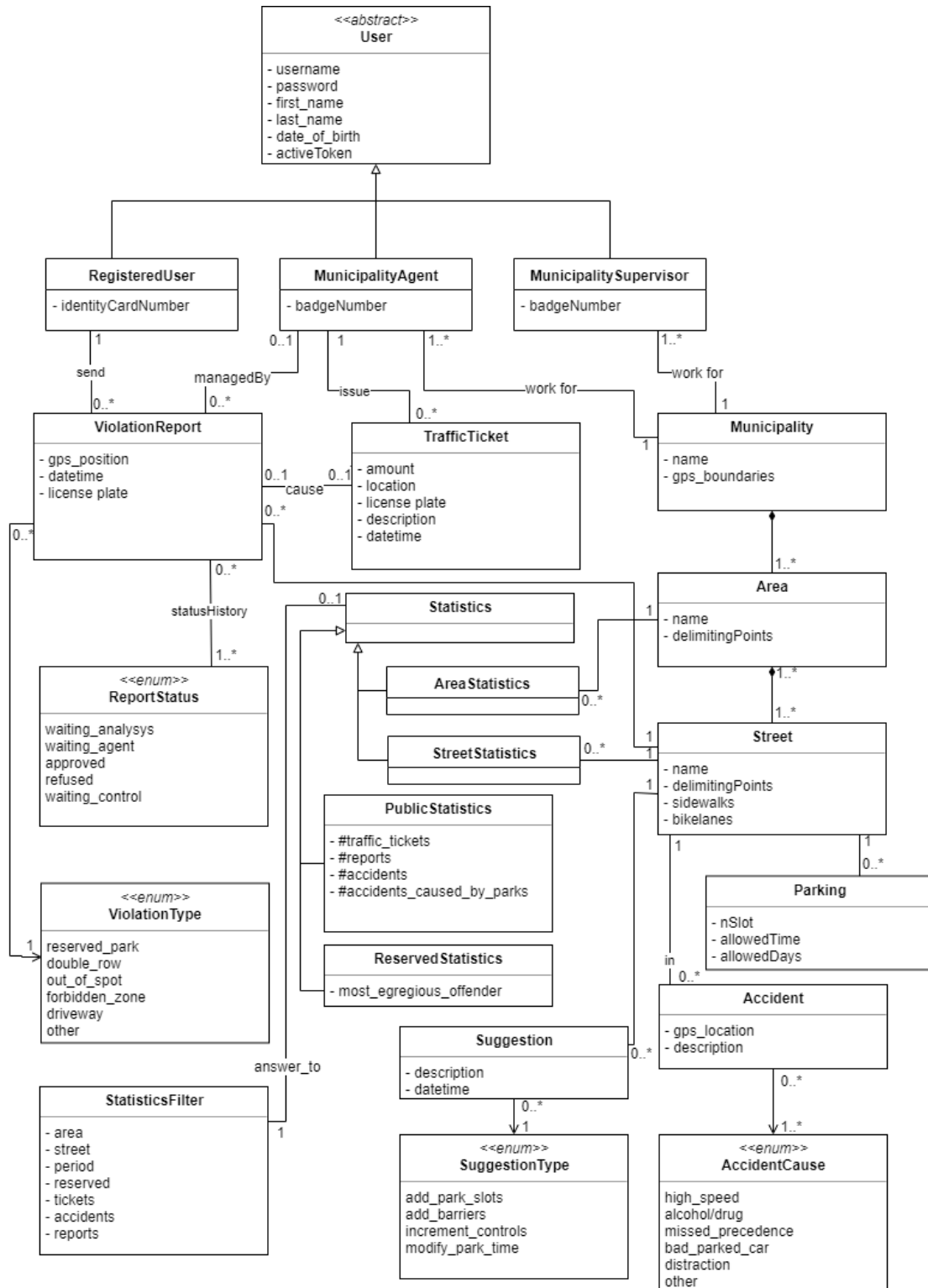


Figure 2.16: UML Class Diagram

2.6 Selected architectural styles and patterns

2.6.1 Model-View-Controller Pattern

The (distributed) Model-View-Controller pattern has been exploited to develop the entire application. This choice has been made to allow fast development, where multiple developers can work together at different levels, and easy debug and update of the application. In section 2.2 the component diagram shows all the high level components that are part of the application. Colors have been used to underline different tiers and so different functionalities: yellow components are View components, orange ones are Controller components and red components are Model Components. For more infos about MVC pattern, see *[MVC-PATTERN]*.

2.6.2 Facade Pattern

The facade pattern has been exploited in our application to hide the complexity of the application: only one component can be seen by the client as a "facade", the Router. This component shows an interface that masks all the complexity of the Application and Data Server to the User Interfaces. For more infos about facade pattern, see *[FACADE-PATTERN]*.

2.6.3 Publish-Subscribe Pattern

The publish-subscribe pattern is used for the notification mechanism implemented by the *NotificationManager* component to send notifications to the user applications (both mobile and web) about reports and suggestions. It is based on the functionalities offered by Google and Apple for Android/Chrome and iOS respectively, namely *Google Firebase Cloud Messaging* and *Apple Push Notification Service*. See *[PUBSUB-PATTERN]* for more information.

2.6.4 Thin client

Our application is based on a three-tier architecture where the client is thin. Thin, in this case, means that client has all (or partial) and only functionalities of the Presentation layer. In particular in this application, client doesn't need to cache anything and doesn't need to know anything about the application logic. No need to store locally statistics, considering that data (and so statistics) can change among two different requests. The client just needs to show the operation that a user can perform and the result of the requests. This is meaningful both for the WebApp and for the Mobile App.

2.6.5 RESTful architecture

In the system, the REST paradigm has been followed. This choice has been made mainly for the sake of scalability and independent development of different components. This goal has been achieved with uniformity of interfaces, client-server stateless interactions and cacheability of data (meaningful only for Server). For more infos about RESTful architecture, see *[RESTFUL-ARCHITECTURE]*.

Note: here interfaces refers to network interfaces in middleware layer, not to be confused with the component interfaces shown before.

2.6.6 Databases

The operational database of *SafeStreets* is a relational database, considering the structured data it has to manage, regarding users, reports, municipalities. For the statistics and the suggestions elaboration, which require the analysis of a big amount of aggregated historical data, a multi-dimensional model is used, with a *DataWarehouse*, obtaining its data both from the relational database and directly from the municipality (through the *DataCollectionManager* component). A conceptual description of the databases is present in the *Database design* subsection.

2.7 Other design decisions

2.7.1 Database design

In figure 2.17 is shown the **Entity-Relationship** diagram for the operational database. It is worth to remark that the data coming from the municipality about tickets and accidents are directly stored in the datawarehouse, while here only data about tickets emitted through the SafeStreet system are stored.

In figure 2.18 is shown the **Dimensional Fact Model** for the datawarehouse. The tree facts of interest are Report, Tickets and Accidents. The diagram highlight the dimension of analysis which can be useful for the Statistics and Suggestions functions.

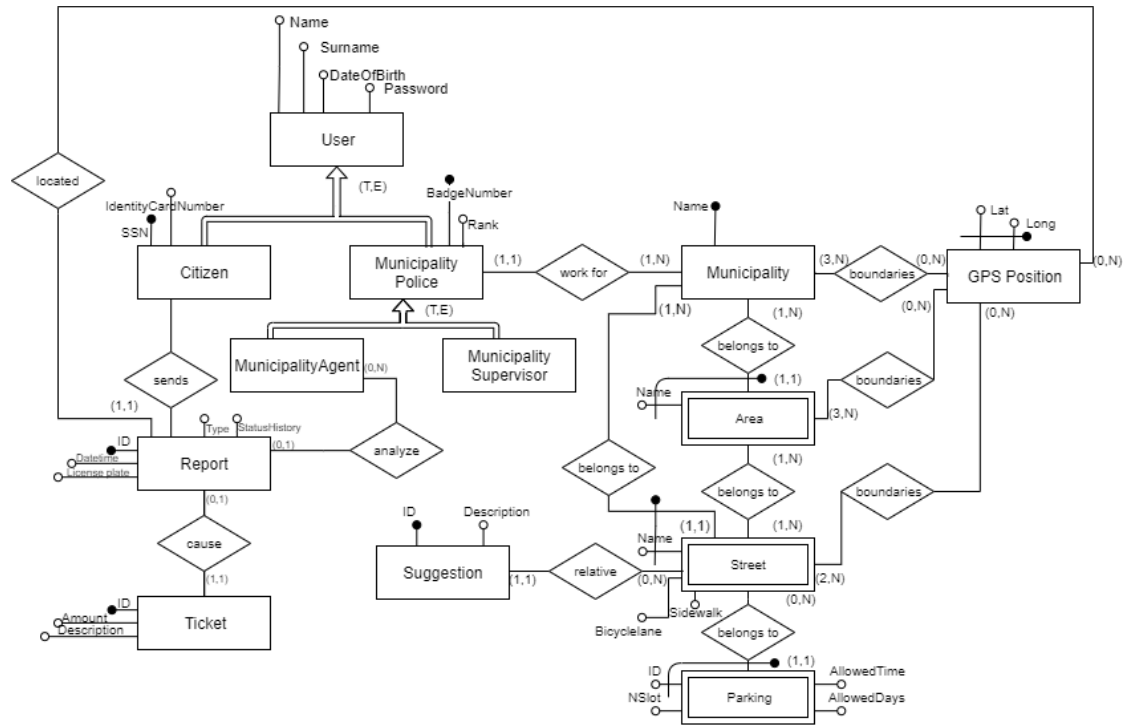


Figure 2.17: Entity-Relationship diagram for the operational database

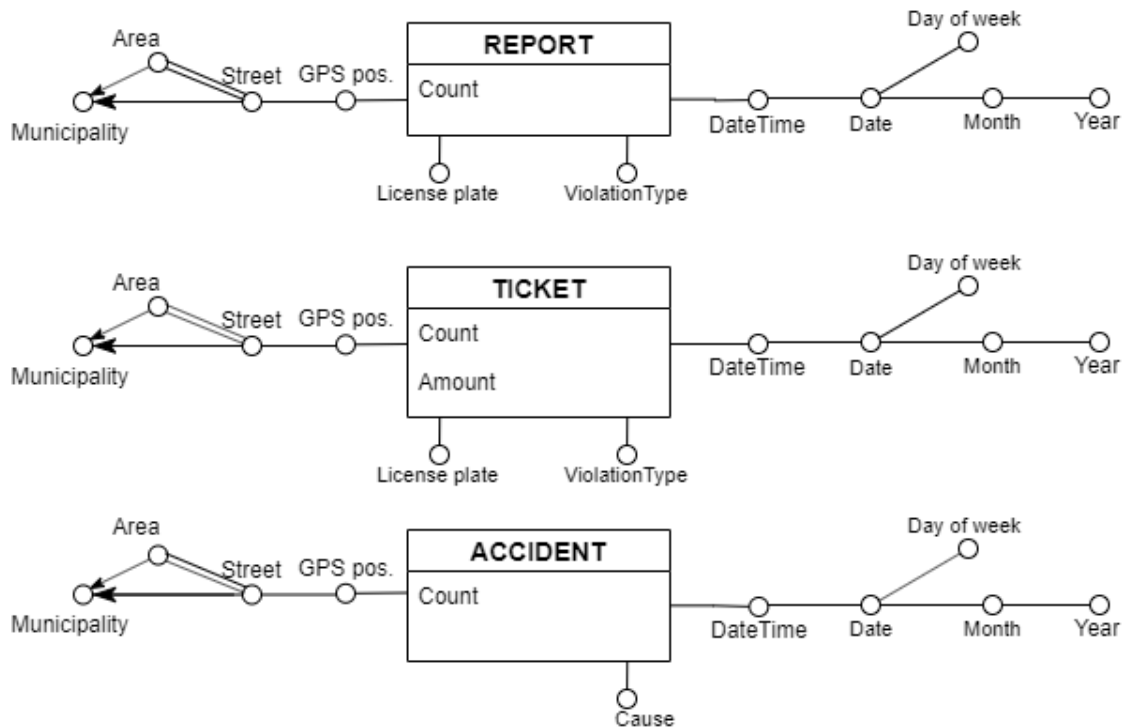


Figure 2.18: Dimensional Fact Model diagram for the data warehouse

2.7.2 Consistency and update strategies among replicas

In this application, we need a client-centric consistency among replicas because end users (and therefore application servers that act as clients towards the DataServers) don't always connect to the same DataServer, due to the presence of the load balancer. Every DataServer can respond to a request to read or write data so we use an active replication protocol. In particular we exploit leaderless replication in which the decision on the value to read and the write to perform is decided by all the replicas or at least a quorum of them. The type of chosen consistency model is the "read your writes": the effect of a write operation by a process on a data item x will always be seen by a successive read operation on x by the same process. Concerning the update propagation we opted for propagating a notification of the operation, assuming that there will be more writes than reads. The propagation strategy chosen is the Gossiping strategy: when a replica is updated then it just propagates that update to all the nodes that it knows; if a replica receives an update that it has already received then the probability of propagating that information is decreased on that replica.

3 User interface design

In this chapter are presented again mockups shown in the RASD with a few additions (figure from 3.1 to 3.11).

In the figures 3.12, 3.13 and 3.14 are shown activity diagrams that describe how a user (citizen, agent or supervisor) can navigate in the UIs offered by the application. The choice of activity diagrams has been made following the suggestions of the paper *[UML-ACTIVITY]*. Notice that Users and Municipality Agents/Supervisors can quit the application from any state to reach the end state of the diagrams. No end state is represented and so are all the arrows that lead to them; this has been done for the sake of readability of the diagrams.



Figure 3.1: Mobile App mockup for the starting page after having opened the app for a User

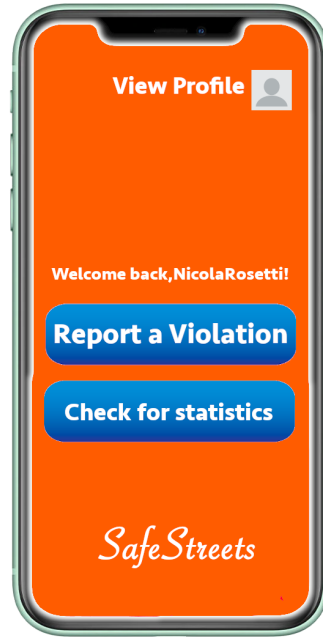


Figure 3.2: Mobile App mockup for the main page of the application after a User has logged in



Figure 3.3: Mobile App mockup for a User when he wants to register

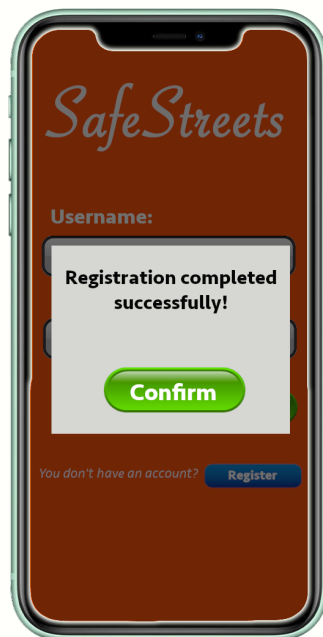


Figure 3.4: Mobile App mockup for the registration confirm popup

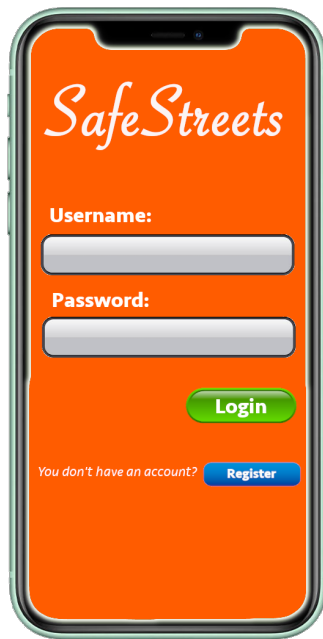


Figure 3.5: Mobile App mockup interface for the login of a registered User




Figure 3.6: Mobile App interface mockup for the reporting of a violation for registered Users



Figure 3.7: Mobile App interface mockup for the statistics for Users

SafeStreets MUNICIPALITY OF MILAN

Giorgio Rossi - 867594
MUNICIPALITY AGENT



[Home](#)[Reports](#)[Statistics](#)

New Reports

ID	Datetime	Location	Description	Manage
128	07-11-2019 09:40	Piazza Lombardia, 1	Car on disabled parking	MANAGE
129	07-11-2019 10:03	Viale Italia, 43	Car on sidewalk	

Waiting Reports

ID	Datetime	Location	Description	Manage
130	07-11-2019 10:15	Via Ponzio, 12	Out of slots	MANAGE

Refused Reports

None

Figure 3.8: Web interface mockup for the report list for a logged Municipality Agent


SafeStreets MUNICIPALITY OF MILAN

Eugenio Verdi - 104981
MUNICIPALITY SUPERVISOR

HomeReportsStatisticsSuggestions

Available Suggestions

ID	Datetime	Location	Description	Cause	Manage
2	02-11-2019 16:00	Via Torino	Add barriers	20 forbidden parkings on sidewalk	DETAILS
1	02-10-2019 16:15	Via Diaz	Increase controls between 12-14	90% of violations are between 12 and 14	DETAILS

 **SafeStreets**

New suggestion available for Via Torino!

×

Figure 3.9: Web interface mockup for the suggestions list for a logged Municipality Supervisor

SafeStreets

MUNICIPALITY OF MILAN

Giorgio Rossi - 867594

MUNICIPALITY AGENT

Home

Reports

Statistics

< BACK

Violation report 259

1/1

Type of violation

Forbidden area

Description

The car is illegally parked on the sidewalk

Position

Via dei papiri, 23

Date and time

2019-11-29 21:25

License plate provided by report author

ART WAR

License plate automatically recognized

ART??AR

Traffic ticket

Description

€ Amount

License plate

➤ ISSUE

👤 ON HOLD


✕ DISCARD


Figure 3.10: Web interface mockup for the report analysis page used by Municipality Agents


SafeStreets MUNICIPALITY OF MILAN


[Home](#) [Registration](#) [Statistics](#)


Municipality user registration


 Name

 Surname

 Date of birth

 Rank

 Badge number

 Password

Register

Figure 3.11: Web interface mockup for the registration page for a municipality user

34

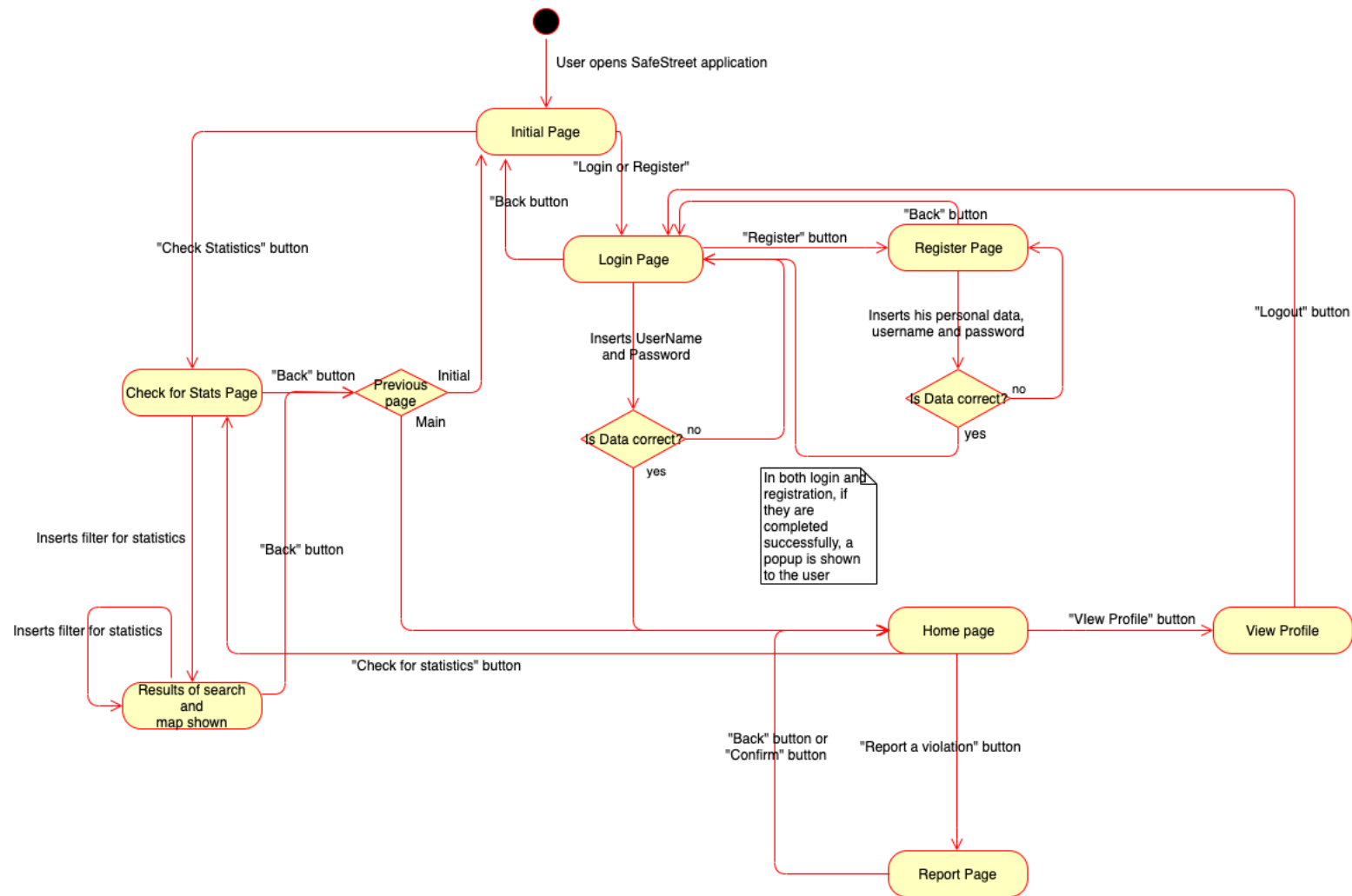


Figure 3.12: UML Activity Diagram for the User Mobile App navigation

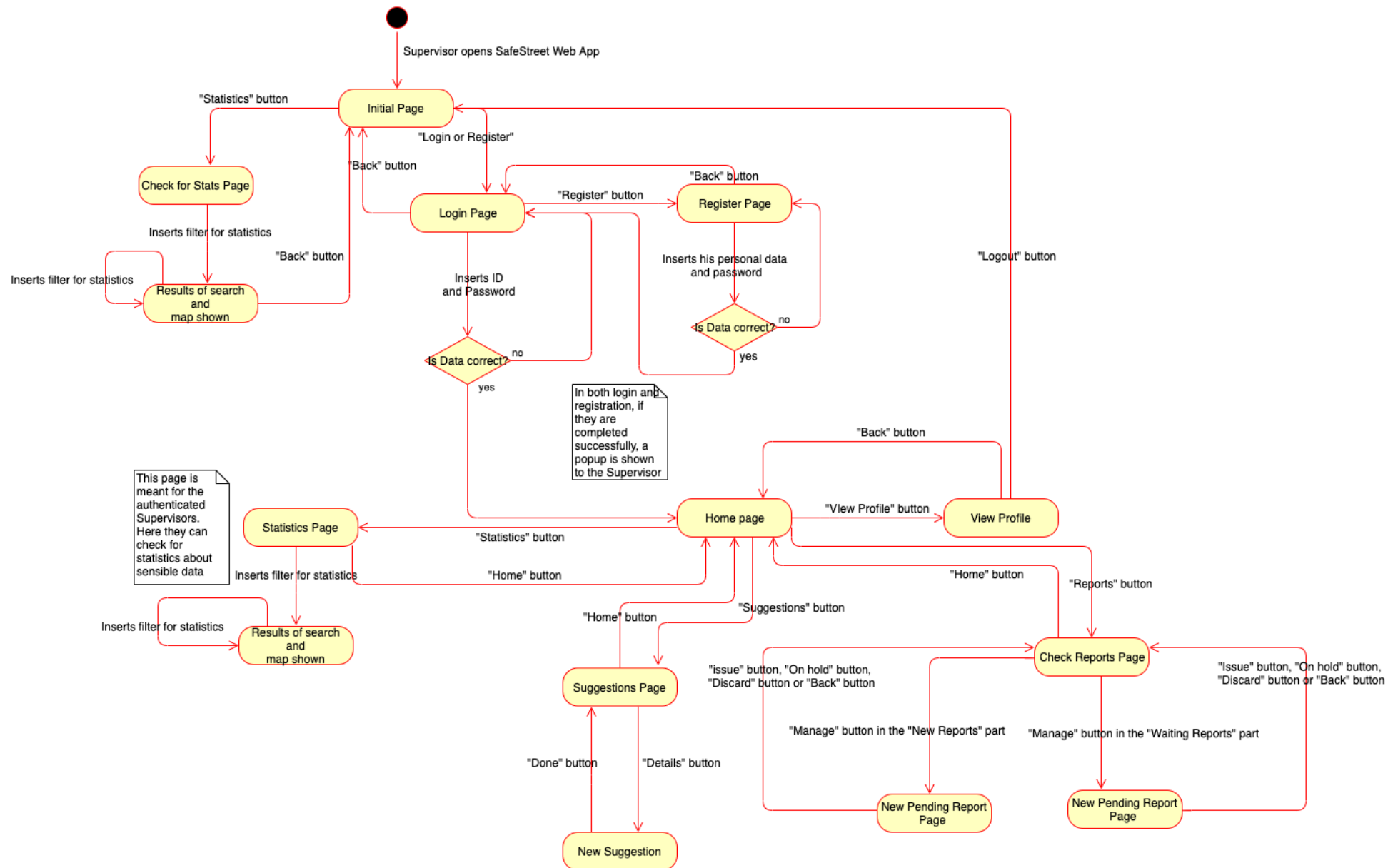


Figure 3.13: UML Activity Diagram for the MunicipalitySupervisor Web App navigation

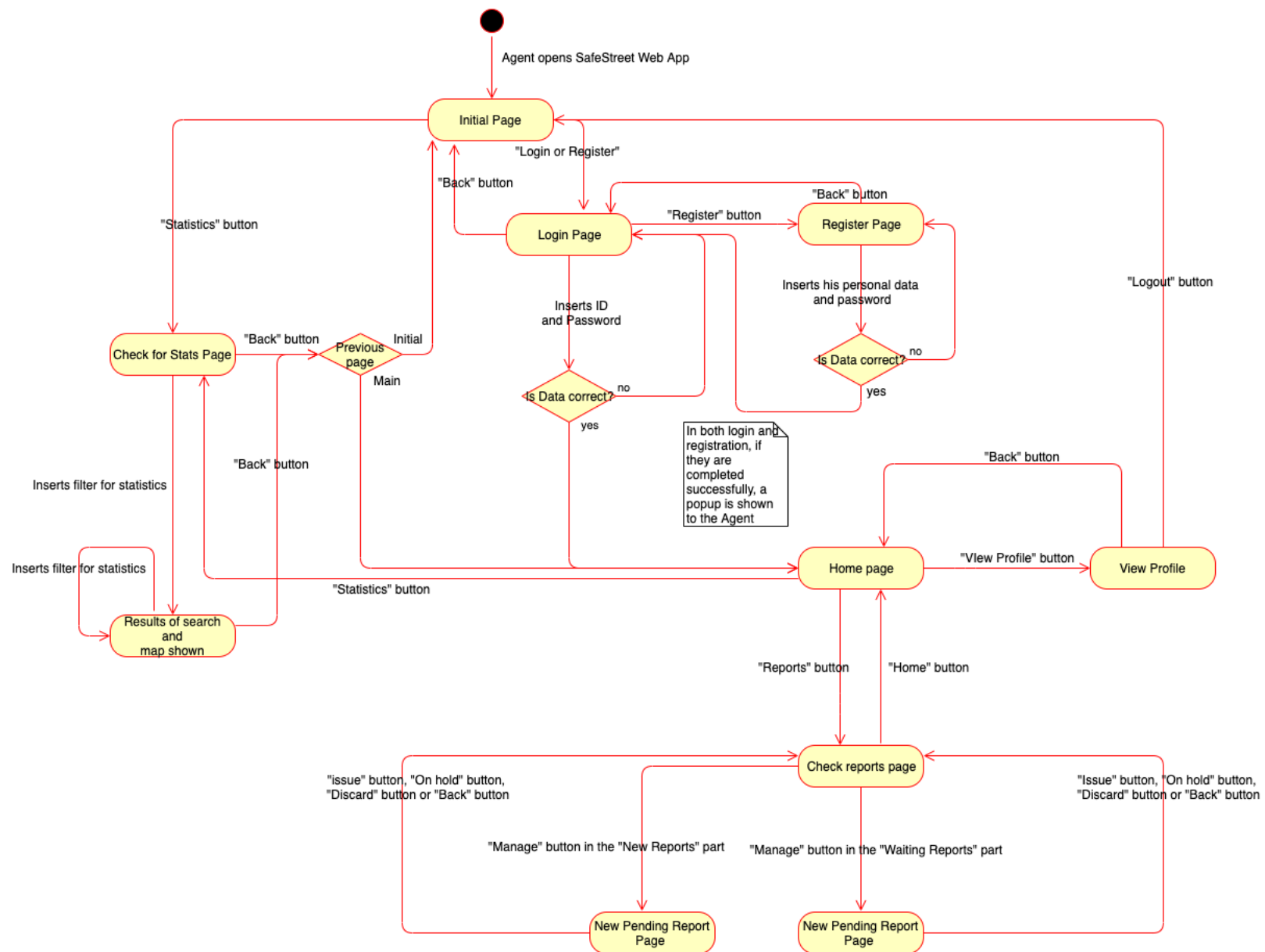


Figure 3.14: UML Activity Diagram for the MunicipalityAgent Web App navigation

4 Requirements Traceability

In this section is shown how the requirements are actually ensured and which components actually ensure them. It's worth to notice that for the sake of simplicity the Router has been ignored but it's always involved when dealing with a request coming from one of the clients. Table 4.1 has been added to highlight for each component what requirements it ensures.

Component	Requirements
User Mobile App	R1,R3, R4, R5, R6, R7, R8, R9, R10, R28, R29, R30, R33, R34,R35
MunicipalityAgent Web App	R19,R20,R21,R22,R23, R24, R26,R28,R29,R30,R33,R34,R35
MunicipalitySupervisor Web App	R28,R29,R30,R34,R35,R36,R37,R38, R40
AccessManager	R1, R2, R3, R25, R26, R39, R40
ReportManager	R11, R12, R13, R14, R15, R16, R17,R18
SuggestionManager	R42
StatisticsManager	R30, R36
NotificationManager	R18, R43
DataCollectionManager	R31,R41
DataWarehouse	R27, R30, R31, R32, R36, R41, R42, R43

Table 4.1: Table for mapping requirements to components

Notice that, for the sake of simplicity, the DataBase has been ignored in the table because it is always involved when a component of the Application Server needs to save or to retrieve some data.

In the following lines is explained how the requirements are provided by the components:

- [R1] The system must allow people to register to it providing personal data (name, surname, birthdate, identity card number, fiscal code) and selecting a username and a password: this requirement is provided by the *AccessManager* and the *User Mobile App* components. The *User Mobile App* allows the user to fill blank spaces with username and password and the *AccessManager* handles requests of registration from the client.
- [R2] The system must verify the correctness of the provided personal data of a registered user checking them from the identity card number, blocking the registration if they are not correct: the *AccessManager* checks the received data from the Client calling the *municipality API services* that actually possesses sensible data of the users.
- [R3] The system must allow registered users to login through their username and password, if they provide them correctly: this requirement is provided by the *AccessManager* and the *User Mobile App*. The *User Mobile app* allows registered users to insert username and password while the *AccessManager* handles the requests of login from the user and blocks them if, checking from the DataBase, the inserted data is not correct.

- [R4] The system must allow the logged user to fill a report violation form: this requirement is fulfilled by the *User Mobile App* component that allows the logged user to choose the "Report a violation" feature.
- [R5] The system must let the user select the type of violation detected: the requirement is fulfilled by the *User Mobile App* component, showing an empty space to fill with the type of violation detected in the "Report a violation" process.
- [R6] The system must allow the user to insert the license plate in a violation report: the requirement is fulfilled by the *User Mobile App* component, showing an empty space to fill with the license plate of the car that committed the violation in the "Report a violation" process.
- [R7] While reporting the violation, the system must allow users to take one or more pictures of the potential violation: the requirement is fulfilled by the *User Mobile App* component, allowing user to click on the "Take a picture" button.
- [R8] The system must not allow users to choose pictures not taken in the moment of the report: this requirement is guaranteed by the *User Mobile App* by not allowing client to just pick some random picture from his local storage.
- [R9] The system must collect the current position of the user, using GPS: the *User Mobile App* component ensures this requirement by taking the current position of the User using GPS location.
- [R10] The system must allow user to confirm or delete the current report: the *User Mobile App* component ensures this requirement by either sending the report for a check through the "confirm" button or deleting the current report clicking on the back button.
- [R11] After confirmation, the system must add the current date and time to the report: the *ReportManager* actually takes the requests from the client and attaches to them the current time and date taken from the application server internal clock.
- [R12] The system must store confirmed report: the *ReportManager* is responsible for this requirement, calling the DataBase to actually store the report.
- [R13] The system must check reports to try to find if the pictures of the violations have been modified: the *ReportManager* manages to check if the pictures are fake or not or have been modified.
- [R14] The system must try to find, according to the GPS position of the user and the pictures sent, if the position is fake or not: the *ReportManager* ensures this requirement by running an algorithm to define if the position is correct or not.
- [R15] The system must discard the report if it has been recognized as fake according to the previous requirements (R13-R14): the *ReportManager* is in charge of this, deleting the report if recognized as fake.
- [R16] The system must try to automatically recognize the license plate in the photo, possibly with the help of the value inserted by the user: the *ReportManager* is in charge of this, running an algorithm for text recognition in the picture(s).

- [R17] The involved municipality must be calculated considering in which city the reported violation has been found, based on the GPS position of the user that has sent the report: the *ReportManager* provides this requirement calculating the municipality in charge comparing the GPS position with the ones associated to municipality borders in the database.
- [R18] The system must send the reported violations to the involved municipality: the *ReportManager* and the *NotificationManager* are responsible for this. After having saved it, the *ReportManager* sends the report to the *NotificationManager* that forwards it to the involved municipality.
- [R19] The system must allow an agent to see the reports for its municipality, checking them in order of arrival: the *MunicipalityAgent Web App* component is responsible for this, showing on screen the reports in order of arrival.
- [R20] The system must allow an agent to issue a traffic ticket to a certain person (i.e. license plate) through the correspondent municipality service: the *MunicipalityAgent Web App* allows agents to issue traffic tickets through a specific button in the Web App, calling the MunicipalityAPI to issue the traffic ticket.
- [R21] The system must allow an agent to put on hold a violation report if it needs to be checked in person: the *MunicipalityAgent Web App* ensures this by showing the possibility of putting a report on hold while checking it.
- [R22] The system must allow an agent to discard a violation if it has been verified as fake or it cannot be verified (the vehicle is not there anymore) or it is a duplicated report: the *MunicipalityAgent Web App* ensures this requirement by allowing the agents to discard a report if needed.
- [R23] The system must allow an agent to retrieve the data of the author of a violation report: the *MunicipalityAgent Web App* asks to the Municipality API data about the author of the report.
- [R24] The system must allow an agent to create an account, providing personal data (name, surname, birthdate, rank, badge number): this requirement is ensured by the *MunicipalityAgent Web App* that allows agents to insert their data to register to the application.
- [R25] The system must verify the correctness of the provided personal data of a registering agent checking them through the Municipality Services, blocking the registration if they are not correct: the *AccessManager* is responsible for this, asking the Municipality API to check the identity of the agent.
- [R26] The system must allow an agent to login, inserting its ID and password, blocking the login if the inserted data is not correct: *MunicipalityAgent Web App* and *AccessManager* are responsible for this. The *MunicipalityAgent Web App* allows agents to insert data, while *AccessManager* checks if they are correct.
- [R27] The system must mine information about streets or area from the reported violations: the *DataWarehouse* is responsible for this, performing data mining on reports stored in the DataBase of the application.
- [R28] The system must allow all type of users (even if not authenticated) to select a city they want information about. The user can choose either the city where he is, using the GPS position, or an arbitrary selected location: the *User Mobile App component*, the *MunicipalityAgent*

Web App and the *MunicipalitySupervisor Web App* ensure this requirement by allowing users to choose the city they want.

- [R29] The system must allow all type of users (even if not authenticated) to select information about streets or areas in the city selected and to specify if he wants information for a specific street or area or a classification of streets or areas: The *User Mobile App* component, the *MunicipalityAgent Web App* component, the *MunicipalitySupervisor Web App* component are responsible for this requirement by allowing users to choose the area or street they want.
- [R30] The system must show the data corresponding to the selection of [R29]: the *User Mobile App*, *MunicipalityAgent Web App*, *MunicipalitySupervisor Web App*, *StatisticsManager* and *DataWarehouse* components are responsible for this requirement; the *StatisticsManager* contacts the MapsAPI to provide an image for the city/area/street selected and contacts the *DataWarehouse* to retrieve the required data. This image is then shown by the *User Mobile App/MunicipalityAgent Web App/MunicipalitySupervisor Web App* component to the user.
- [R31] The system must take information about accidents and tickets from the municipality: this requirement is ensured by the *DataCollectionManager* that periodically queries the Municipality API to retrieve new informations and saves them on the DataWarehouse.
- [R32] The system must use information about accidents and tickets to build statistics, crossing them with reported violations: the *DataWarehouse* component ensure this requirement building statistics exploiting data mining techniques.
- [R33] The system must not allow common users to see confidential data about other people: the *User Mobile App component* and *MunicipalityAgent Web App* is in charge for this requirement by showing only data that does not violate privacy. This requirement is also guaranteed by a subcomponent in the *Router* (i.e. *AuthChecker*) to check if a request made by a specific client is valid or not.
- [R34] The system must allow all type of users (even if not authenticated) to choose a topic: areas or streets with most accidents, areas or streets with the highest number of traffic tickets issued, areas or streets where there have been the best improvements, information for a specific area or street: the *User Mobile App* component, the *MunicipalityAgent Web App* component, the *MunicipalitySupervisor Web App* component provide this requirement by allowing users to choose the topic they're interested into.
- [R35] The system must show to the user the information about the topic selected according to [R34]: the *User Mobile App component*, the *MunicipalityAgent Web App* component and the *MunicipalitySupervisor Web App* component ensure this requirement by showing as an image the topic requested to the user.
- [R36] The system must allow authenticated supervisors to retrieve information about the vehicles with the highest number of violations in a selected area or street: the *MunicipalitySupervisor Web App* allows supervisors to select the statistics they want; the *StatisticsManager* asks the the *DataWarehouse* the type of Data requested by the User.
- [R37] The system must allow supervisors to access only information about their own municipality: the *MunicipalitySupervisor Web App* is responsible for this, allowing supervisor only to check Statistics and suggestions about his city. This requirement is also guaranteed by a

subcomponent in the *Router* (i.e. *AuthChecker*) to check if a request made by a specific client is valid or not.

- [R38] The system must allow a supervisor to create an account, providing personal data (name, surname, birthdate, rank, badge number): this requirement is ensured by the *MunicipalitySupervisor Web App* that allows supervisors to insert their data to register to the application
- [R39] The system must verify the correctness of the provided personal data of a registering supervisor, checking them through the municipality services, blocking the registration if they are not correct : the *AccessManager* is responsible for this requirement, it receives the request and ask the *MunicipalityAPI* to check if the inserted identity is correct.
- [R40] The system must allow a supervisor to login, inserting its ID and password, blocking the registration if the inserted data is not correct: the *MunicipalitySupervisor Web App* and the *AccessManager* are responsible for the requirement. The *MunicipalitySupervisor Web App* allows supervisor to enter his ID and password, while the *AccessManager* checks in the databases if the inserted data is correct.
- [R41] The system must take information about accidents, tickets and street networks (bike lanes, sidewalks, parking areas, ...) from the municipality, exploiting the municipality services: the *DataCollectionManager* is responsible for this requirement periodically asking data to the *Municipality API* and saving it in the *DataWarehouse*.
- [R42] The system must elaborate information about accidents, tickets and streets, combined with reports information, and try to find possible solutions for problems: the *DataWarehouse* component is responsible for this requirement trying to aggregate data coming from the different databases concerning reports, issued traffic tickets and accidents.
- [R43] The system must notify the municipality about new possible interventions: the *NotificationManager* the *DataWarehouse* and *SuggestionManager* components are responsible for this requirement; the *DataWarehouse* component forwards the new intervention to the *SuggestionManager* that forwards it to *NotificationManager* that sends it to the involved municipality.

5 Implementation, integration and test plan

5.1 Implementation and unit testing

To define the implementation and test plan it is useful to identify the level of importance for the customer of the various components and also their difficulty of implementation. They are summarized in table 5.1. The external components (*MunicipalityAPI* and *MapsAPI*) are considered to be ready from the beginning and they will not be further mentioned in this section.

Component	Customer importance	Implementation difficulty
MobileApp	Medium	Low
AgentWebApp	Medium	Low
SupervisorWebApp	Medium	Low
Router	Medium	Medium
SuggestionManager	Low	High
AccessManager	Medium	Low
ReportManager	High	Medium
StatisticsManager	Low	Medium
NotificationManager	Medium	Low
DataCollectionManager	Low	Medium
DBMS	High	Medium
DW	Low	Medium

Table 5.1: Classification of components based on customer importance and implementation difficulty

The implementation plan depends heavily on the dimension and the composition of the development team, which define the level of parallelism. We assume to have 1 programmer expert in data field, 1 for the front end and 2 for the back end components. The front-end developer will develop in order the *Mobile App*, the *MunicipalityAgent WebApp* and the *MunicipalitySupervisor WebApp*.

The data developer will develop in order the *DBMS*, the *Data Warehouse* and finally the *Dat-*

aCollectionManager, considering the lower importance that this last two components have and considering useful to have already the *DataWarehouse* when implementing the *DataCollectionManager*.

The two back-end developers will develop the remaining modules in the following order, considering once again their priority for the customer and their dependencies:

- 1) **AccessManager => Router => StatisticsManager => SuggestionManager**
- 2) **ReportManager => NotificationManager => SuggestionManager**

The *SuggestionManager* component is the most difficult to be implemented, considering the necessity to elaborate an algorithm for the suggestions generation, therefore it will require the two developers to work together.

Figure 5.1 shows a *Gantt chart* for the implementation process.

All modules need to be tested individually, reaching the 80% of code coverage. The white

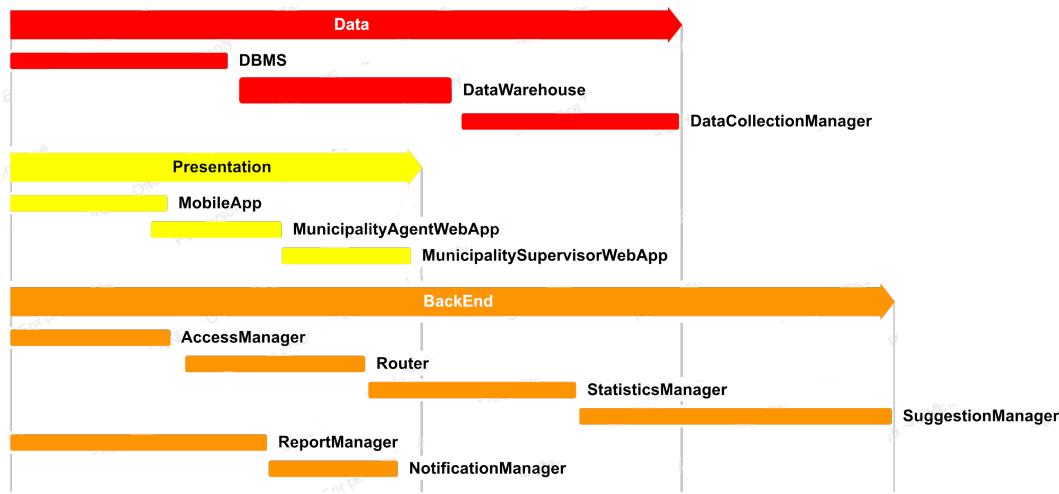


Figure 5.1: Gantt chart for the implementation process

testing technique will be adopted for the unit tests and various components will require stubs:

- *AccessManager*, *ReportManager*, *NotificationManager* will require a stub for the *DBMS*
- *ReportManager*, *SuggestionManager* will require a stub for the *NotificationManager*
- *NotificationManager* will require stubs for *UserMobileApp*, *AgentWebApp* and *Supervisor-WebApp*
- *Router* will require stubs for *ReportManager*, *StatisticsManager*, *SuggestionManager*, *AccessManager* and *DBMS*

For the *AutomaticReportAnalyzer* subcomponent of the *ReportManager* it will be particularly important to have a large number of tests to ensure to have a discrete level of performance in the identification of fakes.

For the *MobileApp* it will be important to test it on various types of mobile devices (smart-phones and tablets), with different versions of Android and iOS. The same holds for the *WebApps*, they will need tests on various browsers (*Google Chrome*, *Mozilla Firefox*, *Microsoft Edge*), versions and operating systems.

5.2 Integration testing

Once that some modules are completed and tested alone they can be integrated, following a thread-oriented bottom-up order:

- Once the *DataWarehouse*, the *DataCollectionManager* and the part of *DBMS* related to *streets* are ready they can be integrated (*data collection thread*)
- Once the *ReportManager* the part of *DBMS* related to *reports*, *tickets*, *streets* and *users* and the necessary function of *NotificationManager* are ready they can be integrated (*report management thread*)
- Once the *AccessManager* and the part of *DBMS* related to *users* are ready, they can be integrated (*access thread*)
- Once the *data collection thread* has been integrated and the *StatisticsManager* is ready, they can be integrated (*statistics thread*)
- Once the *data collection thread* has been integrated and the *SuggestionsManager* and the necessary function of the *NotificationManager* are ready, they can be integrated (*suggestions thread*)
- Once the *ReportManager*, the *StatisticsManager*, the *AccessManager*, the *SuggestionManager*, the *Router* and the part of *DBMS* related to *users* and *accesses* are ready, they can be integrated (*facade thread*)
- Once the *UserMobileApp*, the *MunicipalityAgentWebApp*, the *SupervisorWebApp* and the *Router* are ready, they can be integrated (*presentation thread*)
- Once the *UserMobileApp*, the *MunicipalityWebApp*, the *SupervisorWebApp* and the *NotificationManager* are ready they can be integrated (*notification thread*)

The integration tests will be black box tests, aimed to verify the proper working of all functions in the same way as unit tests, but substituting stubs and drivers with the real corresponding modules.

In figures 5.4, 5.2, 5.3 and 5.5 the integration threads are highlighted on the component diagram.

5.3 System testing

After the integration testing the system must be subjected to system tests.

First of all the system will be subjected to an ***alpha test***, executed by the *SafeStreets* developers in their production environment, checking all functions with a good numbers of test cases (it should be checked that they cover at least 80% of the code).

After the check of the **functional** requirements this test will also verify the **non-functional** requirements, in particular the performances under an heavy load: the database and the datawarehouse will be charged with a big quantity of data (similar to the expected data for tens of municipality in a decade) and all functions will be tested, checking their elaboration and response times.

The second and most important system test is the ***beta test***, which will be executed in a real production environment, with the support of a municipality: a group of voluntary citizens will install and try the mobile app, while a selected group of agents and supervisors will use web

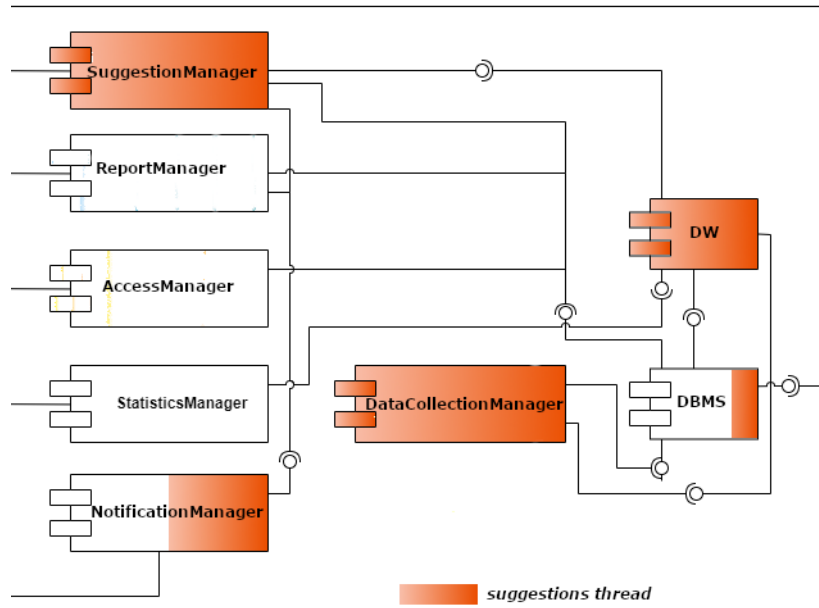


Figure 5.2: Integration diagram for suggestions

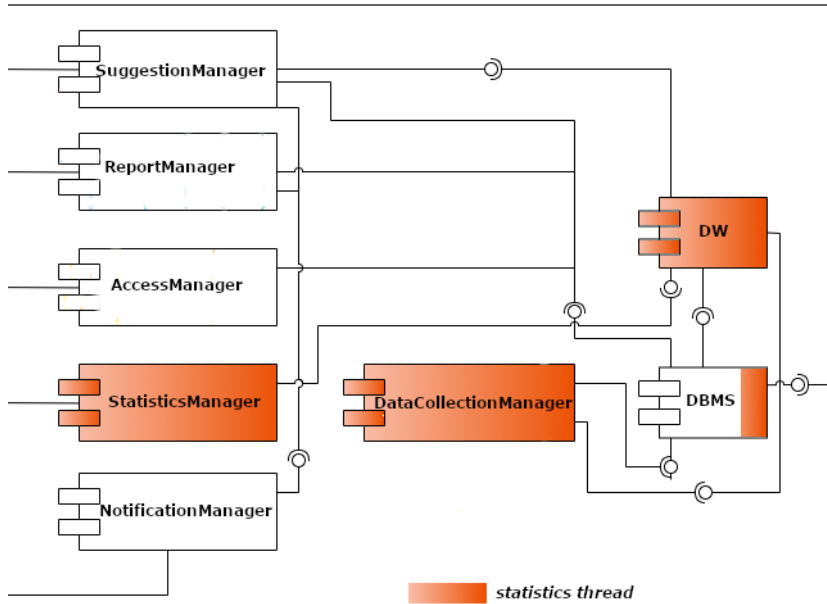


Figure 5.3: Integration diagram for statistics

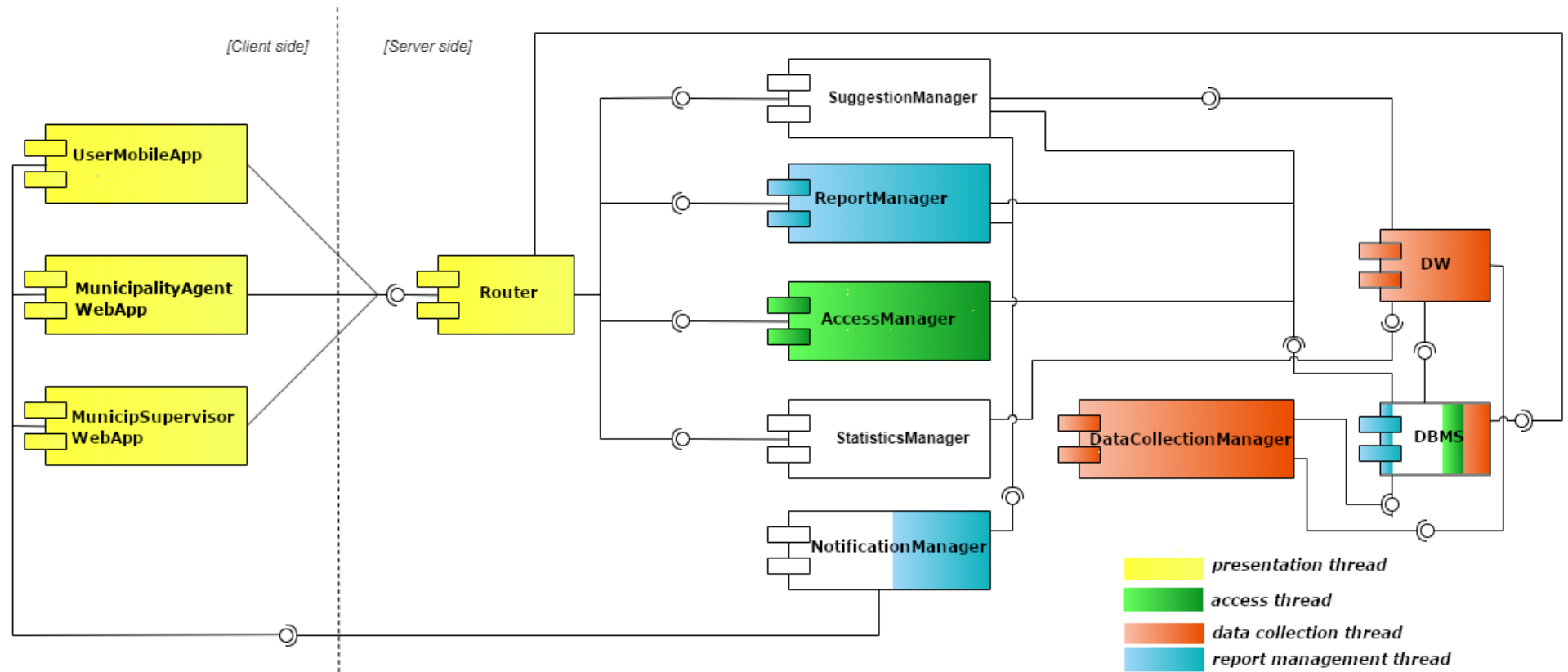


Figure 5.4: Integration diagram for access, data collection, report management and presentation

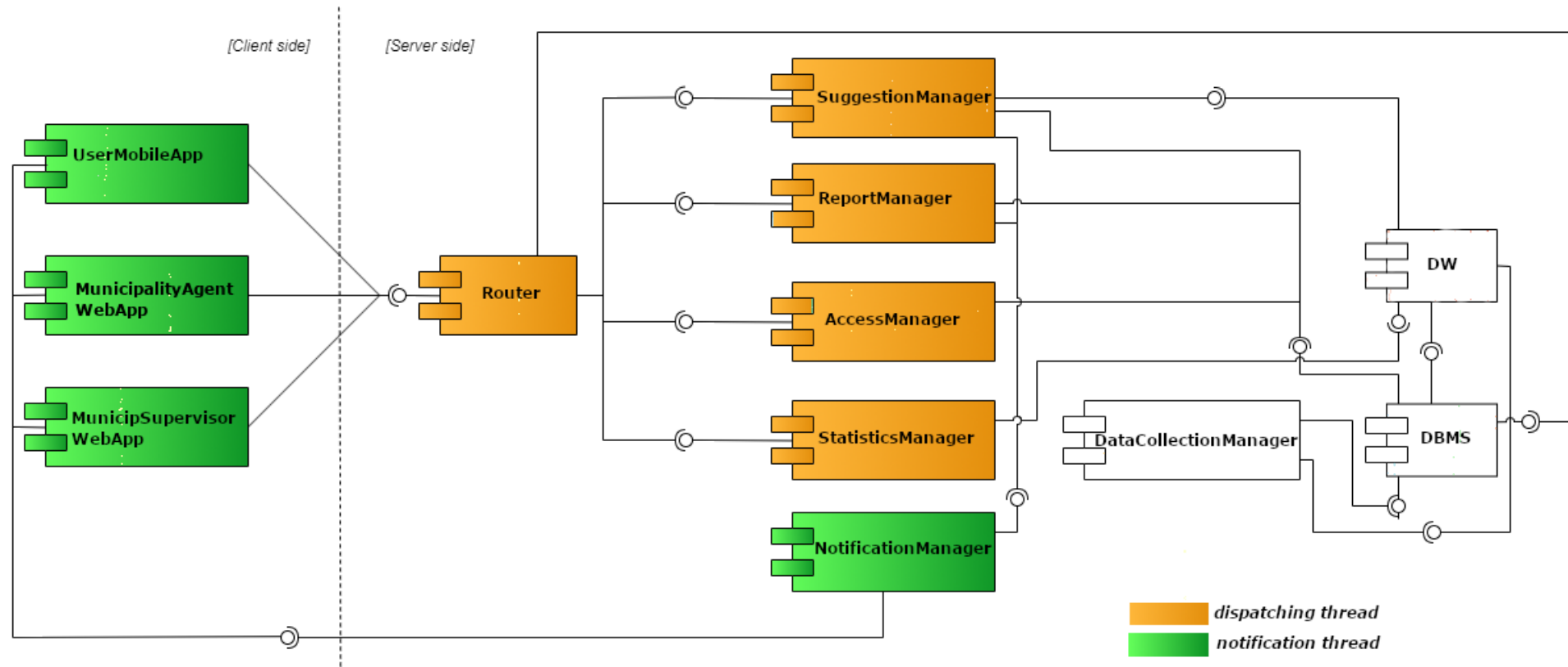


Figure 5.5: Integration diagram for dispatching and notification

apps.

This is particular important to verify another time the correct behavior of the system (i.e. the absence of bugs), in a real distributed environment, but also to verify the **usability** of the system and the correctness w.r.t to the needs of the various types of users (**acceptance test**).

6 Effort spent

Nicola Rosetti		
<i>Date</i>	<i>Hour</i>	<i>Section</i>
18-11-2019	1.5 h*	Component and high-level architecture analysis
21-11-2019	1.5 h	Other design decision part: consistency and replication
23-11-2019	2 h	Introduction and high-level architecture in chap 2
25-11-2019	3 h	Requirements traceability
27-11-2019	1 h	Runtime View
28-11-2019	1 h*	RASD Update
29-11-2019	3 h	Requirements traceability
29-11-2019	2 h	Chapter 1 introduction
29-11-2019	1 h	Deployment view
1-12-2019	2 h	Mockup and general revision
2-12-2019	2 h	Selected architectural styles and patterns section
3-12-2019	1.5 h	Chapter 3 and activity diagram
4-12-2019	1 h	activity diagrams
5-12-2019	1 h*	minor updates and fixes
7-12-2019	1 h	activity diagrams
7-12-2019	1 h	RASD update
8-12-2019	1 h	general revision

Table 6.1

Simone Sartoni		
<i>Date</i>	<i>Hour</i>	<i>Section</i>
18-11-2019	1.5 h*	Componentst and high-level architecture analysis
24-11-2019	2.5 h	System architecture diagram
24-11-2019	1.5 h	Deployment diagram
27-11-2019	1.5 h	Update deployment diagram, requirements traceability revision and Requirements Traceability Matrix creation
28-11-2019	3 h	Requirements Traceability Matrix update and Interface Diagram
29-11-2019	1.5 h	Components Interface Diagram
03-11-2019	3 h	Components Interface Diagram update
04-11-2019	1.5 h	Components Interface Diagram description
07-11-2019	1.5 h	Minor fixes and changes
08-11-2019	2 h	Chapter 2 final revision and Deployments Diagram update
09-11-2019	1.5 h	Chapter 3 final revision and Component Interfaces Diagram update
09-11-2019	2 h	Chapter 4 final revision

Table 6.2

Vittorio Torri		
<i>Date</i>	<i>Hour</i>	<i>Section</i>
18-11-2019	1 h*	Componentst and high-level architecture analysis
21-11-2019	1 h	Components diagram
22-11-2019	0.5 h	Components diagrams
23-11-2019	1.5 h	Components diagrams and components view
24-11-2019	2 h	Sequence diagrams and components view
26-11-2019	1 h	Sequence diagrams
27-11-2019	1 h	Sequence diagrams
28-11-2019	1 h	Sequence Diagram
30-11-2019	2 h	Mockup
01-12-2019	2 h	Deployment and Runtime View, Scope
02-12-2019	1 h	Implementation and testing
03-12-2019	2 h	Implementation and testing
04-12-2019	1 h	Integration diagrams
05-12-2019	1 h*	Minor changes and fixes
06-12-2019	1.5 h	E/R and DFM diagrams
07-12-2019	2 h	Web Registration mockup, gantt chart
07-12-2019	1 h	Class diagram update and minor fixes
08-12-2019	1 h	Minor fixes

Table 6.3

* *Group work*

7 References

- *[FACADE-PATTERN]*https://en.wikipedia.org/wiki/Facade_pattern - definition of facade pattern
- *[MVC-PATTERN]*<https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller> - definition of MVC pattern
- *[RESTFUL-ARCHITECTURE]*<https://restfulapi.net/rest-architectural-constraints/> - definition and characteristics of RESTful architecture
- *[UML-ACTIVITY]*https://www.academia.edu/5041934/User_Interface_Modelling_with_UML-User_Interfaces_Modelling_using_UML, Pinheiro Paulo, Paton Norman, 2002/11/23
- *[PUBSUB-PATTERN]*https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern - Wikipedia page for publish - subscribe pattern