POLITECNICO
MILANO 1863

Software Engineering 2
A.Y. 2019/2020

SafeStreets

Design Document

Alessandro Fulgini — alessandro.fulgini@mail.polimi.it
Federico Di Cesare — federico.dicesare@mail.polimi.it

Version 1.0
December 9, 2019

Deliverable:	DD
Title:	Design document
Authors:	Alessandro Fulgini, Federico di Cesare
Version:	1.0
Date:	December 9, 2019
GitHub Repository:	github.com/fuljo/FulginiDiCesare-SE2
Copyright:	

Acknowledgments

Typeset with LaMBraTeX
Sans-serif font: Titillium by Accademia di Belle Arti di Urbino
Monospace font: Consolas
UML diagrams generated with PlantUML and draw.io

Contents

1 Introduction

1.1 Purpose

1.1.1 General purpose

The general purpose of the SafeStreets project has already been shown in the RASD document, here we only provide the definition of the goals as a quick reference. This document's purpose is to show both general and specific details of the design choices made for the future development of SafeStreets. We will discuss the design from different points of view, starting from a more general and high-level representation of the system in the Overview (see section 2.1). Then a more specific view is given with the Component Diagram (see section 2.2). Other low level choices will be discussed in the following sections. Then all the architectural styles will be shown as well as the design patterns used. A final chapter will be dedicated to the schedule of the development plan and the testing phase.

1.1.2 Goals

G1 The system must allow users to notify authorities of parking violations, by uploading pictures of the violation together with metadata (date, time, position, infraction type, licence plate of the vehicle)

G2 The system must be able to extract the licence plate number from the photos uploaded by the users

G3 The system must persistently store the violations uploaded by the users, while keeping the uploader anonymous

G4 The authorithy must be able to visualize, accept and refuse the single violation reports that occurred in its competence area

G5 The authorithy must be able to visualize violation statistics based on their location, time, type or responsible vehicle

G6 The user must be able to visualize anonymized violation statistics only based on their location, time and type

G7 The system must be able to determine unsafe areas by matching data about violations and data about accidents. It must then use such data to suggest actions that the authorithy can take to help preventing accidents

1.2 Scope

SafeStreets is a service that can be offered and managed by authorities to willing citizens who want to contribute to their city's public order. In this way the authorities can improve their control over traffic violations.

The service is supposed to be provided in Italy, so we assume that each municipality has one reference authority (e.g. police department) that is responsible for handling traffic violations in that area. The authority may activate the service or not in its area.

First of all, the user must register to the service by providing his/her fiscal code, a username and a password of his choice. By requiring the fiscal code to be unique for each user, we can avoid duplicate accounts for the same citizen.

When the user detects a violation, he can open the software on his mobile device and take a picture. The system will elaborate the picture in order to auto-detect the licence plate number. Furthermore, the app will get from the device the date, time and GPS position. The auto-collected data will be shown to the user who can correct it before being sent to the authority. The only data the user must manually insert is the infraction type, which will be chosen from a finite list of possible types. This process will be designed to be as smooth as possible, to offer a clean user experience. The user won't be able to send the report if there exists no responsible authority (that has activated SafeStreets) for the municipality in which the violation occurred.

From the authority point of view, the violation reports are handled by an operator, who is identified by an username (must be unique) and a password. The operator will review the violation and then decide whether to accept it. He may discard the violation when it contains erroneous or incomplete data. For example the photo may simply not show a violation, or it may happen that the licence plate in the picture is partially unreadable and the number inserted by the user is likely to be wrong.

After accepting the violation, the authority might schedule intervention on the ground (e.g. remove the vehicle from the road). Scheduling the intervention, however, is out of the scope of the software to be.

The data collected is of course persistent as the S2B will offer various statistics to citizens and authorities. The authorities will be able to view statistics on the number and types of violations. The statistics can be filtered by violation type, location, date/time and vehicle. Citizens will also be able to view statistics, but in order to keep privacy they won't be able to see the vehicles involved.

For the data about accidents, we assume that a municipality may provide it or not. Therefore the function which suggest actions to prevent accidents can only be offered in municipalities that provide such data. The data is supposed to contain the location, timestamp and accident type for each accident. We also assume that there are a finite pre-defined set of accident types.

The various phenomena that are relevant or partially relevant for the system, are summarized in table 1.1 on the following page.

Table 1.1: Table of phenomena. Phenomena controlled by the world are marked with *W*, while the ones controlled by the machine are marked with *M*

Phenomenon	Controlled by	Shared
A user registers to the service	W	✓
A vehicle commits a parking violation	W	✗
A user takes a photo	W	✓
A user reports a violation	W	✓
An operator checks the reported violations	W	✓
An operator accepts/refuses a violation report	W	✓
The Police emits a fine for a violation	W	✗
The system computes statistics based on the stored violations	M	✗
A user/operator views the statistics	W	✓
The system retrieves data about accidents	M	✗
The system computes suggested actions	M	✗
An operator checks for new suggested actions	W	✓
An operator marks a suggested action as *carried out*	W	✓
The system determines whether a carried out action has been useful or not	M	✗

1.3 Definitions, acronyms, abbreviations

Definitions

User a citizen, identified by his fiscal code, who is able to report violations to authorities and view anonymized statistics. In order to access the functionalities the user must first register with an username and a password.

Authority the public organization which enforces traffic laws in one or more municipalities. The authority is said to be *competent* for those municipalities. The authority is the recipient of the violation reports that occurred in its *competence area*.

Violation a *traffic law infraction*, in particular regarding parkin. In the context of the S2B it refers to the complete set of data sent by users to authorities: photo, date, time, type and licence plate number of the involved vehicle. Also referred to as *infraction*.

Operator an authority employee that reviews the violations sent and decides whether to keep them.

Acronyms

ACID Atomicity, Consistency, Isolation, Durability
API Application Program Interface
CSS Cascade Style Sheet

DB Database
DBMS Database Management System
DMV Department of Motor Vehicles
ER Entity Relationship
GPS Global Positioning System
HTTP Hypertext Transfer Protocol
JSON Javascript Object Notation
ML Machine Learning
OCR Optical Character Recognition
OS Operating System
REST Representational State Transfer
RPC Remote Procedure Call
S2B Software To Be
SQL Structured Query Language
TLS Transport Layer Security
UI User Interface
URL Uniform Resource Locator
UX User Experience

Abbreviations

Gn n^{th} goal
Dn n^{th} domain assumption
Rn n^{th} requirement

1.4 Revision history

Version 1.0 (09/12/2019) Initial version, based on version 1.1 of the RASD

1.5 Reference documents

[4] Elisabetta Di Nitto and Matteo Rossi. *Mandatory Project: goal, schedule and rules.* 2019.

[6] Alessandro Fulgini and Federico Di Cesare. *Safestreets: Requirement Analysis and Specification Document.*

[8] "ISO/IEC/IEEE International Standard - Systems and software engineering – Life cycle processes – Requirements engineering". In: *ISO/IEC/IEEE 29148:2018(E)* (Nov. 2018). DOI: 10.1109/IEEESTD.2018.8559686.

1.6 Document structure

The present document is divided into chapters. Here we give a brief description of each chapter and its intended audience.

Chapter 1 introduces the document and gives a general idea on the contents that will be found in the following chapters. Contains also the revision history, references and the abbreviations and acronyms.

Chapter 2 is the core chapter of the document. In the first part gives a general idea on how the system will be, showing a high-level view of the various parts and the layers. Then it goes into details with the Component Diagram, Interface Diagram and the Database Design. The Runtime View then describes how the components work together during the main tasks. Finally the architectural choices are shown. This chapter is intended for *back-end developers* and the *sysadmin* as it shows how the back-end works and how the various application components interact.

Chapter 3 describes how the User Interface works and how it interacts with the user. The intended audience of this chapter are the *front-end developers*.

Chapter 4 provides a mapping between requirements and the components in charge to satisfy them. Each requirement is analyzed in details and also a brief description of how the component(s) will satisfy the requirement is given. Then there is a section that discusses the non-functional requirements and how the architectural choices fulfill them. This chapter is intended for *developers*, *testers* and also for *designers*.

Chapter 5 outlines a general schedule of implementation and testing. It starts with a general description of the various functionalities and how difficult they are to implement and integrate, then the implementation plan is discussed in details. Finally there is a section dedicated to the testing phase, beginning with code inspection and then ending with unit and integration test. The intended audience for this chapter are *testers*, *developers* and the *Project Manager* who can use this chapter as a reference to check whether the project development is sticking to the plan.

2 Architectural design

2.1 Overview

At a high level, the system is divided into three *layers*:

Presentation layer It contains the two types of clients (mobile application and website) from which the citizens and authorities will access the service.

Application layer It contains all the business logic to handle incoming data from the users and other external sources, to compute suggestions and satisfy queries coming from the presentation layer. Here a Web Server only provides *static content* for the web application, while the *dynamic data* is provided through a REST API which is shared between the mobile app and the web app.

Data layer It is responsible for storing all the data needed by the system, ensure its persistence over time, even in case of failure, and to make it available to the application components.

Each layer sees the other layers as single entities, hiding the internal replication and complexity. In particular the usage of a REST model for client-server communication provides a high level of *uncoupling* between the two layers and allows for efficient *replication* of the application servers, which is a key point to achieve fault tolerance and scalability.

Figure 2.1 on the next page shows an overview of the physical architecture, where we can notice the high degree of replication in the application layer and the need for *load balancers* to manage the replicas. Regarding the database we chose a relational DBMS. Here the entire data-set should be partitioned on different nodes which share a single manager, since a fully-distributed solution is difficult to manage while having to guarantee the ACID properties. We also represented the two external data sources that the system needs to access: the *DMV Service* to validate licence plate numbers and the *Accident Data Service* provided by some municipalities and needed to provide Smart-Suggestions functionality. Finally, we represented a logical overview of the network that connects the various components, underlining the fact that all external communication travels on the internet and also the need for firewalls to protect the internal components (application and data layers) from malicious attacks.

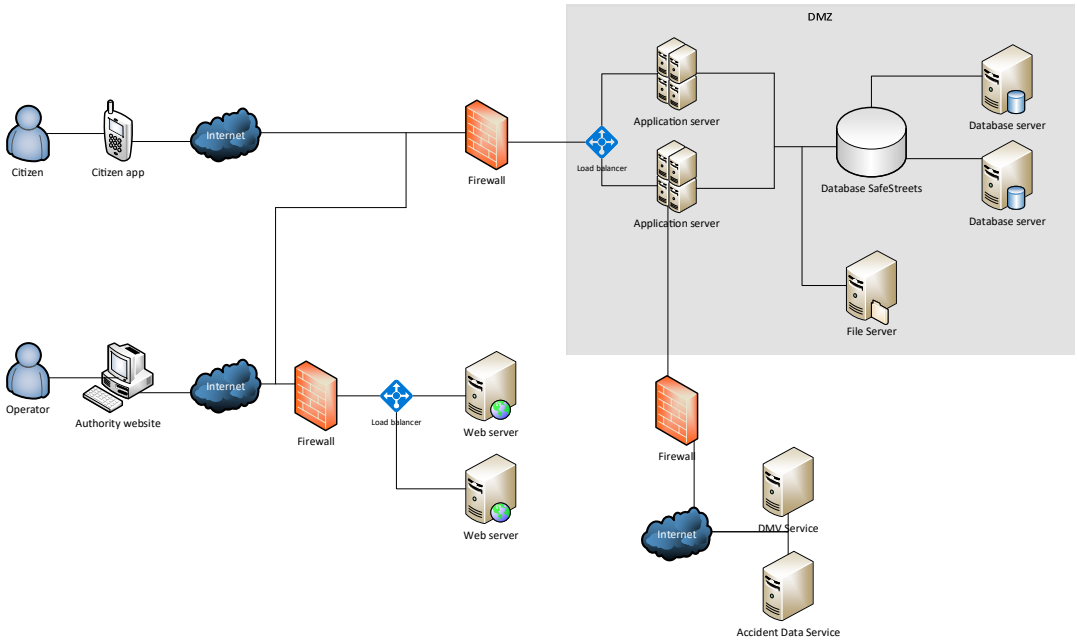

Figure 2.1: Overview of the physical architecture

2.2 Component view

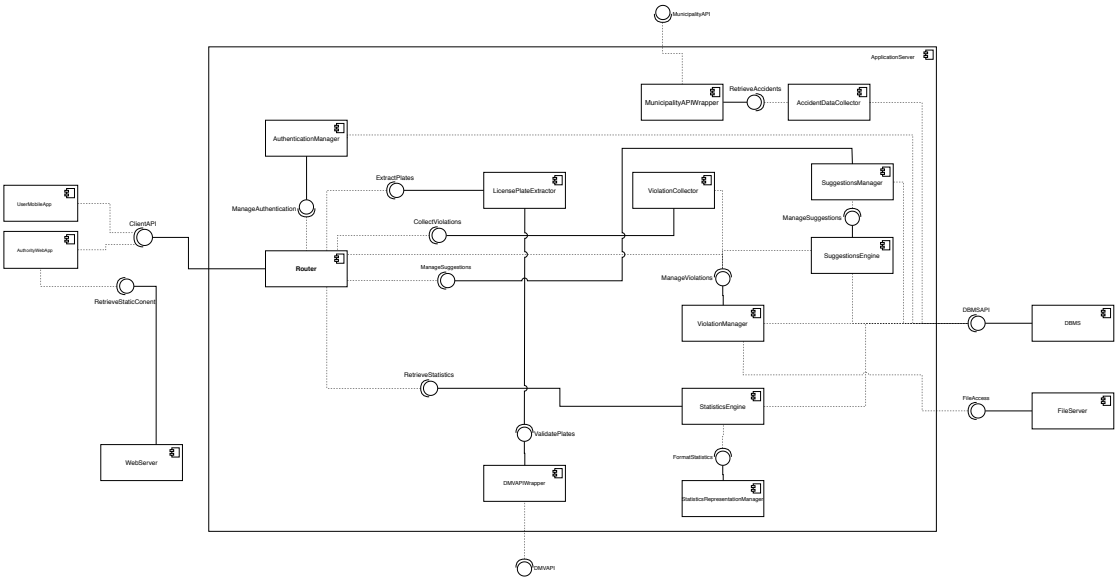

Figure 2.2: Component Diagram, with an exploded view of the ApplicationServer subsystem.

The system is composed of four subsystems:

UserMobileApp This is the smartphone app that provides all the functionalities to the citizens. It handles the presentation and the user interaction locally, while it interacts with the application server by using the *ClientAPI* (RESTful).

AuthorityWebApp This is the web application used by authorities. It handles the presentation and user interaction. The data to be displayed is retrieved using the REST API provided by the application server (*ClientAPI*).

WebServer This is a web server that provides static assets for the *AuthorityWebApp*, such as HTML documents, the JavaScript code that runs the web app, CSS stylesheets and static images such as icons, etc. In this way web server replicas can be brought up or down depending on the current load. Since the replicas are all equal and completely stateless, each request can be equivalently routed to any of them. The only occasion in which the replicas must be synced is when a part of the web app code (or other asset) is changed, which can be considered a maintenance operation.

ApplicationServer This is the component responsible for carrying out the business logic of the system. The application data related to SafeStreets is stored and retrieved from a relational DBMS. The component also accesses a file server to store and retrieve the images of the reported violations. Finally, it uses the interfaces offered by the Department of Motor Vehicles (*DMVAPI*) and the various municipalities (*MunicipalityAPI*).

DBMS This component is the central relational database, used by the ApplicationServer. It stores all the application data that needs to be persisted (users, violations, retrieved accidents, municipality and authority info, access tokens), except the actual pictures of the violations. Of course, this component must not be implemented from scratch, but we just need to choose a commercially available solution that meets the requirements and configure it. It is required for the DBMS to support *distribution* with a single transaction manager and also working with *geographical data types*, to speed up queries based on locations.

FileServer It is only responsible to store the images attached to the violation reports and making them available to be retrieved by the clients via the REST API.

In figure 2.2 on the preceding page we further detail the internal components of the *ApplicationServer* subsystem, which is the core of the system. Here we also provide a description of the various components and their interactions:

Router This component exposes a public REST API, named *ClientAPI* shared between the mobile and the web clients, which they use to perform all requests to the server (authentication and dynamic data in general). Each request type is assigned to a different REST endpoint (identified by an URL) and is then routed to the appropriate component, which will then carry it out autonomously and respond to the client (if required).

AuthenticationManager It is responsible for the registration and login of both citizens and authority operators. Since it has to read and write user data, it needs to interact wit the *DBMS*. This component is also responsible for handling sessions: after a successful login the component generates an *access token* for the specific user, which is valid for a limited time. The token is then stored in the DB and can be accessed by the other components and finally it gets sent to the client that sent the login request. Now, when the client makes a request, it also provides the token which will be checked against the DB by the component that will handle the request.

LicencePlateExtractor This component receives violation photos uploaded by the users and tries to extract the biggest licence plate number it an find in the photo. Then it checks if the detected plate exists by using the DMV API. If a valid licence plate number has been extracted, the component sends it to the client as a response, so it can autocomplete the licence plate field in the request report. This task has been separated from the violation reporting to take load off from the *ViolationCollector* and also to give users the ability to manually insert the licence plate number if it couldn't be detected or to correct it.

ViolationManager This component provides primitives to store, retrieve and review violations (i.e. mark them as accepted or discard them). It is used by the ViolationCollector to store an incoming violation and in this regard it is responsible to check and assign it to a responsible authority, based on the municipality in which

the violation occurred. If this fails, the violation cannot be stored. Storing the violation also includes saving its attached photo to the *FileServer*. In turn, when an operator requests to review violations, this components is activated and it is responsible to only retrieve the violations assigned to the requesting authority and sending them as a response. When the operator decides to mark a violation as accepted or discarded, this component is responsible for applying the change in the DB; note that it is necessary to check that the violation was assigned to the requesting authority by using the access token provided with the request.

ViolationCollector This component is responsible to handle incoming violation reports from the users, in particular to check their validity and then use the *ViolationManager* to store them in the DB. In particular, it checks that the report contains all the required fields and that the licence plate number is valid (by using the DMV API).

StatisticsEngine This component is responsible for computing statistics for both users and authorities. A statistics request will come in, indicating the type of statistics (e.g. violations per month) and possibly various filters (e.g. municipality, year, …); the component will perform a query to the DB and compute the required statistics. These are then passed on to the *StatisticsRepresentationManager* and the result is sent as a response to the client.

StatisticsRepresentationManager Given some *raw* statistics, this component is responsible to generate their appropriate representation to be fed to the client libraries used to show statistics. This takes into account the diagram colors, text labels, measurement units and all the parameters needed by the client libraries to visualize the appropriate diagram. The reason for this to be done server-side is that we don't have to send raw data over the network and we can provide a consistent representation between the mobile app and the website (in terms of colors, labels, etc.).

AccidentDataCollector This component is responsible for periodically collecting data about accidents from the municipalities that provide it and store it to the database. As we stated in the RASD, we assume that the municipalities provide an API to retrieve all accidents that have been registered after a certain date; the component can rely on this to perform incremental updates, rather than retrieving all the data each time. Since the accident data will be used to compute suggestions, a task that requires a lot of accesses, we chose to store it locally rather than continuously contacting the municipality API. This is also dictated by the fact that we have no control over the availability of the external data sources.

SuggestionsEngine This component is responsible to periodically compute new suggestions for the authorities that have activated the *SmartSuggestions* functionality. It reads data about violations and accidents directly from the DB and then computes the suggestions by using Machine Learning algorithms. These suggestions are then stored into the DB, so they can be retrieved by the authority. Note, however, that

no notification is actively sent to the authority. Furthermore, since there can be a lot of municipalities involved, the component must also be able to schedule its jobs over time, in order to keep up to date with new violations and accidents data.

SuggestionsManager Provides methods to retrieve suggestions and mark them as carried out. Analogously to *ViolationManager* it receives requests by authority web apps containing an access token, and it has to allow access only to the suggestions for the authority that owns the token.

DMVAPIWrapper The components that need to check the validity of a licence plate number don't directly interact with the external API, but with this internal component that wraps around the API. For details on the advantages of this approach see 2.6.5

MunicipalityAPIWrapper Similarly to the previous one, we provide a wrapper also for the APIs offered by municipalities to retrieve data about accidents. Here the need for a component that converts the various data formats provided by the external APIs to an internal, shared format is even more evident, since it is quite unlikely that all the municipalities provide such data in the same format.

2.2.1 Database design

Here we present a *conceptual model* of the database through the entity-relationship diagram in figure 2.3 on the next page, according to the *Chen* notation; optional (nullable) attributes are represented with dotted lines.

This model includes all the entities and attributes that have been identified so far. The task of translating it to the relational model is left to the implementation phase, since the choice of the attribute types and other smaller decisions may depend on the specific implementation of the DBMS itself. Furthermore, additional attributes or tables may be added if needed, such as metadata needed by the suggestions engine to keep track of its jobs. It should also be noted that most DBMS systems usually keep track of when each tuple has been *created* and *last modified*; these timestamps can be used by the *SuggestionsEngine* to check if new data has been introduced since the last run, without the need of wasting space by introducing new attributes.
Now we give further details that could not be represented in the diagram:

- in the **AccessToken** entity, the *expiration* attribute is a timestamp that limits the validity of the token.
- in **Municipality** the *APIUrl* attribute identifies the REST endpoint used to retrieve data about accidents, while *lastCollected* and *lastSuggestionsRun* are two timestamps indicating the last time the API was queried to collect new accidents and the last time that the *SuggestionsEngine* was run to compute suggestions for that specific municipality, respectively. In case the municipality does not offer this service, all three fields are set to `NULL`.
- both **Violation** and **Accident** are related to a *municipality*, but they must also be relatable to an *authority* as specified by requirement R11; therefore an additional integrity rule to check this must be put in place.
- both **Violations** and **Accidents** are given artificial unique identifiers, they also have a *type* attribute which is of the enumerator type, since by domain assumptions there are a finite set of types for each of these entities. The *timestamp* attribute indicates when the violation/accident occurred. The format for the *licencePlate*, *longitude* and *latitude* has been described in the RASD (section 3.4.1 – Standards compliance)
- in the **SuggestedAction** entity *latitude* and *longitude* indicate the (approximate) location where the suggested action has to be implemented. The *status* has attributes can take two values: *pending*, given when the suggestion is created and *carried out*, which is set when the suggestion is marked by the operator, in this case the value of *carriedOutTimestamp* must also be set accordingly, to allow the evaluation of the effectiveness of this action by the *SuggestionsEngine*.

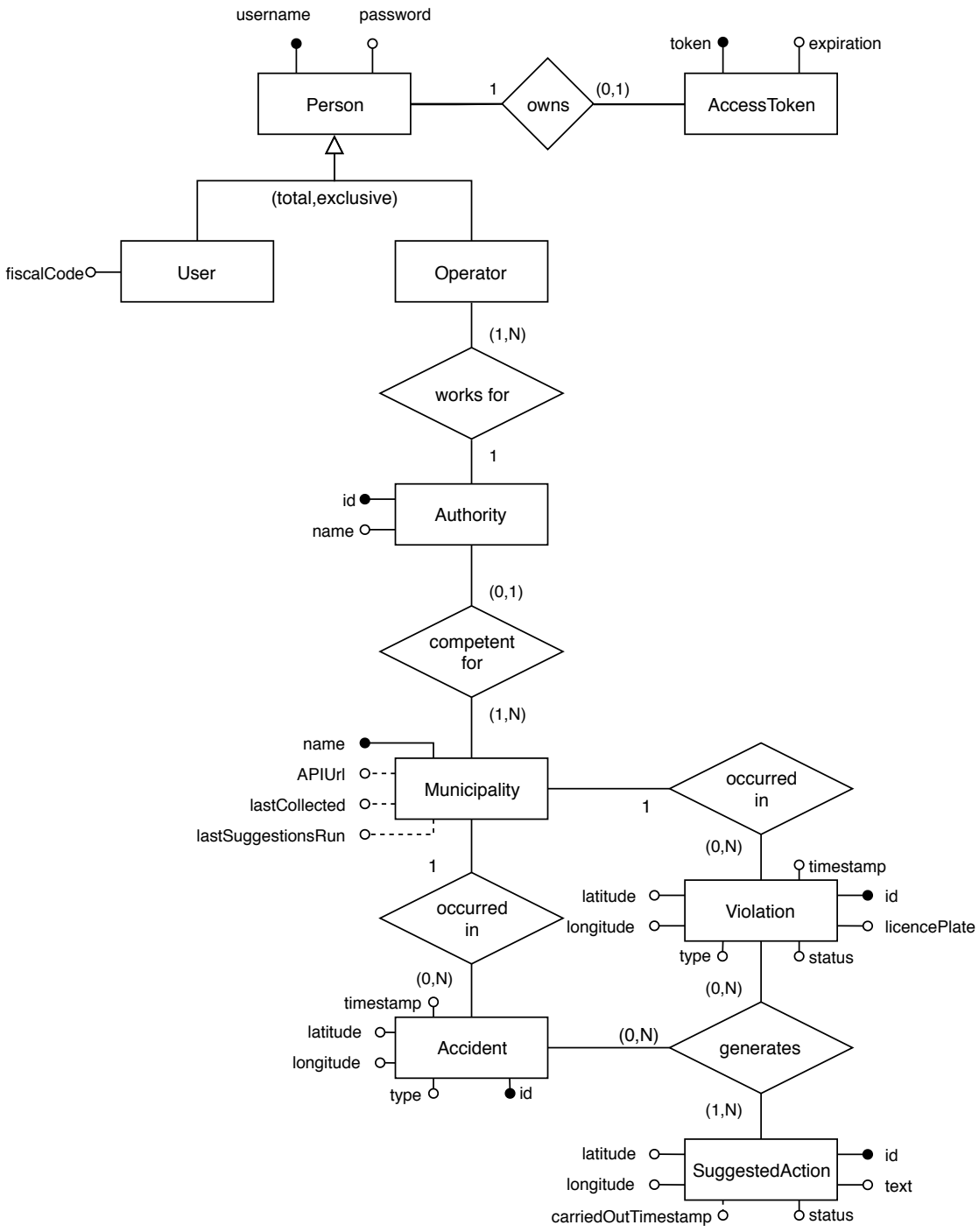

Figure 2.3: Entity-relationship diagram

2.3 Deployment view

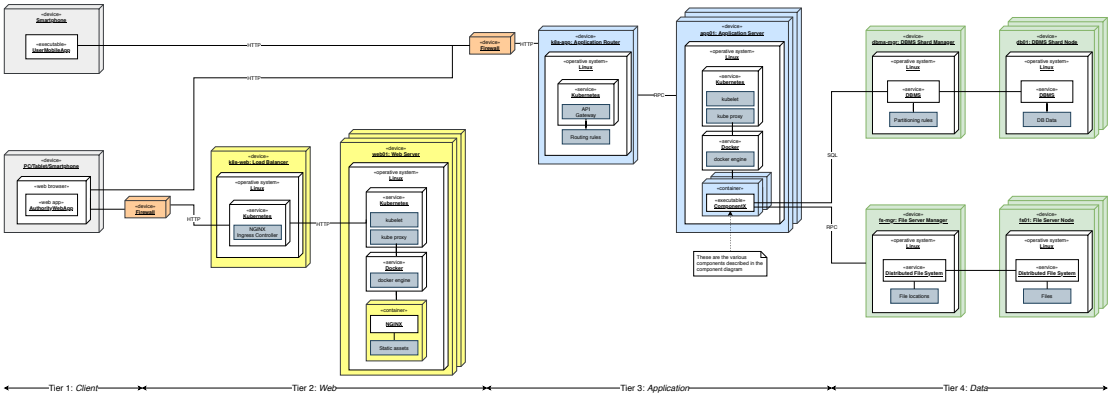

Figure 2.4: Deployment diagram

The diagram in figure 2.4 shows the deployment plan for the various components of the system. External components (such and the DMV and municipality systems) are not shown, since they are already built and deployed, and they can just be accessed from the internet. The diagram shows the *physical devices* in which the components will run, the *communication channels* between the components and their *software environment*. The system is split into *four tiers*, the last three heavily relying on replication to achieve scalability and fault tolerance. It is worth pointing out that tier 1 and 4 correspond to the *presentation* and *data* layers identified in section 2.1, while the *application* layer has been separated in tiers 2 and 3.

Client tier It is composed by the end-user interfaces: the *mobile application* for the citizen and the *web application* for the authority. The first one must be developed for both Android and iOS, as required in the RASD, and communicates directly to the application tier by using HTTP (specifically a REST API).

The website, on the other hand, only requires a web browser to be used, and therefore it can run on PCs, as well as tablets and smartphones, although the focus should be mainly on PCs. It interacts with the web tier through HTTP (to request static assets) and to the application tier also through HTTP, using the REST API.

Web tier This tier contains the *WebServer* component, represented by NGINX, which has been chosen for its reliability and performance for static serving. NGINX actually runs inside a Docker container, which also contains the HTML, CSS and JS files to be served. The containers are orchestrated by *Kubernetes*, which takes care of auto-scaling, load balancing, image and configuration deployment thanks to the *NGINX Ingress Controller*. There is a single node (*k8s-web*) that acts as a Kubernetes manager. It is worth pointing out that, since NGINX already uses internal workers, there are no advantages on deploying more than one container to each replicated node.

Application tier This tier is responsible for the components that implement the business logic of the system, see section 2.2 for more details on them. As in tier 2, we rely on Kubernetes and Docker for replication. Here the Kubernetes manager (*k8s-app*) implements the *Router* component through an API gateway module. The role of this module is to accept REST requests and route them to the correct container as remote procedure calls (RPCs). Each container runs a specific application component and they communicate between themselves through RPC. Differently from tier 2, here it makes sense to run multiple containers on the same node.

Data tier This tier contains the components responsible for persistently storing data: the database and the file server. The database component is implemented as a Relational Database Management System (RDBMS), to which application components send SQL queries. Since the data stored in the DB will most likely exceed the capacity of a single node over time, it is horizontally partitioned over multiple nodes using *database sharding* (see section 2.6.6): there is a single node that acts as the transaction manager, while the actual data resides on the *shard nodes*.

The consideration made for the DB also holds for the file server: at one point the collected images will exceed the capacity of a single node. In the same way the files are partitioned over multiple nodes and there is a single manager that keeps the mapping between the file names and their actual locations. The communication with the application tier relies on RPC.

In this way the application tier only sees one entrypoint for the DBMS and one for the file server, which hide the internal replication.

For more details on the design choices, in particular how data is partitioned between nodes (and also under which assumptions), see section 2.6.

2.4 Runtime View

In this section all the basic interactions will be described. All the use cases from the RASD document are visualized in a deeper way. The Application Server is exploded in the various components, just like in Figure 2.2, and also the interaction between the components is described.

Regarding the methods exposed by the components, please refer to section 2.5 where the interfaces of the main components are described. Note however that also calls to internal methods of the components are shown (always as self-arcs), so they are not included in the interfaces. Also note that the methods called by human actors on the clients are just *placeholders* intended to model the interaction. The interaction with the DBMS is modeled with the SQL verbs `select`, `insert`, `update` and `delete`.

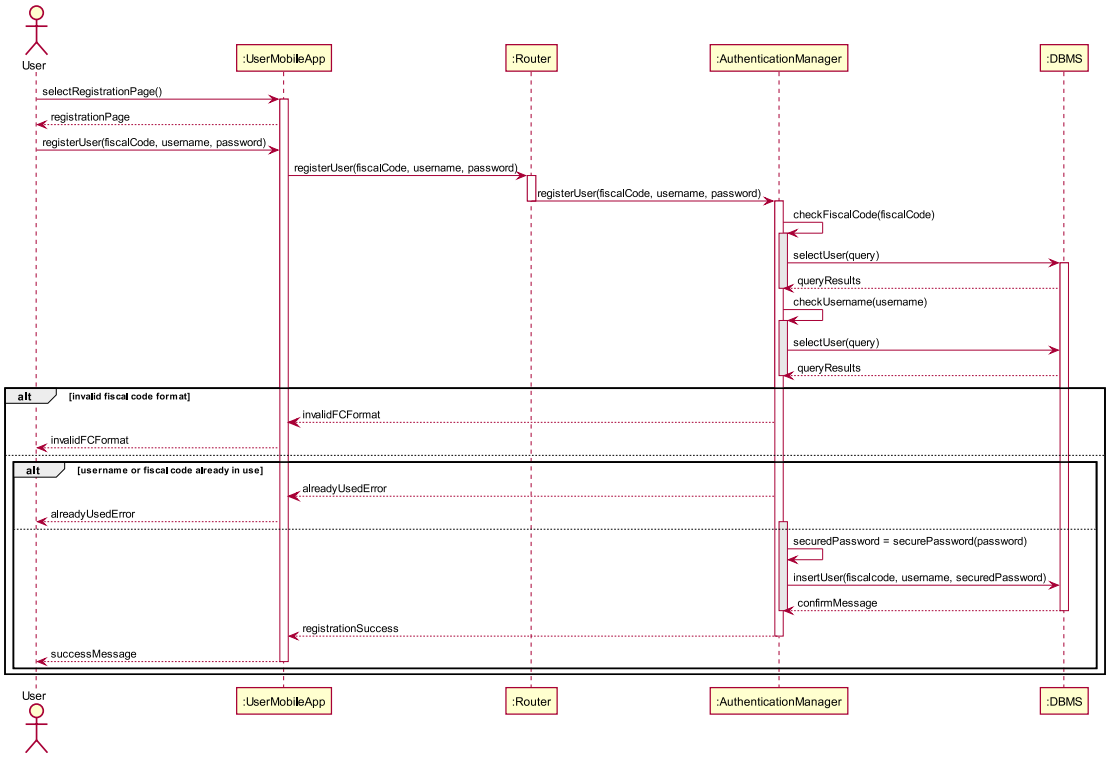

Figure 2.5: Registration process for a new User

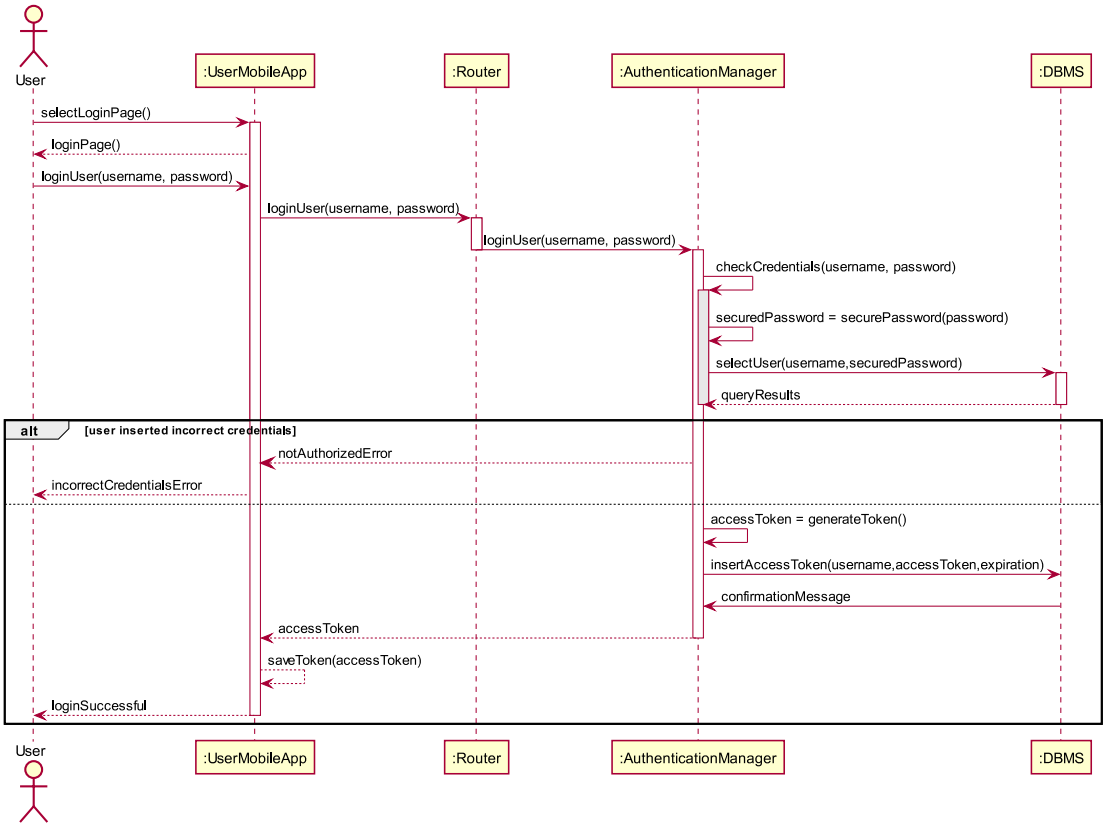

Figure 2.6: Login process for a User

The login process for the operator is analogous to the one performed by the user, shown in figure 2.6. The only difference is that the operator interacts with the *AuthorityWebApp* instead of the *UserMobileApp*.

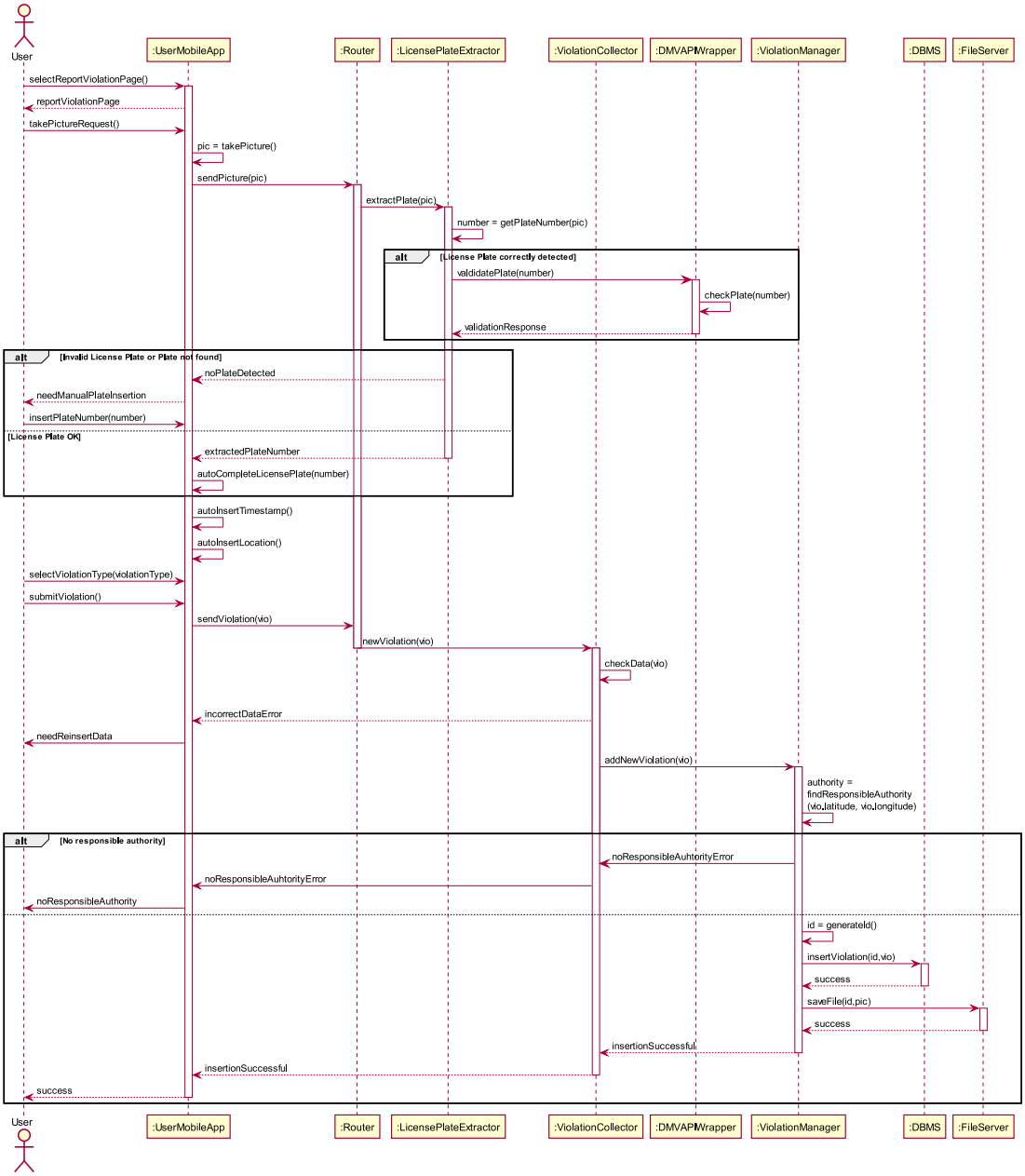

Figure 2.7: Violation Report process for a User

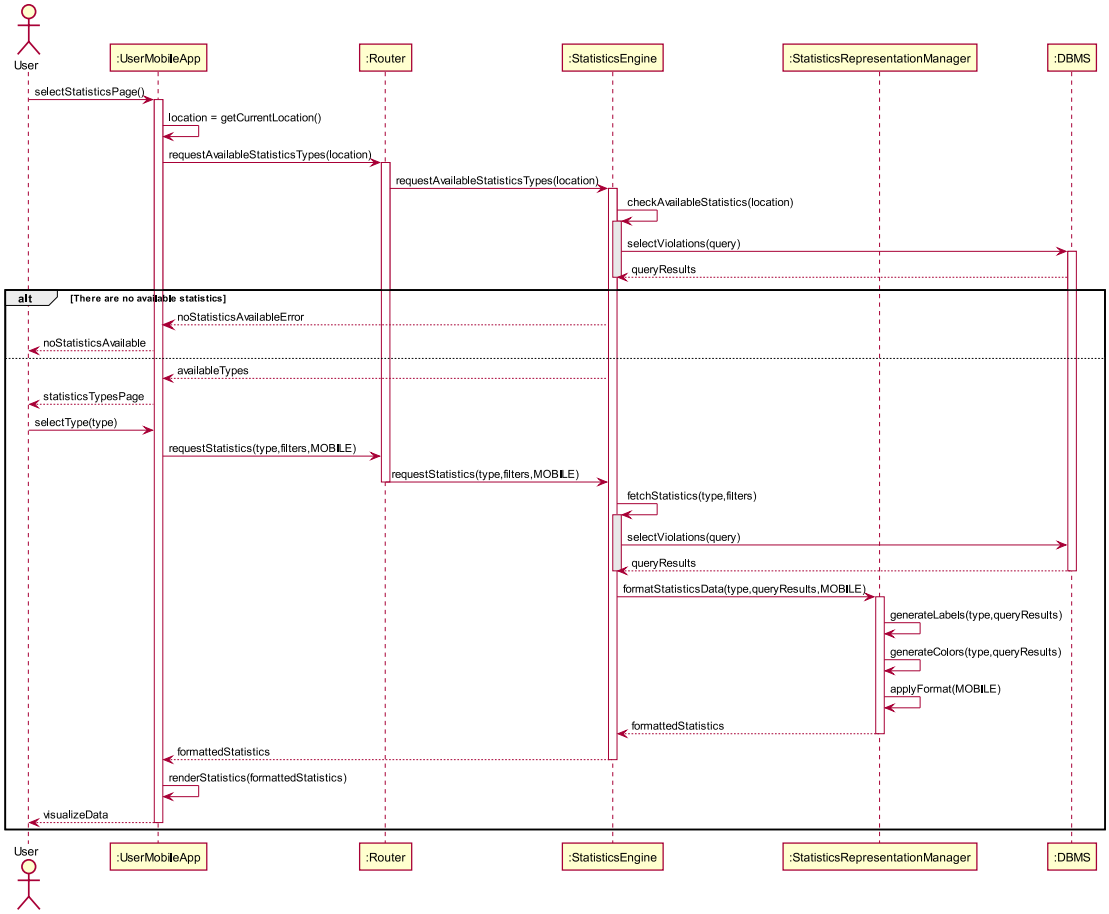

Figure 2.8: Visualize Statistics process for a User

Note that Figure 2.8 shows the interaction between Safe Streets and the User but the same action can be done by the Authority Operator as well. In the latter case, there are no substantial differences from the former case, so another dedicated Sequence Diagram is not necessary.

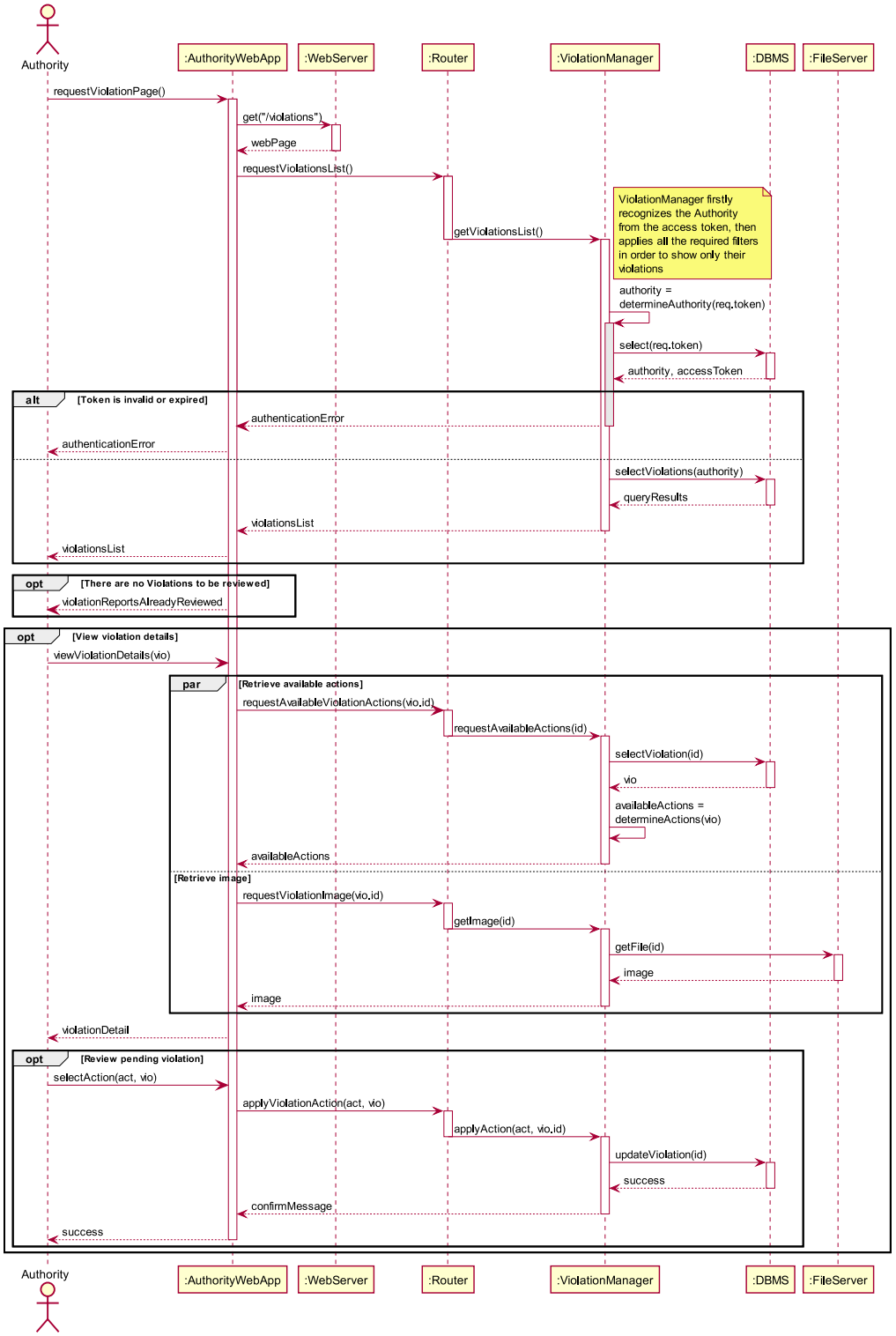

Figure 2.9: Violation management process for the Authority

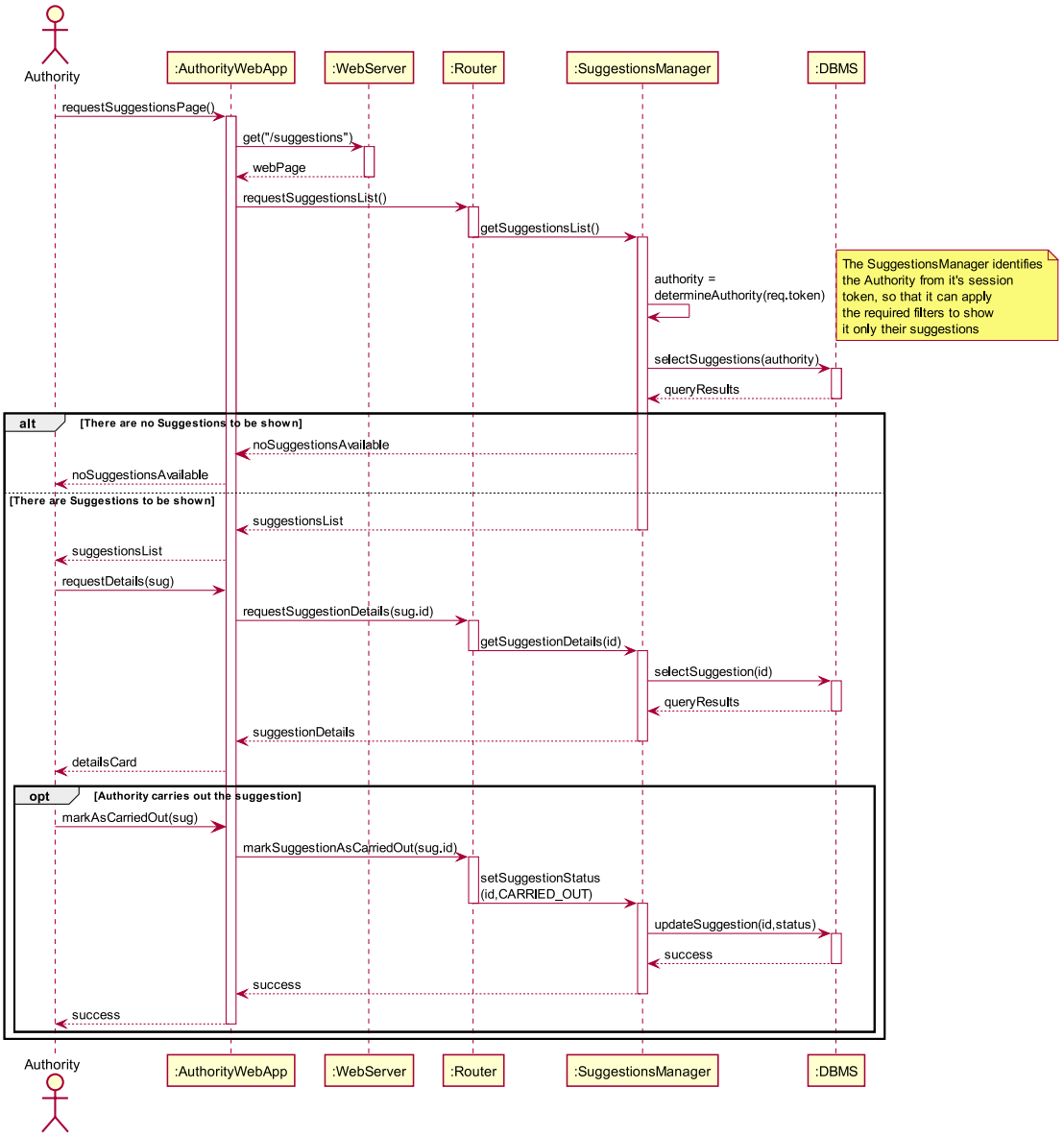

Figure 2.10: SmartSuggestions management for the Authority

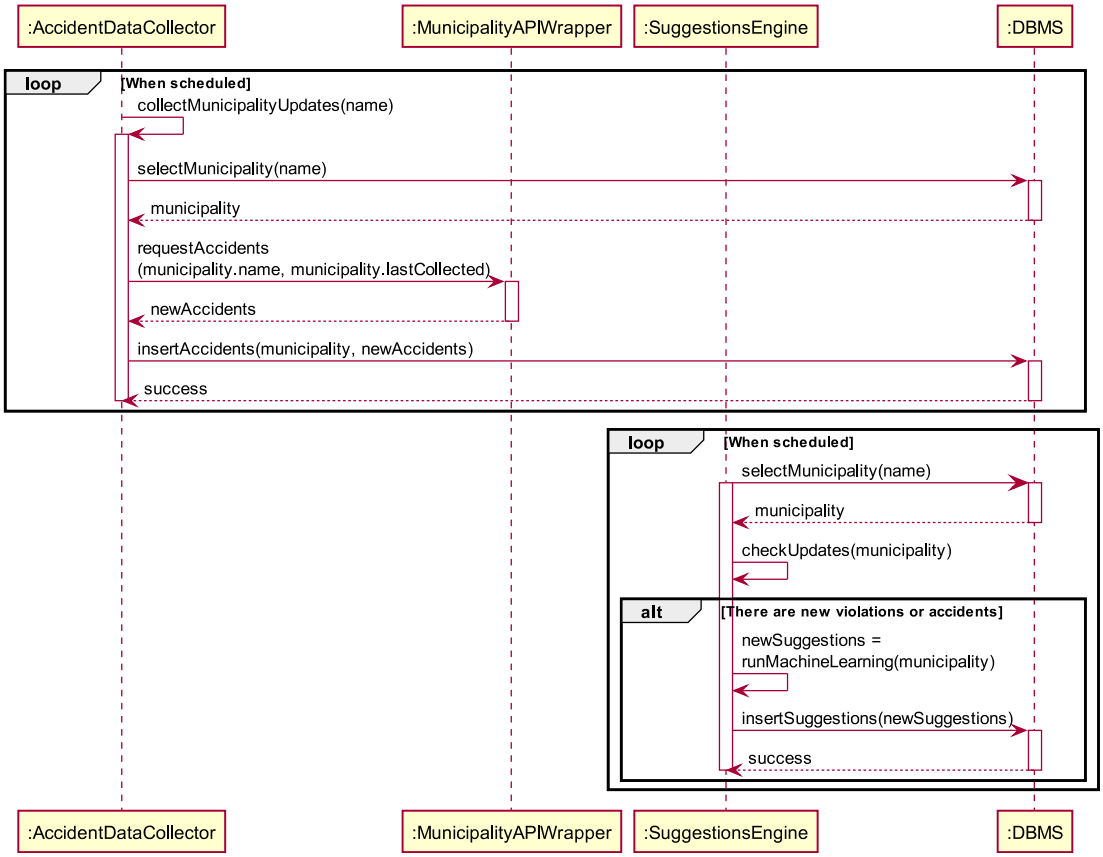

Figure 2.11: Periodic routines to fetch and process accident data

Lastly, in Figure 2.11, we can see how accident data is fetched from the Municipality API, and then how it is processed from the SuggestionsEngine component. It is important to underline that no notification will be pushed to the Authority Web App when new Suggestions are available, they will be shown to the Operator once he selects the *SmartSuggestions* page.

2.5 Component Interfaces

In Figure 2.12 the interfaces of the components described in section 2.2 are shown. The interfaces for the DBMS the WebServer and the FileServer are not represented, since they are commercially available components with standard interfaces. A detailed explanation of the use of the various methods provided by the interfaces is not necessary since those exact names are used in the Sequence Diagrams, furthermore the names are mostly self-explanatory.

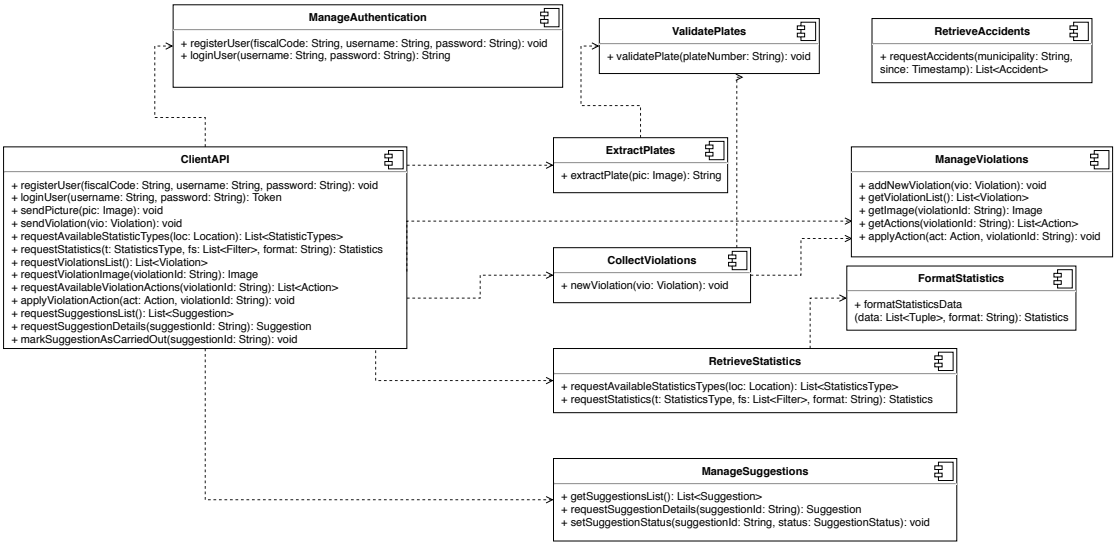

Figure 2.12: Interfaces that are used in the Application Server

2.6 Selected architectural styles and patterns

2.6.1 Four-tier client/server architecture

As shown in section 2.3, we decided to use a client/server architecture, where the components have been divided into four *tiers*. The communication is always initiated by the client, while the server only waits for requests, processes them and finally responds. There are no cases in which the server actively contacts the client (e.g. push notifications). The client/server communication is fully based on HTTP. Furthermore, in order to guarantee security, it is mandatory to use HTTP over TLS (HTTPS) for such communication.

On the other hand the division in tiers enables to *uncouple* the components from one another, which facilitates parallel development, maintenance and also hides the internal complexity of a tier (e.g. partitioning and replication) to its users.

2.6.2 RESTful communication and stateless components

As already mentioned earlier, the communication between the client and the application tier is based on the REST architectural style. Here both the mobile and the web clients can perform operations (register, login, mark violations and suggestions) and request data (violations, suggestions, statistics) in a request-response fashion. In fact all the above mentioned functionalities are designed in such a way that they can each be completed with a single request and response. Furthermore each request also carries an access token (except registration and login, of course) that is used to check the access rights of the client who sent the request. In this way the application components can be *completely stateless* and only rely on the data stored in the DB and file server. This means that we can have multiple instances of the same component and the *Router*, implemented with a Kubernetes API Gateway, can forward an incoming request to any of those instances.

Another advantage of using the RESTful architecture is that data can be represented in a simple and platform-independent way using JSON, allowing for a total uncoupling between the application server and the clients. In fact, even if the citizen client is a native iOS or Android application, while the authority client is a website, they can still share the same API, as the conversion to the concrete data structures happens inside the client itself.

2.6.3 Elastic components and load balancers

The choice of a RESTful architecture with stateless components means that a request can be sent to any instance of a component, without affecting the correctness of the response. These considerations of course hold also for the static web server, which is stateless by definition.

We have exploited this advantage by running each component in a separate *Docker* container and all containers are managed by Kubernetes. Kubernetes always keeps at least one instance of each component running, but when the workload increases it

automatically spawns new containers to replicate stressed components and automatically assigns the requests to the various instances in order to share the workload between them. When the workload decreases again, the replicated containers are deleted accordingly.

The goals of this decision are to achieve *scalability* and also to make an efficient use of the computational resources.

2.6.4 Facade pattern

The facade pattern consist in providing a unique and simplified interface to interact with a complex system. This pattern has been used various times in our design:

- in the *ApplicationServer* subsystem, which provides a unique REST API managed by the *Router*. This hides the fact that the requests are actually fulfilled by other components, which may also be replicated other multiple nodes.
- in the *data tier*, where both the Database and the File Server offer a single entry-point to access data that is actually partitioned over multiple nodes.

2.6.5 Adapter pattern

This pattern consists in allowing an existing interface to be used as another interface. We used this pattern on the *DMVAPIWrapper* and *MunicipalityAPIWrapper*, both providing access to external APIs. The external APIs are usually not consistent with the internal ones, in terms of communication model (RPC in this case), data format (the data structures may be different) and they may also change over time. So the aim of the above mentioned components is to provide interfaces that are consistent with the rest of the internal ones. If an external API changes for some reason we only need to modify its wrapper, rather than all the components that use it.

2.6.6 Database sharding

Database sharding is a database architecture pattern that consists in *horizontal partitioning*: separating the table rows into different sub-tables (*logical shards*) and assigning each of them to a different node (*physical shard*) [2].

This choice has been dictated by the fact that we expect that, at some point, a single DB node wouldn't be able to store all the data, so we needed to apply a *horizontal scaling* technique that allows to efficiently partition the data and share the workload between different nodes.

In order to actually balance the workload between the physical shards, a proper partitioning criterion should be put in place. We chose to create *a logical shard for each authority*, meaning that all the violations and suggestions regarding the municipalities under a specific authority belong to the same shard. The rationales that led to this choice are:

- a single authority has access to all and only the data about the municipalities under its jurisdiction

- the suggestions are computed on a per-municipality basis, so they only access data about a specific municipality
- the rate of requests for a single municipality (or a small subset) is relatively small and can be supported by a single node
- the rate of requests is (approximately) evenly spread between the municipalities

With this technique, the most common operations (retrieve suggestions and violations, insert or update a single violation or suggestion) only require access to a single partition, which makes them efficient.

3 User interface design

As already stated earlier, we have different interfaces for the mobile application and the web application. The main idea is to initially establish a common design language (colors, fonts, look of reusable components) which can then be applied to both interfaces. In this regard we have provided some early mockups of the main pages of both interfaces in the RASD (subsection 3.1.1 – User interfaces).

Here we give a better detail of the flow of the application, also including some pages of minor importance that were not shown in the RASD. In the following diagrams the boxes represent the different pages and the arrows the transitions from a page to another. The arrow labels indicate the conditions(s) which trigger a particular transition.

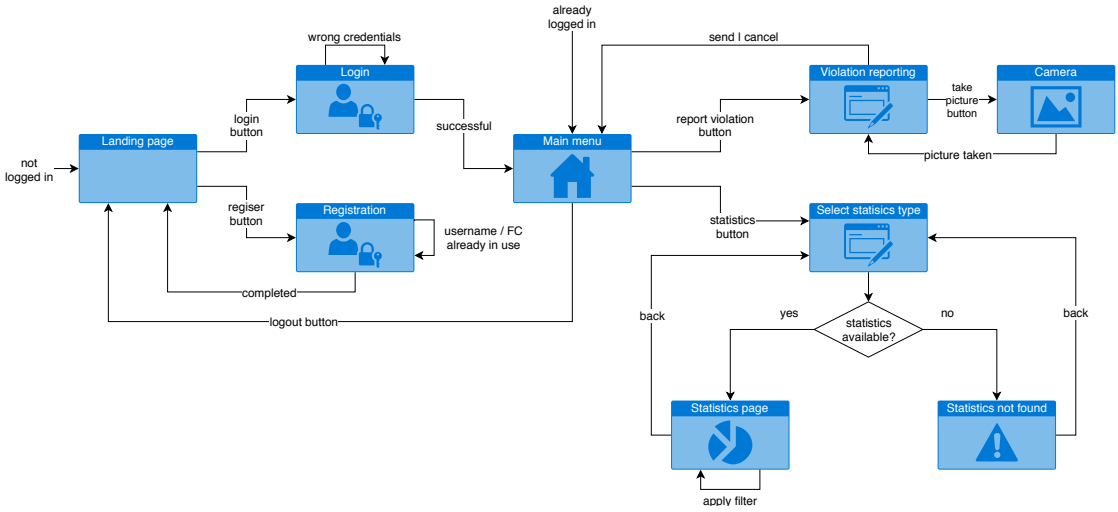

Figure 3.1: UX diagram for the mobile app

Figure 3.1 shows the UX diagram for the mobile application. When the user opens the application on his smartphone, the first check is whether he is already logged in or not. If he's not, the *landing page* will be shown, which enables him to login or register, otherwise the user is directly taken to the *main menu*. Once in the *main menu* the user can decide whether to report a violation, view statistics or log out (this was not shown in the previous mockups). For conciseness the links to come back from this page are not shown in the diagram.

In case the user decides to report a violation, he will be taken to the *violation reporting page*, which is essentially a form that also requires to take a photo. The process of reporting a violation has been already described in the use cases.

In case the user decides to view statistics, he will be first presented with a set of possible types of statistics (e.g. by month, by type). He will select one of these types and the system will try to compute those statistics. In case they are not available (e.g. there is no sufficient data for violations in the surrounding area), then the user is brought to an *error page* from which he can go back and try with another type. If everything goes right, instead, the user is brought to the *statistics page*; note that here the rendering of the statistics (histogram, map, list) depends on the selected type. From here the user can apply additional filters (restrict to a specific area, time period or violation type).

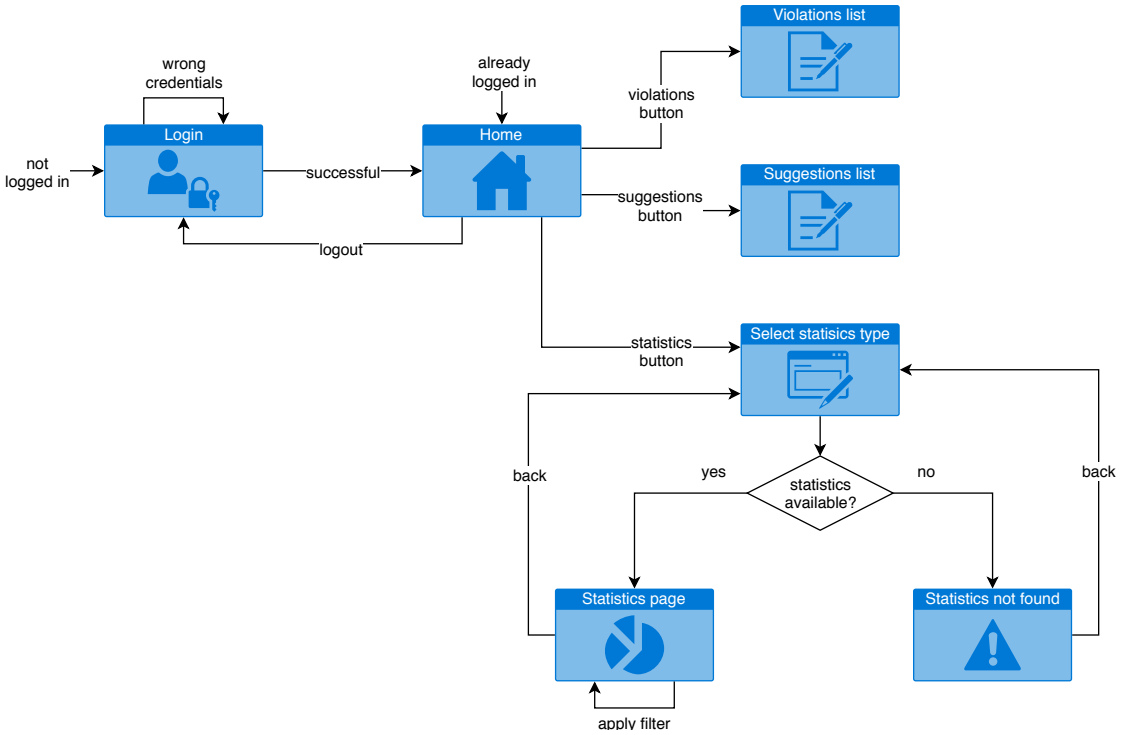

Figure 3.2: UX diagram for the web app

Figure 3.2 shows the UX diagram for the web application. When the operator navigates to the website he will be presented with a *login page* if he's not already logged in, otherwise he will be taken directly to the *main menu*. From the *main menu* the operator can access the main functionalities of the system, and he can also decide to *log out*.

In case he clicks on the violations button, he will be taken to the *violations list* page, which shows all the most recently reported violations that belong to that specific authority; violations that are still pending for approval are shown at the top. From here the operator can click on the *info button* of a violation: this will expand the list item to a card that shows the details of a the report. If the violation is still unrevised, this card will also contain an two action buttons to accept or discard the violation, respectively. The list of violations can be implemented as an *infinite scrolling list*: initially only the most recent N violations are retrieved, enough to fill the page; then as the operator

scrolls down the next N violations are retrieved from the server, and so on.

The suggestions button is only shown if the *SmartSuggestions* functionality is active for the current municipality. If the operator clicks on it he's taken to the *suggestions list* page. This is very similar to the previous one, because it involves viewing a list of suggestions, some of which are new and can be expanded to be marked as *carried out*. Also here we can apply the technique of the *infinite scrolling list*.

If the operator clicks on the statistics button, the process is pretty much the same of the mobile application, detailed before. The only things to keep in mind are that the authority also has the possibility to see the licence plate numbers associated to violations. Here, since this page will primarily viewed on the large screen of a PC, we can show more data in a single page than in the mobile application (as suggested by the mockups).

As an additional consideration, since the website can be viewed from devices (smartphone, tablets, laptops, desktop PCs), its interface can be made *responsive* to adapt to the various screen sizes. At least this should be done for the *violations list* page, which can be useful for field operators.

4 Requirements traceability

The system design described in this document must meet the requirements – both functional and non-functional – defined in the RASD. We dedicate this chapter to mapping each requirement to the various elements we designed to fulfill it.

4.1 Functional requirements

The functional requirements are mainly related to *components*, so for each requirement we list the related components and a brief explanation when needed, for a detailed description of each component's purpose see section 2.2. Please note that for conciseness we only list the components that are *directly related* to satisfy the requirement, other components, such as the DB, that are indirectly used might be omitted.

R1 The system must allow citizens to register by providing fiscal code, username and password

- *UserMobileApp* – interface provided to the user
- *AuthenticationManager* – creates the accounts, enables login

R2 The system must not allow two people (user or operator) to have the same username. Also it must not allow two users (i.e. citizens) to have the same fiscal code

- *AuthenticationManager* – checks unique username and password before creating an account

R3 The users must be able to upload photos from their mobile devices

- *UserMobileApp* – contains page to take photo
- *LicencePlateExtractor* – accepts photos uploaded by users to extract the licence plate
- *ViolationCollector* – accepts photos attached to incoming violation reports

R4 The system must be able to detect the date, time and location of the users' devices

- *UserMobileApp*

R5 The system must allow the user to select the type of violation he wants to report from a finite list of types

- *UserMobileApp* – report violation page
- also enforced by the DBMS schema

R6 The system must allow the user to send violation reports

- *UserMobileApp* – report violation page
- *ViolationCollector* – accepts incoming violation reports

R7 The system must allow the user to edit each field of the report before sending it

- *UserMobileApp* – report violation page

R8 The system must be able to determine whether a plate number is valid (i.e. registered to the DMV) or not

- *DMVAPIWrapper*

R9 The system must be able to detect the biggest licence plate in a photo and read its number

- *LicencePlateExtractor*

R10 The system must be able to persistently store the violation reports, including their attached photos

- *DBMS* – persistently stores the data about the reports, except for the image files
- *FileServer* – persistently stores the photos attached to the violation reports.

R11 Violations with malformed data (invalid date/time, inexistent licence plate number, missing fields) or with no responsible authority must not be stored by the system

- *ViolationCollector* – checks the validity of the fields in the incoming reports
- *ViolationManager* – assigns each incoming report to an authority, if this is not possible the report is rejected, otherwise it is sent to the DBMS
- *DBMS* – enforces integrity constraints on the fields of the stored violation reports

R12 When storing a violation report, the system mustn't save the user who uploaded it. In particular it must not store any identifier (e.g. fiscal code, username) that can provide such association

- *DBMS* – in the schema a violation is not related to a user (also see 2.3)

R13 The system must allow the operators of authorities to login with username and password

- *AuthorityWebApp* – interface for the operator, also saves the access token returned by the server
- *AuthenticationManager* – checks login credentials and creates and returns an access token
- *DBMS* – stores the login credentials

R14 The system must be able to determine the municipality in which a violation occurred based on the location saved in the report. It must be able to assign the violation to the authorithy responsible for that municipality, if any

- *UserMobileApp* – captures the location of the user and sends it as part of the report
- *ViolationManager* – assigns each incoming report to an authority, if this is not possible the report is rejected
- *DBMS* – stores the municipalities for which an authority is responsible

R15 An operator must be able to access only the violation reports assigned to its authorithy. When a new report has been stored, he must be able to mark it as accepted or discarded

- *AuthorityWebApp* – violations list page
- *ViolationManager*

R16 Discarded reports are eventually deleted from the system

- *AuthorityWebApp* – violations list page
- *ViolationManager*

R17 The system must be able to filter and aggregate violation reports by location, date, time, type and licence plate number

- *DBMS* – SQL provides the `where` and `group by` clauses in its query language

R18 The system must be able to compute statistics based on the number of violations

- *StatisticsEngine*

R19 The system must allow an authority operator to view statistics, including filtering for the licence plate of the vehicles

- *AuthorityWebApp* – statistics page
- *StatisticsEngine* – computes statistics, includes the plate number based on the access token in the request
- *StatisticsRepresentationManager* – formats the raw statistics data

R20 The system must allow an user to view statistics which do not include the licence plate of the vehicles involved

- *UserMobileApp* – statistics page
- *StatisticsEngine* – computes statistics, includes the plate number based on the access token in the request
- *StatisticsRepresentationManager* – formats the raw statistics data

R21 The system must be able to retrieve data about accidents from municipalities that provide it (through an API)

- *MunicipalityAPIWrapper*
- *AccidentDataCollector*

R22 The system must be able to determine if there is a causal relation between accidents and violations based on their location, types and timestamps

- *SuggestionsEngine*

R23 The system must be able to determine possible actions that can be taken by the authorithy to reduce the detected accidents, also using the detected causal relations

- *SuggestionsEngine*

R24 An authorithy operator must be able to view the suggested actions for the zone in which the authorithy operates. He must also be able to mark an action as *carried out*, meaning that it has actually been executed

- *AuthorityWebApp* – suggestions list page
- *SuggestionsManager* – get list of suggestions, mark them

R25 The system must be able to detect if a *carried out action* has been useful or not (i.e. the number of accidents and/or violations that it was supposed to prevent has dropped or not) and possibly provide better suggestions based on this knowledge

- *SuggestionsEngine*

4.2 Non-functional requirements

The non-functional requirements, which comprise the *performance requirements*, *design constraints* and *software system attributes* (sections 3.3-3.5 of the RASD), are particularly related to the *deployment strategy* (see section 2.3) and the *architectural patterns* (see section 2.6) that have been applied. Here we analyze in how the crucial aspects have been taken care of.

Scalability A lot of architectural decisions have been made to ensure the scalability of the system:

- tiered architecture
- containerization of components with Docker and Kubernetes
- elastic replication and load balancing
- stateless components
- database sharding

These are also aimed at improving the *reliability*, *availability* and reduce *response times*.

Performance Aside from the response times for the clients, it has been required for the suggestions to be generated periodically. As stated in section 2.2 the *SuggestionsEngine* will take care of scheduling the jobs to generate new suggestions, as new data becomes available.

Security The measures related to security include:

- firewalls to protect the internal components (section 2.3)
- use of HTTPS for communication over the internet (section 2.3)
- hashed passwords in the DB (sequence diagram 2.5)
- session tokens to use the APIs (section 2.2)

Mantainability The use of the RESTful architecture (section 2.6) and the loose coupling between the components are all aimed at providing a better maintainability than a monolithic approach. Furthermore the use of management tools such as Kubernetes provide a way of managing and distributing images of the various components, which facilitates the versioning and deployment processes.

Portability The *mobile application* will be developed for both Android and iOS platforms, keeping a consistent design language and user experience. On the server side, the use of *containerization* reduces the dependency of the components from the hardware, since they run on a virtual environment. This makes it easy to port the containers from one machine to another, provided that they can run Docker and they have the same architecture (we can safely assume to be dealing with x86/64 servers).

5 Implementation, Integration, Test plan

In this chapter three main topics will be discussed: implementation order of the various components, integration testing plan, general testing plan.

5.1 Overview

The *SafeStreets* system is divided in these main parts:

1. User Mobile App
2. Authority Web App
3. Web Server
4. Application Server
5. DBMS
6. File Server
7. External API: DMV and Municipality

As for the External API services, File Server and Web Server, these components deserve special treatments since they won't be developed internally. For example the DBMS is easy to find on the market and since it's a commonly used component it won't be that expensive and it will be already tested and ready to use (after a brief configuration). The User Mobile App and Authority Web App are merely frontend components, thus they can be developed independently from the backend, which requires a more careful approach.

A final note is dedicated to the front-end components: UserMobileApp and AuthorityWebApp. It is strongly recommended that these components are developed using cutting-edge frameworks like *Flutter* or *React Native*. These frameworks natively provide commonly used components like UI, data management, and so on. These components are of course high quality written and already tested. Last but not least these frameworks allow the developer to write the code once and then run it on almost every mobile OS (or browser, for what concerns the web app).

5.2 Functionalities and features

Here is a table showing the various functionalities of the system and their respective importance for the customer and difficulty of implementation. Note that the *Importance for the customer* isn't in any way related to the importance for the system by a structural point of view. For example the Login feature is very important for *SafeStreets* since we truly need to register our users, but from the point of view of the customer, it's something that won't really be noticed. This table will be useful when the development will be scheduled in details. The functionalities that are harder to implement will require more resources and time than the trivial ones, furthermore we can note that ML has a low importance for the customer while the implementation difficulty is high, so a possible option is to launch the product without that functionality and then release it later on.

Feature	Importance for the customer	Difficulty of implementation
Registration and Login	Low	Low
Violation Report	High	Low
Automatic metadata insertion	Medium	Low
Automatic License Plate detection	Medium	High
Visualize statistics	High	High
Manage Violation Reports	High	Medium
Collection of accidents data	Medium	Medium
Visualize and manage suggestions	High	Low
ML algorithm to generate suggestions	Low	High

The development will follow a bottom-up approach, in order to improve components reliability and progressively integrate the single components into the system. Here is a list that describes which components will take care of each feature.

Registration and Login the main component for this feature is *AuthenticationManager*, also the *DBMS* will be needed in order to save the user's data.

Violation Report for this feature *ViolationManager* and *ViolationCollector* are needed, as well as *DBMS*.

Automatic metadata insertion this is a minor feature and it is taken care by the *User-MobileApp*.

Automatic License Plate detection this feature will be handled by *LicensePlateExtractor* and will also involve the *DMVAPIWrapper*.

Visualize Statistics both users and authorities will have access to this feature, it needs *StatisticsEngine* and *StatisticsRepresentationManager*, the rendering of the data is performed by *UserMobileApp*, or the *AuthorityWebApp*. In any case, these components have a little work to do. In order to access stored data, also the *DBMS* will be necessary.

Manage Violation Reports in order to make this feature work, the component which is needed is *ViolationManager*, and of course the *DBMS*.

Collection of accidents data for this feature the system will need the *AccidentDataCollector* implemented. The data will be stored into the database, so also *DBMS* is required.

Visualize and manage suggestions this is the core functionality of *SmartSuggestions* subsystem. The component in charge of this feature is *SuggestionsManager*, which fetches data from the *DBMS*.

Machine Learning algorithm to generate suggestions this is probably the most difficult feature to implement. So we will need to take specific care, the component is *SuggestionsEngine*, as well as the *DBMS* to retrieve and store data.

5.3 Implementation plan

This section shows how the various components will be developed and the timing constraints they will need.

From Figure 2.2, it can be clearly seen that the components don't have specific dependencies, so the development process can be done in parallel. This is especially true if we consider that all the components use RPC for internal communication. This means that once one component has been developed, it can be immediately unit tested. Also the methods that require communication with other components can be tested immediately, this can be done by creating some *ad-hoc stubs* with the required interfaces.

Specifically, the order in which the various components are implemented is not fixed, except for *Router*, *SuggestionsEngine* and *DBMS/FileServer (only configuration)*. The main reason behind these choices is that, for example, the Router is responsible only of taking care of the incoming requests and forward them to the correct component, so it's role isn't critical in the integration of the components, plus it's developing process is quite trivial. As for the storage systems (DBMS and File Server), it is a common practice to test the components using volatile memory, so a working database is not needed from the very beginning. In any case it might be useful to configure these components as soon as possible, especially the DBMS, since it is used by most of the other components. It is important to configure early the DBMS also because the various vendors provide slightly different interfaces so setting the DBMS up as early as possible can only bring benefits. As for SuggestionsEngine, its complexity compels us to start its implementation as soon as possible. It's a machine learning based component so the sooner it is ready, the better it is, because it can start its training and be truly ready for deployment.

Finally despite the fact that the order, as previously stated, is not important, some components should be implemented in pairs. It's the case of *Violation Collector* and *ViolationManager*, *MunicipalityAPIWrapper* and *AccidentDataCollector*, *StatisticsEngine* and *StatisticsRepresentationManager*. This can ben necessary as these pairs of components work closely and their integration should be as easy as possible.

5.4 Testing

The testing phase of a project is one of the most critical ones, a bad testing plan can lead to a massive loss of time and money, so it is important to follow the best practices in order to save time and money.

5.4.1 Inspection

Many researches lead by Aetna Insurance Company, Bell-Northern, Hewlett-Packard, SmartBear and Nasa, pointed out that code inspection has a massive role in unveiling defects of the code as well as bugs. Inspection will be done during the development process, as it can be done before the whole component has been implemented, but must still be done onto "finished" blocks of code. By the way code inspection will be done only on back-end components, the front-end is designed as *thin* so it shouldn't do anything apart from representation. Moreover, the frameworks that are suggested for the front-end applications, natively provide analysis and testing tools.

The inspection process will be done as follows:

- **Planning:** choose participants, entry-criteria, schedule meeting
- **Overview:** mainly assignment of roles and provide a background of the project
- **Preparation**
- **Actual Inspection**
- **Rework:** the developers fix the weaknesses of the code found in the inspection
- **Follow-up:** moderator makes sure that all the weaknesses has been fixed
- **(Re-inspection):** if considered necessary, the inspection process can be repeated, especially if many parts of the code has been changed.

The actual inspection will be done in several meetings of the inspection team. As we can find in the researches mentioned above, it is functional to schedule two daily sessions of two hours each. An idea goal is to inspect approximately 200/250 lines of code per session. Further details on the inspection process can be found in [1].

5.4.2 Unit and integration testing

Once a component has been implemented and inspected, it must be unit tested to make sure it works by itself, then the integration test process can begin. The various components in the sub-groups will be undergoing integration test simultaneously. In the following pictures, the component that must be integrated are shown.

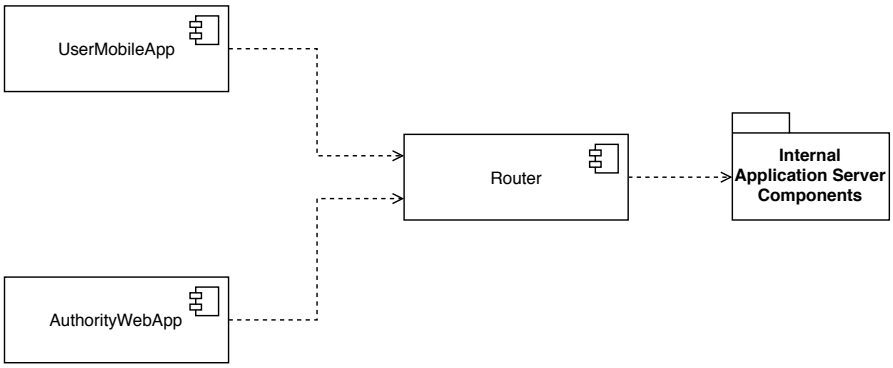

Figure 5.1: Integration testing for the Router

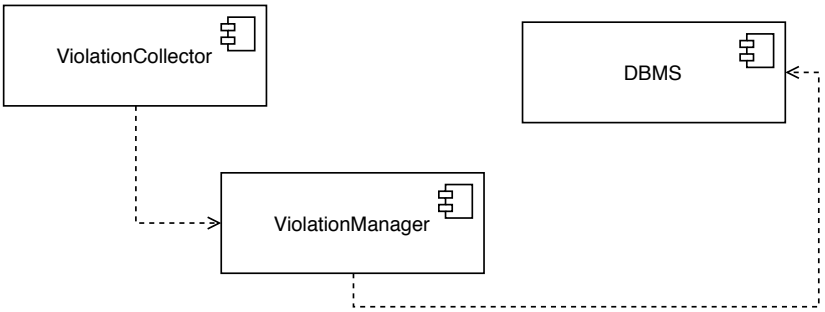

Figure 5.2: Integration testing for Violation components

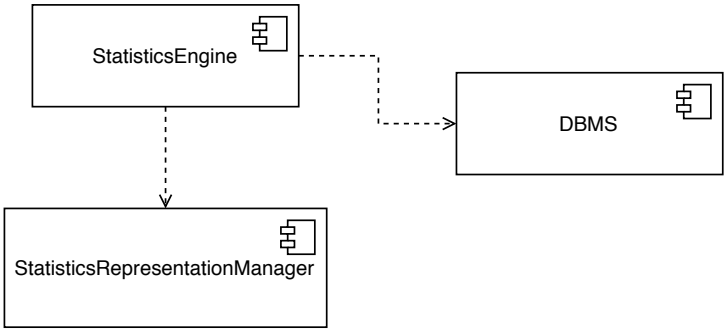

Figure 5.3: Integration testing for statistics components

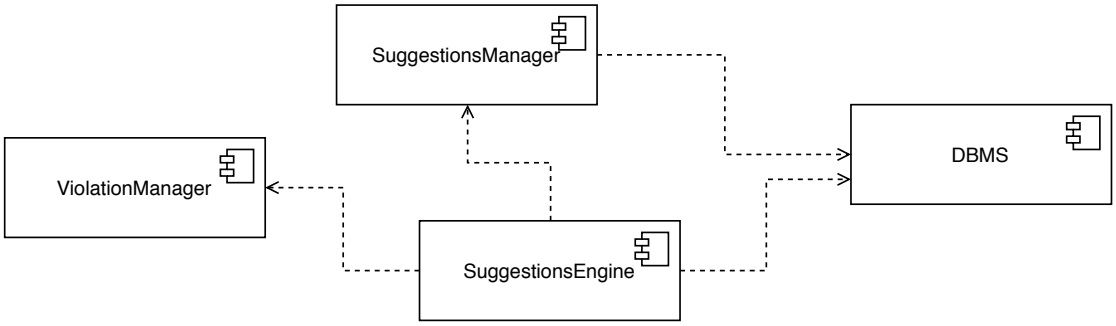

Figure 5.4: Integration testing for *SmartSuggestions* system

Once all the integration test already shown are done, the entire application server should be integrated, in other words, a complete integration test with all the components must be done at this point.

6 Effort spent

Here we show the hours that each of the authors has spent on specific tasks for creating this document.

Alessandro Fulgini

Task	Time spent
Architecture overview	2h
Components	4h
Deployment	6h 30m
Design decisions	4h
User interface design	2h
Requirements traceability	2h 30m
ER model	1h 30m
Runtime view	6h
Total	**27h 30m**

Federico Di Cesare

Task	Time spent
Chapter 1	2h
Component View	3h
Runtime View	12h
Component Interfaces	3h
Chapter 5 overview	30m
Functionalities and Features	1h
Implementation Plan	2h 30m
Testing	4h
Total	**28h**

7 References

[1] SmartBear. *Best Practices for Code Review*. 2019. URL: https://smartbear.com/learn/code-review/best-practices-for-peer-code-review/.