Politecnico di Milano
Dipartimento di Energia,
Informazione e Bioingegneria
Computer science and engineering

Software engineering 2

SafeStreets - DD

Professor:
Prof. Elisabetta Di Nitto

Candidates:
Nicolò Albergoni - 939589
Luca Loria - 944679

Academic Year 2019/2020
Milano - 09/12/2019

Contents

Introduction

1.1 Purpose

The purpose of this Design Document (DD) is to give a rather technical and implementation oriented perspective for the SafeStreets software that is going to be built. While the Requirement Analysis and Specification Document (RASD) gives a more conceptual description of the software and a view of the system that is not strictly related to the actual implementation, the DD provides a three hundred and sixty degrees guide for the developers that will implement the software in the real world. In fact the DD document contains a practical and detailed description of the architecture that will have to be built; it covers all the aspects of the system that are relevant for the developers. For example the second section is entirely dedicated to the description of the system architecture in many useful ways (i.e. component, deployment and runtime view), there is also a section for the design of user interface to be realized as well as a part related to the implementation and testing plan.

1.2 Scope

Here is a brief review of what has already been said into the RASD document about the SafeStreet scope and functionalities.
SafeStreets is a new service that aims, through the help of citizens, to improve the safety of the streets. Users will have the possibility to send reports of illegal behaviour related to street parking to authorities, just by opening the SafeStreets mobile application and take a picture of the violation. Moreover, they can also consult the history of their reports and a map that will highlight the most dangerous streets nearby. SafeStreets also offers a way for the authorities to manage the reports and perform analysis on data. A Web interface will be developed in order to address this purpose. In order to guarantee an efficient use of the ap-

plication by authorities, the system interacts with an external plate recognition service that will extract the car plate number from the report image. In this way, when authorities need to check on a report they immediately find the car plate number of the vehicle that committed the violation. SafeStreets also implements a functionality that performs the interaction with a municipality service that offers data regarding car accidents, if the particular municipality offers one. SafeStreets can cross this information with its owns, in order to get a better idea of the potentially unsafe areas and therefore suggest some possible interventions. As for the performances, the service will have to be scalable, fast and it must be able to cover a great number of users. While for the applications, they must be lightweight and must run of most of the devices available on the market.

1.3 Definitions, acronyms and abbreviations

1.3.1 Definitions

- *Most Dangerous Streets*: Streets with the highest frequency of violation reports.

1.3.2 Acronyms

- EU: European Union;
- CET: Central European Timezone;
- API: Application Programming Interface;
- HTTP: Hyper Text Transfer Protocol;
- GPS: Global Positioning System;
- GDPR: General Data Protection Regulation;
- REST: Representational State Transfer.
- URL: Uniform resource locator
- URI: Uniform resource indicator
- DB: Database

- DBMS: Database management system
- RDBMS: Relational database management system
- SQL: Structured query language
- NoSQL: Not only SQL

1.3.3 Abbreviations

- [Gn]: n-th goal;
- [Dn]: n-th domain assumption;
- [Rn]: n-th functional requirements;
- MDS: Most Dangerous Street;
- LSA: Local System Administrator;
- PT: Police Technician.

1.4 Revision History

- Version 1.0:
  - First release.

1.5 Reference Documents

- SafeStreets assignment document;
- Previous years DDs of the Software engineering 2 project;
- IEEE Std 830--1998 IEEE Recommended Practice for Software Requirements Specifications;
- Slides from the course "Software engineering 2".

1.6 Document Structure

1. In the first part a general introduction of the Design Document is given. The purpose part exposes the substantial differences with the RASD document;

2. The second section it's the core of the DD: it firstly provides an high level overview of the system followed by a description of various aspects of the architecture from different points of view such as: component, deployment and runtime view. It is also present a general explanation of the architectural patterns and styles adopted in the development process. Most of the parts of this section are enriched with UML diagrams to ensure a better understanding of the concepts;

3. This part specifies the user interface design of the mobile application and the web interface. Since the mockups of both applications were already provided in the RASD document, here some UX diagrams are proposed to better describe the navigation and functioning of the applications;

4. Part four exposes the requirements traceability matrix which maps the requirements stated in the RASD document with the corresponding design component;

5. Chapter five provides the proposals for the implementation, integration and testing plans. This plans are created by taking into account, for each functionality, the importance for the customer and the difficulty of implementation/testing;

6. The last part states the hours of work division and the tools used to create all the part of this DD document.

Architectural Design

2.1 Overview: High-level components and their interaction

SafeStreets is a distributed multilayer software application. The software architecture is a thin three-tiers architecture, based on the MVC design pattern:

- *Presentation layer (View)*: the presentation layer is conceived as an high level representation of the application, which will be responsible for the user and third party interaction and data visualization. Also, the presentation layer is the only gateway from which actors can interact with the system;
- *Application layer (Controller)*: the application layer is responsible for the core functionalities of the system: inside the controller, the main functions and the business logic is embedded and performed whenever an actor requires their execution;
- *Data layer (Model)*: the data layer is responsible for the interaction between the system and the data storage. It manages every operation performed by the system that requires data retrieving and storaging.

The logical separation (MVC) allows the system to preserve the safety of sensible data stored in the data layer, as users have only a restricted access to the presentation layer.

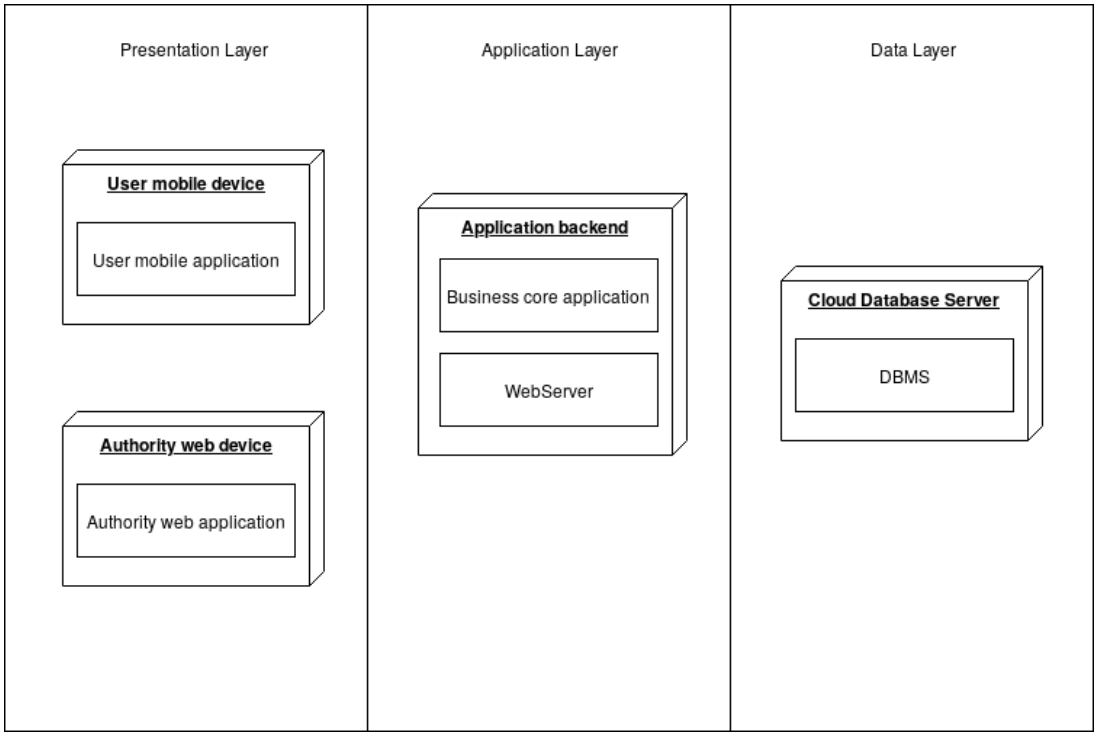

Figure 2.1: Overview diagram

Analyzing the figure 2.1 in details, it is possible to see how the layers are organized:

- **Presentation layer**: the mobile application is run on the user mobile devices and distributed through the Android and IOS official stores. Moreover, the authority web application is accessible through any web browser at a specific URL.
- **Application layer**: a cluster of dedicated servers contains some replicated istances of the backend system that run accordingly to preserve a continuous functioning of the system. The application layer servers also have a local cache necessary for the computation of the ordinary system functionalities. Also, another cluster of servers contain the web server application on which the authority web app runs;
- **Data layer**: data is stored on a cloud DBMS that guarantees a constant and consistent functioning. This allows to reduce the cost of maintainance of the system, as by paying a yearly subscription is much cheaper than running a cluster of server (cost of electricity and cooling systems) and performing periodic maintainance.

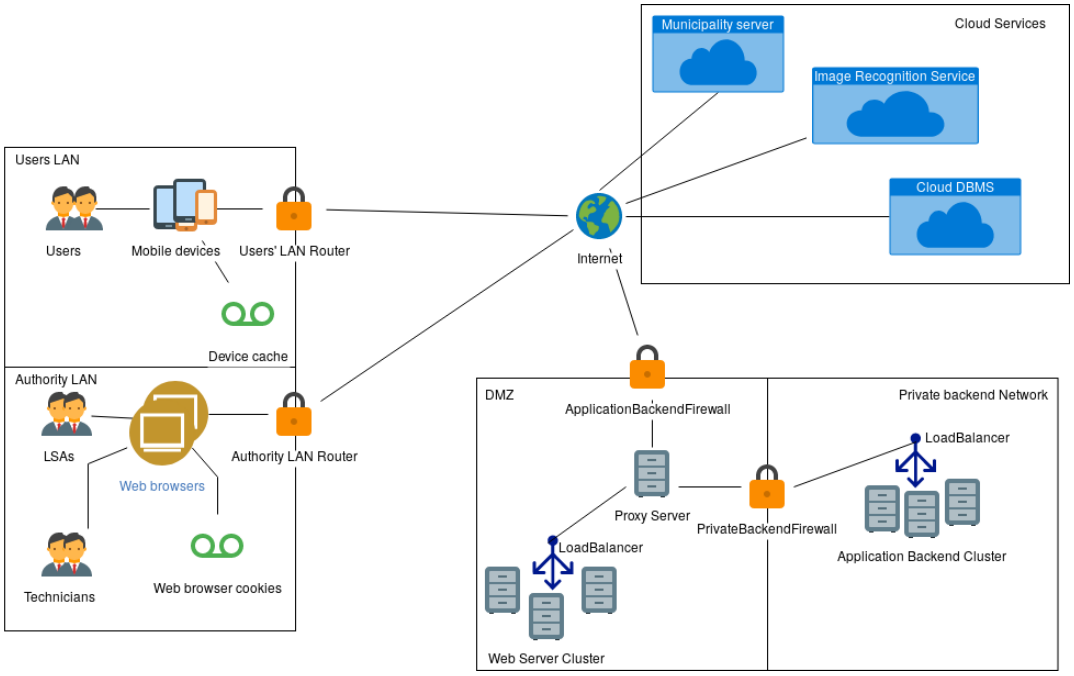

Figure 2.2: Network architecture

The network architecture of the Safestreets system is described in the figure 2.2.

- The presentation layer is phisically separated from the rest of the system by the LAN routers of users and authorities;
- The application layer is splitted in two main areas:
  - A DMZ (DeMilitarized Zone), in which a proxy server filters the requests coming from the internet and forwards them to the web server load balancer and to the private backend network;
  - A private backend network, which has another router/firewall that filters once more the requests in order to guarantee an higher level of safety. Inside the private backend network, the system workflow is automatically balanced through a **load balancer** which forwards automatically the requests to the least stressed node in order to avoid bottlenecks and system overloading.
- The data layer is represented as a cloud server (DBMS) and treated as a black box.

Also, the municipality server and the image recognition service have been pictured for completeness.

This architecture allows the system to meet the non-functional requirements of flexibility and robustness already explained in the RASD document: the backend application is distributed on several nodes distributed on the same cluster, in order to guarantee a corrected and continuous functioning. In case a node crashes, another one is able to take its place automatically and keep the system running. For this purpose, it is necessary to highlight the fact that all nodes are consistent copies of each others: data and logic functions are kept consistent through automatic techniques of node replication that preserves the ACID rules. As it is possible to evince in the figure 2.2, the web servers, backend and data nodes are replicated independently and separately. The division is necessary in order to isolate and preserve the system core functions. This practice is made for:

- *Safety purposes*: having phisically separated nodes prevent the possibility to reach certain hidden layer (for example the data layer) of the system, which needs to be absolutely safe as they contain sensible data.
- *Maintainance purposes*: when maintainance is performed over a certain part of the system, this separation allows to preserve that part of the system which does not require maintainance;
- *Node migration*: if the system will require to be partially moved from a cluster to another in the future, this division allows to transfer only a specific part of the network without the need of transferring the whole system.

2.2 Component view

The figure 2.3 represents the SafeStreets component diagram. Each of the components are wrapped inside a specific box that indicates the subsystem in which they belong to.
Subsystems are groups of components which perform similar operations: this boxing technique is a multilayer logical division that can be useful in order to categorize the components and the whole system at different layers of abstraction.
For the sake of simplicity, the component diagram pictures the application layer (backend) of the system and not the mobile and web app structures. The business

logic and core functions are fully contained in the application layer and therefore only that relevant part of the Safestreets system has been characterized.

–

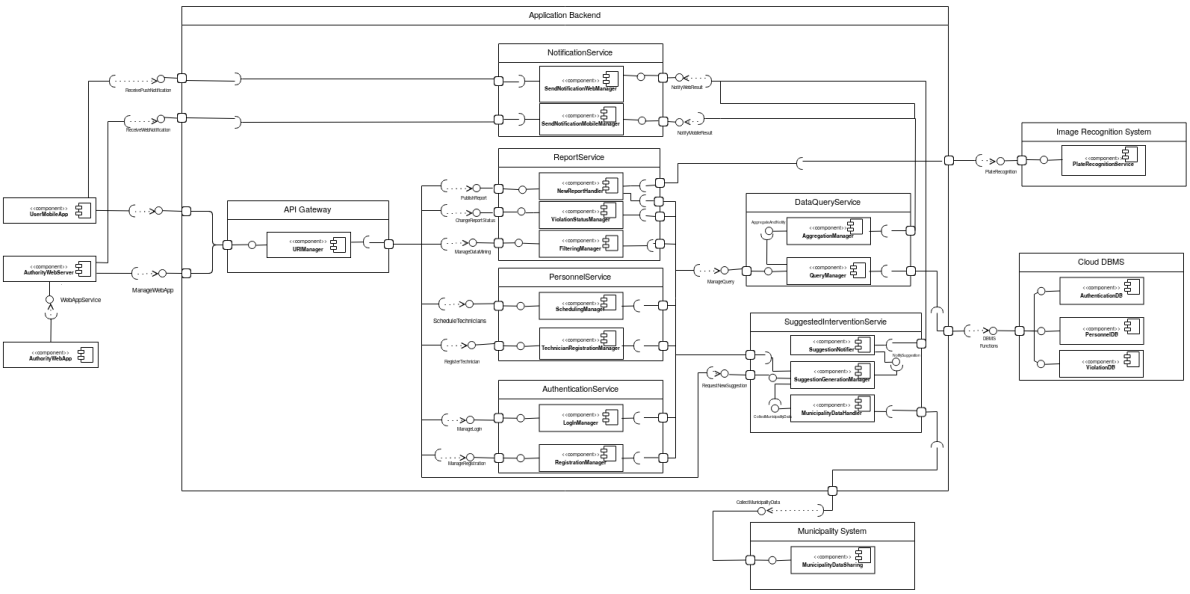

Figure 2.3: Component UML diagram

From the figure 2.3 it is possible to identify several subsystems of the application backend system. Here it is a detailed list of the subsystems:

- **API Gateway**: this subsystem is responsible for handling all the request coming from the frontend applications (web and mobile) and routing them into the appropriate components;
- **ReportService**: this subsystem is responsible for handling all the operations related to the violation reports. It interacts directly with the image recognition service in order to retrieve the license plate transctiptions;
- **PersonnelService**: this subsystem is responsible for handling all the operations related to the personnel management, such as registering new technicians and scheduling them to reports. This subsystem is accessible only by LSA accounts;
- **AuthenticationService**: this subsystem is responsible for handling the login and registration operations;

- **SuggestedInterventionService**: this subsystem is responsible for handling the operation related to the request of suggested interventions. It interacts directly with the municipality system in order to retrieve car accident information;

- **DataQueryService**: this subsystem is responsible for handling all the operations that requires a database interaction. It is responsible for interacting with the Cloud DBMS in order to retrieve all the information required by the application backend;

- **NotificationService**: this subsystem is responsible for forwarding results of the computation to the frontend applications (web and mobile).

Here it is a detailed explanation for each of the components shown in the component diagram:

- **AuthorityWebApp**: this component represents the web application that LSAs and technicians are supposed to use in order to perform any operation with the Safestreets system. In particular, the web app is the end-user application, therefore it is only a frontend interface;

- **AuthorityWebServer**: this component represents the web server that hosts the web application used by LSAs and technicians. The web server receives all the HTTP requests from the AuthorityWebApp and forwards them to the application backend system. Moreover, the web server notifies the web application when the responses of the previous requests are forwarded from the Notification Service;

- **UserMobileApp**: this component represents the mobile application which users interacts with. When users need to perform a specific function offered by the system, the mobile application automatically forwards it to the application backend system. Moreover, the application is always listening for push notifications coming from the Notification service, for example in case one of the previously reported violation is updated by the technicians;

- **URIManager**: this component is responsible for forwarding the requests coming from the frontend system to the backend subsystem which is able to handle the specific request. This routing is possible thanks to the target URI of the frontend request, that uniquely identify the associated component of the application backend system. Before forwarding the requests to

the specific components, the URIManager checks the token that is sent by the frontend application in order to retrieve (and then forward) the account category that performed the request. If the token has expired or it does not exist, the request is automatically rejected and the frontend application is requested for a login operation;

- **NewReportHandler**:this component handles the newly reported violation that are forwarded from the URIManager. The NewReportHandler is responsible for checking the correctness of the new reported data and to delegate the plate recognition to the PlateRecognitionService which is external with respect to the Safestreets system. Once the report has been verified and the plate transctiption has been gathered, the report is forwarded to the QueryManager, specifying an insertion request;

- **PlateRecognitionService**: this component is external from the Safestreets system. It has been created for graphical purposes and it is treated as a black box: given a plate image, it is able to transcribe the license plate and give it back to the NewReportHandler;

- **ViolationStatusManager**: this component is responsible for changing the status of the violation reports. This component can only be invoked only if the requesting account is a technician or an LSA;

- **FilteringManager**: this component is responsible for formulating mining operations. The URIManager forwards the mining information to the FilteringManager, which gathers and aggregate all the query parameter and forwards the formally complete query to the QueryManager specifying a select request;

- **SchedulingManager**: this component is responsible for performing the association between technicians and reports. This component can be invoked only if the requesting account is a LSA. After a series of checks, the verified requests is wired into a query and it is forwarded to the QueryManager specifying an insert request;

- **TechnicianRegistrationManager**: this component is responsible for the registration of new technicians into the system. This component can be invoked only if the requesting account is a LSA. The new technicians are

directly associated to the requesting LSA. All the needed queries are formulated and forwarded to the QueryManager specifying for each query an insert request;

- **LoginManager**: this component is responsible for logging into the system both users and authorities. The request forwarded from the URIManager contains a username and a password: an hash is calculated from the password and the (username, hash) tuple is wired into a select query which will be forwarded to the QueryManager, specifying a login request. Once the login has been approved by the QueryManager, an authentication token is received and backpropagated to the URIManager together with the username;

- **RegistrationManager**: this component is responsible for registering new users to the system. After a series of checks, the registration request is wired into a query and forwarded to the QueryManager specifying an insert request;

- **SuggestionGenerationManager**: this component is responsible for generating suggested interventions when an LSA requests it. The URIManager forwards the generic request of suggestion generation and the system automatically performs the following sequence of actions:

  - If not cached, formulate a query on the LSA competence area, asking for the most frequent categories of violations occured in that set of streets;
  - Save on a local cache the information;
  - Formulate a data request of car accident to the MunicipalityDataHandler and retrieve the response;
  - Aggregate the two kind of data and compute suggested interventions;
  - Forward the suggested interventions to the SuggestionNotifier.

- **MunicipalityDataHandler**: this component is responsible for taking the data requests coming from the SuggestionGenerationManager and formulate them into the municipality standard. Once this operation is performed, the request is forwarded to the MunicipalityDataSharing interface together with an authentication token provided by the municipality. If the token has

expired, a new one is requested and the operation is repeated. When data is provided, it is backpropagated to the SuggestionGenerationManager;

- **MunicipalityDataSharing**: this component is external from the Safestreets system. It has been created for graphical purposes and it is treated as a black box: when a formally correct request is received together with an authentication token, the MunicipalityDataSharing system forwards the requested information to the MunicipalityDataHandler;

- **SuggestionNotifier**: this component is responsible for forwarding the suggested interventions set to the LSA account web app through the SendNotificationWebManager;

- **QueryManager**: this component is responsible for interacting with the data layer of the Safestreets system. The QueryManager is able to connect directly with the Cloud DBMS and perform queries on the several databases of the Safestreets system. When this component receives query information, it acts as following:

  - Perform queries depending on their kinds:
    * Insert: tries to insert the specified data into the database;
    * Select: retrieve the filtered data;
    * Login: retrieve the specified data and checks wether the result contains at least one correspondency. If this check is approved, the QueryManager generates a token that is associated with the username and the category of the account. This token is then backpropagated to the LoginManager and embedded in the result of the query;
  - Forwards the result of the query to the AggregationManager.

- **AuthenticationDB**: this component is external from the Safestreets system. It has been created for graphical purposes and it is treated as a black box: it contains all the information associated to the Safestreets accounts;

- **PersonnelDB**: this component is external from the Safestreets system. It has been created for graphical purposes and it is treated as a black box: it contains all the information associated to personnel of the authorities: any kind of association between LSAs and technicians relation and the technicians schedules;

- **ViolationDB**: this component is external from the Safestreets system. It has been created for graphical purposes and it is treated as a black box: it contains all the information associated to the violation reports;
- **AggregationManager**: this component is responsible for packing the retrieved data from the QueryManager into a standard format and forward it to the SendNotificationWebManager or SendNotificationMobileManager depending on the kind of account that performed that specific operation;
- **SendNotificationWebManager**: this component is responsible for forwarding the result of the requested operation to the web application of the account whom requested such operations;
- **SendNotificationMobileManager**: this component is responsible for forwarding the result of the requested operation to the mobile application of the account whom requested such operations;

2.3 Deployment view

The following image represents the deployment diagram of the SafeStreets architecture. It shows the logical division of the software architecture as well as the distribution of the software components to their target nodes, on which they will be deployed.

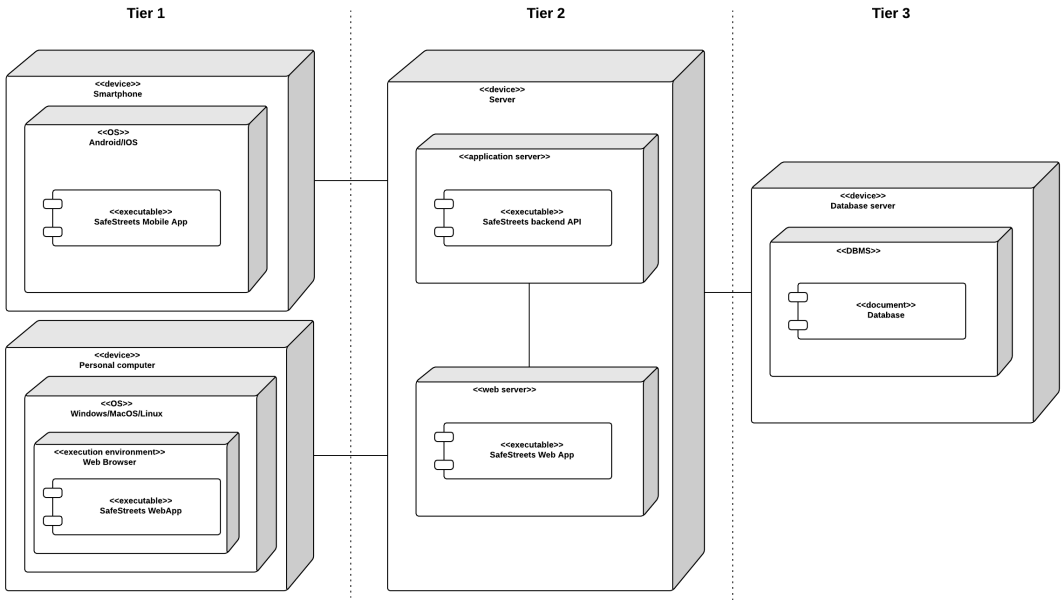

Figure 2.4: Deployment diagram

The diagram represents the architectural pattern chosen for the deployment of the SafeStreets service. It's a three-tiers architecture that has been conceived in order to provide a fast and lightweight client for both the users and the authorities. It is also important to note that the previous diagram is focused only on those components that will have to be effectively developed,that's why, for example, the plate recognitions services are not displayed in the diagram, as they already exists. Here is a short explanation of what each tier does:

- **Tier 1**: The first tier is the presentation tier of the architecture. The main function of this layer is to host the application of the users and the Web app of the authorities. It is worth noting that there's little or nothing business logic present in this layer apart from those that are strictly required for the correct functioning of the applications; the mobile application and the Web App are only presentation client that will have to interact with the second tier in order to get/elaborate information. As for the deployment aspect: the mobile application must be developed in order to cover the most of the devices, so it must be runnable at least on Android and IOS (to reduce the development time one could also think to use a cross-platform development framework) while the Web app must be compatible with the most modern web browser such as: Edge, Chrome, Firefox and Safari. As stated

before, the applications in this tier are just thin clients so they need to interact with the application server in order to get information. The Mobile application asks the information directly to the application server via the backend API while the Web app communicate with the web server that will eventually ask to the application server for data. All the communications are performed with RESTful calls over a secure connection;

- **Tier 2**: The second tier of the architecture is called the logic layer. It is the layer in which all the business logic is located; all the functionalities and the elaborations are computed here. The implementation is achieved via a server that runs two subservices separately: the Application server and the Web server. The Application server hosts the core of the system that is to say the SafeStreets Backend API, this is the piece of software that will handle all of the requests and the offered services. Instead the Web server hosts the content for the web app. In case some pages need to be created with some data, the Web server can communicate with the backend API in order to retrieve the information needed;

- **Tier 3**: The final layer is the Data tier. It handles the data access with the remote databases. The choice to move this into a separate tier is due to the fact that with this approach data is independent from the business logic.

2.4 Runtime view

2.5 Component interfaces

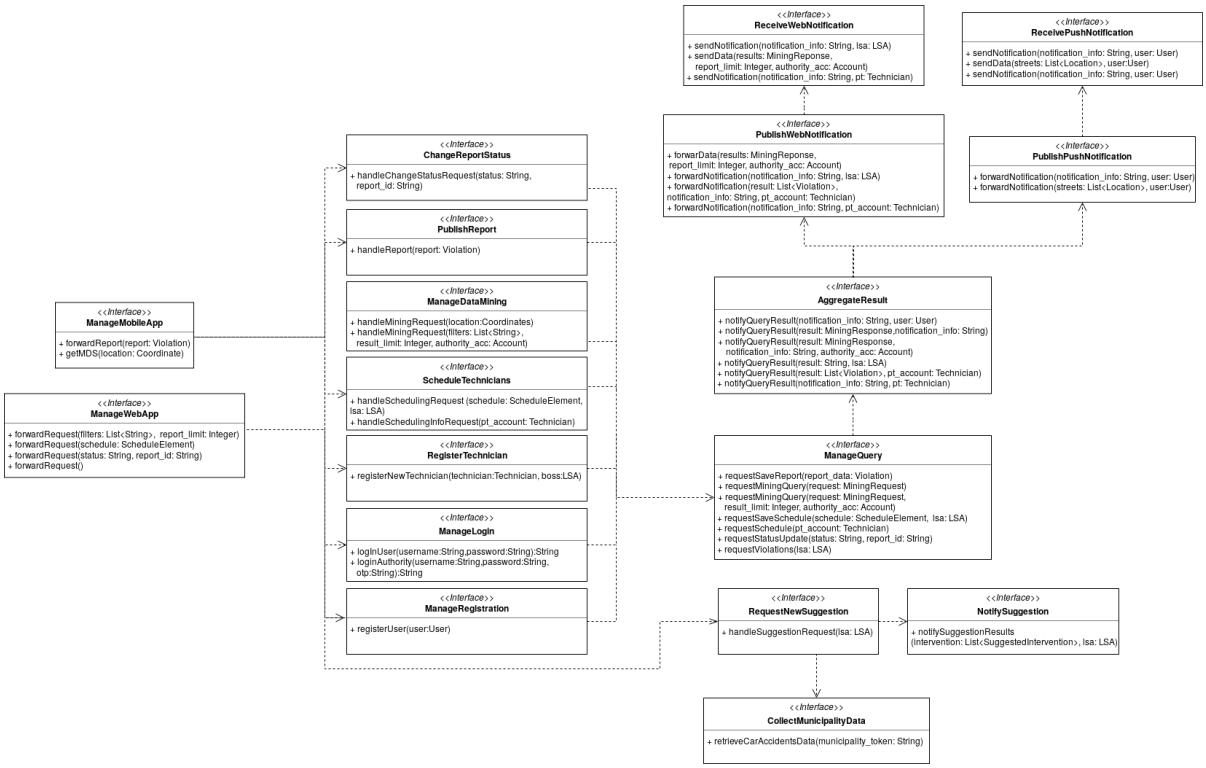

Figure 2.5: Interface diagram

In the figure 2.5 the component interfaces belonging to the application server are represented with respect to what was shown in the Component diagram. The arrows represent dependency relations and depict the dependency tree of the application backend interfaces. The functions represented in the interface diagram have the following properties:

- The signature of this functions are built only for conceptual showing purposes and they might differ in the final implementation;
- The main part of this diagram's functions have been pictured for guaranteeing a correct correspondency with the sequence diagrams of the previous paragraph, therefore some functions are submitted to the overloading technique;

- they are offered by the application backend component to one another or to other Safestreets system components (mobile application and web application).

In order to better understand how the interfaces work and how they are correlated, the following list of clarifications (assumptions) have been considered:

- The frontend interfaces have been generically represented by the ReceivePushNotification and ReceiveWebNotification interfaces for dependency purposes. For the sake of simplicity, they have not been analyzed in depth as it is not relevant for the system global functioning;
- The workflow of the operation of the Safestreets system is the following:
  - In order to perform any of the functionalities offered by the application backend, the external applications interacts to the ManageMobileApp and ManageWebApp by a RESTful API service;
  - The functions of these two interfaces forward the requests to the interface of the component related to the called API, propagating also all the parameters embedded to the request (if correctly formalized);
  - When one of its function is called, the interface of the target component takes in charge the request and let the component do its computation;
  - Once computation is done, the ManageQuery interface is called in order to interact with the database;
  - At this point, the component related to the ManageQuery interface execute all the necessary queries and directly forwards the results to the AggregateResult interface;
  - In this component, the result of the computation is formalized and forwarded to the PublishWebNotification interface in case the request came from a web application (LSA or Technician accounts) or from a mobile application (User)[1];
  - The publishing interfaces collect the responses and forward them to the devices that performed the operation[2];
  - The ReceiveWebNotification and ReceivePushNotification API interfaces receive asynchronously the responses and let the local applications handle them.

It is paramount to underline that the workflow of operations cannot be traversed in the opposite way. For this reason, all interface functions does not have a return type;

- The calls of the interfaces functions are performed synchronously: as explained at the previous point, the system operations flow only one way and the output of the previous ones is piped as input to the next one. Without a synchronous function calling system, the piping could not be implemented;

- The ManageQuery interface's functions don't have a queryType parameter because it can be inferred from the parameters of the functions: the requestMiningQuery function can be univocally inferred as a SELECT request, the requestSaveSchedule can be univocally inferred as an INSERT and so on;

- The ManageQuery interface is the only interface that leads to the QueryManager component. Therefore, all of the previous interfaces depend on it for query executions;

- The LoginManager's functions are the only ones that return a value (String): this is due to the authentication token, which has to be backpropagated to the URIManager component in order to perform its functions.

[1]: all of the functions of the interfaces that requires a response have the submitter's account as a parameter: this is both necessary for the specific computation of the offered functions and for mapping the response to the submitter.
[2]: the Safestreets system handles a table with <Account, Address> tuples, which allows the Notification Service components to forward the results of computation.

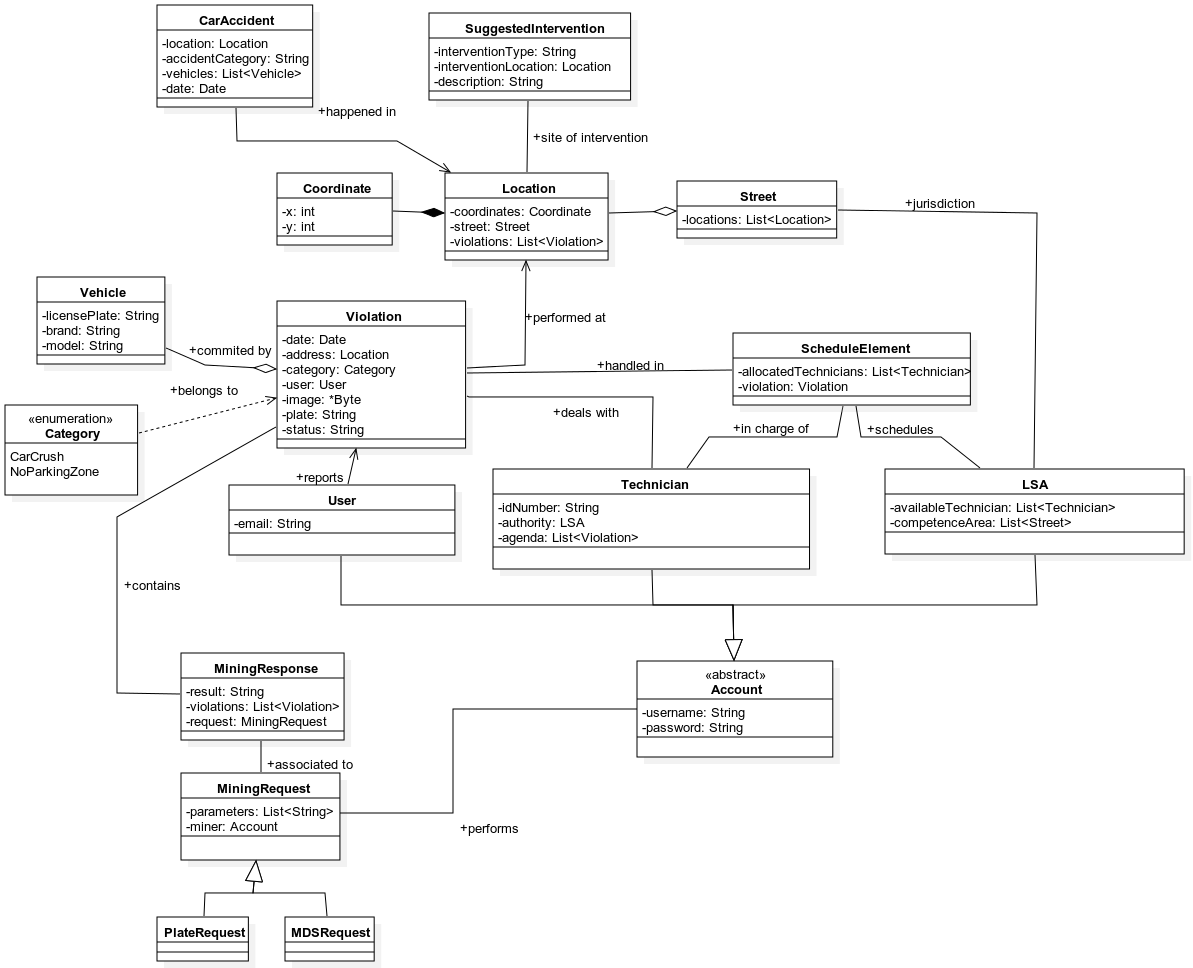

Figure 2.6: Class diagram

In figure 2.6, the Safestreets class diagram is represented. This version of the class diagram is very similar to the one present in the RASD document, a part from:

- The Mining request generalization, in order to add a new useful and extendible design pattern;
- The ScheduleElement class, which represents the n-n relation between Technicians and Violations: instead of representing this multiple relation in a singleton class which associate Technicians and Violations, the ScheduleElement objects represent tuples that links a single violation to many Technicians (<5 as stated in the RASD document). In this way, the system allows LSAs to schedule many Technicians to a specific violation as pictured in the web application mockups. The opposite operation (Technician to many violation) is not allowed up to this version of the system;

- The CarAccident class, which represent the aggregation of a single datum collected from the Municipality information system.

2.6 Selected architectural styles and patterns

2.6.1 RESTful API architecture

A RESTful API is based on representational state transfer (REST) technology, an architectural style and approach to communications often used in web services development. A RESTful API breaks down a transaction to create a series of small modules. Each module addresses a particular underlying part of the transaction. This modularity provides developers with a lot of flexibility. Every specific module of the REST API can be invoked remotely by an URI which uniquely identifies that service. Once an API has been casted, a certain function is executed on the RESTful API server: the result of this computation is then sent back to the invoking client by a callback function that it offers. As far as the SafeStreets system is concerned, both the web and the mobile application make use of RESTful APIs offered by the application backend system:

- the mobile application directly interacts with the APIs through their URIs;
- the web application retrieves data and performs operations that require the application backend system through API invokation through a **Promise-Deferred** asynchronous calling system.

Also, the mobile application make use of the Google Maps API for the mapping service.
At the same time, the application backend system make use of the RESTful APIs offered by the Municipality information system and the Image recognition service. Therefore, a certain information system can be both offering and using RESTful APIs for its purposes.

2.6.2 MVC design pattern

As deeply explained in the overview paragraph of this section, the MVC design pattern is the core of the functioning of the SafeStreets system. In order to provide a detailed explanation of how this popular design pattern works, here it is a detailed description from the Wikipedia website:

- **Model**: *" The central component of the pattern. It is the application's dynamic data structure, independent of the user interface. It directly manages the data, logic and rules of the application."*
- **View**: *"Any representation of information such as a chart, diagram or table. Multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants."*
- **Controller**: *"Accepts input and converts it to commands for the model or view. In addition to dividing the application into these components, the model–view–controller design defines the interactions between them. The model is responsible for managing the data of the application. It receives user input from the controller. The view means presentation of the model in a particular format. The controller responds to the user input and performs interactions on the data model objects. The controller receives the input, optionally validates it and then passes the input to the model. As with other software patterns, MVC expresses the "core of the solution" to a problem while allowing it to be adapted for each system."*

The main advantages that the MVC design pattern provides are:

- Simultaneous development – Multiple developers can work simultaneously on the model, controller and views;
- High cohesion – MVC enables logical grouping of related actions on a controller together. The views for a specific model are also grouped together;
- Loose coupling – The very nature of the MVC framework is such that there is low coupling among models, views or controllers;
- Ease of modification – Because of the separation of responsibilities, future development or modification is easier;
- Multiple views for a model – Models can have multiple views.

In the SafeStreets system, the model is represented by the Cloud DBMS and the DataQueryService subsystem of the backend application and the Municipality-DataHandler of the SuggestedInterventionService; the view is represented by the web and mobile application; the controller is represented by all the application backend subsystem except from the DataQueryService and the Municipality-DataHandler of the SuggestedInterventionService.

2.6.3 Three-tier architecture

Three-tier architecture is a client-server software architecture pattern in which the user interface (presentation), functional process logic ("business rules"), computer data storage and data access are developed and maintained as independent modules, most often on separate platforms. The Wikipedia website states that in the three-tier architecture it is possible to evidence:

- **Presentation tier**: *"This is the topmost level of the application. The presentation tier displays information. It communicates with other tiers by which it puts out the results to the browser/client tier and all other tiers in the network. In simple terms, it is a layer which users can access directly (such as a web page, or a mobile application)."*

- **Application tier**: *"The logical tier is pulled out from the presentation tier and, as its own layer, it controls an application's functionality by performing detailed processing."*

- **Data tier**: *"The data tier includes the data persistence mechanisms (database servers, file shares, etc.) and the data access layer that encapsulates the persistence mechanisms and exposes the data. The data access layer should provide an API to the application tier that exposes methods of managing the stored data without exposing or creating dependencies on the data storage mechanisms. Avoiding dependencies on the storage mechanisms allows for updates or changes without the application tier clients being affected by or even aware of the change. As with the separation of any tier, there are costs for implementation and often costs to performance in exchange for improved scalability and maintainability."*

Although the three-tier architecture may seem very similar to the MVC design pattern, they present a crucial difference: in the three-tier architecture the client tier is forbidden to communicate with the data tier (as it is implemented in the SafeStreets system) while the communication in the MVC pattern is triangular. As far as the SafeStreets sytem is concerned, only the application tier is responsible for the interactions between different layers.

2.7 Other design decisions

2.7.1 Lightweight thin client

The client software is narrowly purposed and lightweight: only the host server or server farm needs to be secured,rather than securing software installed on every endpoint device (although thin clients may still require basic security and strong authentication to prevent unauthorized access). One of the combined benefits of using cloud architecture with thin client desktops is that critical IT assets are centralized for better utilization of resources. Unused memory, bussing lanes, and processor cores within an individual user session, for example, can be leveraged for other active user sessions.
The simplicity of thin client hardware and software results in a very low total cost of ownership, but some of these initial savings can be offset by the need for a more robust cloud infrastructure required on the server side. Therefore, the SafeStreets mobile application can be run on any device (even cheap ones) as the computational power required is extremely low.
Mobile devices and web browser are responsible only for showing the results of the computation of the backend system: this functiong allows to speed up the execution of the system functions that are performed on a powerful hardware on the server side.

2.7.2 Relational database

Relational databases are the most used technique of data storaging. A relational database has at least to guarantee the following aspects:

- Present the data to the user as relations (a presentation in tabular form, i.e. as a collection of tables with each table consisting of a set of rows and columns);
- Provide relational operators to manipulate the data in tabular form.

In order to build a formally correct relational database, it is necessary to create a **Relation model** of the data that the database is going to contain. The relational model organizes data into one or more tables (or "relations") of columns and rows, with a unique key identifying each row. Rows are also called records or

tuples.Columns are also called attributes. Generally, each table/relation represents one "entity type" (such as user or violation). The rows represent instances of that type of entity and the columns representing values attributed to that instance (such as username or date).
Moreover, tables are connected through relations, which are a logical connection, established on the basis of interaction among these tables.
In order to interact with a relational database, it is necessary to use a RDBMS, such as PostgreSQL or MySQL and so forth.
The main advantages of using a relational database are:

- **Accuracy**: Data is stored just once, eliminating data deduplication;
- **Flexibility**: Complex queries are easy for users to carry out;
- **Collaboration**: Multiple users can access the same database;
- **Trust**: Relational database models are mature and well-understood;
- **Security**: Data in tables within a RDBMS can be limited to allow access by only particular users.

In the SafeStreets system, relational databases are very useful as:

- data is structured in a fixed way, therefore NoSQL databases are not very useful;
- many users and authorities accounts have to perform frequent queries to the DB;
- the hierarchical structure of the RDBMS allow SafeStreets to perform some of the privileges controls directly on the inside the RDBMS.

2.7.3 Cloud database

A cloud database is a database that typically runs on a cloud computing platform, and access to the database is provided as-a-service. Database services take care of scalability and high availability of the database. Database services make the underlying software-stack transparent to the user. The design and development of typical systems utilize data management and relational databases as their key building blocks. Advanced queries expressed in SQL work well with the strict relationships that are imposed on information by relational databases. However,

relational database technology was not initially designed or developed for use over distributed systems. This issue has been addressed with the addition of clustering enhancements to the relational databases, although some basic tasks require complex and expensive protocols, such as with data synchronization.
The main advantages of adopting a cloud DBMS are:

- Scalability: scaling along any dimension generally requires adding or subtracting nodes from a cluster to change the storage capacity, I/O operations per second, or total compute available to bring to bear upon queries. That operation, of course, requires redistributing copies of the data and sending it between nodes. Although this was once one of the hardest problems in building scalable, distributed databases, the new breed of cloud-native databases can take care of these issues efficiently;

- Reduced Administrative Burden: a cloud-hosted, mostly self-managed database doesn't eliminate a database administrator, but it can eliminate unnecessary features that typically consume much of a DBA's time and efforts. That allows a DBA to focus his or her time on more important issues;

- Improved Security: by running databases on in-house servers, it's SafeStreets responsibility to think about security. It is necessary to ensure that databases have an updated kernel and other critical software, and it is necessary to keep up with the newest digital threats. By delegating all these operations to the cloud DBMS company, a lot of work is saved.

All of this advantages relieves an heavy burden from the budget of the project, as the expenses of a Cloud DBMS service is much lower than affording a local DBMS.

User interface design

Requirements traceability

Implementation, integration and test plan

Effort Spent

6.1 Luca Loria

Day	Hours	Topic
20/10/2019	1.5	Text assumptions
22/10/2019	1	Domain assumptions
23/10/2019	2	Design constraints
24/10/2019	2	Revision ch 1 and 2
26/10/2019	3	Software system attributes
28/10/2019	0.5	Performance requirements
29/10/2019	2	External interface req
30/10/2019	5	Mockups creation
31/10/2019	1.5	Revision ch 3
01/11/2019	1	Alloy signatures
04/11/2019	1.5	Alloy facts part 1
06/11/2019	3	Alloy facts part 2 and world predicates
07/11/2019	4	Finish alloy, fix class diagram and start impaginating
08/11/2019	3.5	Reviewing requirements and sequence diagrams. Create references and Revision. Create alloy world section in RASD. Start impaginating correctly
09/11/2019	2	Shared phenomena matrix, frontpage and pagination fixes
10/11/2019	2	Final Revision

6.2 Nicolò Albergoni

Day	Hours	Topic
22/10/2019	2.5	Purpose
23/10/2019	3	Scope, Current system
24/10/2019	1.75	Goals, Overview
26/10/2019	3	Product perspective, Product function
27/10/2019	3	Product function, Revision ch 1 and 2
29/10/2019	2.5	Scenarios
30/10/2019	4	Use cases
31/10/2019	3	Use cases, Revision ch 3
01/11/2019	1	Finish use cases
02/11/2019	2.75	Requirements
05/11/2019	2.25	Finish requirements
06/11/2019	2.5	Sequence diagrams
07/11/2019	3.25	Sequence diagrams
08/11/2019	1.75	Finish and reviewing of sequence diagrams
09/11/2019	2	Use case diagrams, pagination fixes
10/11/2019	2	Traceability matrix, final revision

References

- LateX Workshop extension for Visual Studio Code:
  https://github.com/James-Yu/LaTeX-Workshop/
- LateX compiler:
  https://www.latex-project.org/
- StarUML for UML diagrams:
  http://staruml.io/
- DrawIO for some UML diagrams
  http://draw.io
- MVC design pattern wikipedia:
  https://en.wikipedia.org/wiki/Model-view-controller
- Multitier architecture wikipedia:
  wikipedia
- Relational database advantages:
  https://searchdatamanagement.techtarget.com/definition/relational-database