POLITECNICO
MILANO 1863

Design Document

Software Engineering II Project - A.Y. 2019-2020

SafeStreets

Authors	ID Numbers
Andrea Furlan	944774
Cosimo Russo	945891
Giorgio Ughini	944710

December 10, 2019 Version 1.1

Contents

1 Introduction

1.1 Purpose

The Design Document is intended to provide a deeper functional description of the SafeStreets system-to-be by giving technical and design details and describing the main architectural components as well as their interfaces and their interactions. The relations among the different modules are pointed out using UML standards, a traceability table and other useful diagrams showing the structure of the system. This document aims to guide the software development team to implement the architecture of the project, by providing a stable reference and a unified vision over all parts of the software itself and clearly defining how every part interacts with the others.

In summary the document will help the developers to create the software by referring to:

- A high level architecture.
- The design patterns to use.
- The main components and the interfaces they provide to communicate with one another.
- The Runtime behaviour of the system.
- The implementation, integration and testing plan.

1.2 Scope

As SafeStreets has a wide user target, the main focus of our system design phase will be to create an application capable of reaching the vast majority of them. Thus the architecture must be designed with the intent of being very easy to use, but also maintainable and extensible, foreseeing future changes that will be applied to make SafeStreets compatible with other Police Departments. In addition our system should be maintainable enough to

keep it up to date with all the mobile phone updates that will come in the next years. This document aims to drive the implementation phase so that cohesion and decoupling are increased in full measure. In order to do so, individual components must not include too many unrelated functionalities and they should reduce interdependence between one another.

1.3 Definition and Acronyms

1.3.1 Definitions

As this document is the natural continuation of the Requirements Analysis and Specification Document, many of the definitions used in that document will be used in this Design Document. Thus, to be fully clear and many self-explanatory as possible we include a subset of definitions taken from the Requirement Document.

- *Data Mining*: The process of discovering patterns in large data sets involving methods at the intersection of machine learning, statistics, and database systems.
- *Computer Vision*: The Computer Vision includes methods for acquiring, processing, analyzing and understanding digital images, to extract high-dimensional data from the real world.
- *Report*: The data that a user has provided to the authorities that witness a violation by a vehicles.
- *Ticket*: Notice issued by a law enforcement official to a motorist or other road user, indicating that the user has violated traffic laws.
- *Violation*: The general infraction that a vehicle has done and that has been reported by a SafeStreets user.

1.3.2 Acronyms

- *API*: Application Programming Interface
- *AI*: Artificial Intelligence
- *UML*: Unified Modeling language
- *GPS*: Global Positioning System
- *ACID*: Atomicity, Consistency, Isolation, Durability
- *IEEE*: Institute of Electrical and Electronic Engineers
- *RASD*: Requirements Analysis and Specification Document

1.4 References

- The 2019-2020 Software Engineering 2 Project Assignment document
- The IEEE Standard for DD
- The RASD of SafeStreets

1.5 Document Structure

This document is divided in five parts:

- **Introduction**:

  a description of the purpose and the scope of this Design Document is provided. A subsection dedicated to the understanding of some acronyms and definitions is also present.

- **Architectural Design**:

  in this section the main system components, their sub-components and the relationships among sub-components are shown.

  This section will mainly focusses on design choices, interactions, used architectural styles and patterns.

- **User Interface Design**:

  based on the RASD's user interface section, a more detailed overview on how the user interface should look like and behave is provided.

- **Requirements traceability**:

  This section describes how the design elements and components present in this DD satisfy and map the requirements defined in the RASD of SafeStreets.

- **Implementation, integration and test plan**:

  an explanation about the strategies that will be used to implement and to test SafeStreets' application that will take place in the development part of this project is provided.

In the last part of the document a short note about the softwares used and the effort spent in producing this DD by its authors is shown.

2 Architectural Design

2.1 Overview

This section of the document gives a detailed view of the physical and logical infrastructure of the system-to-be.
It provides the different types of view over the system as well as the description of the main components and their interactions. The logical division of the application consists of 3 layers: presentation, application and data. Every layer, then, is made of several microservices.

1. Subsection 2.2.1 **Presentation layer**: The Presentation Layer is the space where interactions between humans and machines occur. The goal of this interaction is to allow effective operation and control of the machine from the human end, whilst the machine simultaneously feeds information back to the server.

2. Subsection 2.2.2 **Application layer**: The Application Layer is the part of the program that encodes the real-world business rules that determine how data can be created, stored, and changed.

3. Subsection 2.2.3 **Data layer**: The Data Layer is where the data is physically stored. In this layer happen all the retrieve and update operations on the data.

Then, in the following sections, a top down approach will be adopted to describe the architectural design of the system:

2.2 High-Level components: A description of high-level components and their interactions.

2.3 Component View: A detailed insight of the components described in the previous section.

2.4 Deployment view: A set of indications on how to deploy the illustrated components on physical tiers.

2.5 Runtime View: A thorough description of the dynamic behavior of the software with diagrams for the key-functionalities.

2.6 Component Interfaces: A description of the different types of interfaces among the various described components.

2.7 Selected Architectural styles and patterns: A list of the architectural styles, design patterns and paradigms adopted in the design phase.

?? Other design decisions: A list of all other relevant design decisions that were not mentioned before.

2.2 High-Level components: general architecture identification

The logical division of the application consists of 3 layers that will be shown here: presentation, application and data.
Here we provide for each tier the definition, choice reasons and used technology:

2.2.1 Presentation layer

the presentation layer consists of a mobile application for the users and a website for the officers. The application will be available from the major app stores while the website will be distributed through a web server. It is important to note that both the application and the website will only provide the user interface and will retrieve data from the application layer.

2.2.2 Application layer

This layer will employ a microservices architecture. Each microservice will handle a single functionality of the application in an atomic and stateless manner. Also, each service will expose a REST interface accessible over

HTTPS to be able to handle requests from the clients, the police systems and other microservices.

Microservices will be deployed in containers that will be able to efficiently scale based on the load on the single service, thus ensuring maximum scalability and elasticity and never wasting resources. Since microservices are stateless by definition, redundancy can be easily implemented. This is a key point toward the availability requirement.

This architecture opens the possibility for some services to be bought instead of being implemented from scratch. For example the login/registration service will use the App ID service from IBM instead of a homemade solution.

The main services are:

- **Users Management** - Login and registration
- **Violations and Tickets management**
- **Tickets Hashing system** - Allows the police to check for the integrity of a ticket by comparing an hash
- **Statistics**
- **Tickets checking** - Exploits computer vision and data mining to automatically check tickets, using information from different sources. Tickets that are suspected to be invalid are then sent to the local police for a double check.
- **Unsafe positions** - Uses an AI to extract unsafe positions from violations and accidents and to provide a possible solution to it. Such solutions are then available for the local police.
- **Computer Vision** - Extract info from pictures, like the car color and model
- **Data Mining** - Extracts patterns from huge amounts of data useful for other services

2.2.3 Data layer

Each microservice will have its own storage engine that cannot be directly contacted by other services. Pay-per-use services, such as *Users management*, have their internal storage system that is abstracted, for the services that need a storage system it will be hosted in the cloud and managed by IBM, billed per GB used. This system is the best possible abstraction for scalability.

All storage systems are guaranteed to have replicas and scheduled backups in order to avoid data loss.

2.3 Component View

2.3.1 Database overview

Each microservice will have its own data storage system but not all of them will result to be a database. In particular, those microservices who will need object storage for training purposes (such as Computer Vision and Data Mining) will use a transparent-to-us data layer.
However, to make SafeStreets data's persistent, these database will be deployed:

- **Violations DB**: This database will store all the data regarding a single violation except for images. As the data inside it are well structured this database will be a relational database.
- **Statistics DB**: As the above database, the Statistics DB will be deployed to store all the data regarding the usage of SafeStreets. This data are well structured over time. Indeed, the mechanism chosen for such DB is the relational database container.
- **Suggestion DB**: Every time the Data Mine microservice runs, it will produce possible intervention to some areas considered dangerous. Such advises that comes from the Data Mining algorithm will be stored in a

highly unstructured database, formerly in a NoSQL document-oriented database.

- **Hash DB**: This database allows SafeStreets to ensure that the chain of custody is never broken. Indeed, when someone reports a violation and opt in for the automatic ticket, a hash of the entire violation object along with its media elements, is stored here. This database should be configured once as read-only as this is the key-aspect of its usage.

Obliviously, all the used systems must obey to the laws regarding data protection (like GDPR), which means that (list not exhaustive):

- Sensible data such as passwords and personal information must be encrypted properly before being stored in the selected microservice.
- Users must be granted access only upon provision of correct and valid credentials.

Below we include a description of each microservice (including DBMS) to better understand them all, then we'll describe each single database in section 2.3.4.

2.3.2 IBM Services

In this subsection it is explained in detail what IBM services will be used. For services that are used with the pay-as-you-go mechanism it is briefly explained how they work and some diagrams are included to better understand the concept.

- **Login - IBM App ID**: This microservice is about User Authentication and Management, and provides a log-in framework. App ID will be used to add authentication to our web and mobile apps securing our Cloud-native applications and services on IBM Cloud. By requiring users to sign in to our app, we will be able to store user data such as app preferences or information from public social profile to recognize

user that reports violations and customize each user's experience within the app.
When a user is successfully authenticated, the application receives tokens from App ID. The service uses three main types of tokens to complete the authentication process: Access Token, Identity Token and Refresh Token.

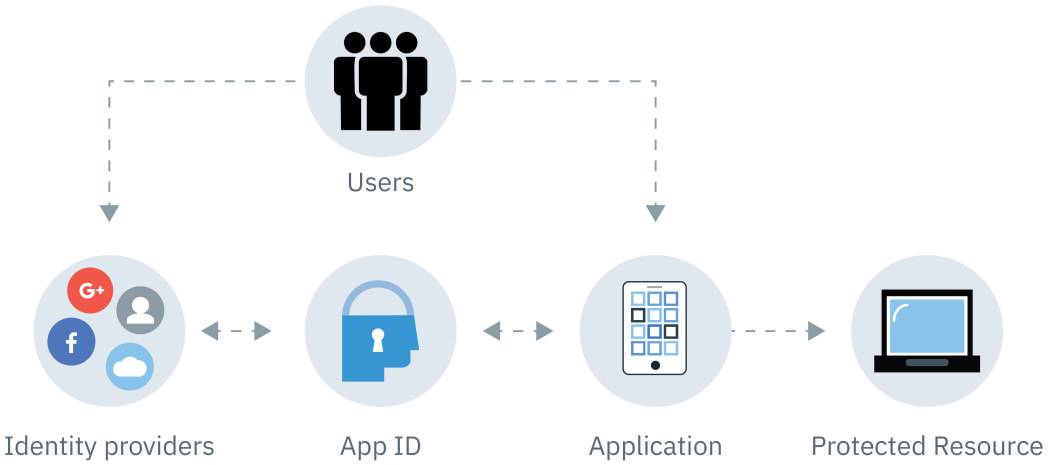


Figure 1: IBM App ID working diagram.

- **Computer Vision - IBM Watson Visual Recognition**: This service uses deep learning algorithms to analyze images for scenes, objects, and other content. In our scenario it will be used to certify automatic tickets and to extract various information from violation's pictures.

  Its usage is as simple as making an HTTP request to its POST API sending the image then waiting for it to finish its work. The response that it returns includes keywords that provide information about the content along with an estimate probability of them being identified.

  In addition, IBM Watson Visual Recognition supports high availability with no single point of failure. Recovering from potential disasters that affect an entire IBM location requires planning and preparation but can always be done saving all of our custom training model.

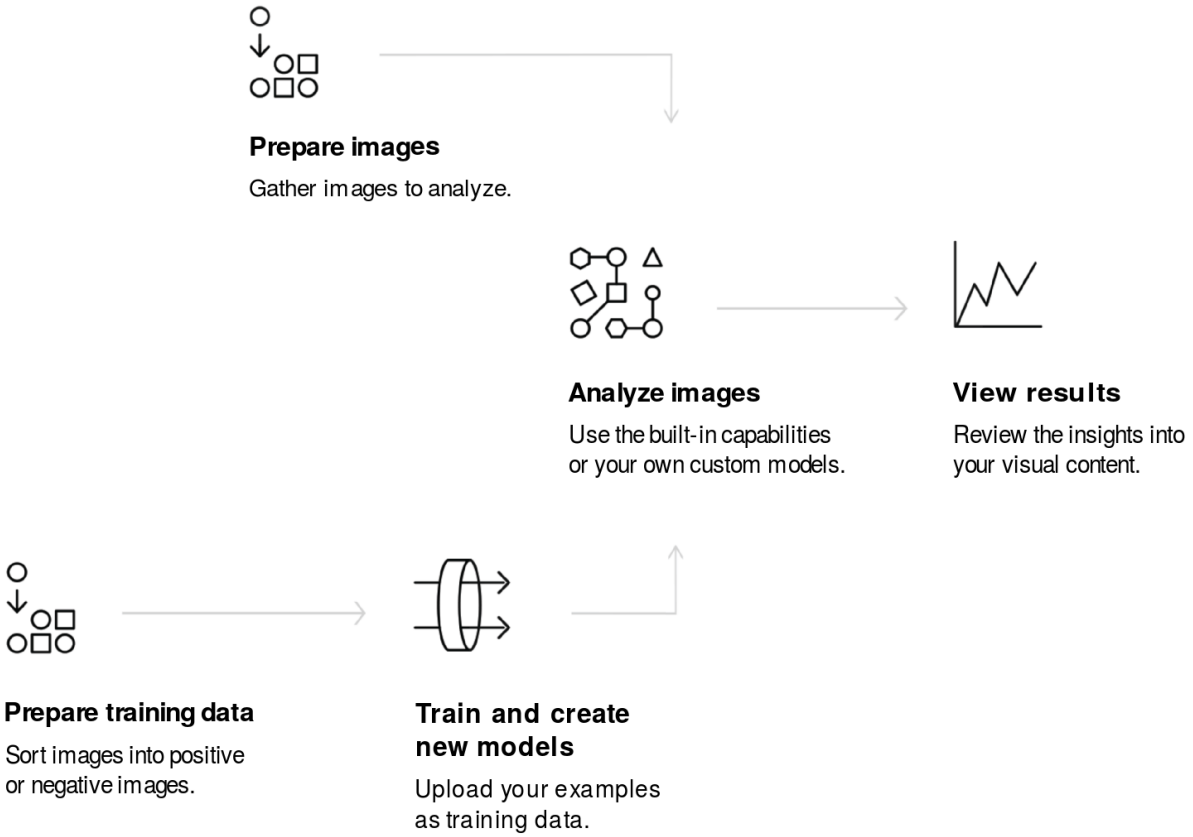


Figure 2: IBM Watson Visual Recognition working diagram.

- **Data Mining - IBM Discovery**: IBM Discovery makes it possible to rapidly build cognitive, cloud-based exploration applications that unlock insights hidden in unstructured data.

  This microservice is particularly useful when, at a given time everyday, SafeStreets crosses data coming from the municipality with its own data to extract potential suggestions.

  With Discovery, it only takes a few steps to prepare our unstructured data coming from different sources and to create a query that will pinpoint the information SafeStreets needs. Discovery automatically uses data analysis combined with cognitive intuition to take the unstructured data and enriches it to discover hidden information.

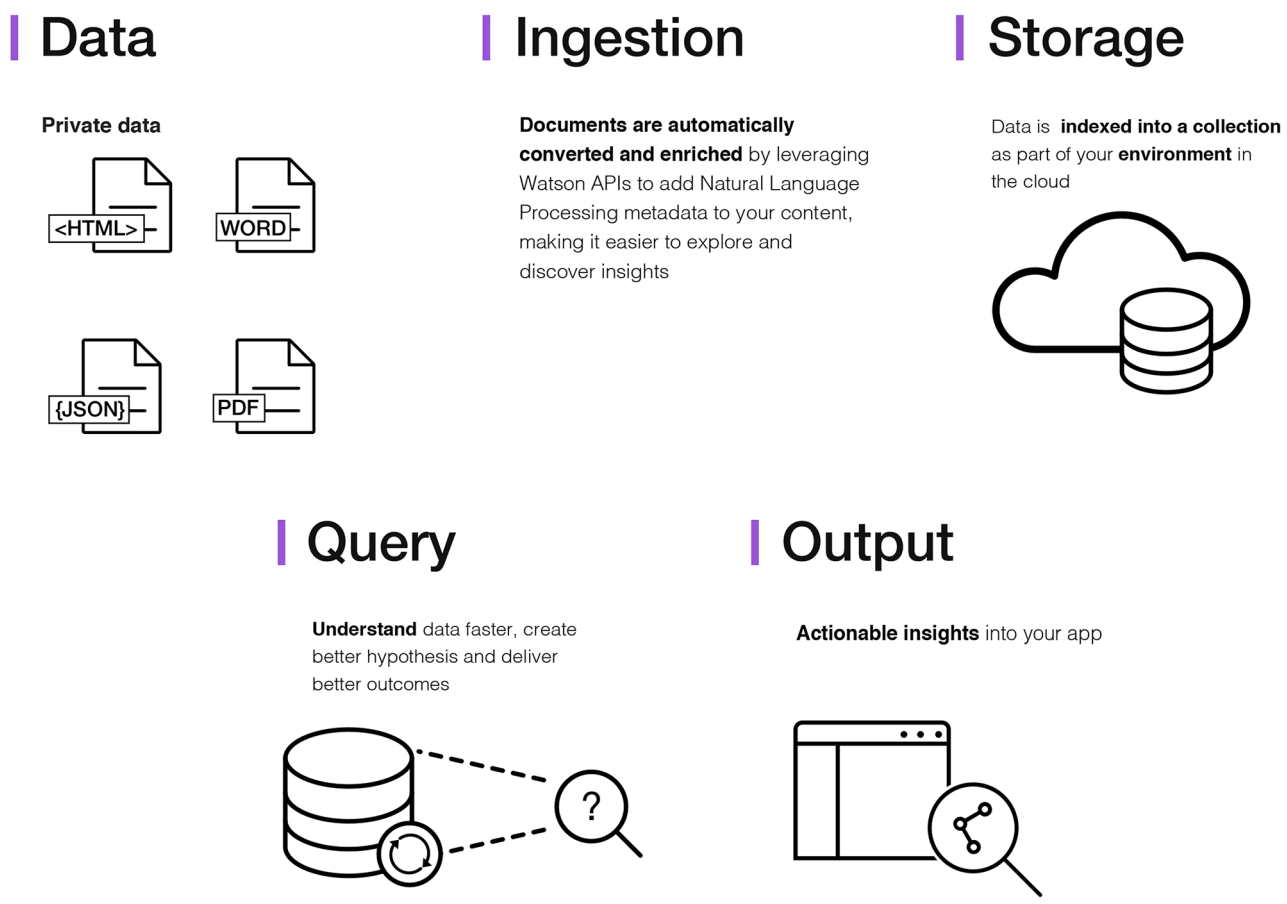


Figure 3: Complete architecture of our Discovery solution.

- **Image Storage - IBM Cloud Object Storage**: When a user sends one or more pictures in a report along with various information on the violation he is reporting, these files must be obviously stored somewhere. In our case, to ensure resiliency and the other proprieties expressed in the RASD, we opted for storing these image in the IBM Cloud. Indeed, information stored in the IBM Cloud Object Storage is encrypted and dispersed across multiple geographic locations, and accessed over HTTP using a modern RESTful API. In addition, IBM Cloud Object Storage allows SafeStreets to separate metadata from the image, but store them as objects in the same storage. That means that for every image, SafeStreets can associate it with highly unstructured

data, different for each image.

For example when someone reports a violation, the related images are immediately saved in the Storage, along with their metadata. Then, when the Computer Vision algorithm elaborates them, the metadata of such images will be updated by means of a PUT request with the information that it will extract (such as the presence of no-parking signs).

Last but not least, the IBM Cloud Object Storage can be easily associated with IBM Discovery as an external source for training.

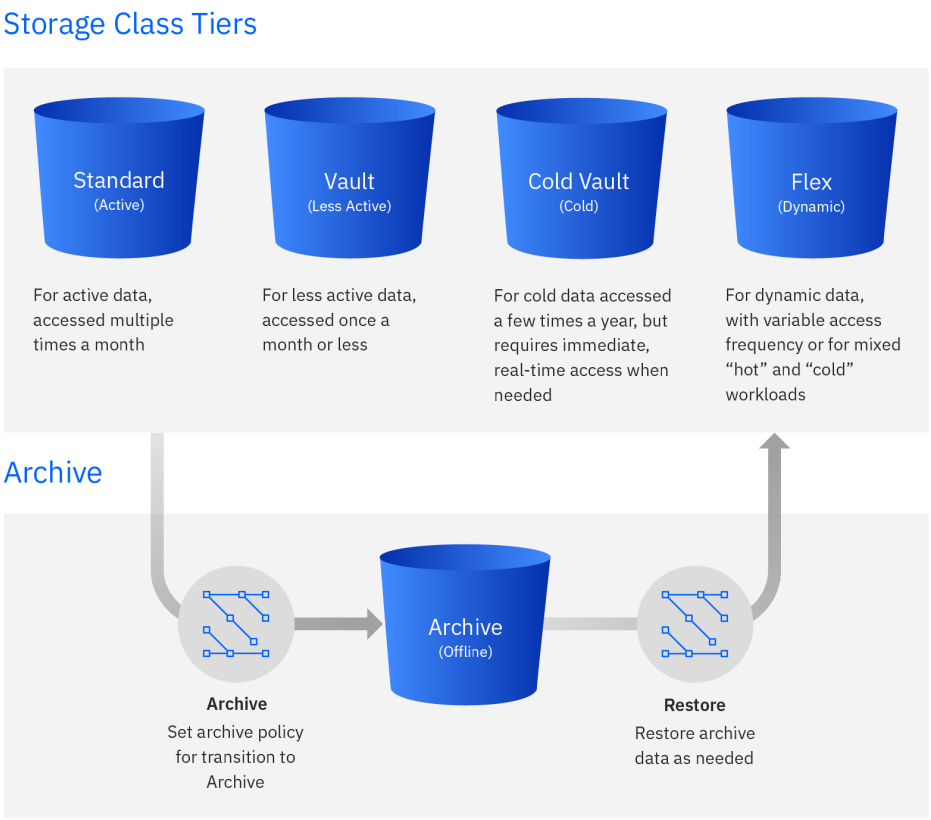


Figure 4: IBM automatically tier data to the lowest-cost archive.

- **NoSQL Database - IBM Cloudant**: IBM Cloudant is a distributed database that is optimized to handle heavy workloads that are typical

of large, fast-growing web and mobile apps. Available as an SLA-backed, fully managed IBM Cloud service, Cloudant elastically scales throughput and storage independently.
SafeStreets will use Cloudant every time a NoSQL Database is required, for example to manage the highly unstructured data in the Suggestion DB. Moreover, Cloudant is ISO27001, SOC 2.2 compliant and HIPAA ready. All data is encrypted over the wire and at rest. The service integrates with IBM Authentication and Management (IBM App ID) for granular access control at the API level.
Using this service, SafeStreets will not need any server (Cloudant is serverless) and its data will be automatically replicated closer to all the places it needs to be, for uninterrupted data access.

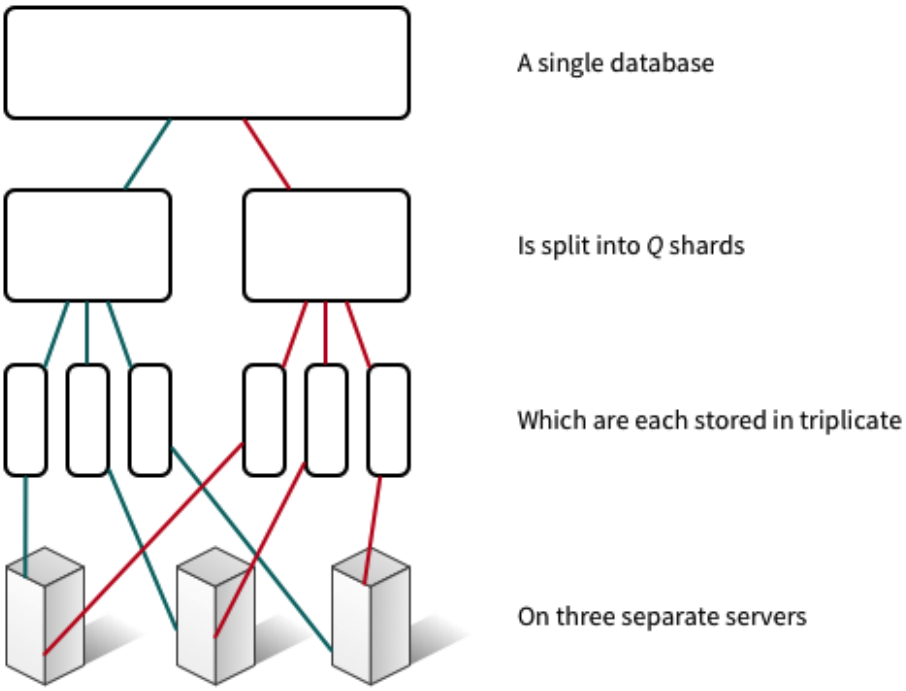


Figure 5: How data is stored in IBM Cloudant

All Q shards together contain the data within database. Each shard is stored in three separate copies. Each shard copy is called a shard replica. Each shard replica is stored on a different server. The servers

are available within a single location data center. The collection of servers in a data center is called a cluster.

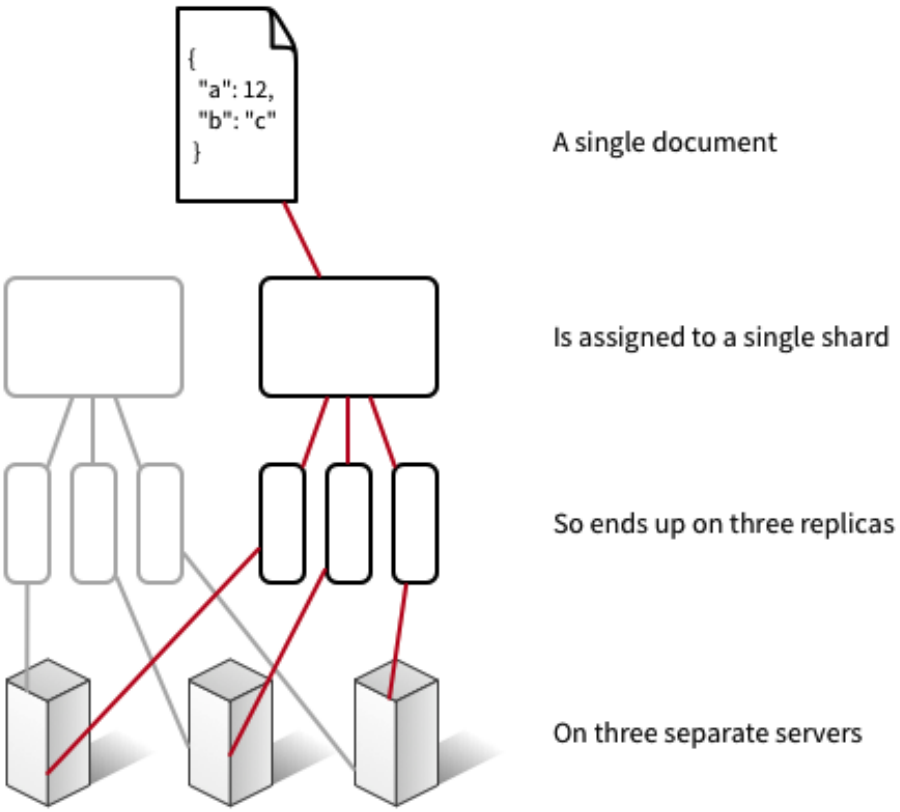


Figure 6: How data is stored in IBM Cloudant

- **Relational Database - IBM Cloud Databases for PostgreSQL**:
  A relational DBMS was chosen for all the Databases that contain structured data and on which many queries will be executed (Violation, Statistics). Postgres was picked over all database because it is a powerful, open source object-relational database with JSON support, that gives the best of both the SQL and NoSQL worlds. The advantages in using an IBM Cloud Service to deploy such service relies on the fact that it allows us to scale disk and RAM independently to best fit SafeStreets application requirements.
  IBM Cloud Databases provide out-of-the-box integration with IBM

Identity and Access Management that integrates completely into the SafeStreets architecture.

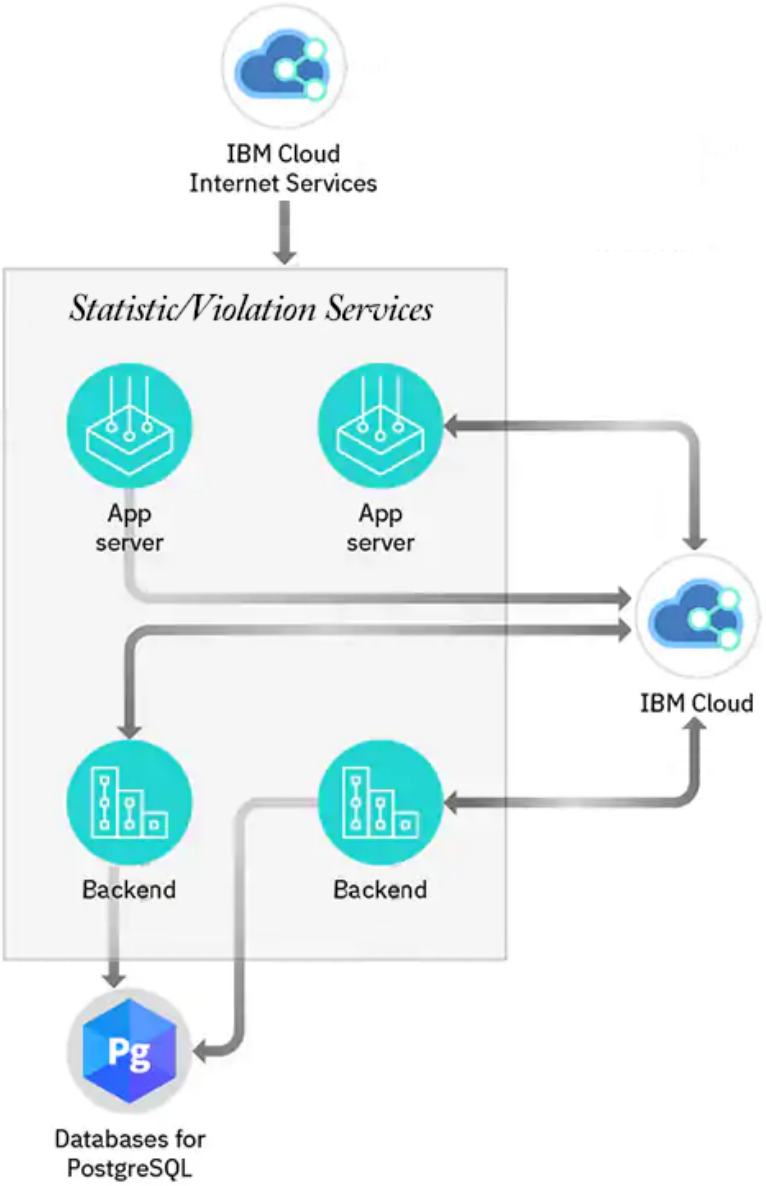


Figure 7: Scalable SafeStreets architecture to handle millions of users.

- **Container orchestration system - Kubernetes Service**: As SafeStreets popularity is unpredictable because no such service actually exists, the resource usage optimization is mandatory. Considered that all the SafeStreets microservices will run in a containerized environment, we will use Kubernetes to manage all their workloads across multiple hosts, and to offer management tools for automating, monitoring, and scaling such containerized apps with no manual intervention. In addition, using such architecture allows SafeStreets to reduce cluster downtime such as during master updates with highly available masters. Here we include an image that details all the components of the Kubernates architecture that it abstracts into our managed node.

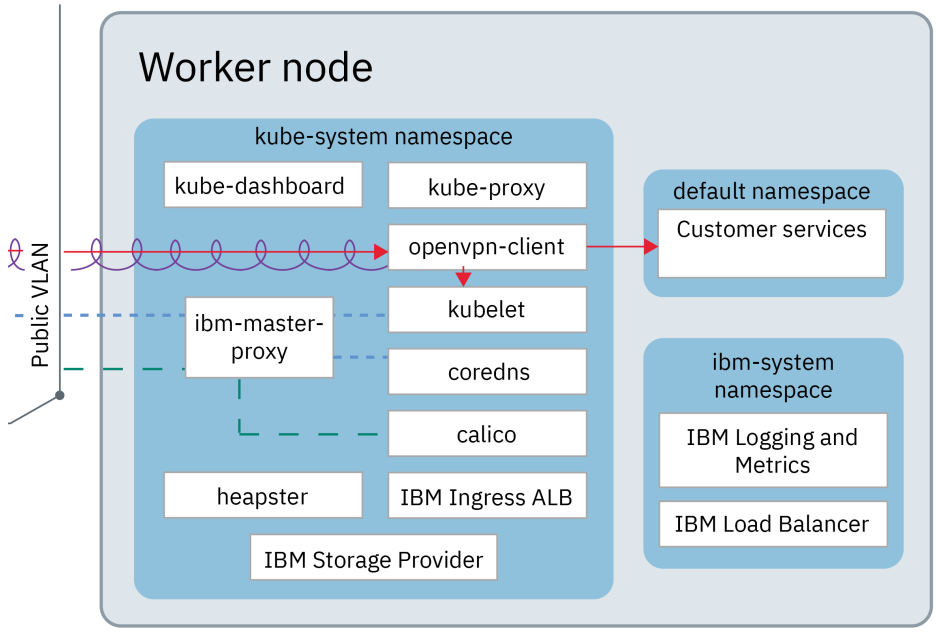


Figure 8: Kubernates SafeStreets managed node

Then, in the following page, we are sticking the full version of the Kubernates SafeStreets architecture, rotated to be readable even on paper documents.

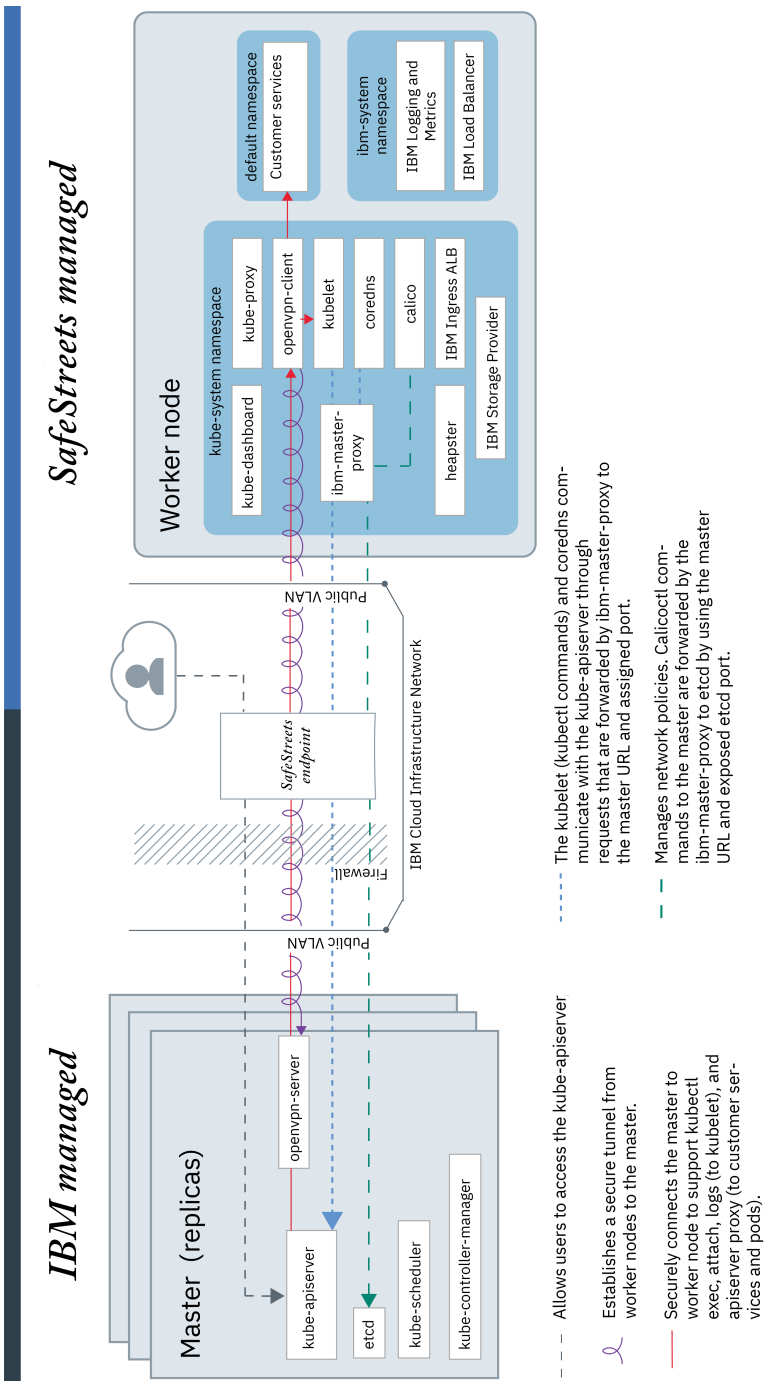


Figure 9: Kubernates Full SafeStreets Architecture

2.3.3 Our microservices

All of our custom microservices will be stored in the Cloud offered by IBM and every external service is picked from the IBM Cloud Platform.

- **Webserver**: The webserver is containerized in a Docker Apache container and orchestrated by the Kubernetes service by IBM. This composition allows to scale the server horizontally just by adding or removing containers based on the load on the server.
- **Cloud Object Storage**: All the SafeStreets' images will be stored on this component and will be accessible through a public URL. In addition, it allows us to receive metadata from the Computer Vision service that will be stored along with the image they belong to.
- **Violations and Tickets**: Receives violations from users and gives access to violations to officers and other microservices. This is our core service that consists in a proprietary web application exposing a REST API and connected to a managed PostgreSQL database.
- **Tickets Hashing system**: Stores the hashes of the violations with automatic ticket option enabled on a database that allows only inserts and reads. It provides an API for the *Violations and tickets* service to POST a ticket, of which it will calculate the hash, and for the police to send and hash for a ticket that will be compared with the one stored in its database.
- **Statistics**: This is a simple Python service that every day, when the load on the system is low, contacts the *Violations and tickets* service to grab the violations for that day and builds new statistics that stores inside its Cloudant database. It also provides an API for users and officers to query such statistics.
- **Suggestion mining**: It uses the power of the IBM services related to computer vision and data mining to identify unsafe areas and to provide suggestions on how to fix them. It consists of a Python application that queries and handle data from such services and provides an API for the local police to retrieve the suggestions found, which are stored in its database.

- **Tickets Checking**: Uses the data mining service to identify potentially incorrect tickets and, if it is sure enough, it requires an officer to manually approve it.
- **Other services** The computer vision, data mining and User management services are well described in other sections and do not have any components relevant here since they are pay-per-use services.

2.3.4 Database components

Having provided a detailed description of the types of DBMS services we will use to store data, we can proceed to explain in detail how they will be organized.

- **Violations Database - Relational PostgreSQL Cloud Database**: The Violations Database is the most important Database of the whole SafeStreets architecture. Inside it, it will be found all the reported infraction along with a link to the Object Storage where the pictures of each violation are. For this reason, there will be two read replicas of the PostgreSQL Violations DB to offload traffic from our leader database. As IBM Cloud Databases allows to scale disk and RAM to best fit the application requirements, it is not needed to state how much memory and disk space the database must have, it just needs to be deployed. After the deployment, all the microservice that needs to access data inside this database, will be given a connection string that allows read or read/write access depending on which service is.
- **Statistics Database - Relational PostgreSQL Cloud Database**: The Statistics Database where all the application usage statistics are. These are not so important data so for what concerns the replication, it will not physically duplicated by us, but we rely on the internal data replication of IBM.
  As already stated, IBM Cloud Databases allows to scale disk and RAM to best fit the application requirements, but in this case is defined a maximum number of resources to be allocated because the statistics page is not a SafeStreets core functionality. After the deployment, everyone can query this database trough an apposite microservice, but

the write access is restricted only to enabled services inside the IBM Cloud VPN.

- **Suggestion Database - IBM Cloudant**: This database represents the core heart of the memory that stores all the data that comes from the AI and Data Mining algorithms.
  Our system will be configured such as when it detects that is in a low-traffic moment, if it's in a time shift to be configured (late night) then it will start to mine the data of the past day to find possible suggestion to unsafe areas. When it finishes, the newly created suggestion will be stored in this database.
  As every suggestion comes from an Artificial Intelligence agents, it would be substantially different in structure from each other, and so it needs to be stored in a NoSQL database such as Cloudant. Figure 10 shows some possible documents that can be stored inside this database. This instance of Cloudant will be configured to accept read-only queries only from verified Public Police Officers.

```
{
  "infraction": "Illegal parking",
  "timeShift": 2019
  "area": "Via Bassini",
  "reportsCount": 1821,
  "ticketsGiven": 1437,
  "unsafeArea": true,
  "incidents": 24,
  "causedDeath": false,
  "suggestions": [
        {
          "suggestion": "Avoid illegal parking"
          "actions":
            [
                  {
                    "action": "Warn drivers with a flyer",
                    "score": 60.2
                  },
                  {
                    "action": "Make more tickets",
                    "score": 58.7
                  }
            ]
        },
        {
          "suggestion": "Make ZTL from 5PM"
          "actions":
            [
              {
                "action": "Install cameras",
                "score": 80.2
              }
            ]
        }
  ]
}
```


Figure 10: Example of suggestion document.

2.3.5 High-level component diagram

This section provides a high level diagram of the architecture of SafeStreet, listing all the microservices and how they are structured in the cloud.

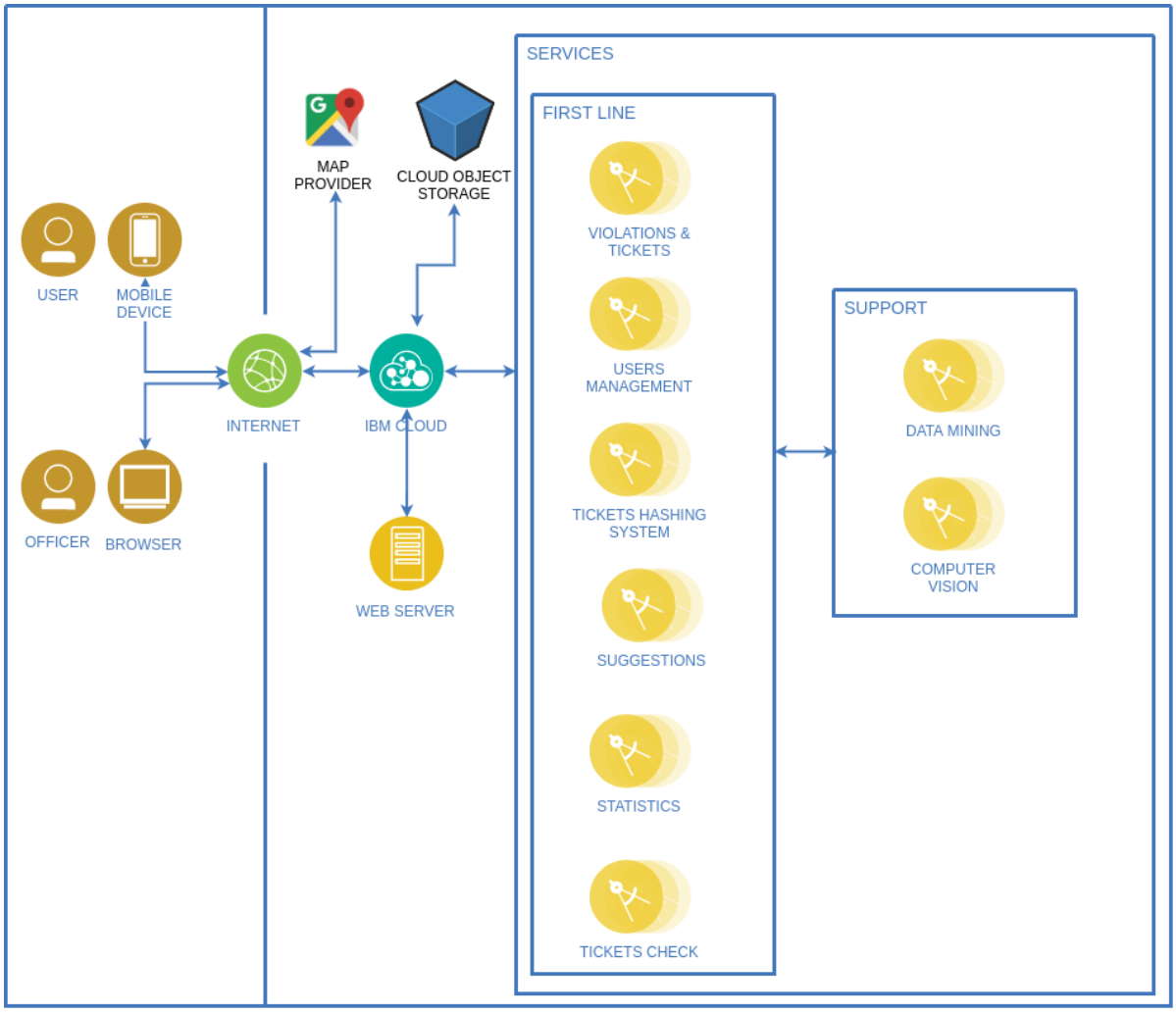


Figure 11: High level architecture

As show from the figure above, our architecture is composed of several microservices and a web server hosted inside the IBM cloud.

The service inside the box called *first line* are directly accessible for the

user, so they will expose an API through a public URL. Those inside the *support* box provide functions and data to the first line service and will not be exposed outside of the IBM network.

The only purpose of the web server is to serve the static webpages for the CMS system of the local police.

2.3.6 Low-level component diagram

In this section each component of the high level components diagram is analysed in more details, explaining its internal structure.

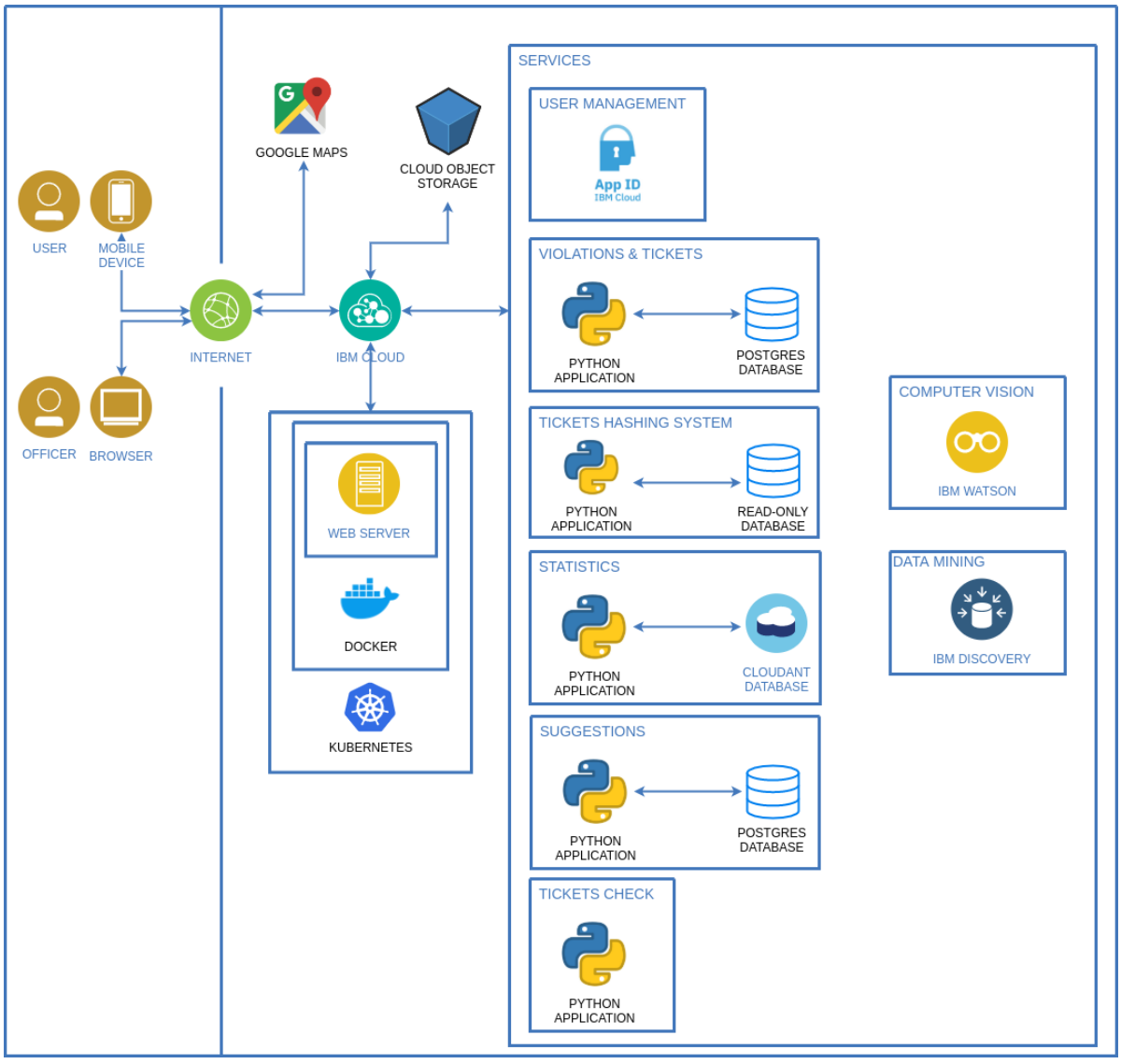


Figure 12: Low level components diagram

2.4 Deployment View

The following diagram explains how the application is deployed. Since most of the physical details are hidden behind the IBM services the diagram is custom but still quite straightforward.

Users connect through a smartphone application, while officers use a website that runs in all major browsers. Both the devices can expolit a cache system allowing them to save bandwidth.

The IBM cloud provides a firewall and DDoS protection system, it is included by default when using IBM services and does not need to be managed.

The web server shows 2 instances of the docker container in the graph, which is the starting point for the application. However, it is almost immediate to add new instances and IBM offers an automatic load balancer service for Kubernetes that works out of the box.

All the services inside the *services* box are able to talk to each other without exposing an URL to the network outside of IBM exploiting IBM connection mechanisms. In particular, those inside the green box *private connection* have a reserved connection that can hide them even from other IBM services. This schema is used to isolate databases, while the Python applications are exposed to other microservices and to the external world by means of an API.

The *App ID* service exposes an API to the outside world, while the *Data Mining* and *Computer Vision* services are only available inside the IBM network.

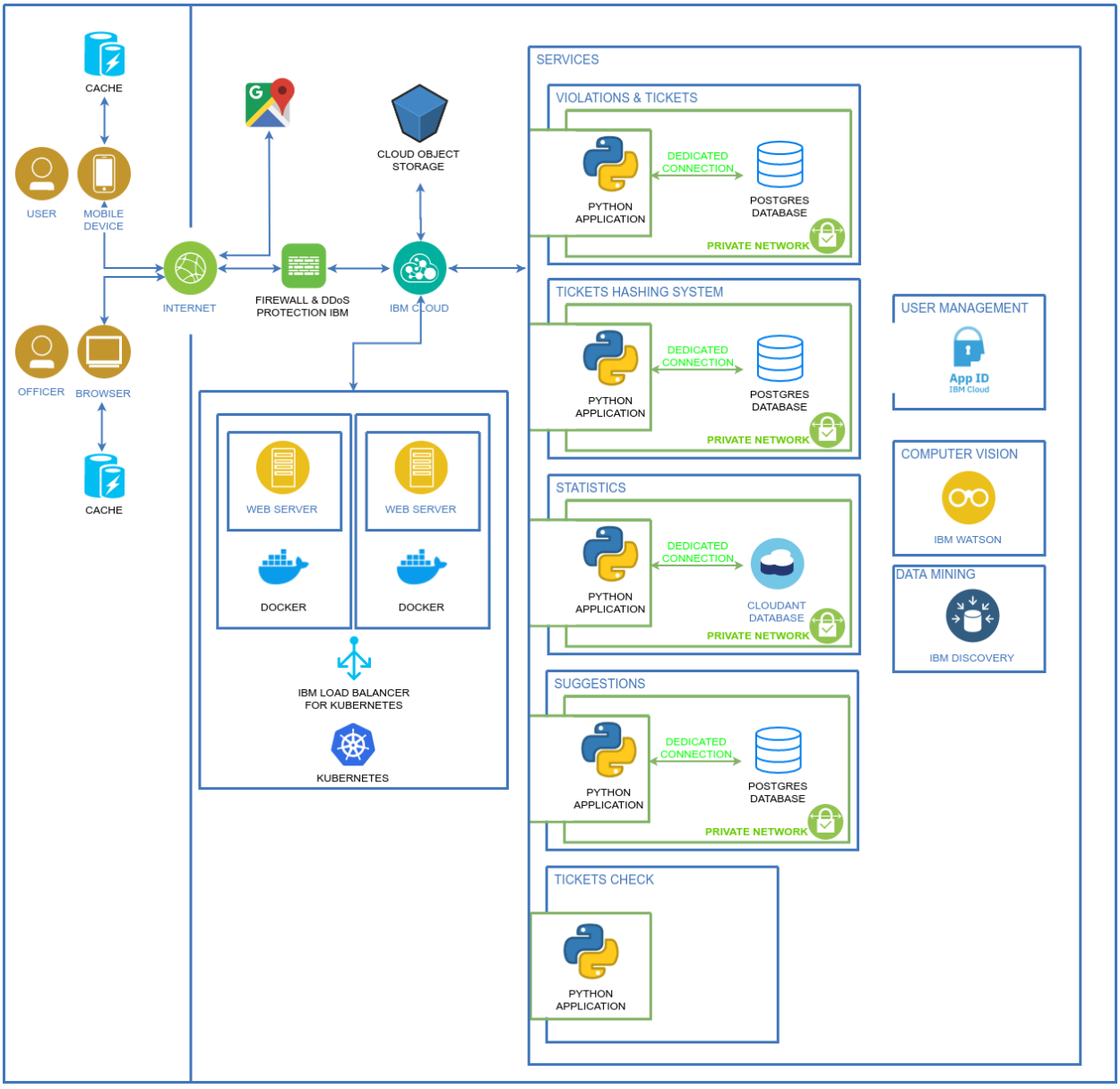


Figure 13: Detailed structure of the deployment

2.5 Runtime View

2.5.1 Signup/Login

This diagram shows how the app handles all the unauthorized requests. It works in the same way for all the requests that require authentication and for all the microservices.

When the user contacts the microservices without being logged in, he is redirected to the login/registrazion page, where he sends the login information directly to App ID, which responds with a token. The user can then use this token to identify itself with all the microservices, even if his path started from another one.

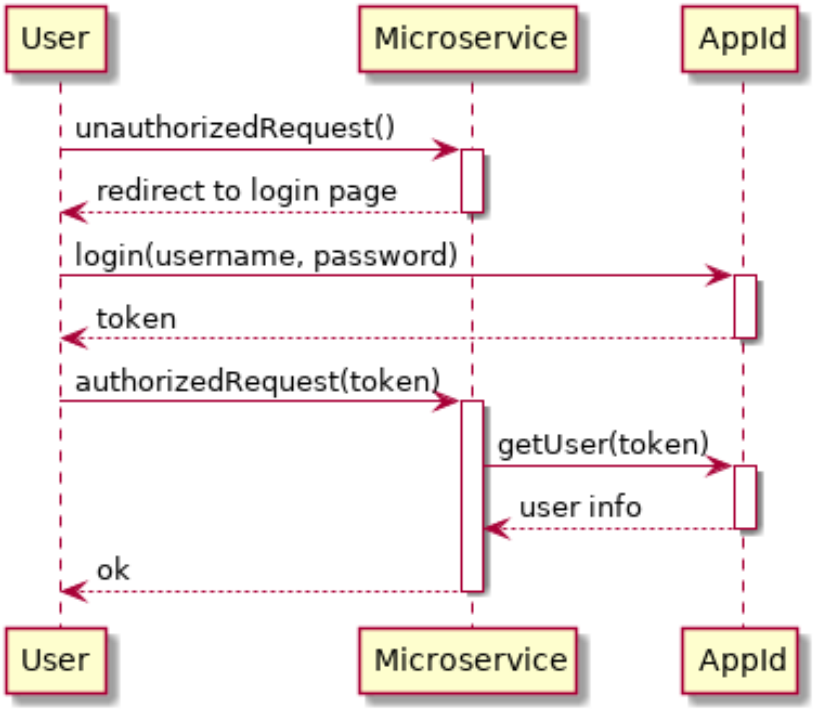


Figure 14: sequence diagram of the login process

From now on we will suppose that each request that requires authentica-

tion already has a token.

2.5.2 Create Violation

When a user submits a violation, the request is split into 2 parts: first, the pictures are stored in the Cloud Object Storage and their URLs are collected by the user client, then the text fields, along with the URLs, go directly to the violations microservice.

The pictures are sent to the computer vision service to retrieve metadata, identifying objects in the picture. In parallel, the data regarding the violation (vData) is stored inside the violations database. Once the metadata have been retrieved, the couple (vData, metadata) is sent to both the statistics and the data mining microservices.

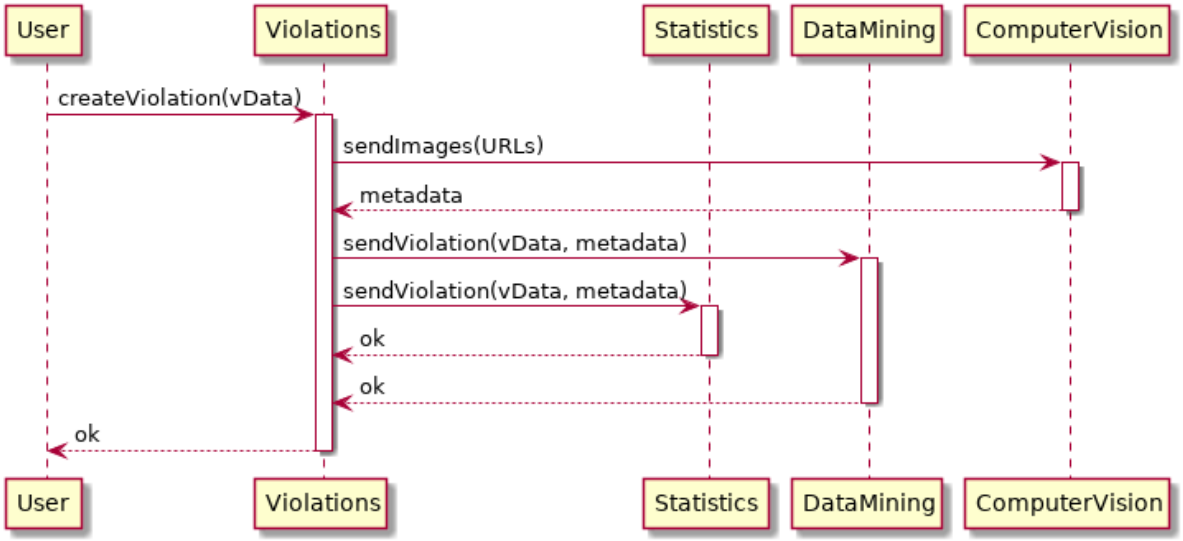


Figure 15: sequence diagram of the creation of a violation

The following diagram provides more details on how a violation is stored in the violation microservice: The images are sent directly to the cloud object storage which returns the URLs to the client which are sent along with the request. In the violations database only the URLs and the unprocessed data

are stored.

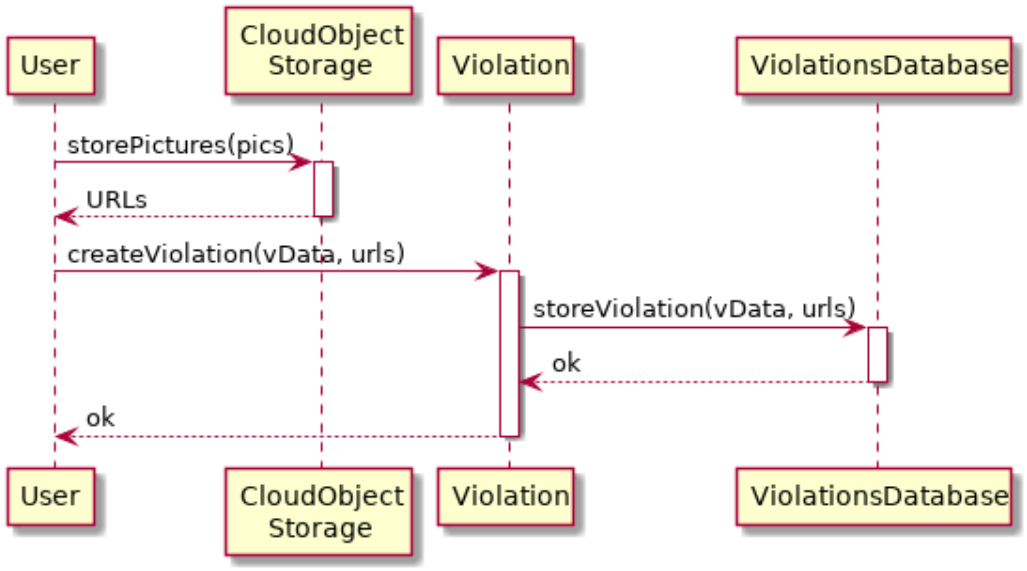


Figure 16: sequence diagram detail of the violations microservice

2.5.3 Automatic Ticket

An automatic ticket works like a violation, but 2 more controls are put in place:

- First, data about the ticket (pictures included) are sent to the Tickets Hashing System which calculates an hash and stores it into its database (which does not allow updates). When the police is sent to the police system for the automatic ticket, it must calculate the hash on the same data and check it against the one stored in the hashing system. Only if the hashes are equal the ticket is accepted.
- Also, the ticket information is sent to the Tickets Check service which uses the data mining service to do an automatic check of the ticket to control whether it is a legit ticket. If the system does not find anything

suspicious the ticket is automatically accepted, otherwise it must be controlled by an officer

All the other calls from the violation service, as they are in figure 14, are executed after the Tickets Hashing System confirms that the ticket has been succesfully stored.

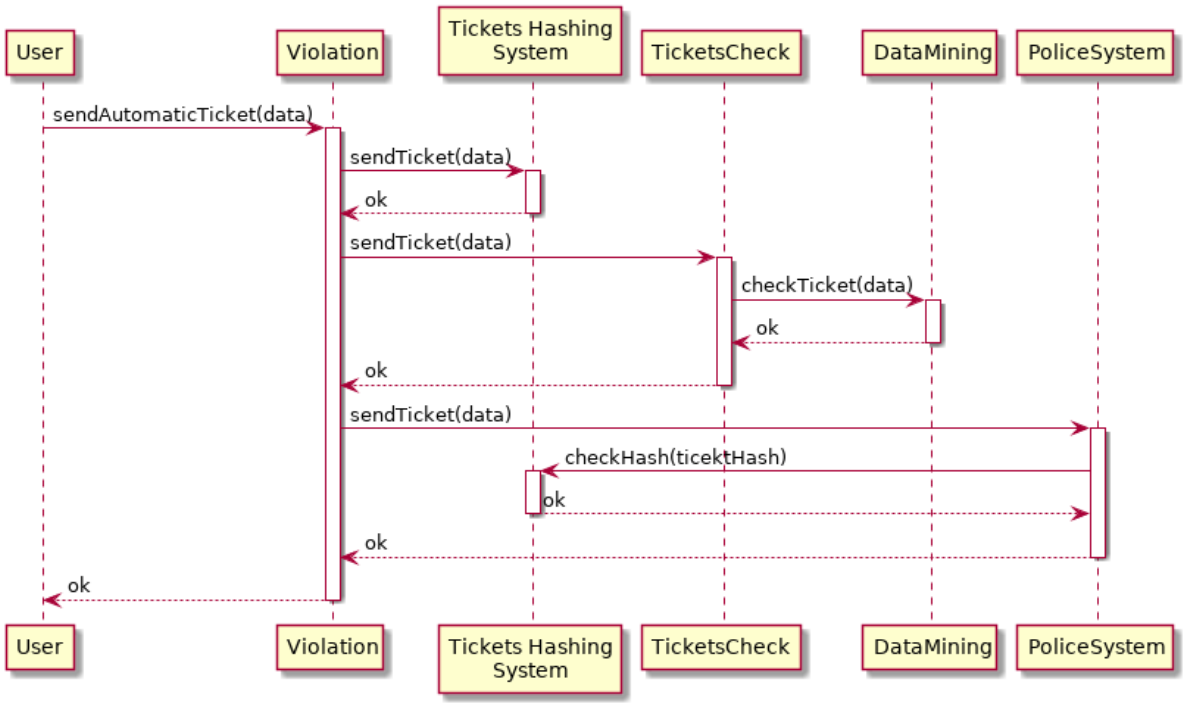


Figure 17: sequence diagram of a succesful ticket

More details on how the tickets hashing microservice works are in the following diagram:

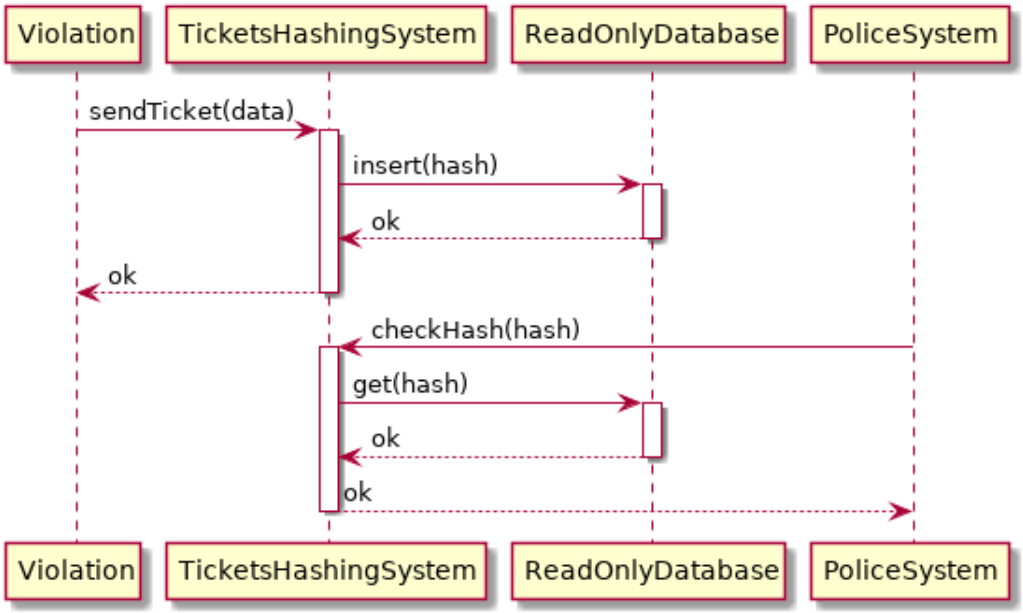


Figure 18: sequence diagram of the tickets hashing system

If the TicketsCheck service is sure enough that the ticket is invalid, the ticket must be first checked by an officer

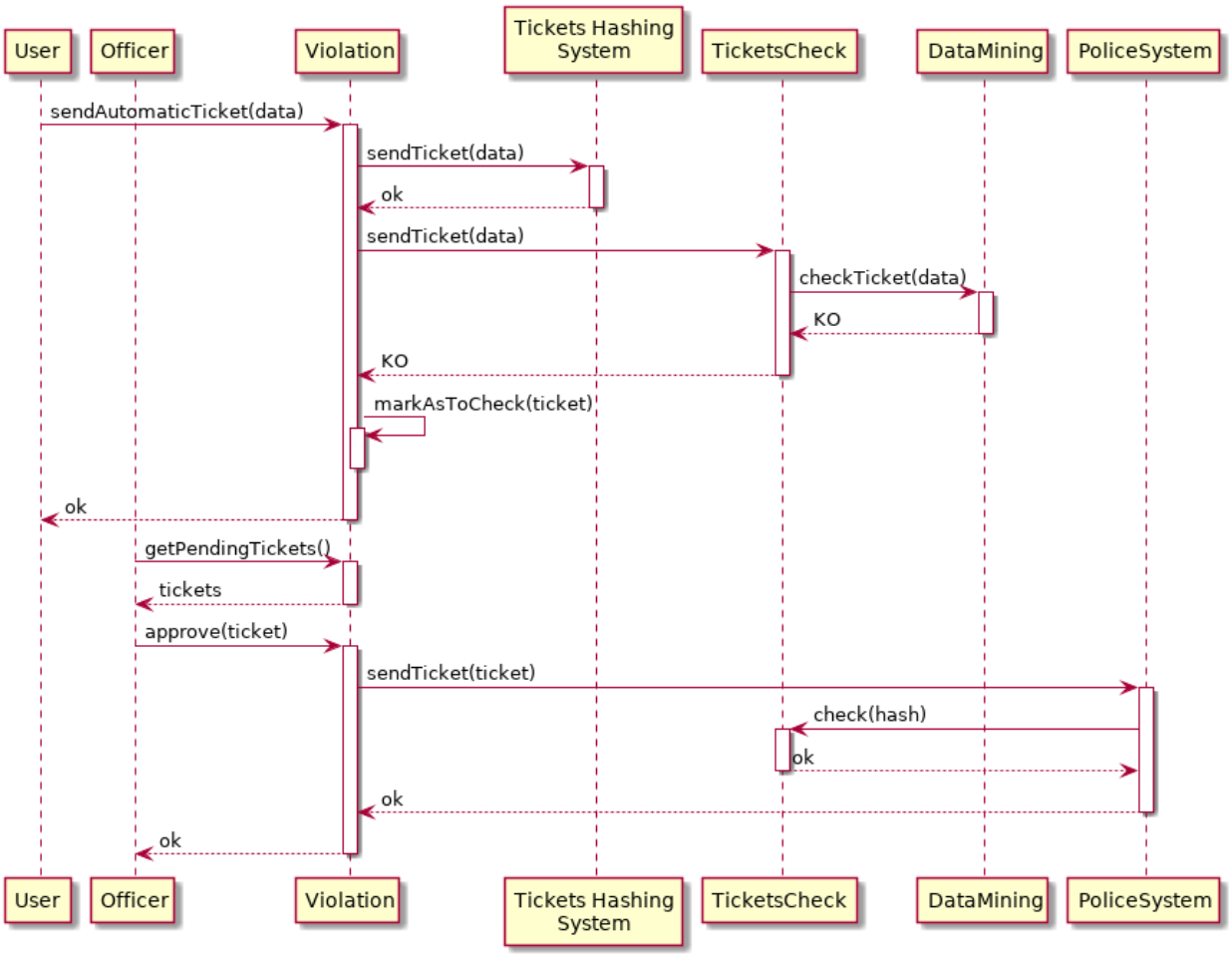


Figure 19: sequence diagram of a ticket that is initially rejected by the Data Mining

2.5.4 Suggestion System

Once a day, when the load on the system is low (each night at midnight), the suggestions service queries the data mining service to retrieve new suggestions that stores inside its database.

The officer can then query for new suggestions. The service queries its own database in which it had previously stored the suggestions and returns

them to the officer, that can handle and delete them one by one.

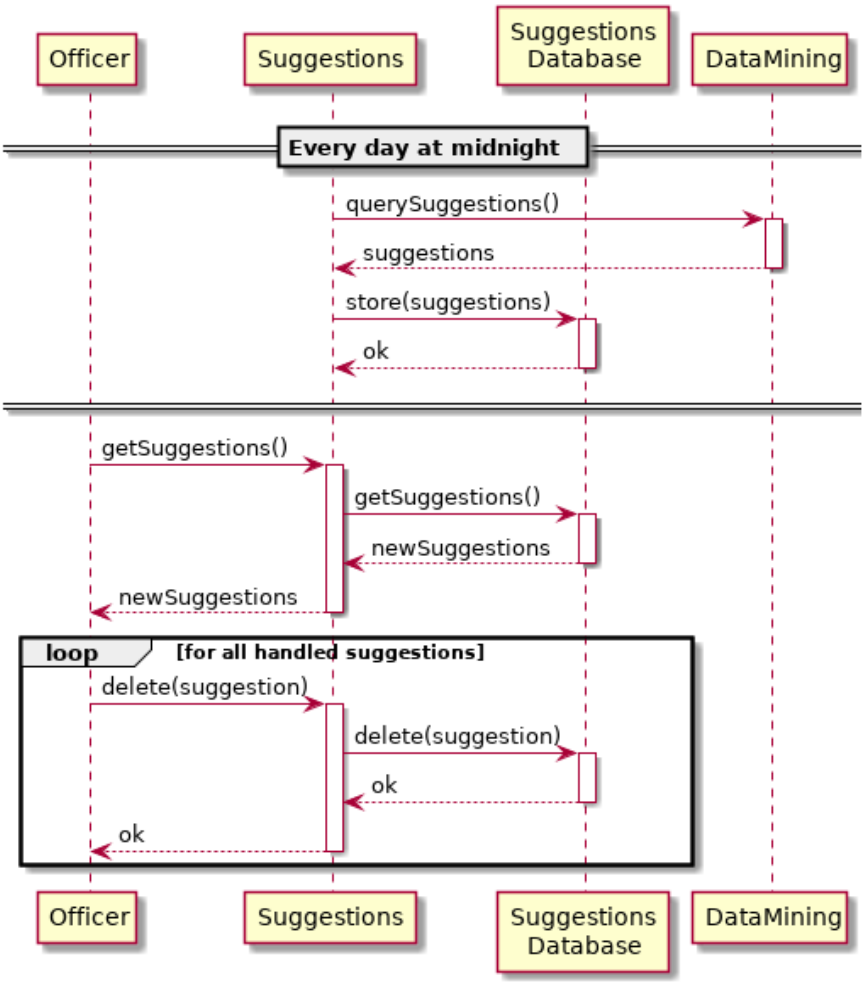


Figure 20: sequence diagram of the suggestions system

2.5.5 Statistics

Statistics are updated every night at midnight using the violations of the day. The application gets them from the db, computes the new statistics and stores them on server.

When a user queries for statistics, these are already built and can be returned directly with an easier and faster query.

In the following picture a particular query is executed but the process is the same for all the possibile statistics.

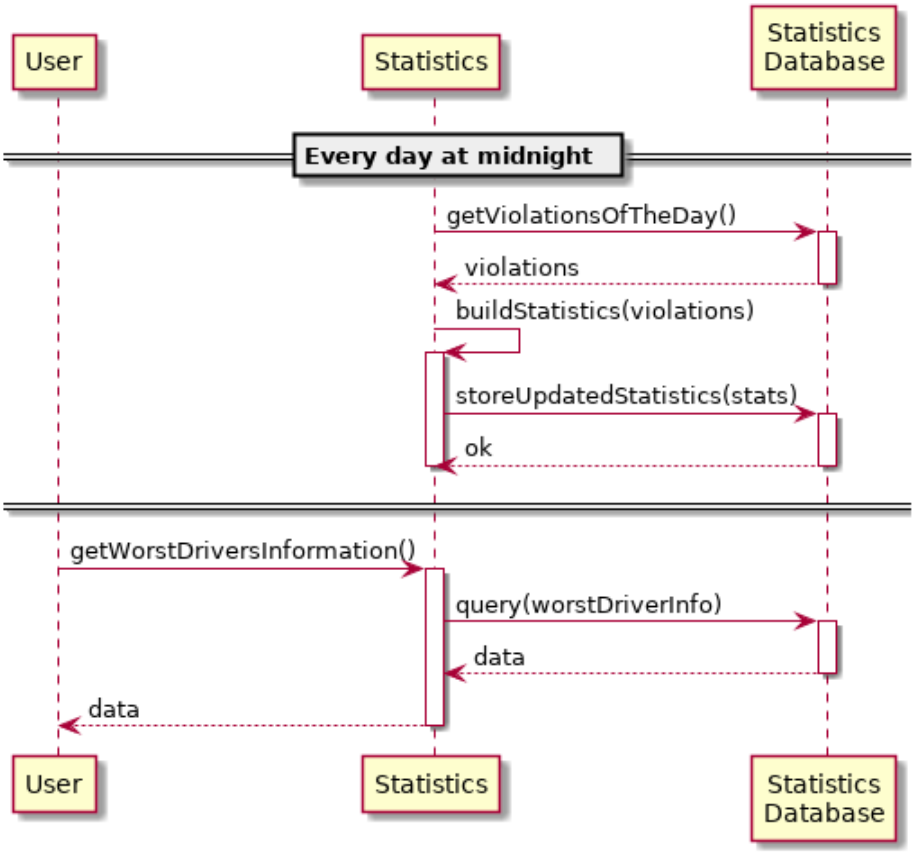


Figure 21: sequence diagram of the statistics microservice

2.6 Component Interfaces

This sections includes further details on the interfaces between the custom internal components of the system and the IBM micro-services illustrated before. The communications among components should take place only through APIs, some of them need to be custom developed and others are already exposed by the services described in 2.3.2.
A list of APIs needed for the application's well being are briefly described in the following list, grouped by functionality that interacts through them:

User Data Management:

- **[POST] changePassword(userId,newPassoword):** Used from the client application to change the password of its user.
- **[POST] banUser(userId):** Used from the officers portal in order to ban a malicious user of the application.

the *login* and *register* interfaces are not described here since they are provided by the IBM App ID service.

Violations:

- **[POST] createViolation(vData, URLs):** Used from the client application when a user wants to send a new ticket violation. The URLs of the picture are attached and point to objects in the Object Storage System.
- **[POST] sendAutomaticTicket(data, URLs):** Used when the user wants to send an automatic ticket. The URLs of the picture are attached and point to objects in the Object Storage System.
- **[GET] getViolationById(violationId):** Used from the officers portal to get the information of a specific violation.
- **[GET] getViolationsByUser(userId):** Used from the client application to get all the user's violations.

- **[GET] getPendingTickets():** Used from the officers portal to get all the tickets that are yet to be processed.
- **[GET] getPendingViolations():** Used from the officers portal to get all the violations that are yet to be processed.
- **[POST] approve(ticket):** Used from the officers portal to approve a ticket that was initially marked as invalid from the Tickets Check service.
- **[POST] changeViolationState(violationId,newState):** Used from the officers portal to change the status of a violation just analyzed from PENDING to ACCEPTED/DENIED.
- **[GET] getPositionWithGPS(x-coordinates,y-coordinates):** Used from the client application to get the user's position given their GPS position.

Data Mining:

- **[GET] getLicensePlate(images,oldLicensePlates):** Used from the client application to get a license plate from the images provided if found. The API should also accept a set of wrong license plate in case the precedent output of the API call was wrong and the user realized that the license plate given didn't correspond to the actual license plate of the vehicle committing the violation.
- **[POST] checkTicket(data):** Used by the Tickets checking service to check if the ticket might no be legit.
- **[GET] querySuggestions():** Used by the Suggestions service to query for new suggestions.
- **[POST] sendViolation(vData, metadata):** Used by the Violations service to send new violations into the data mining engine

Tickets Check:

- **[POST] sendTicket(data):** Called when the violation service receives an automatic ticket, is used to check if the ticket might not be legit.

Tickets Hashing System:

- **[POST] sendTicket(data):** Called when the violation service receives an automatic ticket, is used to store the hash into the microservice.
- **[GET] checkHash(hash):** Used from the officers portal to check the integrity of a specific violation (chain of trust).

Statistics:

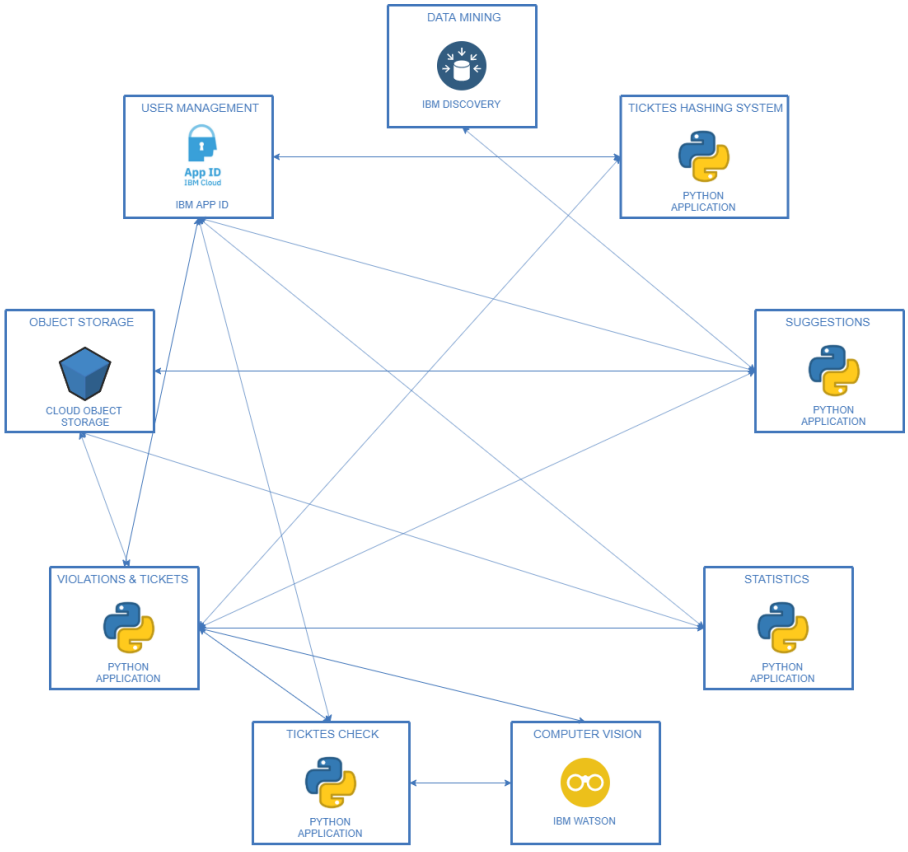
- **[GET] getAreaInformation(area):** Used from both the client application and the officers portal to retrieve the information of an area that may be unsafe (both) or not (only officers).
- **[GET] getWorstDriversInformation():** Used from both the client application and the officers portal to retrieve the information about the worst drivers in the SafeStreets database. The officers should be able to acknowledge more information of the drivers than the users.
- **[GET] getData(bounds):** Used from both the client application and the officers portal to get the information of a specific thing given some proper bounds.
- **[GET] getStatisticsOverview:** Used from the client application so that the user could have a more general idea when he accesses the statistic section.
- **[POST] sendViolation(vData, metadata):** Used by the Violations service to send new violations into the statistics database

Suggestions:

- **[GET] getSuggestions():** Used by the officers to get new suggestions.
- **[POST] delete(suggestion):** Used by the officers to delete a managed suggestion.

Of course, the mentioned above APIs are not to be intended as strict and not-modifiable, but there can be added, removed or edited APIs to/from the list during the implementation of the application if there is the need to use a different approach.

In order to provide the correct application functioning, it is trivial to say that all the components needs to expose APIs that can allow them to be in communication with each others.



2.7 Selected architectural styles and patterns

2.7.1 Microservices

In a microservices architecture the system is divided into several autonomous services, each one is self-contained and implements a single business capability.
All the functionalities provided by the microservices to the final users are exposed through a REST API, while the internal communications service is provided, in our case, by IBM itself.

Some of the main advantages offered by this architecture are:

- **Decoupling** - This allows to easily build, scale and alter services indepentently, which is particularly important for SafeStreets since it is difficult to foresee how it will scale.
- **Autonomy** - Developers can work indepentently on different services without complicated merges or the need to communicate too much, thus increasing speed.
- **Responsibility** - A single microservice does not focus on the entire application but on a single component that is handled as a product for which it is responsible
- **Decentralized Governance** - Each service can be developed with its own application stack that is the best for it, so you are not stuck with a single technology just because you started the project with it.
- **Agility** - Microservices support agile development. Any new feature can be quickly developed and discarded again.

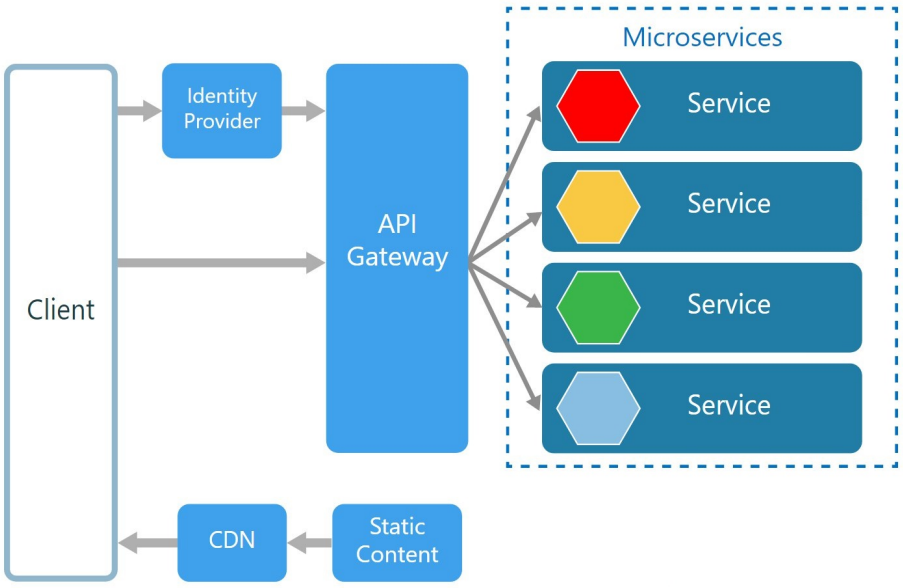


Figure 22: General scheme for a microservices architecture

2.7.2 Client - Server

The most common pattern used on the web, it consists of several clients (mainly smartphones and computers) that contact a server which is always available to answer to requests and is reachable through its public IP.
In our case the concept of server is abstract because the application is not provided by a single, monolithic server but most principles remain.

2.7.3 REST API

REST stands for Representational State Transfer, is an architectural style for designing network application. It is the most obvious choice to use with microservices if scalability is required because it forces statelessness.
REST relies on a client-server architecture and uses the HTTP protocol.

3 User Interface Design

3.1 UX Diagram

In order to show the navigation that the user can experience, moving into different pages of the application, a visual flow of screens that explain the User Experience is provided.

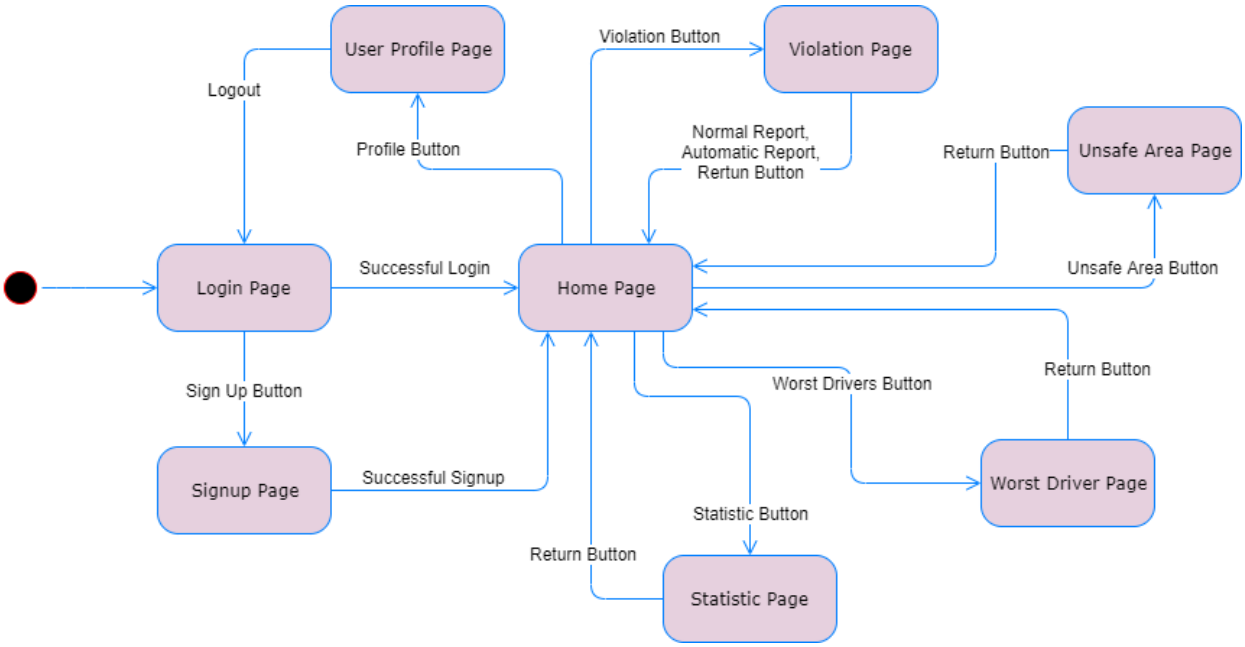


Figure 23: The UX diagram of the client application

Since the navigation of the officers is different than the users' one, a different User Experience graph, the one that involves officer, more simple and dynamic is provided.

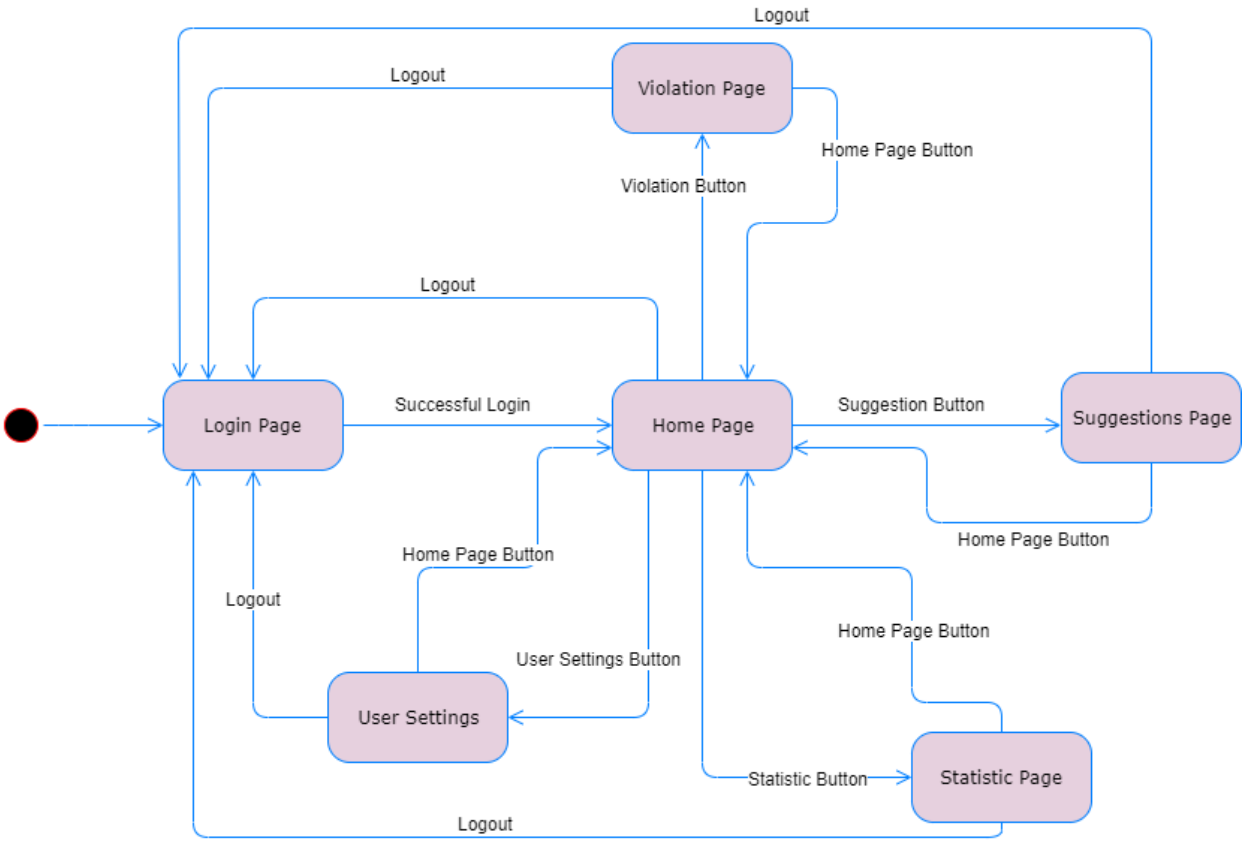


Figure 24: The UX diagram of the officers portal

3.2 Mockups of the User and Officers Interfaces

Users Login Page:
Since logging into SafeStreets is mandatory, this is the first page that the user will face when the app does not recognizes him. Of course, cookies or sessions could avoid to force the user to login every time.

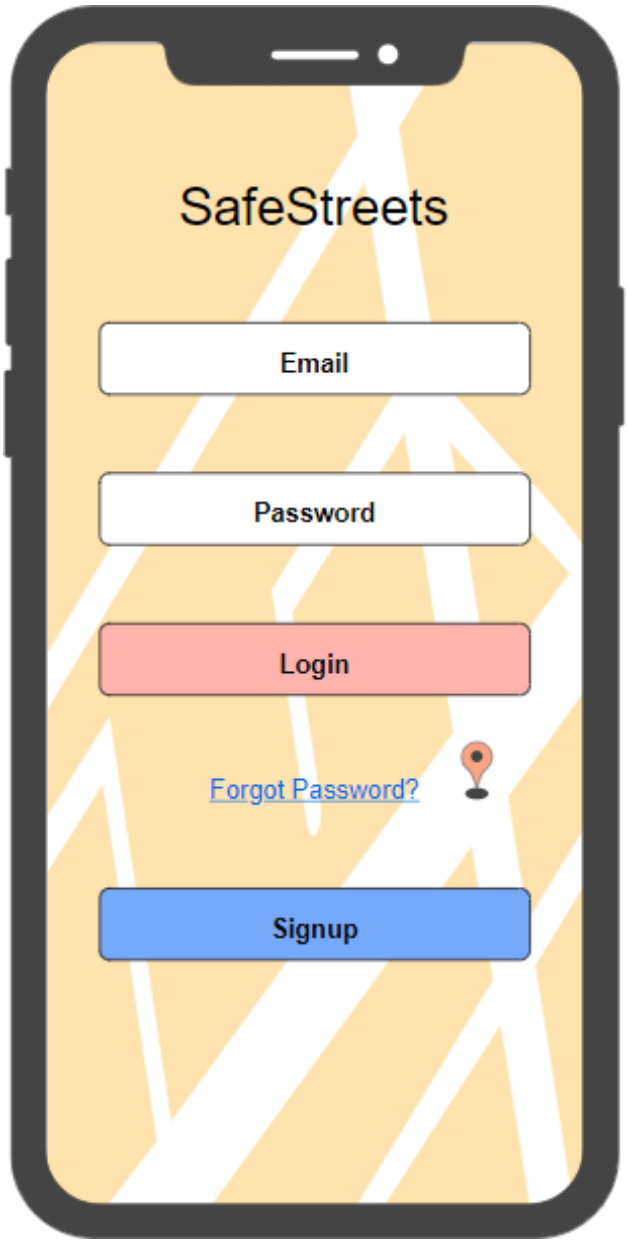


Figure 25: SafeStreets Login page.

Officers Login Page:
Since logging into SafeStreets is mandatory, this is the first page that the officer will face when the portal does not recognizes him. Of course, cookies or sessions could avoid to force the officer to login every time. A Sign-up page is not expected since an officer charged with SafeStreets jobs should also receive an account.

Figure 26: SafeStreets Login page.

Users SignUp Page:
Since logging into SafeStreets is mandatory, if the user does not have an account he needs to make one. Every field except the photo should be mandatory.

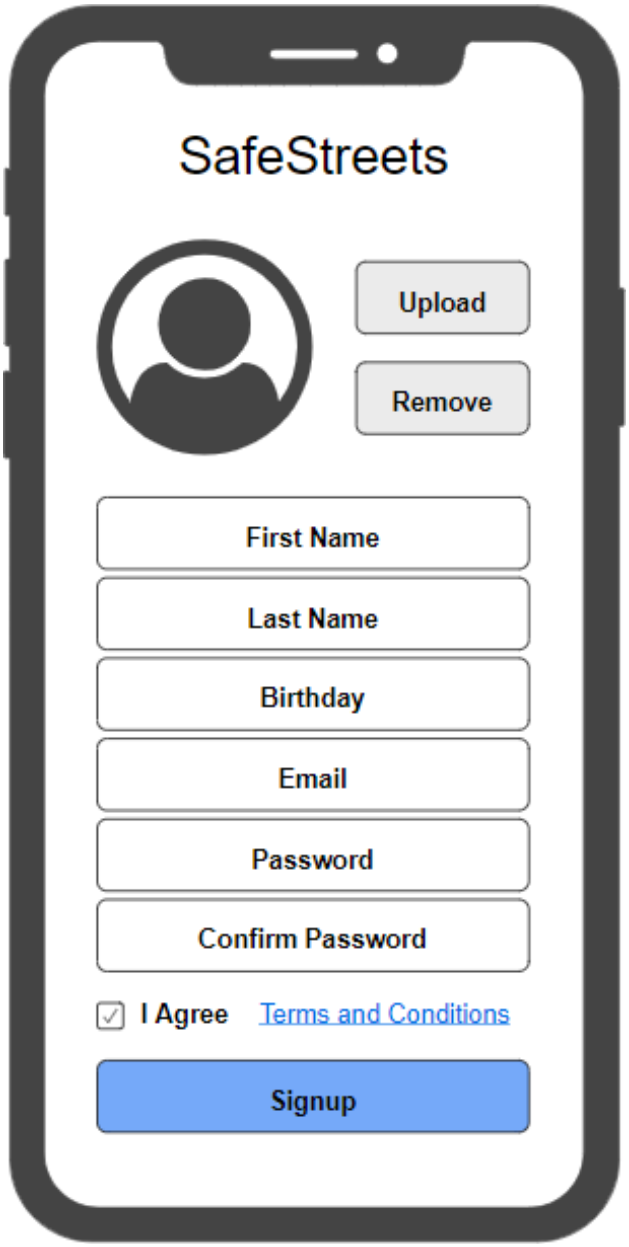


Figure 27: SafeStreets SignUp page.

Users Home Page:
When the user logs into SafeStreets successfully, an home page should be provided with the options for the user to use every functionality of the application.

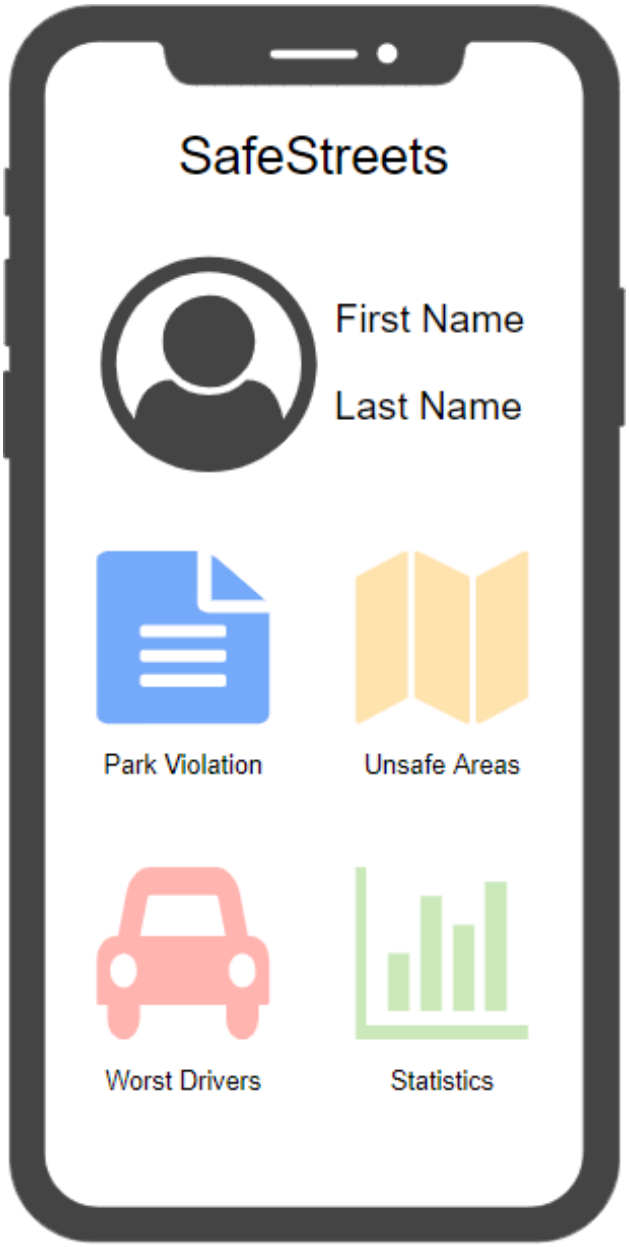


Figure 28: SafeStreets Home page.

Officers Home Page:
When the officer logs into SafeStreets successfully, an home page should be provided with the options for the officer to use every functionality of the application.

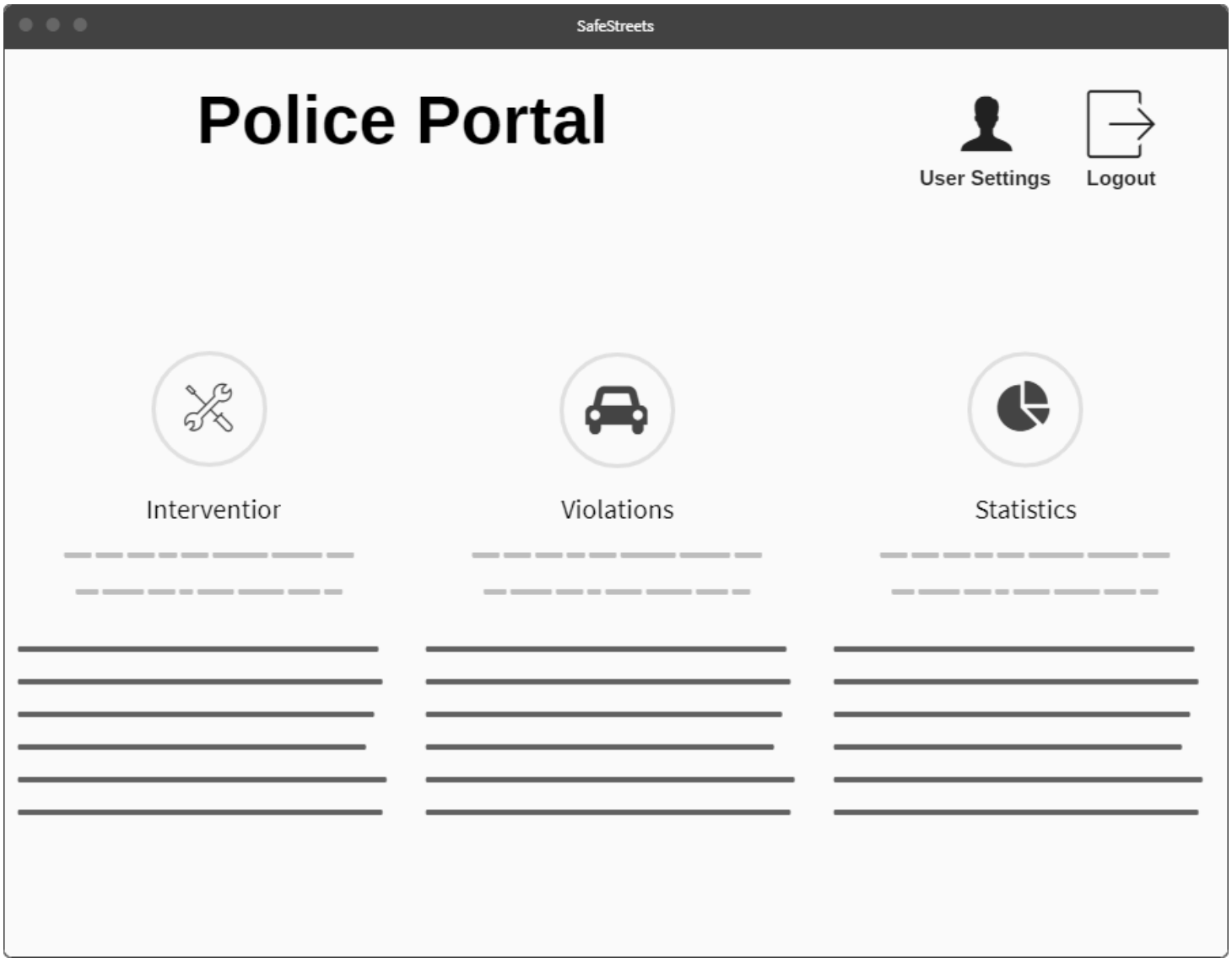


Figure 29: SafeStreets Home page.

Users Violation Page:
If the user wants to send to the officers a parking violation of some sorts a form to be filled is provided. The user can either send a report that will generate an automatic ticket or that will need to be checked by the officers.

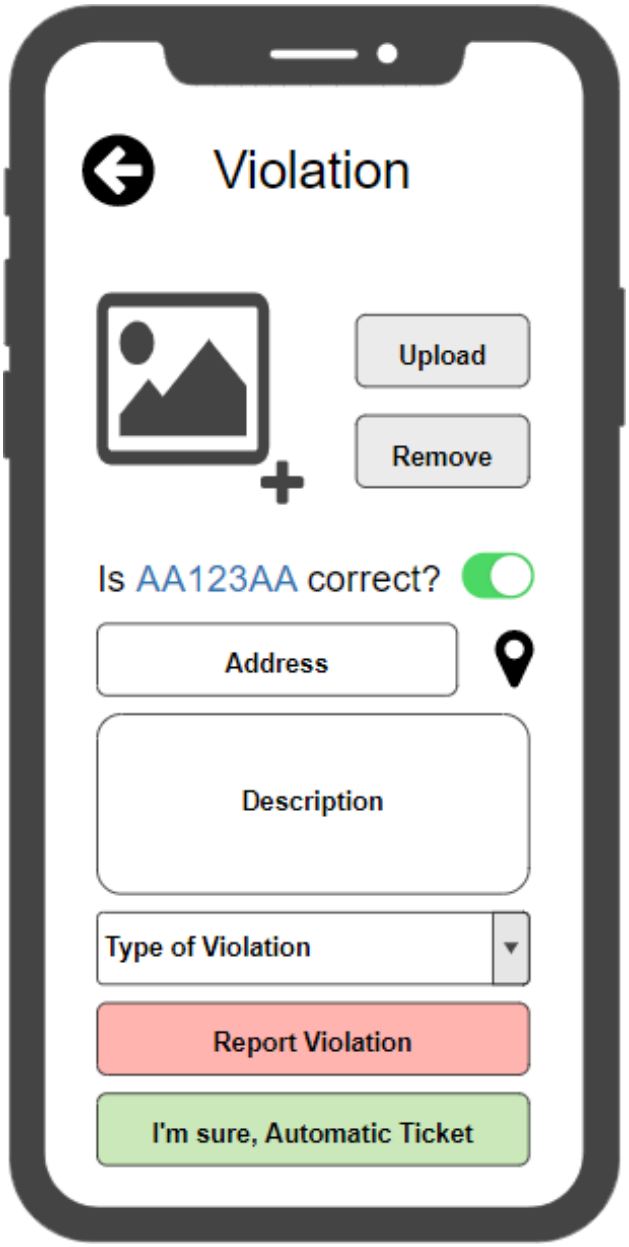


Figure 30: SafeStreets Violation page.

Officers Violation Page:
If the officer needs to check the tickets that the users sent to the portal an interface to handle them is provided. The officer can either abort the violation if it is not considered valid, or file the ticket if the ticket is enough for a fine.

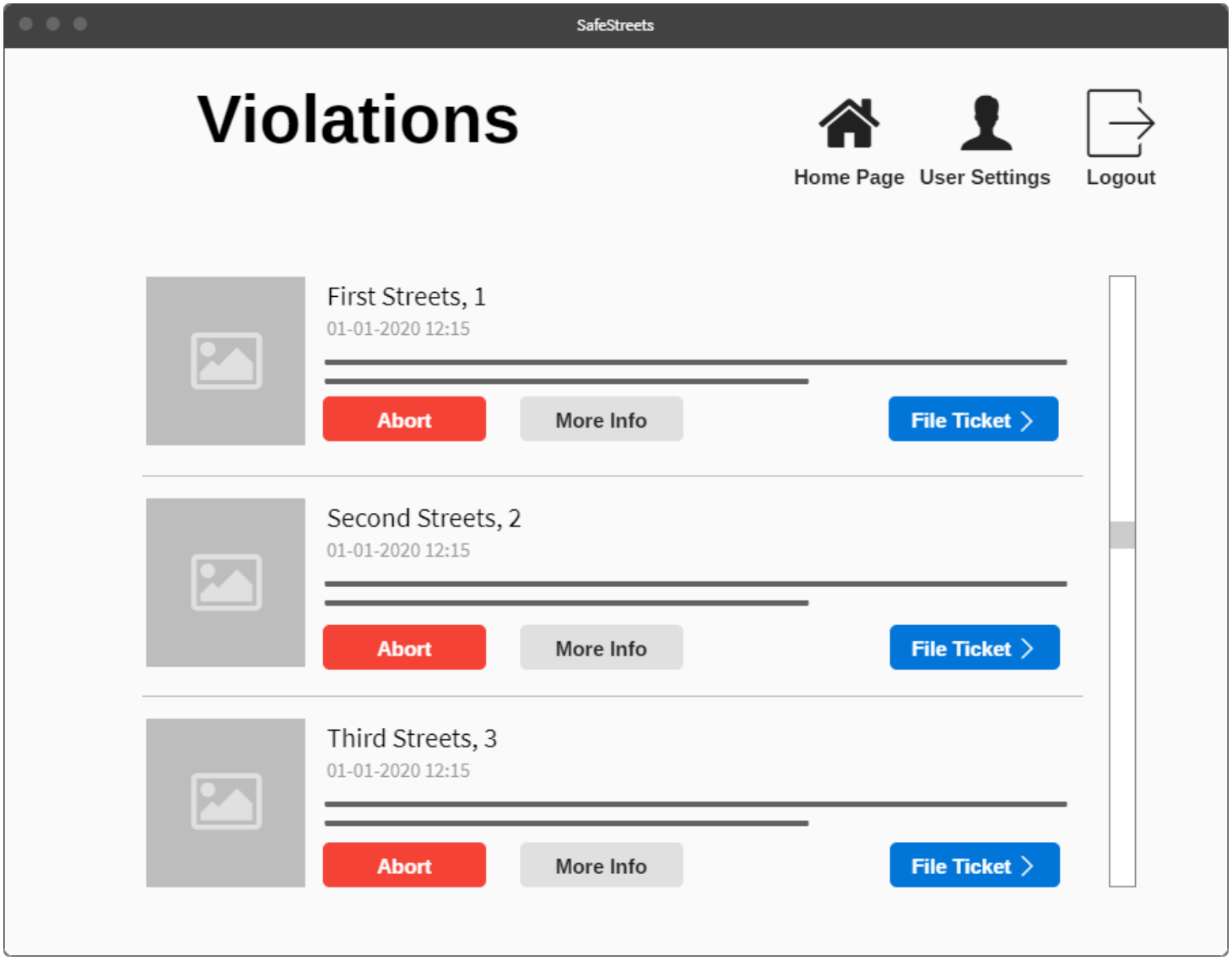


Figure 31: SafeStreets Violation page.

Users Unsafe Areas Page:
If the user wants to see which areas are the most subject to violations, an interface to search among all areas should be implemented.

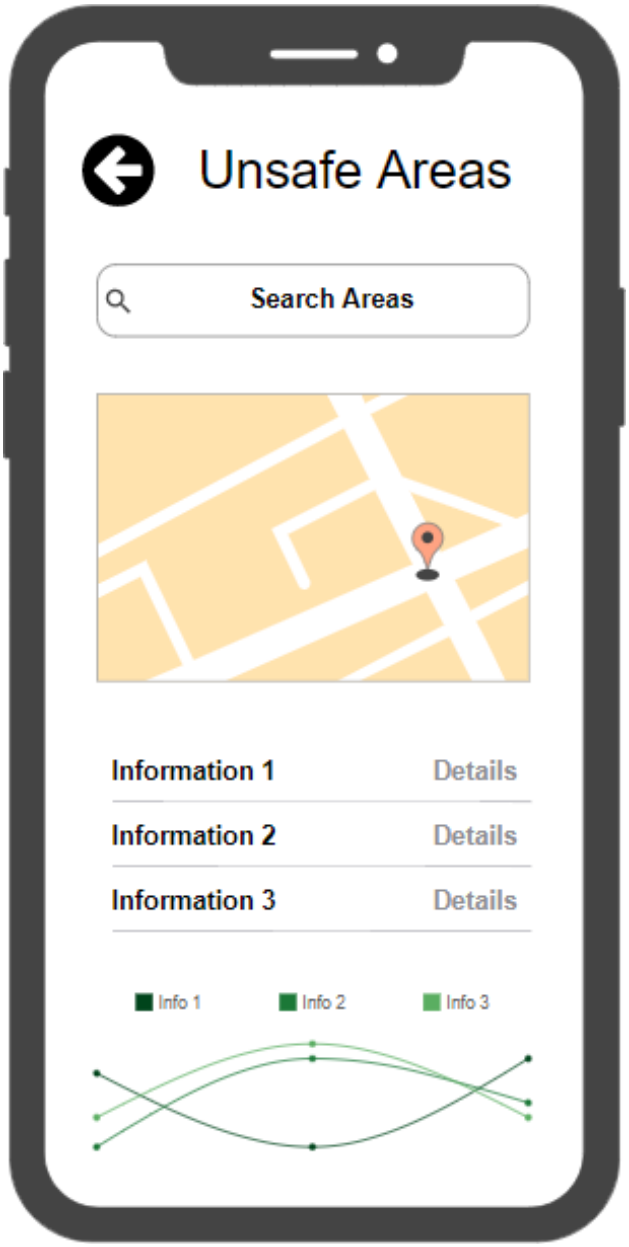


Figure 32: SafeStreets Unsafe Areas page.

Users Worst Drivers Page:
If the user wants to see which vehicles tends to not follow the city rules, an interface that shows this needs to be present in the application.

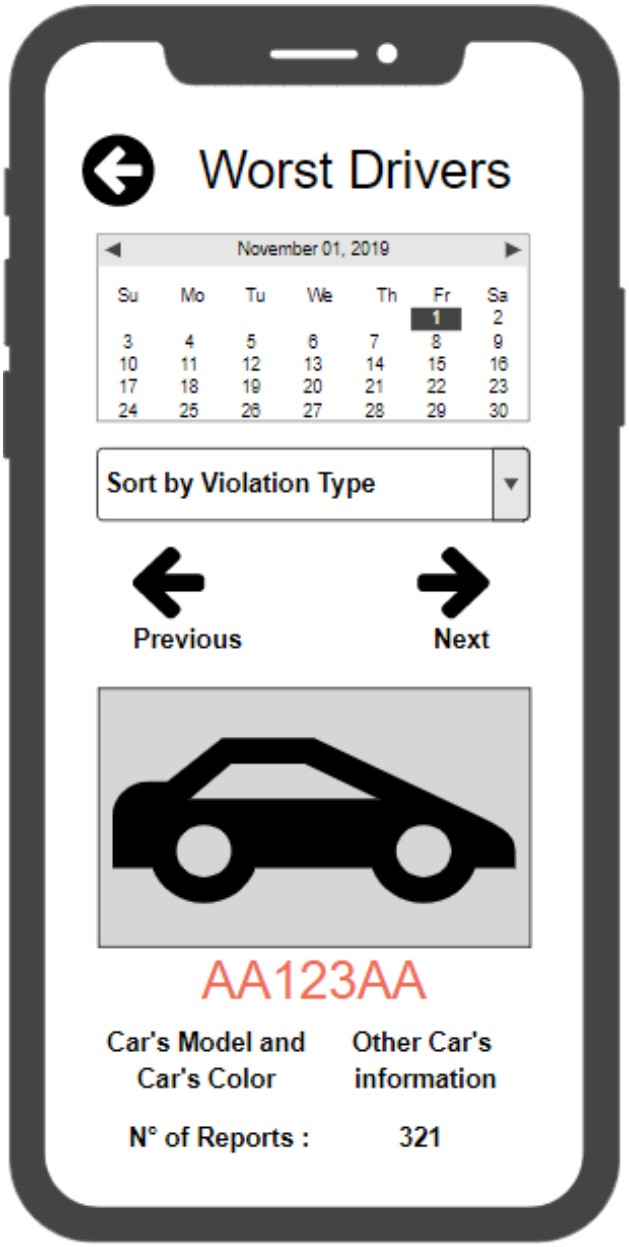


Figure 33: SafeStreets Worst Drivers page.

Officers Suggestions Page:
If the officer needs to check possible suggestions and interventions for a specific area that the data mining procedures have enlighten, a page that enable this is provided. The officer can given a map select the spots that have a possible intervention to apply and decide if it is a valid or not suggestion.

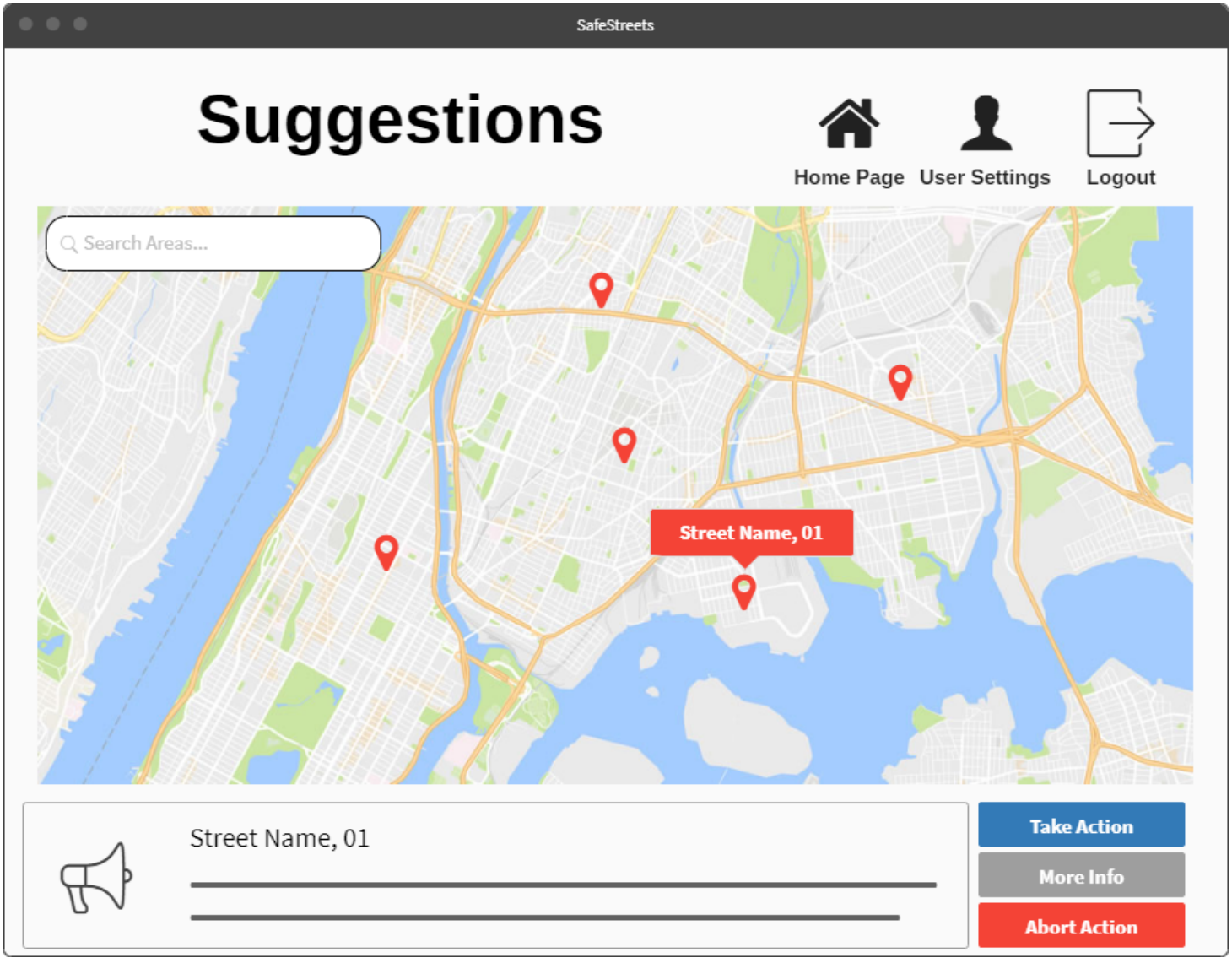


Figure 34: SafeStreets Statistics page.

Users Statistics Page:
A complete page that exhibits all of SafeStreets data in a complete and detailed manner, that also enlightens SafeStreets effectiveness.

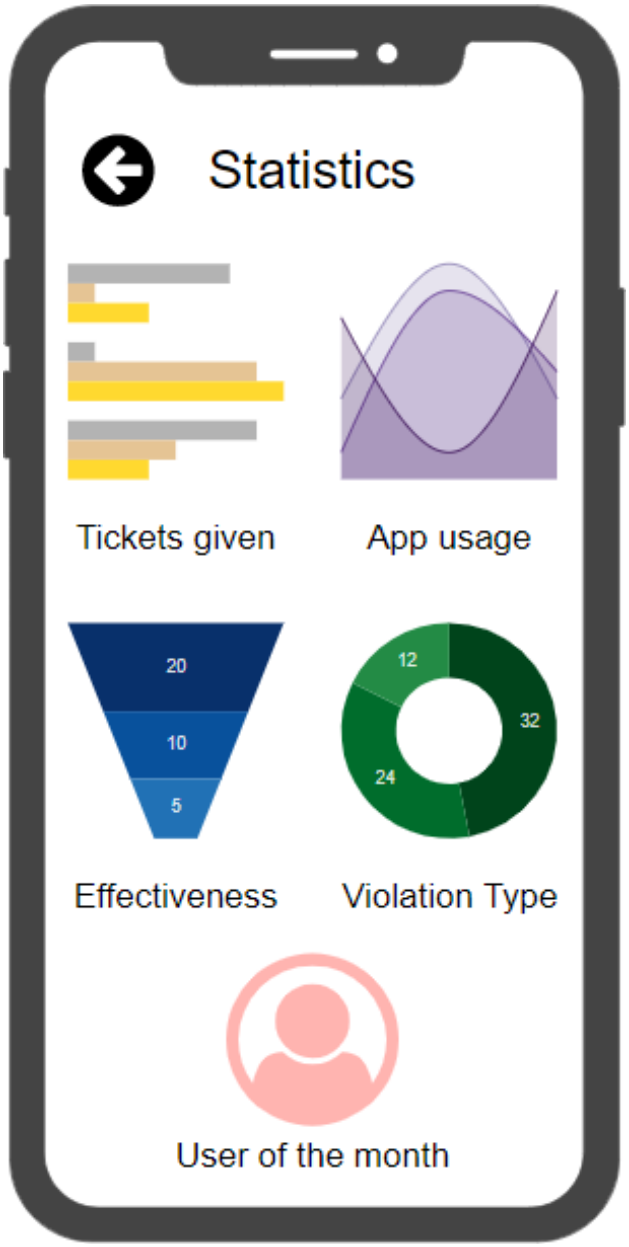


Figure 35: SafeStreets Statistics page.

Officers Statistics Page:
A complete page that exhibits all of SafeStreets data in a complete and detailed manner, that also enlightens SafeStreets effectiveness. Within this page the officers have access to the unsafe areas and worst drivers information, as long as more sensitive data such as the detailed information of the owners of the worst car users.

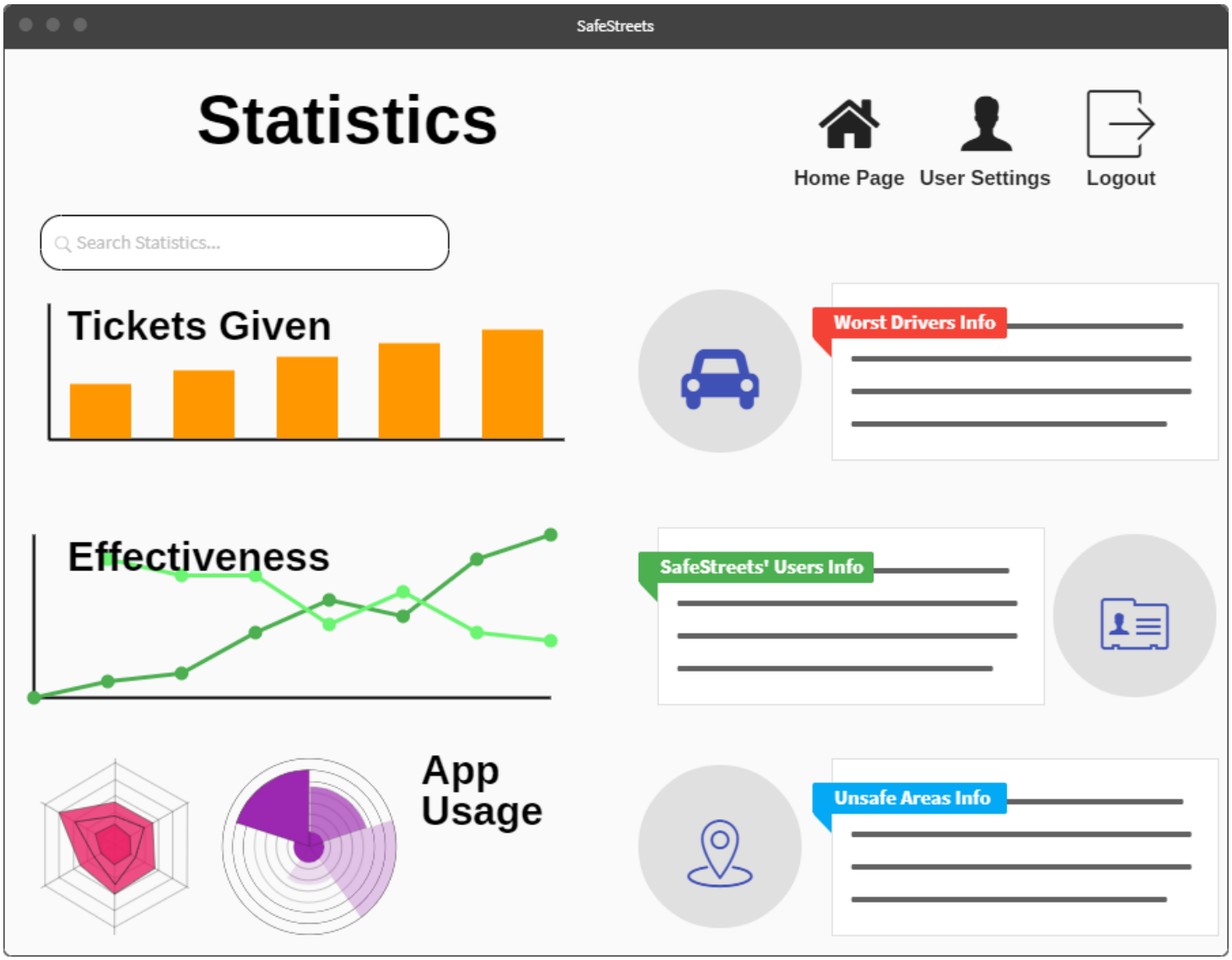


Figure 36: SafeStreets Statistics page.

User Profile Page:
If the user wants to change its profile picture or his password, if he wants to check his reports or to logout, he should be able to do all these things.

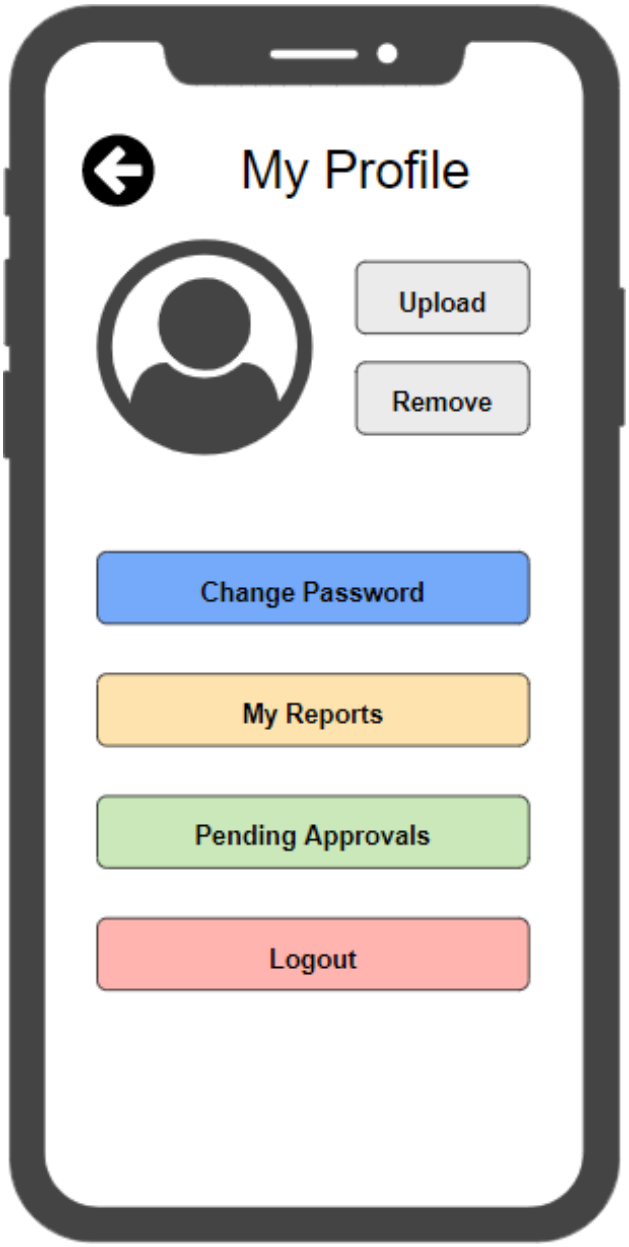


Figure 37: SafeStreets User Profile page.

4 Requirements Traceability

In order to have a simplified and schematic view over the designed components that are going to satisfy the requirements and provide the functionalities which were identified in the RASD we define a formal mapping that explicitely correlates the Goals and functions with one or more Components that should be able to achieve the goal and create the functionality.

Goal	Functions	Components
[G1]: Allowing users to notify officers when particular parking violations occur.	**[F1]:**Notification of Violations	**[M2:]**IBM Watson Visual Recognition **[M4:]**IBM Cloud Object Storage
[G2]: Permitting both users and officers to learn which are the most unsafe areas in their jurisdiction.	**[F2]:**Data Mining	**[M3:]**IBM Discovery **[M6:]**IBM Cloud Databases for PostgreSQL
[G3]: Suggesting possible interventions to potentially unsafe areas in the city.	**[F3]:**Request for interventions	**[M3:]**IBM Discovery **[M5:]**IBM Cloudant
[G4]: Allowing the authorities to generate automatic parking violation tickets from users reports.	**[F4]:**Automatic Tickets	**[M2:]**IBM Watson Visual Recognition **[M3:]**IBM Discovery
[G5]: Permitting both users and officers to learn which vehicles commit the most violations.	**[F2]:**Data Mining **[F5]:**Statistics	**[M3:]**IBM Discovery **[M6:]**IBM Cloud Databases for PostgreSQL
[G6]: Permitting customers to apprehend the possible effectiveness of these mechanisms.	**[F2]:**Data Mining **[F5]:**Statistics	**[M3:]**IBM Discovery **[M6:]**IBM Cloud Databases for PostgreSQL
[G7]: Allowing both users and officers to learn the analytics of the tickets given in their city.	**[F2]:**Data Mining **[F5]:**Statistics	**[M3:]**IBM Discovery **[M6:]**IBM Cloud Databases for PostgreSQL
-	**[F0]:**User data management	**[M1:]**IBM App ID

Figure 38: Requirements Traceability

5 Implementation, Integration and Test Plan

5.1 Implementation Order

Here we explain how a possible good order of implementation of sub-components might help us to fulfill and build the application. Its achievement implies the following steps:

- The very first thing to be implemented is the SafeStreets link with the municipality, implemented in form of encrypted HTTP APIs. The correct and secure definition of this link is something that the entire SafeStreets architecture relies on.
- At the second stage, we plan to create and configure all the server components/microservices (since they are those which will be queried by the client application afterwards).
  Every microservice component, will be implemented following the goals and functionalities described in this documents and in the SafeStreets RASD. The order of the implementation of each microservice is not relevant, but only the development of one microservice at a time will be allowed. Basically, our implementation order ranges from creating a user pool (IBM App ID) for data synchronization and authentication to setting up the cloud storage (IBM Cloud Object Storage and IBM Cloudant) for archiving data.
- Then, when all microservice are implemented, we will focus on the creation of the client application such as the mobile app and the dashboard for the municipality.

As said, in our architecture, every microservice needs to be developed one small functionality at a time, and should be as atomic as possible. Once a microservice is ready, it will be tested on its own, in order to have a more efficient way of testing the application.

5.2 Macrocomponents to be integrated

Here we describe how integration strategies are supposed to be performed. Specifically, we explain how different components are integrated in our system and which strategy we adopt in order to integrate all the components efficiently.
The following list represent every component grouped into several subsystems into which it will be integrated:

Core Data Storing

- IBM Relational Databases for PostgreSQL
- IBM Cloud Object Storage

Account Management

- IBM App ID for Authentication
- IBM App ID for Account Management

Violation & Ticket Management

- Violations Manager
- Tickets Manager

Basic Server

- Kubernates
- Apache2 custom microservice

Data Handling

- Statistics custom microservice
- IBM Watson Visual Recognition

Suggestion Management

- Suggestion custom microservice
- IBM Watson Discovery
- IBM Cloudant

Interfaces

- Web Application GUI
- Mobile Application GUI (Android and iOS)

5.3 Integration Testing Strategy

The first integration testing strategy that we'll use to test our application during its development, is the **Thread approach**. In particular, the **Bottom-up strategy** will be used to test and integrate modules within every thread. At first, only single portions of the modules (the ones that don't need any stubs) will be integrated and tested.
Small drivers needs to be created in order to give inputs to the portion of each module until a single feature is completed, then others threads will replicate the same procedure with the goal of reaching the completeness of the whole application.
Using a global strategy like this one will allow to have an application that works in the early stages of the implementation; this could permit to anticipate some testing and could minimize the costs of repair in the eventuality of an error.
Having said that, we will perform a second integration test when each module is fully implemented

In this new tests, we will impersonate a user who is actually using our application and who tries to get a result in response to a certain action.
This kind of test verifies not only the correct behavior of every single element, but also ensure the correct relationships with the other components of the application, listed in the above subsection.
The integration test that we will set up will use a real browser or application in order to perform, on the pages of the application, a certain number of actions in a programmatic way (such as the submission of a violation), to then verify a specific output at the end of the test. Following the example of the violation submission, the user will expect a successful message.
Although the integration test is undoubtedly the most complete and reliable, since it runs the entire stack during its execution, it is also the slowest, especially for the functionality that to be tested requires interaction with Computer Vision or Data Mining.
In our case, we would have needed to collect thousand of test results, but integrating the component with the thread approach while developing allow ou to proceed with fewer integration tests and then lower the execution times.

6 Softwares Used, Effort Spent and History

Software Used

Task	Software Used
Edit and compile LATEX code	TeXstudio and VisualStudioCode
UML modelling	PlantUML
Images and charts	draw.io
Mockup	Moqups and mockflow

Table 1: Softwares used

Effort Spent

Student	Hours spent
Andrea Furlan	close to 34
Cosimo Russo	close to 35
Giorgio Ughini	close to 31

Table 2: Effort spent

Revision History

- **v1.1 on 10/12/2019:** Minor Fix in component interfaces