POLITECNICO DI MILANO 1863

SOFTWARE ENGINEERING 2 PROJECT

Design Document (DD)

SafeStreets

Version 1.0

Authors
Tiberio Galbiati
Saeid Rezaei

Supervisor
Dr. Matteo Rossi

Download page: https://github.com/TiberioG/GalbiatiRezaei.git

November 24, 2019

Contents

List of Figures

List of Tables

1 Introduction

1.1 Purpose

This is the Requirement Analysis and Specification Document (RASD) of SafeStreet application. Goals of this document are to completely describe the system in terms of functional and non-functional requirements, analyze the real needs of the customer in order to model the system, show the constraints and the limit of the software and indicate the typical use cases that will occur after the release. This document is addressed to the developers who have to implement the requirements and could be used as a contractual basis.

1.1.1 Description of the given problem

SafeStreets is a crowd-sourced application that intends to provide users with the possibility to notify authorities when traffic violations occur, and in particular parking violations. The application allows users to send to authorities pictures of violations, including their date, time and position. Examples of violations are: vehicles parked in the middle of bike lanes, in places reserved for people with disabilities, on foothpats, double parking etc.

SafeStreets stores the information provided by users, completing it with suitable meta-data every time it recieves a picture. In paricular it is able to read automatically the license plate of a vehicle and store it without asking the user to type it. Also it stores the type of the violation which is input by the user from a provided list. Lastly it stores the name of the street where the violation occurred, rereiving it automatically from the geographical position where the user took the picture. Then the application allows both end users and authorities to mine the information crowd-souced. Two visualizations are offfered: the first is an interactive map where are high-lighted with a gradient color the streets with the highest frequency of violations. The second is a list of the vehicles that committed the most violations (available only to authority users).

In addition the app offers a service that creates automatically traffic tickets which can be approved and sent to citizens by the local police. This is done using the data crowd-soucred by the users. The application guarantees that every picture used to generate a ticket has't been altered. In addition, the information about issued tickets is used to build statistics. Two kind of statistics are offered: a list of people who received the highest number of tickets and some trends of the issued tickets over time and the ratio of approved tickets over the violations reported.

1.2 Scope

1.3 Definitions, acronyms, abbreviations

1.3.1 Definitions

- Heatmap : A heatmap is a graphical representation of data that uses a system of color-coding to represent different values
- Enduser : a regular citizen which will use the app
- Authority user : someone who's working for an authority like police, municipality etc.
- Geocoding : the process of converting addresses (like a street address) into geographic coordinates (latitude and longitude)
- Reverse geocoding: the process of converting geographic coordinates into a human-readable address

1.3.2 Acronyms

- ALPR : Automated Licence Plate Recognition
- GUI : Graphical User Interface
- GDPR : EU General Data Protection Regulation
- API : Application Programming Interface

1.3.3 Abbreviations

1.4 Revision history

This is the first released version 10/11/2019.

1.5 Reference Documents

References

[1] Robert C. Martin, Clean Architecture: A Craftsman's Guide to Software Structure and Design, Prentice Hall, Year: 2017 ISBN: 0134494164,9780134494166

[2] OpenALPR Technology Inc. , OpenALPR documentation http://doc.openalpr.com

[3] MongoDB Inc, The MongoDB 4.2 Manual https://docs.mongodb.com/manual/

[4] Node.js Foundation, Node.js v13.1.0 Documentation https://nodejs.org/api/

[5] StrongLoop, IBM, and other expressjs.com contributors, Express.js website http://expressjs.com

[6] GOOGLE inc, Google Maps Platform Documentation | Geocoding https://developers.google.com/maps/documentation/geocoding/start

[7] GOOGLE inc, Google Maps Platform Documentation | Heatmap https://developers.google.com/maps/documentation/javascript/heatmaplayer

1.6 Document Structure

This document is divided in five parts.

1. **Introduction**
2. **Architectural Deisgn**
3. **User Interface Design**
4. **Requirements Traceability**
5. **Implementation, Integration and Test Plan**
6. **Effort spent** contains the tables where we reported for each group member the hour spent working on the project

2 Architectural Design

2.1 Overview

In order to design our application we need two main parts: one is the The general architecture of our system has three tiers. We have a mobile app running on mobile devices, smartphones or tablets with ios or Android. then we have a server

the kind of architecture is distributed logic as explainde in the slides.

2.2 Component view

Here are proposed the component view for both part of the system, the mobile application and the application server.

2.2.1 Mobile App

The component view for the mobile app

Entities Entities are the domain of the system, they represent the business objects of the application. In our case entities are plain objects with methods that don't have any dependency on other part of the system (eg. frameworks). Since the core of our system is based on **Users**, **Violations** and **Tickets** we have included those entities.

Use Cases Use cases are components that represent our system actions, they are pure business logic which describe what is possible to do do with the application. We have one component for every possible use case.

We encapsulate all use case in a **Use Case interactor** which manages all possible use cases, it depends on the entites and has communication ports with the Controller and Presenter. In fact the use case interactor has two ports: an input, which interfaces with the Controller and an output port connected with the presenter As an example if there is data coming from the camera, this is acquired by an adapter of the controller (described later) and is passed to the Use case interactor which coordinates entities, Use cases and the data just acquired. After data is processed is passed to the Presenter and

Two sub com

CoreUtils This component encapsulates all the libraries and classes with methods that can be needed by any use case. Here we list some of these functions

- Input validation: to check if
- Error handling and reporting
- Exceptions
- Data conversion

Controller The controller component is an adapter that encapsulates all the specific adapters which are devoted to retrieve and store data from different sources such as the local filesystem, the device sensors, the camera and lastly our apllication service API which is described in section 2.2.2. Each component of the controller in fact implements the interfaces required by each use case.

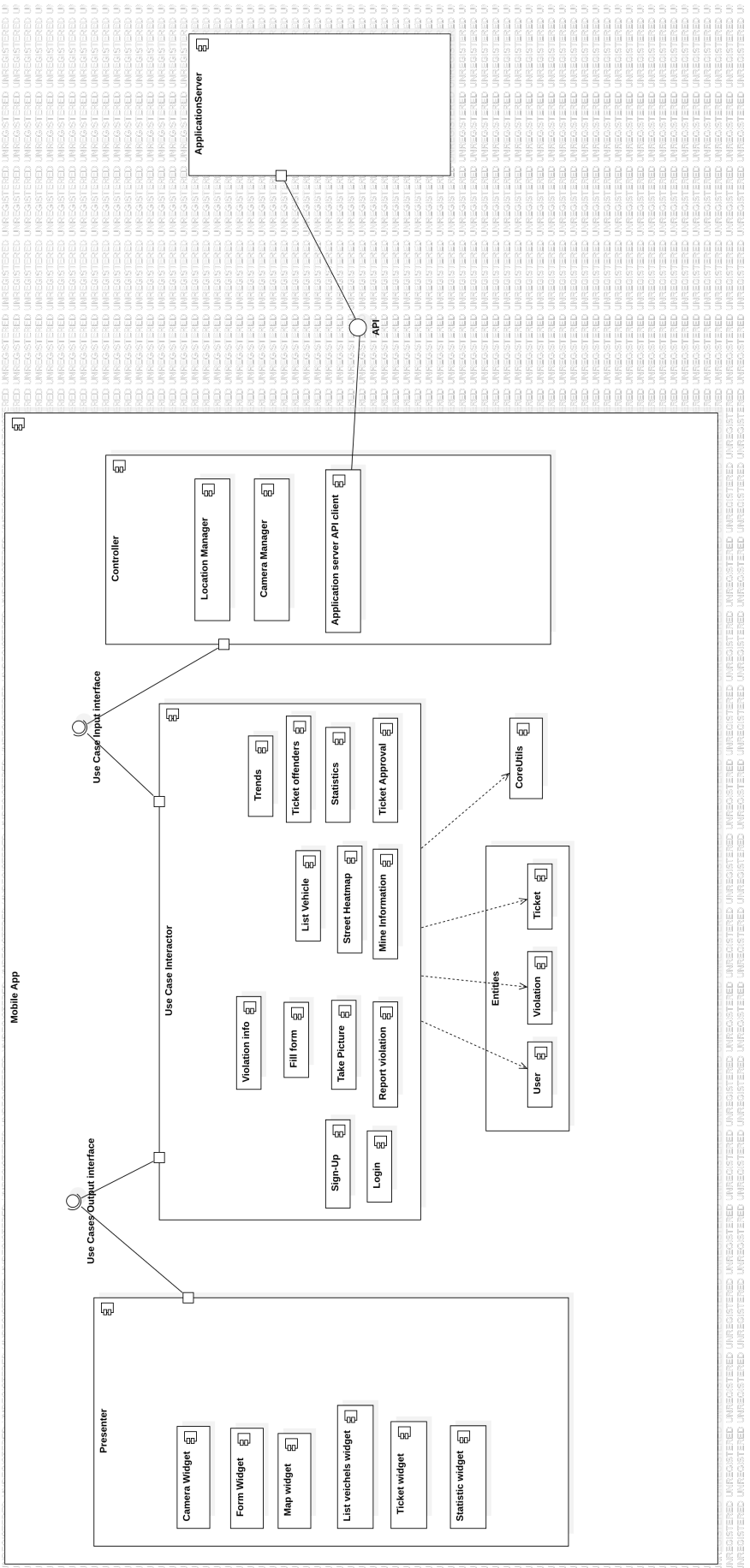

Figure 1: Component diagram

Presenter The presenter

2.2.2 Application Server

The architecture of the Application server looks the same as the Mobile Application,

Entities Entities for the application server are the same as Since the core of our system is based on **Users**, **Violations** and **Tickets** we have included those entities.

Use Cases The use cases component for the server-side have the same name as the ones as the application but they have a completly different logic. As an example, on the application side the use case "Take picture" has to interact with the device physical camera in order to take the picture and then send it as POST HTTP request to the server. The use case with the same name, but on server side has to fetch the HTTP requets, parse the content, send the picture to the OpenALPR service to get the decoded plate and then store the picture.

CoreUtils This component encapsulates all the libraries, classes with methods a middleware that can be needed by any use case.

- Input validation
- Error handling and reporting
- Exceptions
- Data conversion
- Body parser for HTTP requests

Controller The controller component is an adapter that encapsulates all the specific adapters which are devoted to retrieve and store data from different sources such as the local filesystem, the device sensors, the camera and lastly our apllication service API which is described in section 2.2.2. Each component of the controller in fact implements the interfaces required by each use case.

Presenter The presenter

2.3 Deployment view

In Figure 3 is shown the Deployment diagram.

The deployment consist of three main devices. The first tier consist is **Mobile device** the user will use, which can be a smartphone or a tablet using as operating system either iOS or Android. The exection environment is the built Flutter app.

The second tier is the **Application Server**. It is supposed to be a dedicated server running a linux distribution specific for server use. As an example of OS we choose Centos 7. Other distros can be used like Red Hat Enterprise Linux, Debian, OpenSUSE. As execution enviorment we install Node.js which is an open-source JavaScript runtime environment that executes JavaScript code outside of a browser. Inside Node.js we use the web application framework Express.js which is designed for building web applications and APIs.

The third tier is the **DB Server**. It consists in another server where we run the DB system MongoDB. We choose to run the database in a separate server and not in the same as the ApplicationServer in order to increase scalability. MongoDB is a cross-platform document-oriented

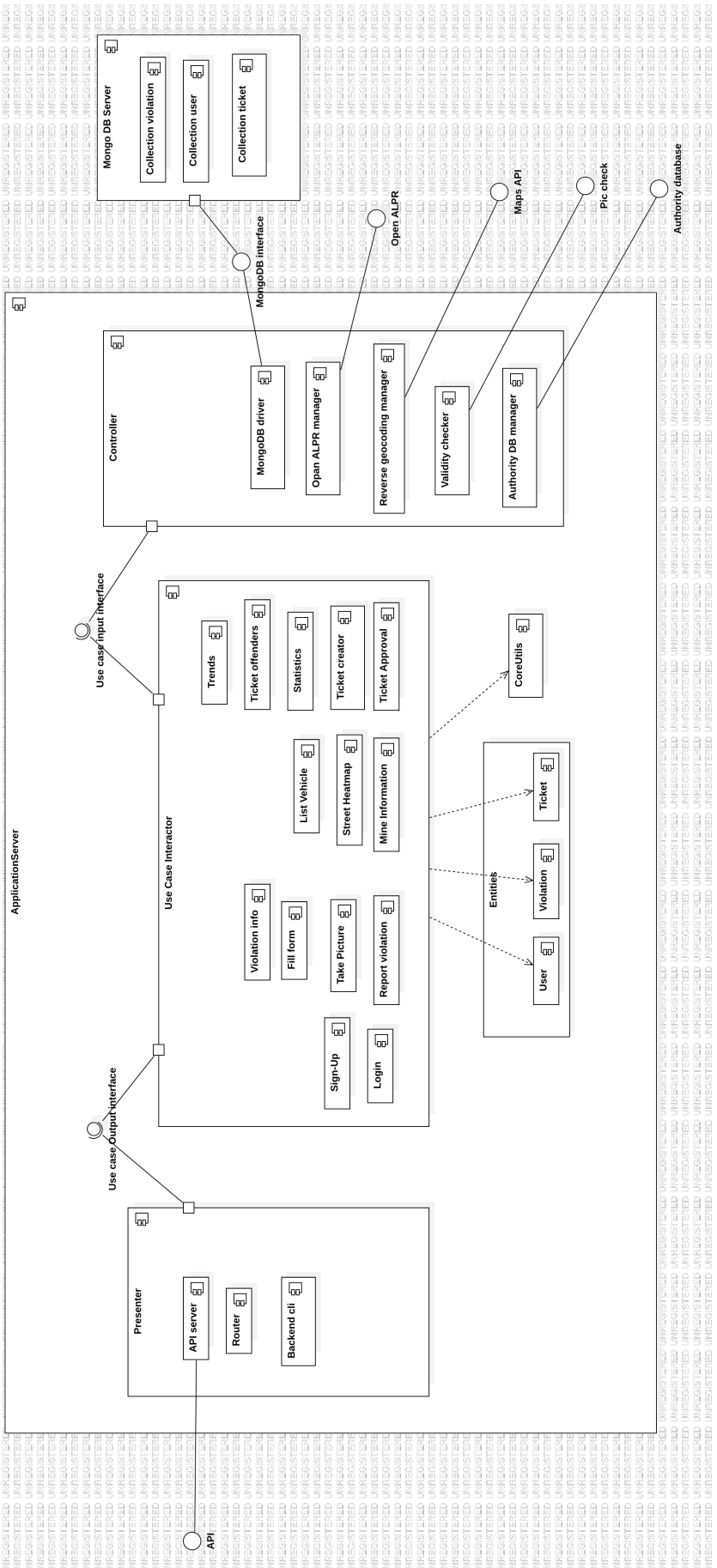

Figure 2: Component diagram

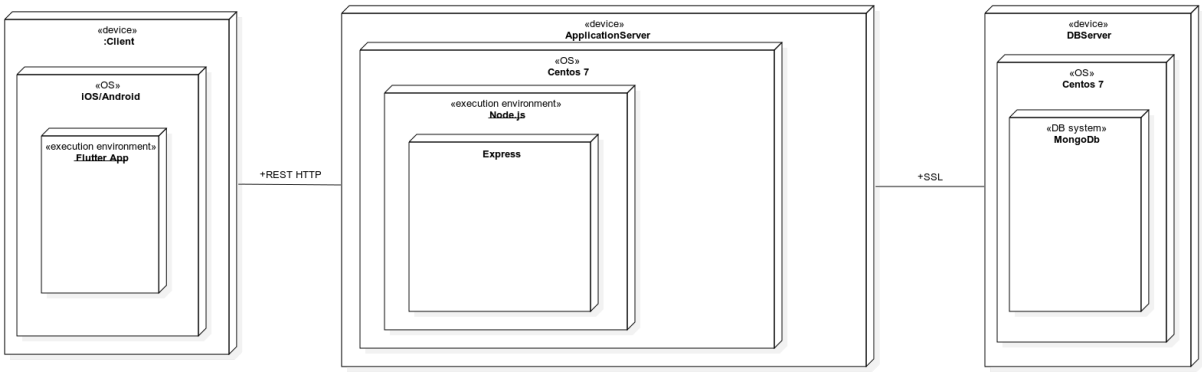

Figure 3: Deployment diagram

database program. Classified as a NoSQL database program, MongoDB uses JSON-like documents with schema.

2.4 Runtime view

api endpoints

2.5 Selected Architectural styles and patterns

As already introduced each part of our system will use the Clean Architecture, proposed by Robert C. Martin. We apply this architecture both on the mobile application and in the Application Server.

2.5.1 Dependency Rule

"The concentric circles in Figure 4 represent different areas of software. In general, the further in you go, the higher level the software becomes. The outer circles are mechanisms. The inner circles are policies. The overriding rule that makes this architecture work is the Dependency Rule: *Source code dependencies must point only inward, toward higher-level policies.* Nothing in an inner circle can know anything at all about something in an outer circle. In particular, the name of something declared in an outer circle must not be mentioned by the code in an inner circle. That includes functions, classes, variables, or any other named software entity." [1]

2.5.2 Entities

"Entities encapsulate enterprise-wide Critical Business Rules. An entity can be an object with methods, or it can be a set of data structures and functions. It doesn't matter so long as the entities are the business objects of the application. They encapsulate the most general and high-level rules. They are the least likely to change when something external changes. For example, you would not expect these objects to be affected by a change to page navigation or security. No operational change to any particular application should affect the entity layer" [1]

2.5.3 Use Cases

"The software in the use cases layer contains application-specific business rules. It encapsulates and implements all of the use cases of the system. These use cases orchestrate the flow of data to and from the entities, and direct those entities to use their Critical Business Rules to achieve the goals of the use case. We do not expect changes in this layer to affect the entities. We also do not expect this layer to be affected by changes to externalities such as the database, the UI,

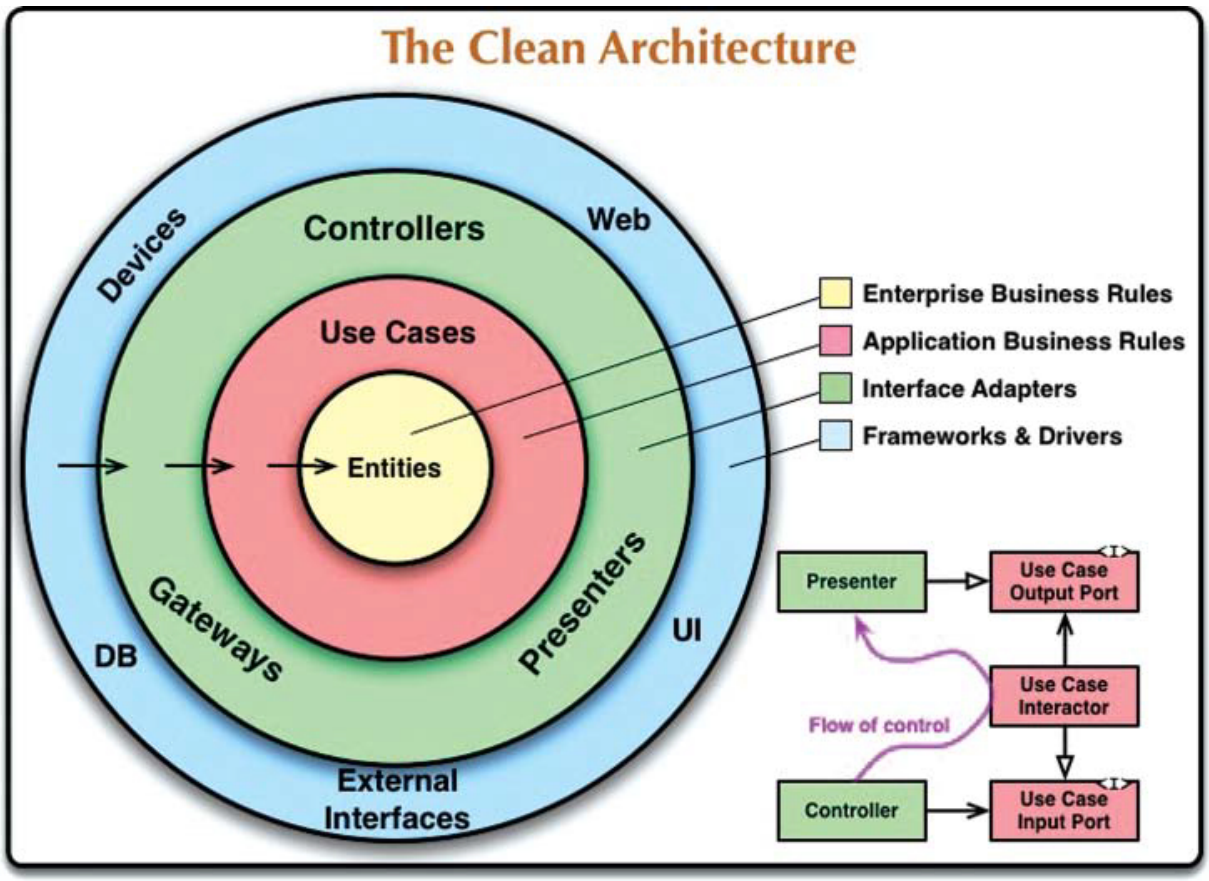

Figure 4: Clean Architecture [1] p. 203

or any of the common frameworks. The use cases layer is isolated from such concerns. We do, however, expect that changes to the operation of the application will affect the use cases and, therefore, the software in this layer. If the details of a use case change, then some code in this layer will certainly be affected." [1]

2.5.4 Interface Adapters

"The software in the interface adapters layer is a set of adapters that convert data from the format most convenient for the use cases and entities, to the format most convenient for some external agency such as the database or the web. The presenters, views, and controllers all belong in the interface adapters layer. The models are likely just data structures that are passed from the controllers to the use cases, and then back from the use cases to the presenters and views." [1]

2.5.5 Adantages of Clean architecture

Following are some reasons which a good architectural pattern for our app:

- All business logic is in a use case, so it's easy to find and not duplicated anywhere else
- Good monolith with clear use cases that you can split in microservices later on

2.5.6 REST

The communication between the mobile application and the microservice will be done via HTTP requests following REST principles. REST (Representation State Transfer) is an architectural style for communication based on strict use of HTTP request types. One of the most important REST principles is that the interaction between the client and server is stateless between requests. Each request from the client to the server must contain all of the information necessary to understand the request. The client wouldn't notice if the server were to be restarted at any point between the requests.

2.6 Other design decisions

Here we describe the frameworks and languages hat should be used to produca a state-of-art application.

Flutter

Node.js To build a microservice

MongoDB MongoDB is a noSQL database. It stores data as document which can have different shemas.

This application will be developed with the MVP architectural pattern. In general, the MVP pattern allows separating the presentation layer from the logic, and this feature can be useful when we test the app. MVP is a user interface architectural pattern, which eases automated unit testing and it helps with providing clean code. This pattern consists of three parts which are Model, View and Presenter. In this pattern, model does not communicate with the view directly, it is the Presenter's responsibility to communicate with the Model and update the View. SafeStreet application will be developed with the MVP architectural pattern in place of the MVC (model, view and controller) because of the test advantage mentioned above and compared to MVVM the architecture does not fit to the project design. MVVM does not give us a relation 1-1 between Presentation and View. For that reason, during this project it is recommended to utilize the MVP

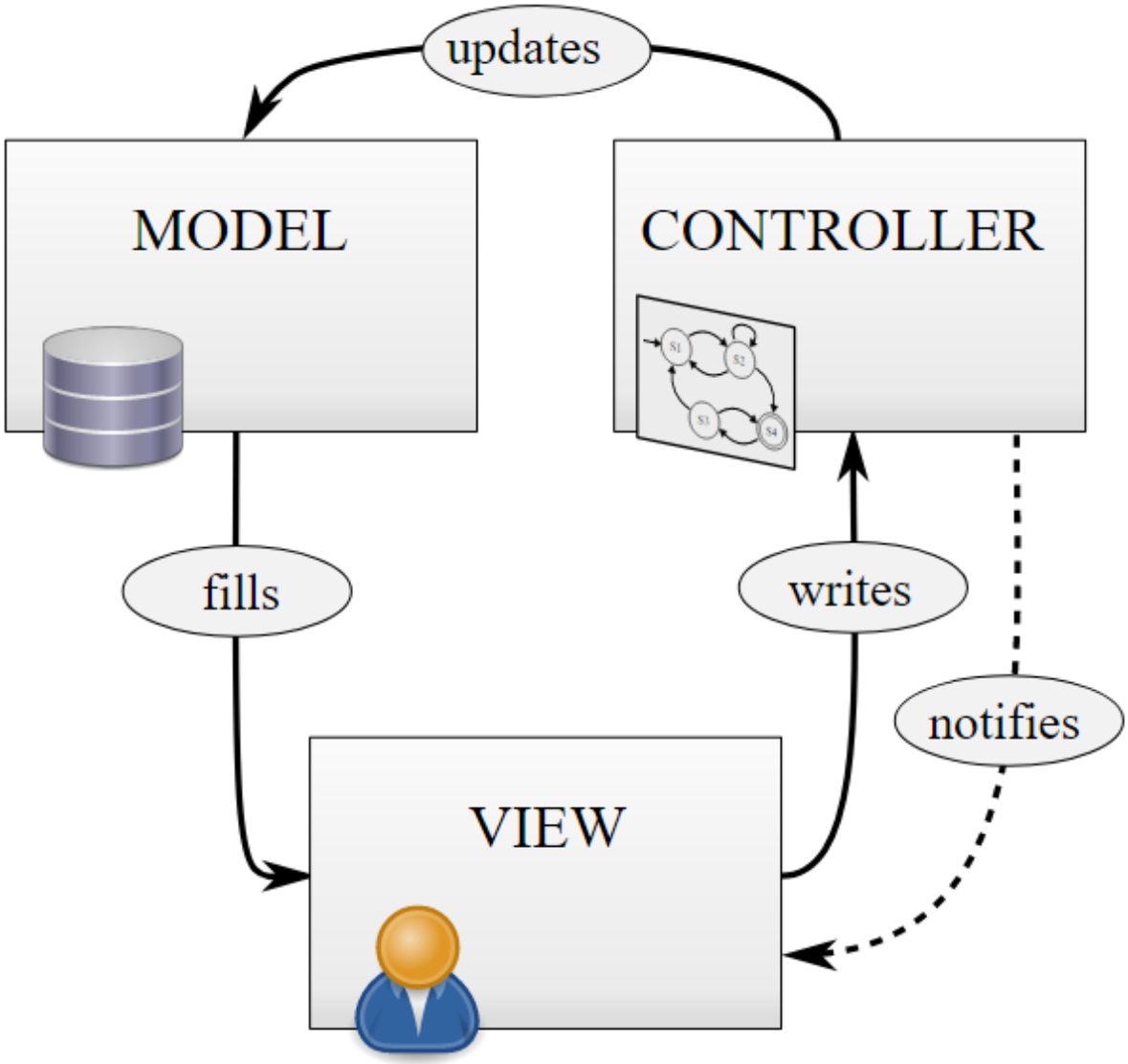

Figure 5: MVC Architectural diagram

architectural pattern. There are more architectural patterns that we considered and discarded, like "Client-server pattern" or "Layered pattern", but as mentioned above the MVP architecture would be the best fit for the SafeStreet Application.

2.6.1 Model

The model component stores data and its related logic. It represents data that is being transferred between controller components or any other related business logic. It responds to the request from the views and also responds to instructions from the controller to update itself. It is also the lowest level of the pattern which is responsible for maintaining data. In this project we use MongoDB database to store all the useful data. As well as we will use NODE.JS as the application server to build and run the application.

2.6.2 View

A View is that part of the application that represents the presentation of data. Views are created by the data collected from the model data. A view requests the model to give information so

that it resents the output presentation to the user. In order to implement SafeStreet application we are going to use Flutter framework. Flutter helps app developers build cross-platform apps faster by using a single programming language. Although there are some other frameworks to implement cross-platform apps, according to [https://nevercode.io/blog/flutter-vs-react-native-a-developers-perspective/] Flutter is more efficient than others and has entered the cross-platform mobile development race very strongly.

2.6.3 Controler

Controler is the mediator between View and Model which hold responsibilities of everything which has to deal with presentation logic in the application. Presenter does the job of querying the Model, updating the View while responding to the user's interactions. It monitors Model and talks to View so that they can handle when a particular View needs to be updated and when to not. In this project we will use Representational state transfer (REST) API in order to communicate between View and Model.REST is the software architectural style of the World Wide Web. REST gives a coordinated set of constraints to the design of components in a distributed hypermedia system that can lead to a higher-performing and more maintainable architecture. To the extent that systems conform to the constraints of REST they can be called RESTful. RESTful systems typically, but not always, communicate over Hypertext Transfer Protocol (HTTP) with the same HTTP verbs (GET, POST, PUT, DELETE, etc.) which web browsers use to retrieve web pages and to send data to remote servers. We have decided this API because it guarantees to achieve important non-functional requirements such as:

- Scalability: every node belonging to our architecture can be multiplied without redesign the whole system.
- Portability: Every platform it's able to interact with the server since it's just a matter of HTTP request and JSON response.
- Reliability: If suddenly an instance crashes, the load balancer detaches it and will be replaced by a new one automatically.

2.6.4 Why do we use MVP architectural pattern?

Following are some reasons which makes MVP a good architectural pattern for our app:

- Makes debugging easier in Applications: MVP enforces three different layers of abstractions which makes it easier to debug your applications. Moreover, since business logic is completely decoupled from View, it is more easier to perform unit testing while developing your application.
- Enforces better separation of Concerns: MVP does the great job of separating out the business logic and persistence logic out of the Activity and Fragment classes which in turn better enforce good separation of concerns.
- Code Re-usability: In MVP, the code can be better reused since we can have multiple presenters controlling our Views. This is more important as we definitely don't want to rely on a single presenter to control our different Views.

2.7 Other design decisions

3 User Interface Design

uiiiiii

4 Requirements Traceability

The advantage of using clean architecture is that we have a component for each use case so it's very easy to keep track of the requirement traceability.

	Requirement	**Component**
[R1]	User must be able to choose the kind of violation from a list	Fill form
[R2]	User must be able to read detailed information about each kind of violation he can report	Violation info
[R3]	Date, time and position should be automatically added to the violation reported	Report violation
[R4]	We should require the user to send again a picture in case the plate is not visible	Take picture
[R5]	The user must be able to select the vehicle to report in case there are other vehicles in picture	Take picture
[R6]	Application must automatically determine the street name where User is	CoreUtils
[R7]	Application must be able to count occurrency of violations	Street Heatmap
[R8]	Application must be able to count violation for each vehicle	List vehicles
[R9]	Application should show all the vehicles ordered with the number of violations	List Veichle
[R10]	Application should visualize the areas where violation occurred	Street Heatmap
[R11]	Application must use a gradient of color to show the occurrencies of violations as an overlay of a interactive map	Street Heatmap
[R12]	Regular users cannot mine data about plates of the offenders	Mine Information
[R13]	Authority users can know the exact licence plate when mining data about offenders	Mine Information
[R14]	Only authority users can access the ticket approval section	Ticket approval
[R15]	Application must be able to read every violation stored and automatically generate a ticket	Ticket Creator
[R16]	Application should offer to authorities the possibility to approve tickets or not	Ticket approval
[R17]	Application must store all the tickets created	ticket creator
[R18]	Application must read all the history of tickets created	Trends
[R19]	The application must be able to know if a picture has been altered	Ticket Creator
[R20]	If a picture has been altered the applcation must automatically flag as not valid the corresponding ticket	Ticket Creator

Table 1: Requirements Traceability matrix

5 Implementation, Integration and Test Plan

iiiii

6 Effort Spent

Tiberio	
Task	**Time**
Structure of document	1h
Component diagrams and study of REST	2h
Component diag	1h 30 min
Meeting design	1h 30 min
Study clean architecture and DeploymentDiagram1	3h
Study clean architecture	1h
New Component diagrams	3h
Design patterns and frameworks	2h
Requirements traceability	2h
Total	**17 h**

Saeid	
Task	**Time**
Meeting design	1h 30 min
Total	**36 h 30 min**