POLITECNICO DI MILANO 1863

SOFTWARE ENGINEERING 2 PROJECT

Design Document (DD)

SafeStreets

Version 1.0

Authors
Tiberio Galbiati
Saeid Rezaei

Supervisor
Dr. Matteo Rossi

Copyright: Copyright © 2019, Tiberio Galbiati - Saeid Rezaei – All rights reserved

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

December 8, 2019

Contents

Copyright © 2019 Tiberio Galbiati - Saeid Rezaei

List of Figures

List of Tables

Copyright © 2019 Tiberio Galbiati - Saeid Rezaei

1 Introduction

1.1 Purpose

This is the Design Document (DD) of SafeStreet application. The purpose of this document is to discuss more technical aspects regarding architectural and design choices that must be made, so as to follow well-oriented implementation and testing processes.

1.2 Scope

The scope of the SafeStreet application can be found in the RASD.

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
- UI : User Interface
- GDPR : EU General Data Protection Regulation
- API : Application Programming Interface
- REST : REpresentational State Transfer
- JSON : JavaScript Object Notation
- RASD : Requirements Analysis and Specifications Document

1.3.3 Abbreviations

- APP : The SafeStreet mobile application to be developed

1.4 Revision history

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

Copyright © 2019 Tiberio Galbiati - Saeid Rezaei

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

Copyright © 2019 Tiberio Galbiati - Saeid Rezaei

2 Architectural Design

2.1 Overview

The system to be developed has a three tier client/server structure. Here we list the three main tiers:

1. The first tier consists in a mobile app running on mobile devices: smartphones or tablets with iOs or Android. This is the application the users will interact with
2. The second tier consists in an Application Server which provides the service as a RESTful API to the clients which are the running applications, and it's connected to the third tier, the Database Server where all data is stored
3. The third tier is the Database where data is stored

There are some advantages of our system architecture: the first one is the modularity of this approach of having different subsystems. The second is scalability, which is the ability of the system to manage changes in the scale of demand. Since we have separated the client, the server and the database, it's easy to manage for example an increase of the data stored, just updating the database with a bigger one, whithout modifying the application server.

Moreover the sofware that runs on clients and on the server will be developed with a layered Clean architecture, as described in detail in Section 2.8. This pattern is organized through abstraction levels, starting from Entities to Use Cases and finally to Controllers and Presenters. One big advantage of this architecture is that we have a separated component for each use case.

2.2 Component view

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

2.2.1 Mobile Application

Figure 1 shows the component diagrams for the Mobile Application. In the following paragraph we describe all the components of each cluster.

Entities Entities are the domain of the system, they represent the business objects of the application. In our case entities are plain objects 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 and is passed to the Use case interactor which coordinates the Use Cases and the data just acquired. After data is processed, it goes to the Presenter and visulalized by the widgets of the UI.

Here follows the list of every Component of the Use case group. We have to add a note here: when it's written for example that a component interacts with the Application Server, it doesn't

Copyright © 2019 Tiberio Galbiati - Saeid Rezaei

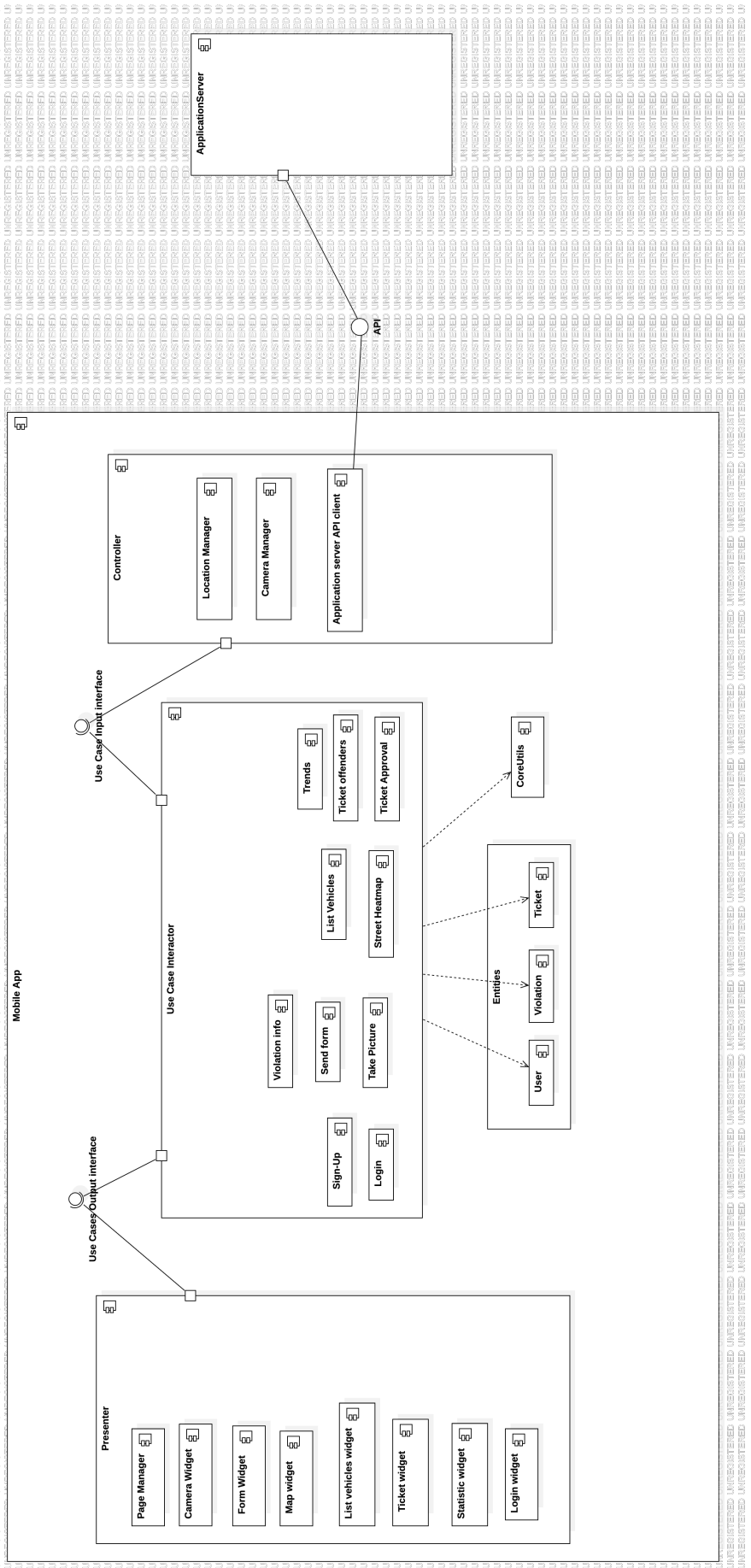

Figure 1: Component diagram for Mobile Application

mean that this component interacts directly with the Application API. As we have written before, every use case has to interact firstly with the controller, which handles then the connections with the external interfaces. The same goes for communicating with the UI, every data transfer happens via the Use-Case/Presenter interface. In order to keep the following descriptions short, we just say what the component does in a higher level.

- **Sign-Up**: This component allows to add new users to the system. It has to get the input strings (username, password, user data etc.) from the sign-up widget, validate them before sending the request to the Application Server. After, it has get the answer from the Server and communicate to the user if the sign-up was successful or not.
- **Login**: This component allows the login to the App. It has to get the input from the Login widget, communicate the strings to the Application Server and interpret the response if the user is autorized to login or not. It must also manage the API key, received by the Application Server, which is needed to ensure identification of each user and to provide a restricted access to users to some funcionalities.
- **Take Picture**: This component makes possible to take picture of violations. After the picture has been taken, it has to send it to the Application Server and wait for an asnwer. If the answer is that a plate is found, than it calls the next use case "Send form". If no plates are found it asks the user to take a picture again discarding the previous one. If more than one plate is found, it activates a "brush tool mode" (in the Camera Widget) to ask the user to cover the other plates present in the picture. Then it has to manipulate directly the picture changing the color of the pixels selected by the user. Lastly it can send the picture to the Server.
- **Send form**: This component presents the user the form in which he can choose the type of volation referred to the picture just taken in the previous use case. After the form has been submitted by the user, it is sent to the Application Server in order to be stored.
- **List Vehicles**: This component calls the Application Server in order to get the list of vehicles that committed the most violations. Then it opens the widget to show a scrolling list view of plates with the count of violations.
- **Street Heatmap**: This component asks the Application Server to provide the heatmap visualization of the areas where the violations happened.
- **Ticket Approval**: This component is responsible of showing the tickets available for approval. Every time the Authority user opens this or refersh the list, the component has to call the Application Server in order to get the data about the pending tickets. Moreover this component offers the feature of approving or discarding a pending ticket using two buttons.
- **Ticket Offenders**: This component is responsible of showing the list of the most egregious offenders by querying the Application Server.
- **Trends**: This component has to show the desired statsistics about the issued tickets, querying the Application Server.

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

Copyright © 2019 Tiberio Galbiati - Saeid Rezaei

- Error handling and reporting
- Exceptions
- Data conversion

Controller The controller component 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 application service API which is described in section 2.2.2. Each component of the controller in fact implements the interfaces required by the use cases. Here is the description of each sub-component:

- **Location Manager**: this component is responsible of getting the location of the user, accessing the libraries of the OS of the device
- **Camera Manager**: this component is responsible of getting access to the camera of the device
- **Application Server API client**: this component handles all the HTTP requests to and from our Application server

Presenter The presenter is a macro component that communicates with all the components of the UI. Here follows the list of every component with the description:

- **Page Manager**: this component coordinates every page shown in the mobile application, and provides the navigation bar
- **Login Widget** : this is the login page and sign-up page, with the required input fields
- **Camera Widget** : this shows what is recorded by the camera, has a button for taking pictures, activate zoom and other camera functionalities. It also shows the picture once taken and implements the "brush tool mode". This is the graphic interface used to cover some parts of th epicture like with a black brush
- **Form Widget**: this shows a list of possible description of violations the user can select
- **Map Widget** : this is responsible for showing the street heatmap
- **List Vehicles Widget** : this widget shows the vehicles that committed the most violations as a scrolling list, showing the plate string and the count of violations per plate
- **Ticket Widget**: this shows the list of tickets the authority user can approve or not. It also shows information about each one after the user clicks on it. It has also two buttons to approve or discard the selected ticket
- **Statistics widget**: this can show some data visualization about the issued tickets

2.2.2 Application Server

Figure 2 shows the component diagrams for the Mobile Application. In the following paragraph we describe all the components of each cluster. The architecture of the Application server looks the same as the Mobile Application, but has completly different components in the Presenter and Controller.

Entities Since the core of our system is based on **Users**, **Violations** and **Tickets** which are the main Entities of the Systemm, those are the same as the Mobile Application.

Copyright © 2019 Tiberio Galbiati - Saeid Rezaei

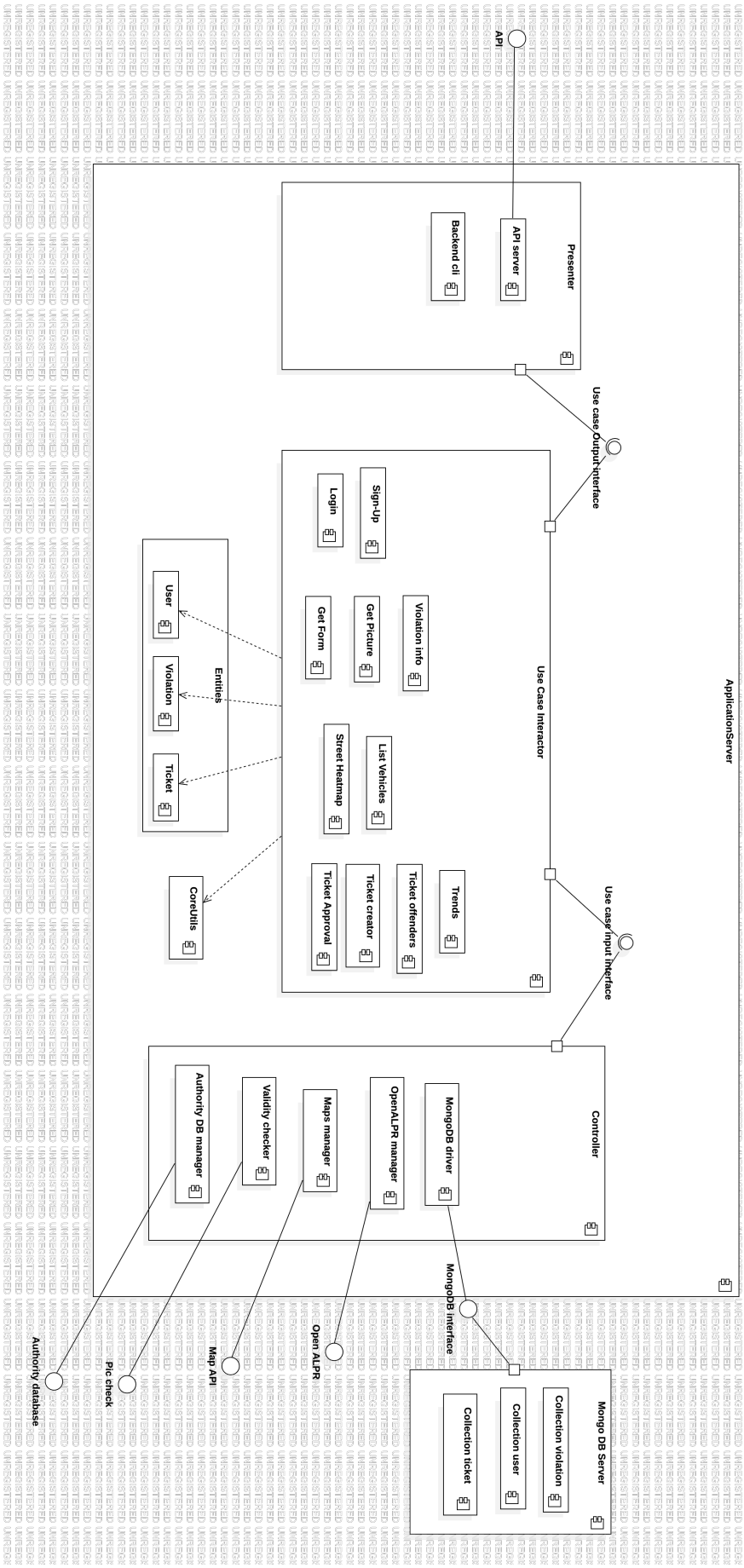

Figure 2: Component diagram for Application Server

Use Cases The use cases components for the server-side have almost the same names as the ones as the application, but they have a completly different logic. As an example: on the application side we have the use case "Take picture" which 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, whereas on server side we have the "Get picture" use case, which fetches the HTTP requets, parses the content, sends the picture to the OpenALPR service to get the decoded plate and then stores the picture.

- **Sign-Up**: This component receives from the APP the username, password and personal information of a new user who is registering to the service. It must be able to check if the username is already in use or not. Then it must encrypt the password. Lastly it stores the new user in the database.

- **Login**: This component receives from the APP the request of a user trying to login to the service. It has to send back a successful login answer if the user exists and the password is correct. It has also to communicate an API key to the mobile app that will be used for every successive API call. This API key is needed to ensure identification of each user and to provide a restricted access to users to some funcionalities.

- **Get Picture**: This component receives the POST request from the APP containing the picture of the violation and the coordinates of the location of the user. The picture is stored temporarily in the filesystem and then is sent to the external OpenALPR API which has to decode the plate. If no plates are returned by the OpenALPR service, it has to send to the APP a message of error asking to take a picture again. If more than one plates are detected it has to send to the APP a message of error so the app can ask the user to use the "brush tool" to cover the other plates. If one plate is correctly found, it sends to the APP the plate just decoded with a success message.

  This component has also to send the coordinates to the reverse geocoding service of the Maps API in order to get the street name and number. Then it moves the picture in a specific directory and saves in the database a new record containing the path where the picture is stored, the received decoded plate as string, the raw coordinates, the string containing the name and the number of the road.

- **Get form**: This component receives the PUT request from the APP containing the content of the form assocciated to the picture of the violation. This request is PUT type because it must update the existing violation docment in database adding the kind of violation. Once the request is received it parses the content and updates the database.

- **List Vehicles**: This component is used to query the database of violations in order to get the list of vehicles with the associated count of violations occurred.

- **Street Heatmap**: This component is responsible of getting the map from the external Maps API and then forward it to the Mobile Application. It receives as input the coordinates where the map should be centered via a GET request from the Mobile Application. It must also query the database in order to create a map layer containing the coordinates of every violation.

- **Ticket Approval**: This component is used to provide the mobile APP the list of ticket available for approval.

- **Ticket Creator**: This component is used to automatically create a new ticket, based on the data present in the violation database.

Copyright © 2019 Tiberio Galbiati - Saeid Rezaei

- **Ticket Offenders**: This component does the query of the ticket database in order to return to the mobile APP the list of people who received the highest number of tickets.
- **Trends**: This component contains the logic in order to provide some aggregated data about the emitted tickets.

CoreUtils This component encapsulates all the libraries, classes with methods and 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 encapsulates all the specific adapters which are devoted to retrieve and store data from different sources such as the exteral APIs and the Database 2.2.2.

- **MongoDB driver** This is the component responsible of managing every interaction with the database
- **OpenALPR manager** This is the component responsible of managing every API call to the external service OpenALPR, used to decode the plates of a given picture
- **Maps manager** This is the component responsible of managing every API call to the external Maps Service
- **Validity checker** This is the component responsible of managing every API call to the external service used to check if a picture has been modified or not
- **Authority DB manager** This is the component responsible of managing every API call to the external database of authorities.

Presenter The presenter is the macro component that encapsulates all the frameworks needed to provide an API interface.

2.3 Deployment view

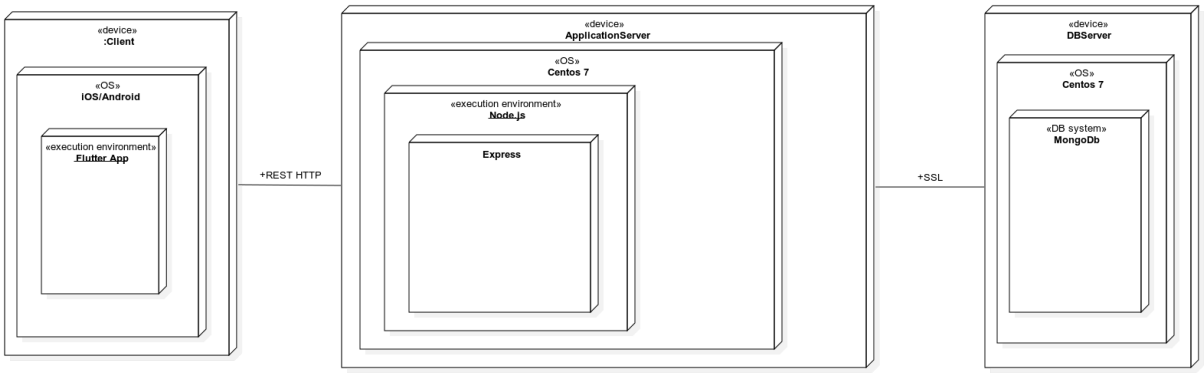

Figure 3: Deployment diagram

Copyright © 2019 Tiberio Galbiati - Saeid Rezaei

In Figure 3 is shown the Deployment diagram of all the system.

The deployment consist of three main devices. The first tier consist in the **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 SafeStreet 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 database program. Classified as a NoSQL database program, MongoDB uses JSON-like documents with schema.

2.4 Runtime view

A note for the sequence diagrams, to keep the diagrams more readable, we omit that every call to and from a specific use case must pass from the use case interactor which manaes the component interfaces.

Copyright © 2019 Tiberio Galbiati - Saeid Rezaei

2.4.1 Reporting Violation

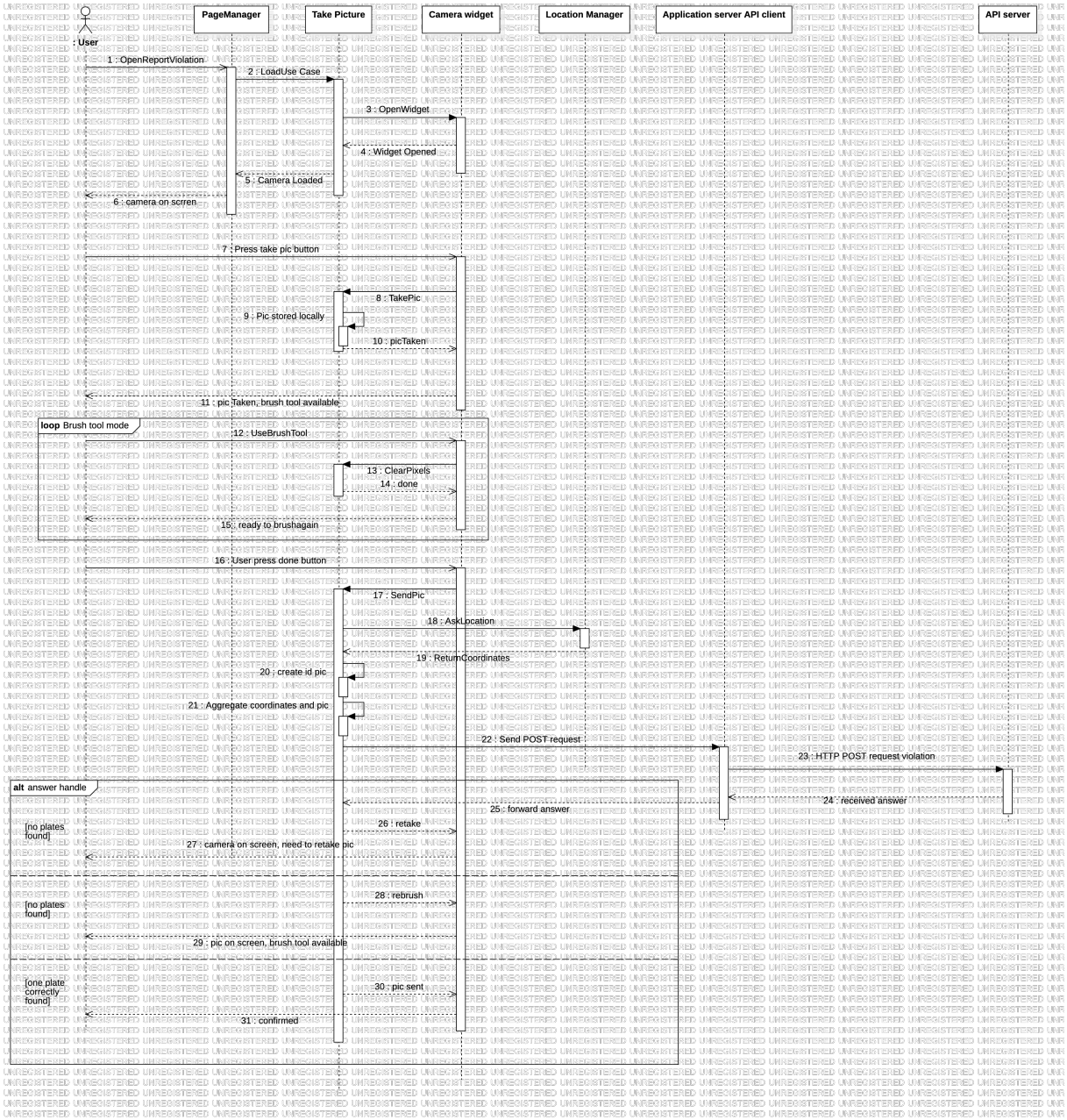

Figure 4: Sequence diagram for picture upload via the Mobile Application

In figure 4 is shown the sequence diagram for the use case of taking a picture using the mobile application. The user interacts with the **PageManager** pressing the button to open the "Report Violation section", so the **TakePicture** use case is loaded because it's the first step needed to start the reporting of a violation. This opens the **CameraWidget** that is the component of the Presenter responible of showing on screen the camera recording. When the widget is correctly started the user can take the picture pressing the main button. The widget has to tell the **TakePicture** use case that the button has been pressed so the picture can actually be taken and stored locally. The camera widget can now show the picture just taken and show the brush tool mode. If there is need the user can cover whith is finger some areas of the picture. The widget communicates to the use case the location of those areas so the image can be manipulated

Copyright © 2019 Tiberio Galbiati - Saeid Rezaei

by the **TakePicture** use case and stored. This can happen until the user presses the "done button". Now the the **TakePicture** use case asks the the **LocationManager** to return the GPS coordinates of the device. An Unique identifier is generated which is passed with the picture to the **Application Server API client** which has to actually make the POST HTTP request to the Application Server (endpoint /API/v1/violations/upload/id=xxx). What happens in the applcation server is described later in the following Sequence diagram in Figure 5. When an answer is received there are three options possible: if no plates have been found, the use case has to start again, so it tells the widget to show a message for the user, asking to retake the picture including the plate. If more than one plates are found, the use case has to ask the widget to show a message asking the user to use again the brush tool mode to cover the plates not required. If one plate is correctly identified the use case can tell the widget to show a success messae and terminates.

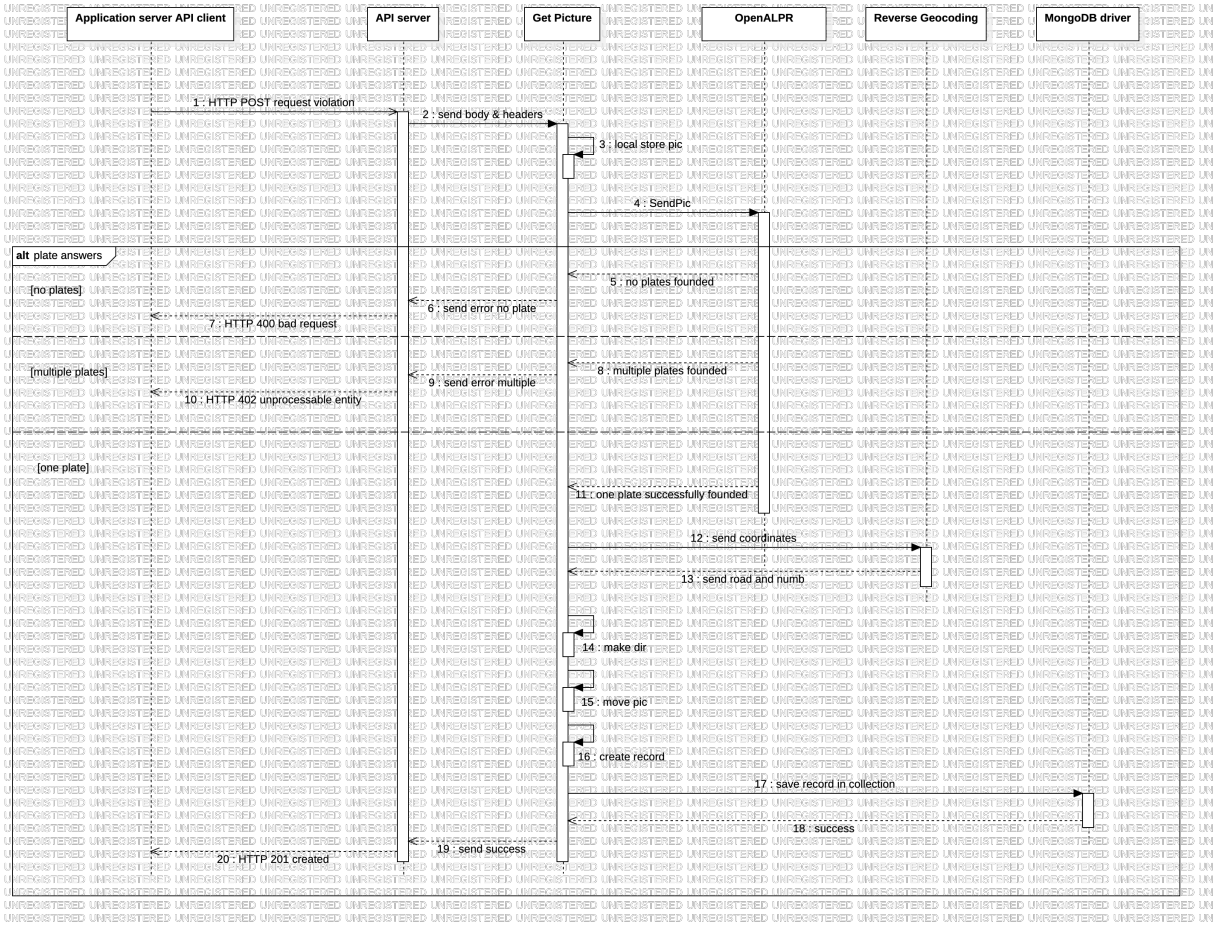

Figure 5: Sequence diagram for picture storage by Application Server

In figure 5 is shown the sequence diagram for the Application Server when it receives a POST request at the endpoint /API/v1/violations/upload/id=xxx containing a new picture in the body. When the requests is receied the the picture is stored on a temporary folder of the server by the component **Get Picture**. This use case then has to call the external API **OpenALPR** sending the picture of violation in order to decode the plate number. We have three options, depeending on the answer of the call to the **OpenALPR** API.

If no plates are found, the usecase **Get Picture** has to ask the **API server** to answer to the mobile app, informing that no plates were present in the picture. One possible HTTP answer for this error is 400 bad request.

Copyright © 2019 Tiberio Galbiati - Saeid Rezaei

If multiple plates have been found, the usecase **Get Picture** has to ask the **API server** to answer to the mobile app, informing that more than one plate were present in the picture. One possible HTTP answer for this error is 402 unprocessable entity.

If one plate is correctly found, then there is need to get the street name and number from the coordinates received from the POST request. This is done by sending the coordiantes to the external **Reverse Geocoding** API which returns the name and the road number. We don't store the pictures in the database, rather the use case moves the picture of the violation on a specific directory of the filesystem and then creates the document containing the path of the picture and the metadata. This is stored in the database by the **MongoDB driver**. The HTTP answer sent to the mobile app in this case is 201 created.

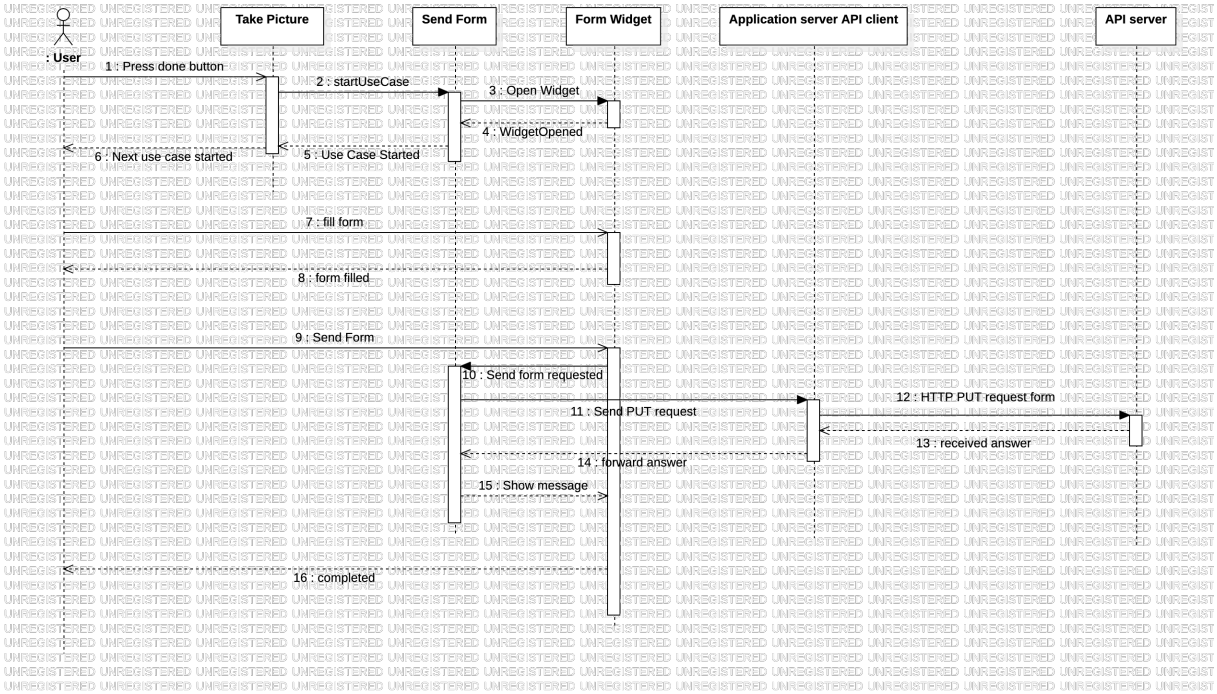

Figure 6: Sequence diagram for filling form on Mobile App

In figure 6 is shown the sequence diagram for the use case of filling the violation form in the Mobile App. After the User has done the take and upload picture use case, the next use case **Send Form** is started. This use case requires the loading of the **Form Widget** whic shows the list of possible types of kind of violation. The user fills the form and the widget communicates to the use case the choice made. This selection then is communicated to the Application Server with a PUT request in order to update the document containing the violation document with an attribute containing the kind. This is done by the **Application server API client** using the following endpoint: PUT /API/v1/violations/upload/form/id=xxx.

Copyright © 2019 Tiberio Galbiati - Saeid Rezaei

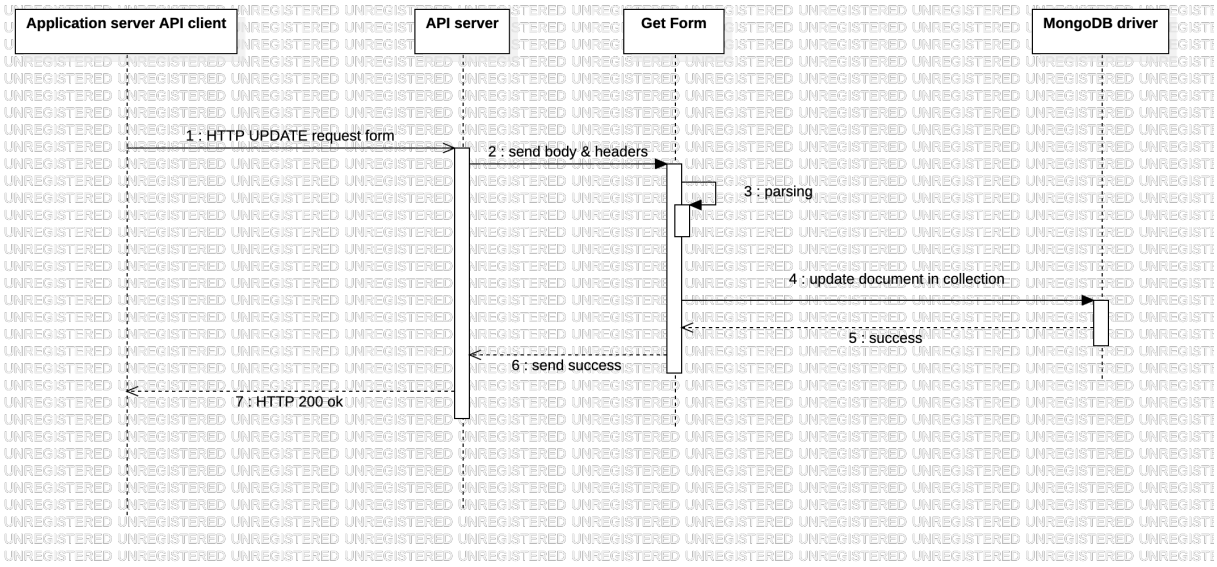

Figure 7: Sequence diagram for getting the form by Application Server

In figure 7 is shown the sequence diagram for the use case of getting the content of the filled form about the kind of violation by the Application Server. When the **API server** receives a PUT request at the following URL /API/v1/violations/upload/form/id=xxx it has to forward the content to the use case **Get Form** which parses the content and updates the document stored in the databse about the violation, completing the attribute about the kind of violation. The HTTP answer sent to the mobile app after the success of this case is 200.

2.4.2 Streets Heatmap

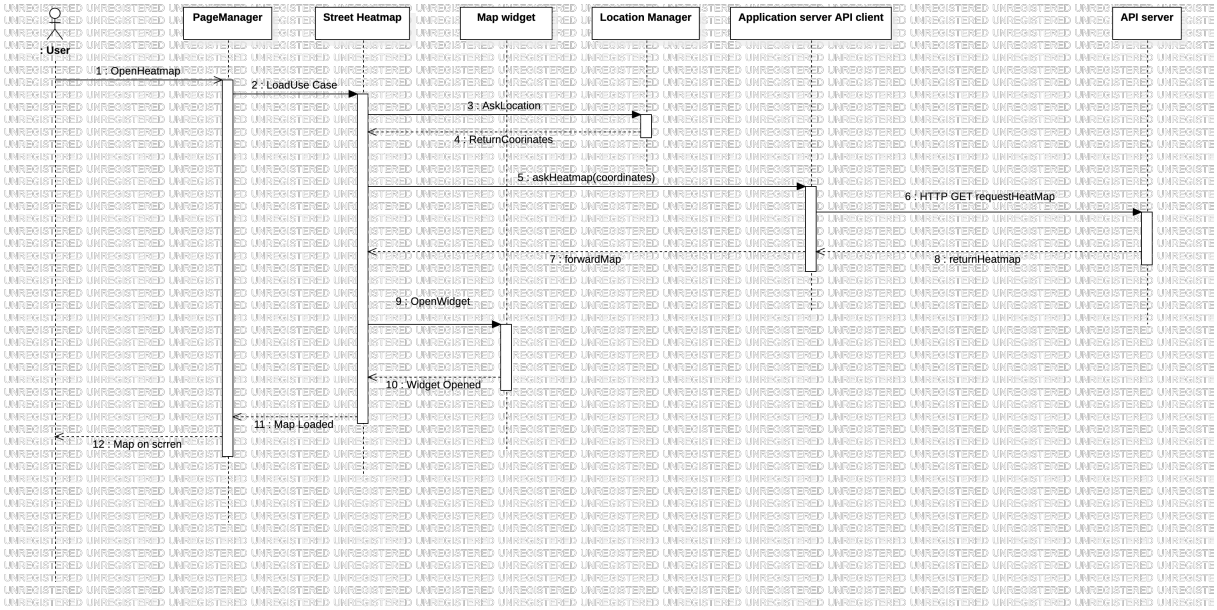

Figure 8: Sequence diagram for showing Heatmap on Mobile App

In Figure 8 is shown the sequence diagram for the use case of showing the Heatmap of violations. The user interacts with the **PageManager** asking to open the section with the Heatmap, with the result that the use case **Street Heatmap** is loaded. This use case first has to use the internal

component interface with the controller in order to know the location of the user. This is done by calling the **Location Manager**. Then it has to interact with the API in order to retrieve the actual map with the following request: GET /API/v1/heatmap/lat=xxx&long=xxx. Once the map is returned the **Map Widget** can be opened.

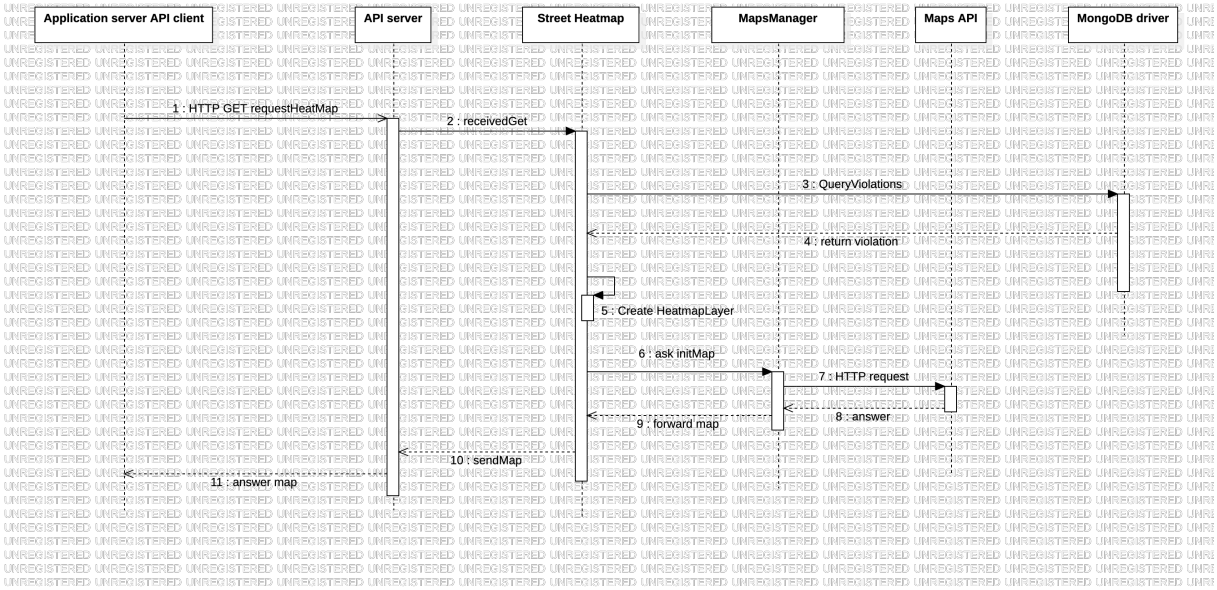

Figure 9: Sequence diagram for creating Heatmap on Application Server

In figure 9 is shown the sequence diagram for the use case of creating the Heatmap on the Application Server. Once the server receives a GET request from the endpoint devoted offer the Heatmap (example: /API/v1/heatmap/lat=xxx&long=xxx) the first thing that it does is querying the database containing all the violations, getting the coordiantes of each violation reported. The Heatmap Layer is part of the google.maps.visualization library, which must be loaded. Using the data from violations it's now possible to create a HeatmapLayer object containing the latitude and longitude of every violation that must be visualized [7]. After instantiating the HeatmapLayer object it has to be added to the map by calling the setMap() method of the Google Maps API [7]. All the data of the map must be than forwarded to who made the GET request. Using this logic we are in fact "mirroring" the Google Maps API on our Application Server. This approach has the following advantage: the mobile APP has only to interact with our Application Server which manages all the map creation (using the external service). So on the mobile app there is no nead to add a dedicated controller for the Maps service.

Copyright © 2019 Tiberio Galbiati - Saeid Rezaei

2.4.3 Vehicle list by violations

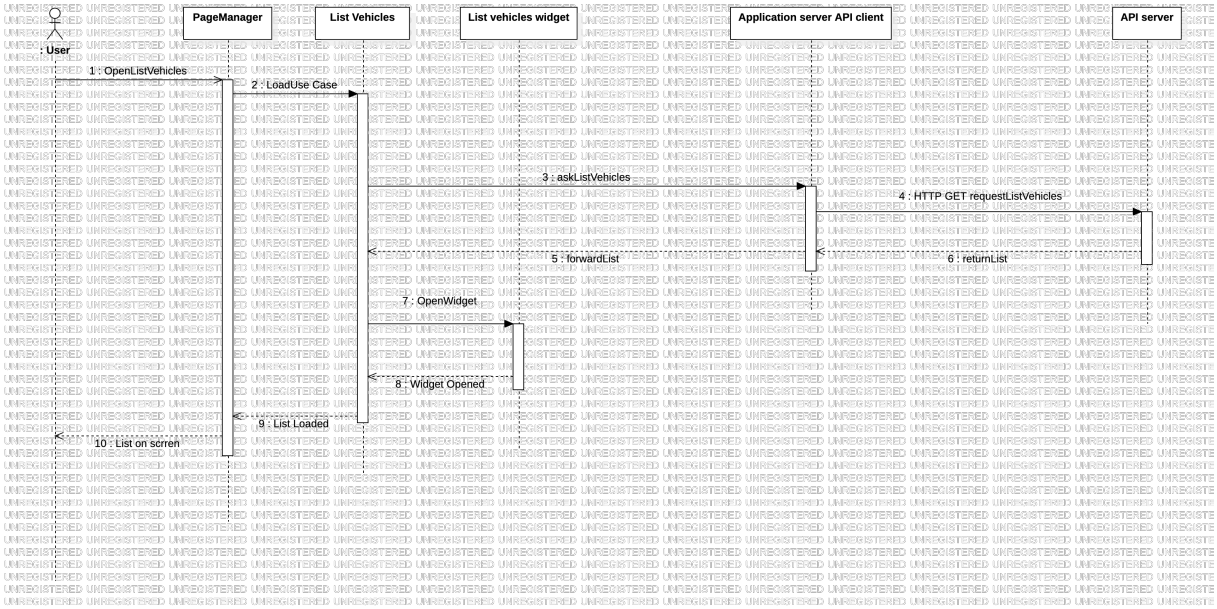

Figure 10: Sequence diagram for showing list of vehicles ordered by violations in Mobile App

In Figure 10 is shown the sequence diagram for the use case of showing the list of plates ordered by number of violations. The user interacts with the **PageManager** asking to open the section with the list of violators, with the result that the use case **List Vehicles** is loaded. This use case has to interact with the API in order to retrieve the list of vehicles associated with the count of violations. This is dove with a GET request that must also specify the number of records than needed to be retreived. As for example: GET /API/v1/violations/plates.json?countby=violations&sort=desc&depth=x. So the use case calls the **Application server API client** which performs the actual call to the REST API. Once a JSON file containing the data required is returned by the API, the use case can open the **List vehicles widget** responsible of showing on the app the list.

Copyright © 2019 Tiberio Galbiati - Saeid Rezaei

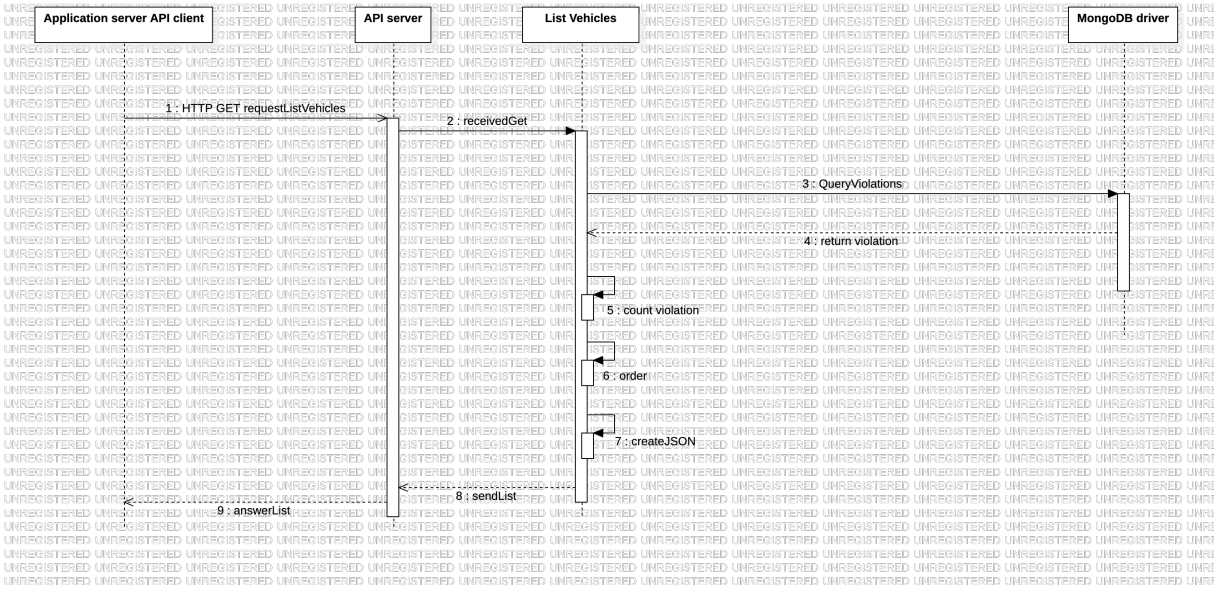

Figure 11: Sequence diagram for getting list of vehicles ordered by violations in Application Server

In Figure 11 is shown the sequence diagram for the use case of getting from database the list of vehicles that committed the most violations, ordering and then offering the API endpoint. Once the server receives a GET request from the endpoint devoted to the list of violations (example: /API/v1/violations/plates.json?countby=violations&sort=desc&depth=x) it has to query the database of violations and count the number of violation for each unique plate. Then it must sort them in descending order and creating a JSON file that must be returned to who called the API. The numner of record returned was specified in the GET request.

Copyright © 2019 Tiberio Galbiati - Saeid Rezaei

2.4.4 Ticket Creation

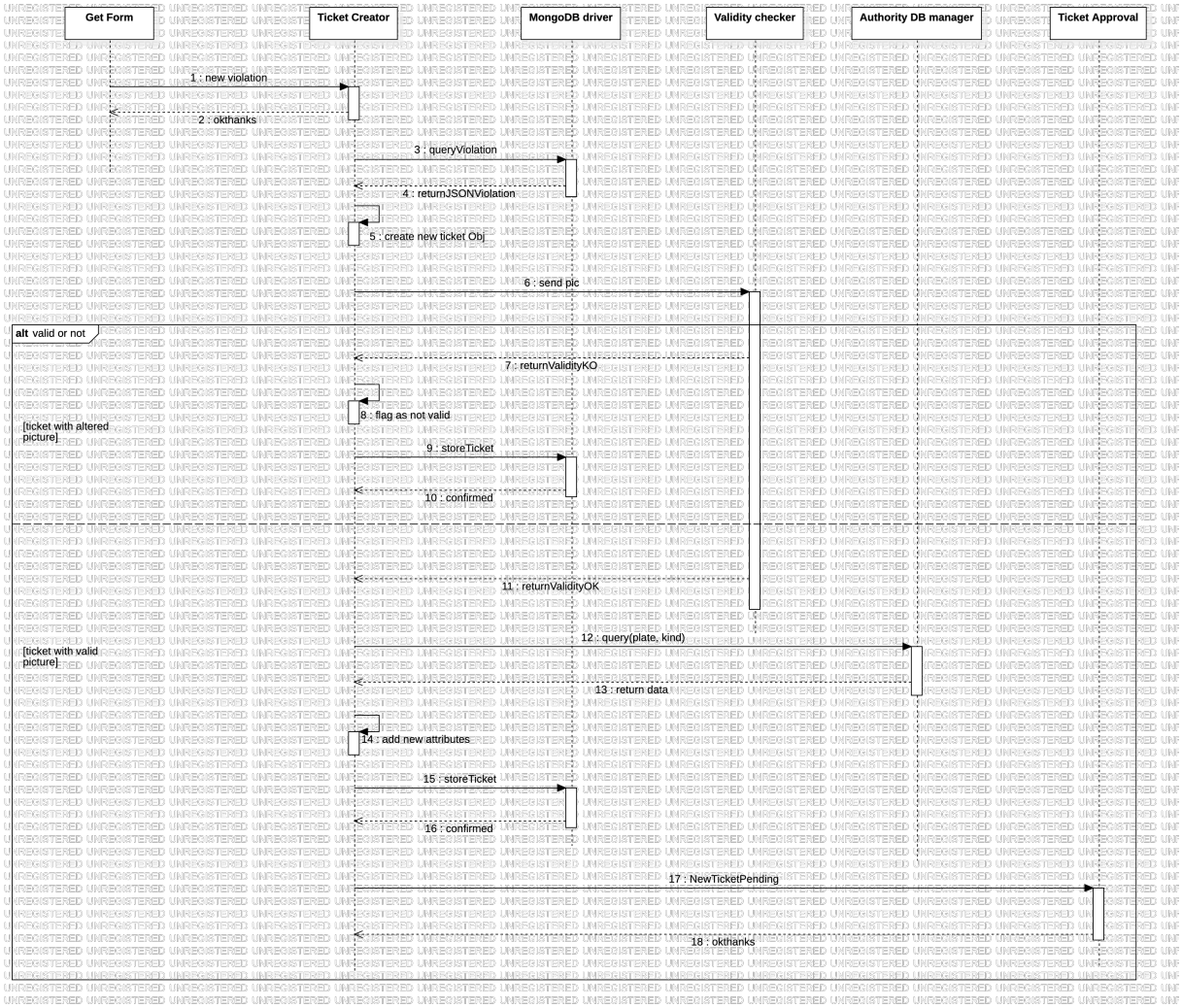

Figure 12
Sequence diagram for ticket Creation in Application Server

In Figure 12 is shown the sequence diagram for the use case of creating a ticket from an inserted violation in the Application Server. When a new violation has been successfully inserted, the use case **Get Form**, sends a message to the use case **Ticket Creator** informing that a new violation is available in the database so it's possible to start creating the corresponding ticket. Now the **Ticket Creator** knows the ID of the new inserted violation so it can query the Database for that specific violation. Then it uses the queried data of the violation to create a new Ticket Object. To check if the picture of the violation hasn't been altered it passes to the component **Validity checker** the path of the picture so it can be sent to the external service. Once the answer is received, we have to alternetives. If the picture is considered fake, the ticket attribute valid is set to FALSE and the ticket is stored in the database for debu purposes or for cheater identification which can be later implemented. If the picture was authentic the ticket attribute valid is set to TRUE. Then there is need to complete some attributes of the ticket quering the external Authority database. This is done by calling the component **Authority DB manager**. These attributes are: the name, surname and address of the owner of the violator, the amount of money to be paid and the deadline of payment for that kid of violation. After the new attributes are added to the Ticket object, it is stored in the database. Lastly a message to the use case **Ticket Approval**

Copyright © 2019 Tiberio Galbiati - Saeid Rezaei

is sent, telling the ID of the new isnerted ticket.

2.4.5 Ticket Approval

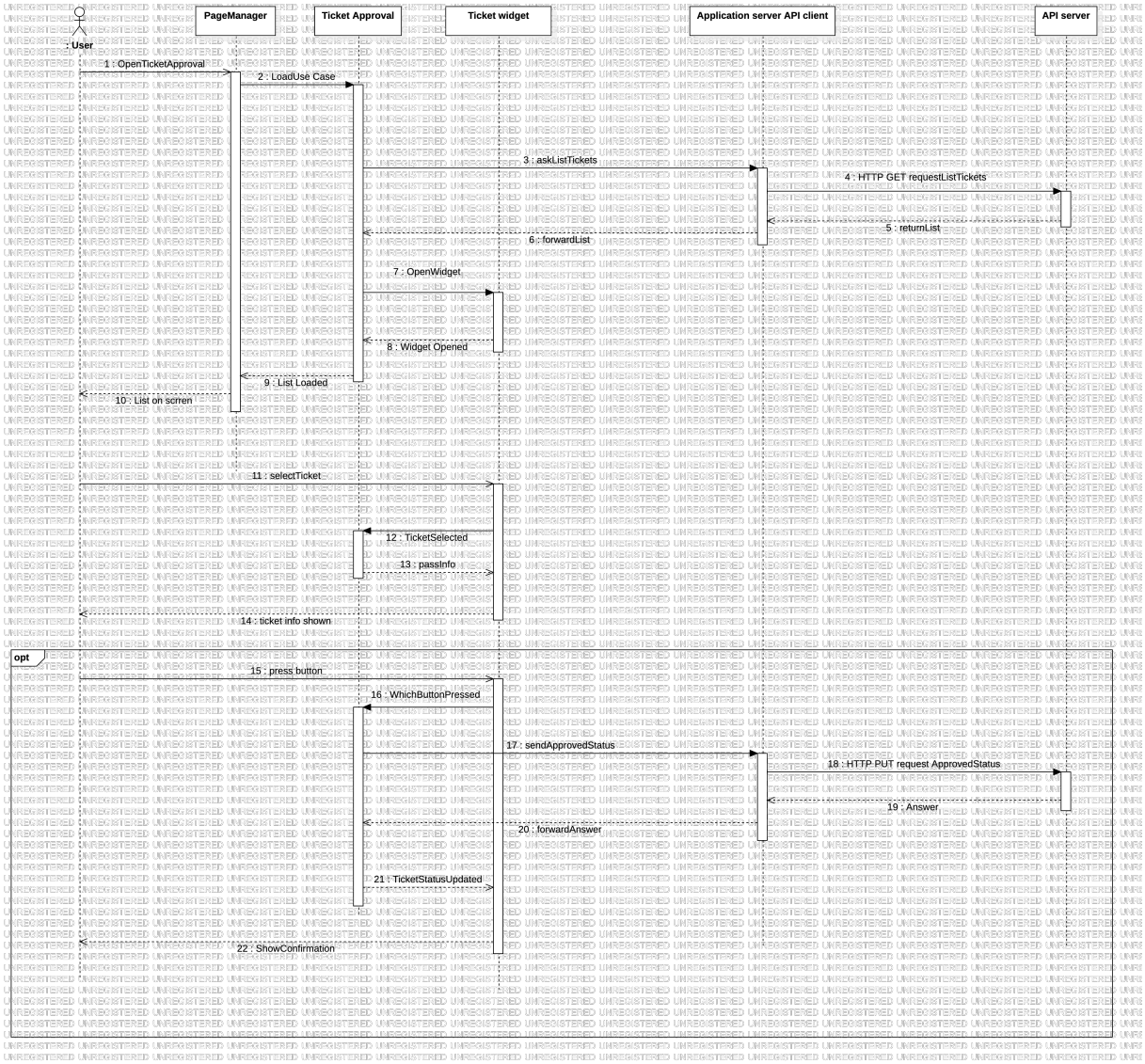

Figure 13
Sequence diagram for Ticket Approval on the Mobile Application

In Figure 13 is shown the sequence diagram for the use case of approving a ticket on the mobile Application. The user interacts with the **PageManager** asking to open the section of ticket approval, with the result that the use case **Ticket Approval** is loaded. The **PageManager** only offers this function to the authority users, as stated in the RASD, this is a function only for policemen. This use case has to interact with the API in order to retrieve the list of pending tickets. This is dove with a GET request. As for example: GET /API/v1/tickets/pending.json. So the use case calls the **Application server API client** which performs the actual call to the REST API. Once a JSON file containing the data required is returned by the API, the use case can open the **Ticket widget** responsible of showing on the app the list. If the user selects a ticket, the widget sends a message to the use case with the ID of the ticket and receives back the information relative to that one. So the deatil of the ticket is shown on screen with a button for approval or

Copyright © 2019 Tiberio Galbiati - Saeid Rezaei

discarding. When one of those buttons is pressed, the **Ticket widget** communicates to the use case which selection has been made by the user. This selection then is communicated to the Application Server with a PUT request in order to update the document containing the ticket with an attribute to determine if has been approved or not. The choice made is sent in the body of the following request: PUT /API/v1/tickets/ticketid

2.5 Component interfaces

In our architecture we have two main interfaces between the Use Cases and the other components: one is the **Use Cases Output interface** devoted to interface the use cases with the Presenter and the **Use Cases Input interface** devoted to interface the use cases with the Controller. We choose to keep the methods of those interfaces very generic, so thay can be used by different Use cases, actually the methods describe in most cases only the kind of data tha needs to be exchanged so can be adapted for different purposes. We use heavily JSON documents so the structure of the data is very flexible.

2.5.1 Mobile App

In the Mobile App the **Use Cases Output interface** is used to communicate with the UI. So we have listed some methods used to interact with the widgets of the UI which are needed to load, close, and display strings and other kind of data to the widgets. Even though the name is "Output interface" it's still possible to use this interface to get data input by the user in the UI tha needs to be passed to the use cases. The **Use Cases Input interface** is used to communicate with the adapters of the controller, to perform the calls to the Server API and to interact with the device physical sensors. In Figure 14 are listed some possible methods of the interfaces of the Mobile App. More can be added during implementation whithaut affecting the one already existing and the Use cases.

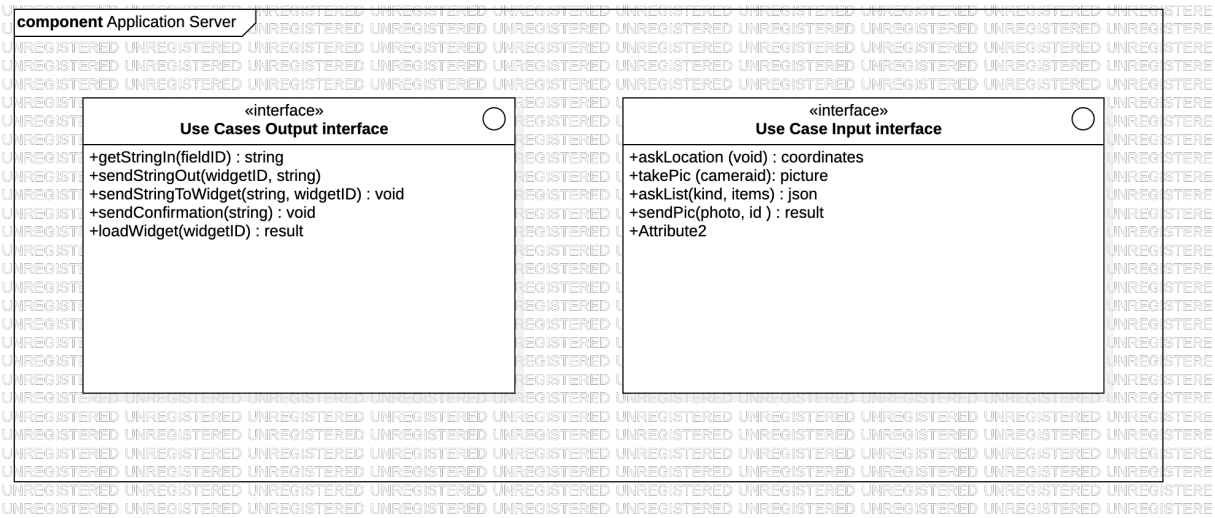

Figure 14: Mobile Application Component interfaces

2.5.2 Application Server

In the Application Server the **Use Cases Output interface** is used by the use cases to provide the required API. The **Use Cases Input interface** is used to communicate with the adapters of the controller, to perform the calls to the externals API and to interact with the database.

Copyright © 2019 Tiberio Galbiati - Saeid Rezaei

In Figure 15 are listed some possible methods of the interfaces of the Application Server. More can be added during implementation whithaut affecting the one already existing and the Use cases.

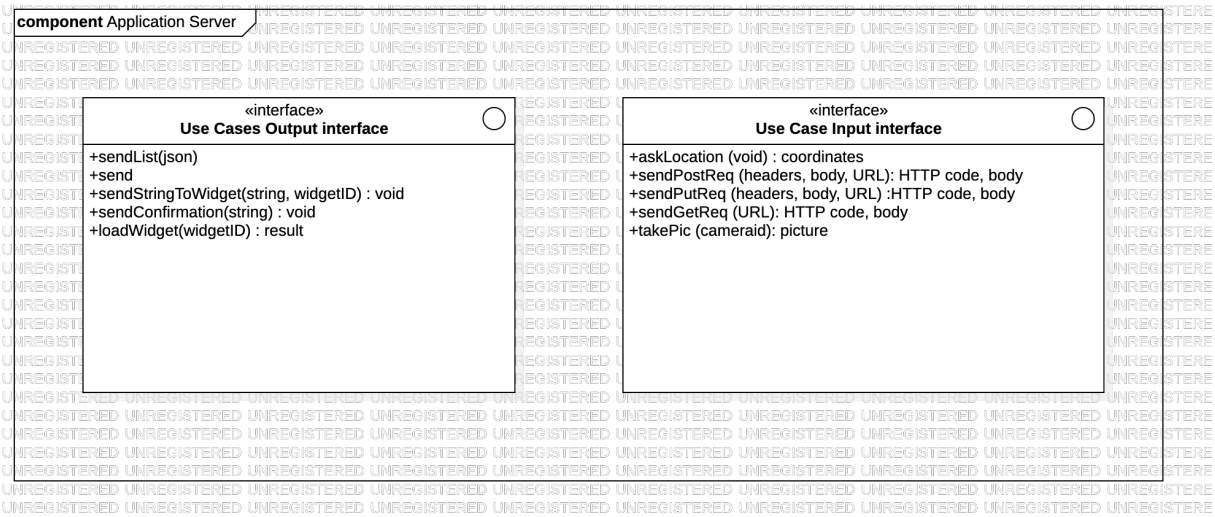

Figure 15: Application Server Component interfaces

2.6 API interfaces

The Application Server has to provide a RESTful API. Here we list the URI of the endpoints the Server must offer:

- /API/v1/violations/
- /API/v1/heatmap/lat=xxxlong=xxx
- /API/v1/violations/upload/id=xxx
- /API/v1/violations/upload/form/id=xxx
- /API/v1/violations/plates.json?countby=violationssort=descdepth=x
- /API/v1/tickets/pending.json
- /API/v1/tickets/ticketid

The description of those is given in the Runtime View section.

2.7 Selected Architectural styles and patterns

2.7.1 Three-tier client/server architecture

Our system is based on a client/server architecture with three layers. Presentation, application and data access are physically separated repectively in Mobile App, Application Server and Database server, as shown in the Deployment View 2.3.

We choose this kind of architecture in order to have a modular system, whick keeps the client application very light and all the heavy computation is done on the server side. Moreover we separated physically the database from the application server, to increase scalability and security, and it's also possible to use a distribuited database.

Copyright © 2019 Tiberio Galbiati - Saeid Rezaei

2.7.2 Clean Architecture

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 Client (Mobile application) and in the Application Server.

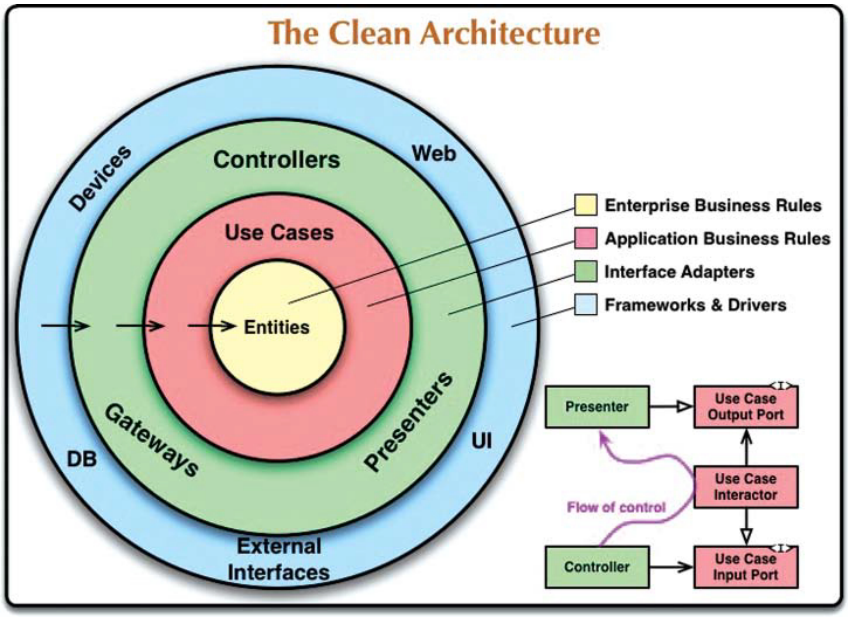

Figure 16: Clean Architecture [1] p. 203

Dependency Rule "The concentric circles in Figure 16 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]

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 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]

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, 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]

Copyright © 2019 Tiberio Galbiati - Saeid Rezaei

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]

Adantages of Clean architecture Here we present some reasons why we choose the Clean Architecture

- Very clear dependencies between components
- Modularity of having clear use cases as components.
- Separation of presentation layer from logic. The use cases don't depend on the UI, so it' possible to change the presentation layer without touching the core of the system
- Separation of interfaces from use cases. We have a dedicated layer with the code that acts as a bridge between the outside world and the use cases and entities. So it's possible to change the external services without touching the core of the systems. The same goes for the database: it's possible to change the DBMS just changing the relative adapter, without touching the use cases.
- Easier testing, due to the modularity of the system

2.7.3 REST

The communication between the Mobile application and the Server 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. Here we list the main REST principles:

- **Client/Server**: There is separation of concerns between client and server. The server stores and manipulates information and makes it available to the client. The client takes that information and displays it to the user. This separation of concerns allows both the client and the server to evolve independently.
- **Stateless**: That means the communication between the client and the server always contains all the information needed to perform the request. There is no session state in the server. this means for example thatthe client needs to authenticate itself with every request in case the access to a resource is restricted and requires authentication.
- **Cacheable**: The client, the server and any intermediary components can all cache resources in order to improve performance.
- **Uniform interface**: This simplifies the architecture, as all components follow the same rules to speak to one another.
- **Layered system**: Individual components cannot see beyond the immediate layer with which they are interacting. This allows components to be independent and thus easily replaceable or extendable.

2.8 Other design decisions

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

Copyright © 2019 Tiberio Galbiati - Saeid Rezaei

Flutter Flutter is a free, open-source mobile SDK that can be used to create native-looking Android and iOS apps from the same code base. Flutter helps app developers build cross-platform apps faster by using a single programming language, Dart, an object-oriented, class defined programming language. We chose to use this Framework in order to have a mobile app that works both with iOs and Adroid, with performances similar to the native development for each platform but having to write only once the code. In Flutter, every piece of the user interface is a widget like text, buttons, check boxes, images. There are also Container widgets thac contain other widgets. Widets can be stateless, which are immutable, or stateful, which have a mutable state and are used when we are describing a part of the user interface that can change dynamically.

Lastly we want to remember that Flutter is not the only framework that can be used to buld cross-platform mobile apps, another oprions can be React Native, based on JavaScript. We decided to use Flutter because of it's advantages in comparision with React Native and other UI software development kit, which are better performances and increasing popularity due to it's simplicity of develop.

Node.js Node.js is an open-source, cross-platform, JavaScript runtime environment that executes JavaScript code outside of a browser. It is the most used server side runtime environment. We use Node.js with Express, which is a web application server framework, designed for building single-page, multi-page, and hybrid web applications. It is the de facto standard server framework for Node.js which simplifies development and makes it easier to write secure, modular and fast applications and APIs.

MongoDB MongoDB is a noSQL distributed database which allows ad-hoc queries, real-time integration, and it's indexing efficient. It represents the data as a collection of documents rather than tables related by foreign keys. Those documents are collected in collections and can have different schemas. We choose to use MongoDB because it's so well intergrated with Node.js, because there are very useful libraries such as Mongoose whick make easier to communicate with the DB.

Copyright © 2019 Tiberio Galbiati - Saeid Rezaei

3 User Interface Design

uiiiiii

Copyright © 2019 Tiberio Galbiati - Saeid Rezaei

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	Take Picture, Get Picture
[R4]	We should require the user to send again a picture in case the plate is not visible	Take Picture, Get 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	Take Picture, Get Picture
[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 vehicles
[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	List vehicles, CoreUtils
[R13]	Authority users can know the exact licence plate when mining data about offenders	List vehicles, CoreUtils
[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

Copyright © 2019 Tiberio Galbiati - Saeid Rezaei

5 Implementation, Integration and Test Plan

This section will discuss the implementation, integration and testing of the SafeStreet application. The section aims to prevent any setbacks entailing changes on the project once started its development and do multiple tasks twice. First of all, it is important to divide the project into smaller components in order to have more concrete goals that help keep the developers' motivation high and make a straightforward development of the project. Therefore, we will divide the project in dfferent components, starting from the ones that are responsible of the basic features and going on until the end, where we will develop the most specific ones. It is not a new method, because it has been studied during this course and it is referred to as 'bottom-up' strategy. This method will help providing a better integration of the project tier-by-tier and make different tests of the behaviour of the application before it ends. The System can be divided into three main parts as following:

- **Mobile Application**
- **Application Server**
- **External components**

External components contains services such as the ALPR, to extract the license plate from a photo, Map API and so on. It should be noted that the external systems' components need not to be implemented and tested just because they are external and they can be considered reliable. Going back to our project, we will divide the Mobile App components and Application Server components to three main entities, corresponding to User, Violation and Ticket. The services that have to be created for each entity are:

- *User*: Sign-UP, Login
- *Violation*: Take Picture, Fill Form, Violation Info, List Vehicles, Street Heatmap, Mine information.
- *Ticket*: Ticket Approval, Statistics, Ticket offenders, Trends.

In figure 17, the overall process structure of the implementation has been shown. First, the blue part, will be implemented and tested. After the blue part works e ciently, the orange part, services related to report a violation, will be implemented and tested. Finally, if the blue and orange part will work e ciently, the final green part, Ticket, will be implemented and tested. During the implementation and testing process, we will make the component integration once we have finished each tier and tested each isolated component, meaning we will mix the different phases and try to detect possible defaults in our design.

As mentioned in 17, we will start with developing the User component. The Sign-Up and Login components are clearly an entry condition for the right functioning of the system, but they are not the main features and they are not very compicated. These two components can be implemented and tested at any order. After Implemeting and testing User component, we will continue with the next part of the implementation of the software which is Violation component. We will integrate all the services and test whether the whole Violation component will work as it is expected. If the result is positive, we will continue with the next last of the implementation of the software. If the result is negative, we should study if something is wrong in our code or if we should make any changes in the project development.

5.1 Sequence of component integration

The diagrams below describe the sequence of the component implementation, integration and testing of the system. The arrows starts from the component which uses the other one.

Copyright © 2019 Tiberio Galbiati - Saeid Rezaei

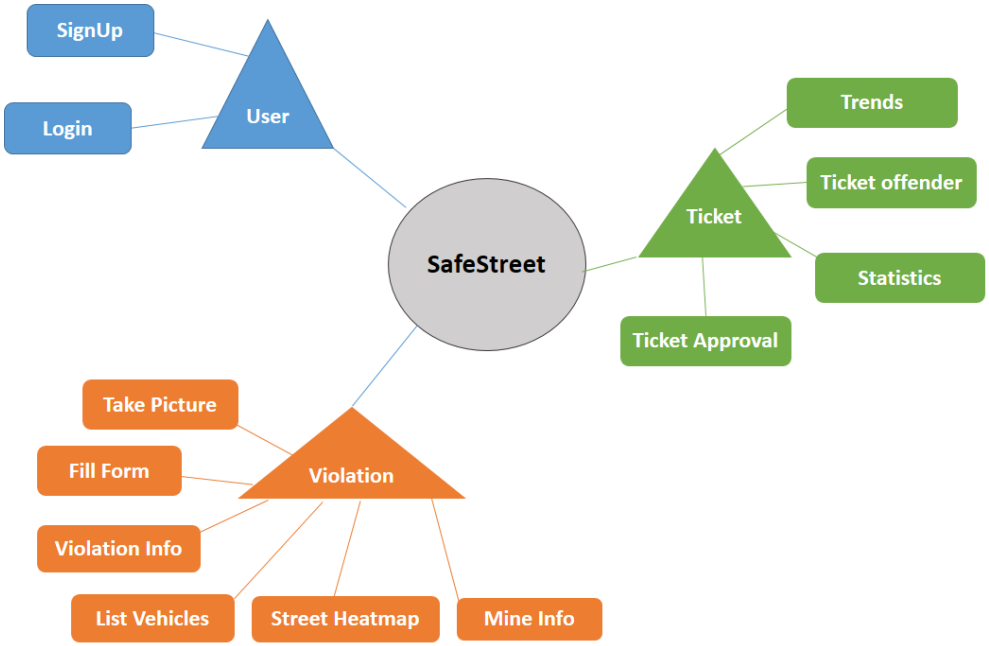

Figure 17: A general overview of the service implementation

Integration of the all components of the Mobile Application Firstly all the Mobile Application components will be implemented and tested with unit tests. Then the integration process will proceed and the integration is tested as well. All the Use Case Interactor should be integraterd with their corresponding Presenter and Contoroler. In the following integration between main components are shown.

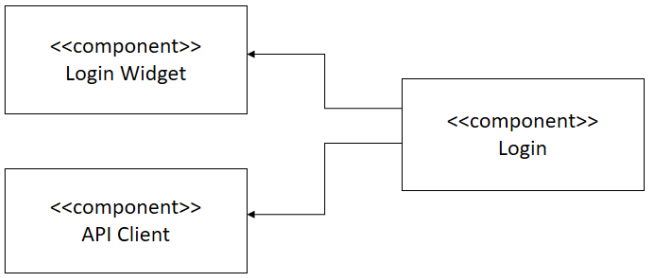

Figure 18: Mobile application login integration

Copyright © 2019 Tiberio Galbiati - Saeid Rezaei

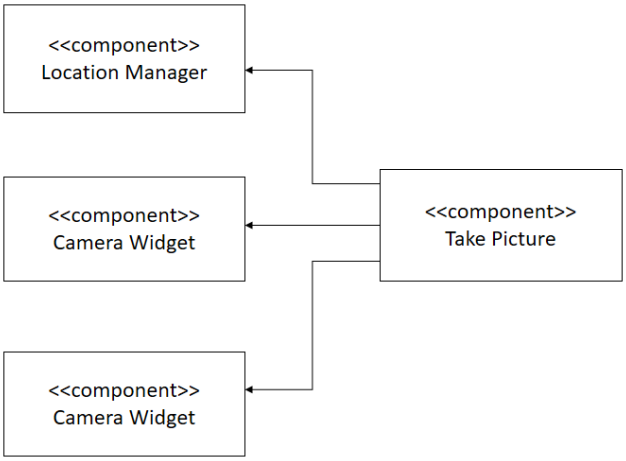

Figure 19: Take picture integration

Figure 20: Send form integration

Copyright © 2019 Tiberio Galbiati - Saeid Rezaei

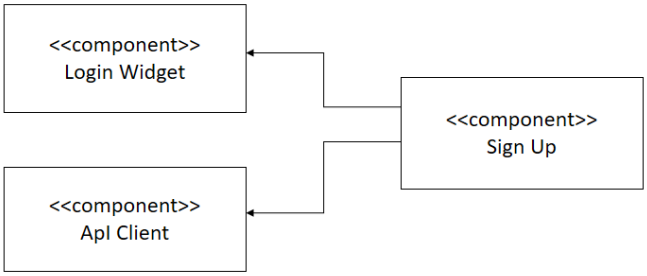

Figure 21: Mobile application sign up integration

Integration of the all components of the Server Application When all the Mobile Application components implemented, tested and integrated, we should do the the same for Server Application components. In this part all the Use Case Interactor should be integraterd with their corresponding Presenter and Contoroler. In the following integration between main components are shown.

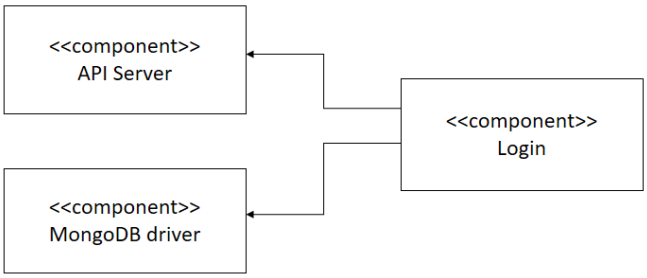

Figure 22: Server login integration

Copyright © 2019 Tiberio Galbiati - Saeid Rezaei

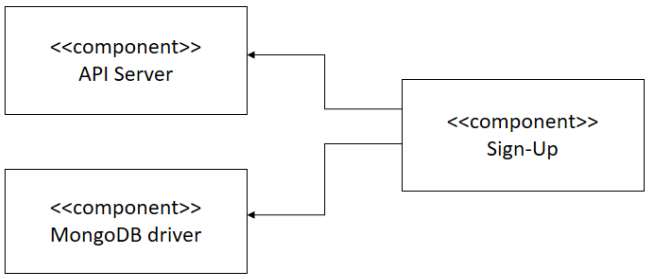

Figure 23: Server sign uP integration

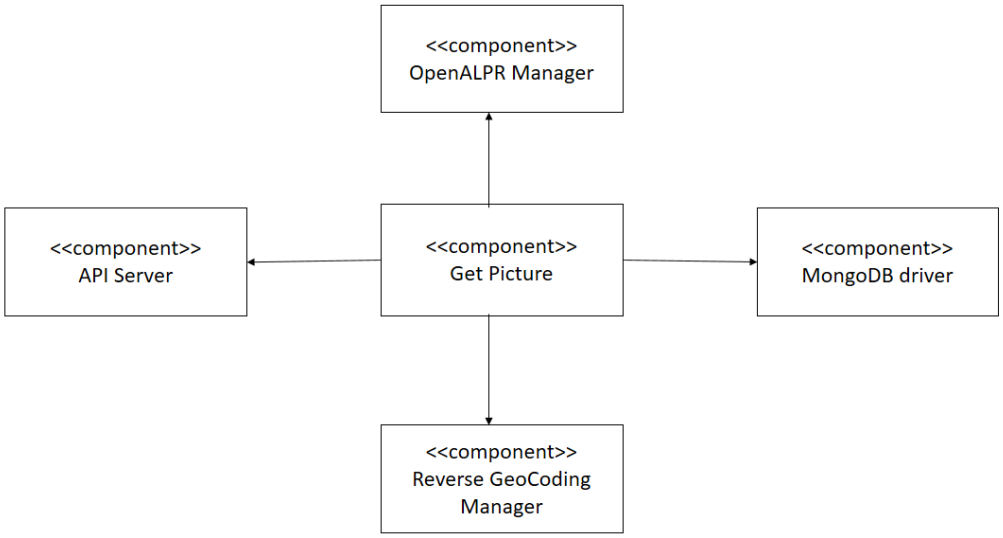

Figure 24: Server get picture integration

Copyright © 2019 Tiberio Galbiati - Saeid Rezaei

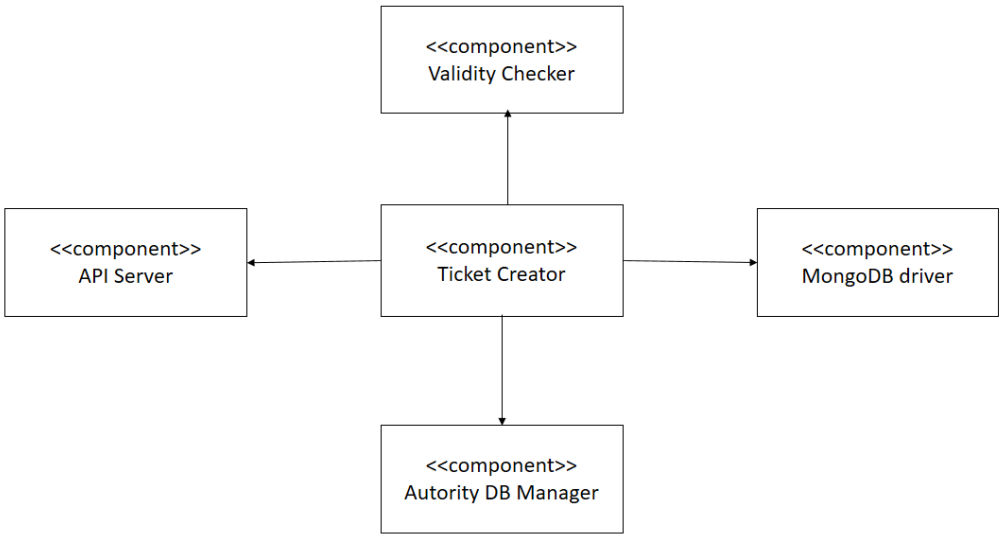

Figure 25: Ticket Creator integration

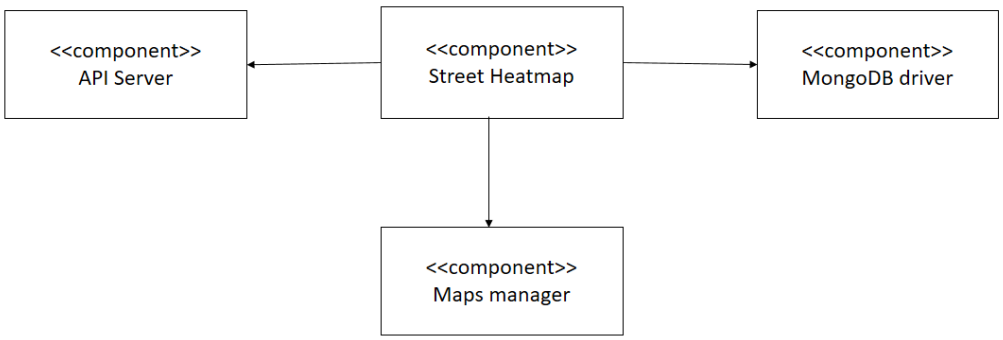

Figure 26: Street Heat map Integration

Integration of the frontend with backend Once all the components are implemented and tested among each others, the Mobile Application will be integrated and tested within the Server Application.

Copyright © 2019 Tiberio Galbiati - Saeid Rezaei

Figure 27: Integration of Mobile Application and Server

Integration of all subsystems Once theMobile Application has been integrated with the-Server side, the system will be fully tested, with the respect of all the components shown in the Component Diagram.

Copyright © 2019 Tiberio Galbiati - Saeid Rezaei

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
meeting	1h 30 min
Work on sequence diagrams	2h
User case components	3h
Sequence diagrams and explanation	2h
Sequence diagrams new	3h
meeting	1h
Sequence diagrams ad API	2h 30 min
Sequence diagrams and typos	2h
All doc review and interfaces	3h
Interfaces design	3h
Total	**17 h**

Saeid	
Task	**Time**
Meeting design	1h 30 min
Study previous projects	2h
Study Architectural patterns	2h
Architectural styles and patterns	2h
Architectural styles and patterns	1h 30min
meeting	1h 30 min
Test Plan	1h 30 min
Integration and Test Plan	2h 30 min
Component diag analysis	1h 30 min
Integration diagram and explanation	2h
meeting	1h
Total	**36 h 30 min**

Copyright © 2019 Tiberio Galbiati - Saeid Rezaei