SmartCityAdvisor

POLITECNICO

MILANO 1863

ASSIGNMENT 2 – DD

Design Document

A Software Engineering II Project,

made by Luca Favano 899224

V1.0 - 12/05/2018

Table of Contents:

1 - Introduction

1.1 Purpose

The purpose of this Design Document (DD) is to show the architecture that lays below the application. It will show in detail which components are necessary, how they interacts together and the algorithms used in the application. This document will also provide details of the user interface chosen.

All the requirements previously shown in the RASD document will then be specified more in details.

This document is particularly useful to project managers, developers and testers.

1.2 Scope

SmartCityAdvisor is a server-sided system that offers an app that let Server and Clients interact together. This interaction requires an interface that can be easily accessible by any device that satisfy the given hardware requirements.

To develop this application, we need a system that is both flexible and easily maintainable, so that in the future it could be easily improved and functionalities could be added as planned by the SmartCityAdvisor organization. Particular care should be applied to the application architecture which code should use coding best practices and principles like encapsulation.

The main issue that will be discussed in this document is how to make all the system work together flawlessly, in fact the system should provide the services in a reliable and easy to access way while keeping everything synchronized using an Internet connection.

1.2.1 Goals

[G1] Limit the traffic accordingly to current CO_2 levels in Milan.

[G2] Warns the citizens of a change of the current status

[G3] Provide users with up-to-date information retrieved from the sensors

[G4] Manage the traffic accordingly to special events taking place

[G5] Simplify the process of finding and reserving an available parking spot

[G6] Providing Superusers the possibility to manage events and emergencies

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

- Sensor: A component of that communicates with the system at regular intervals providing useful information for the system to work correctly.
- Actuator: A component that allows the system to have an impact on the environment based on choices that can be taken automatically or manually,
- City Center: An area of Milan which characteristics need to be preserved and kept constant over the time.
- CO_2 Level: Current amount of CO2 particles detected by the sensor and measured in ppm (particles per million).
- Superuser: A user of the app that has higher privileges compared to a "normal" user which is a common citizen that downloaded the app.

1.3.2 Acronyms

- RASD: Requirement Analysis and Specification Document
- XMPP: Extensible Messaging and Presence Protocol
- HTTP: Hypertext Transfer Protocol
- DBMS: Database Management System
- UML: Unified Modeling Language
- IEEE: Institute of Electrical and Electronic Engineers
- DD: Design Document
- UX: User Experience

1.4 Revision History

- V0.2: "First Version", Released on 11/05/2018
- V0.3: "Updated First Version", Released on 11/05/2018
- V0.9: "Main structure completed, need refinements", Released on 12/05/2018
- V1.0: "Complete Version", Released on 07/06/2018

1.5 Reference Documents

- Specification Document: "Assignment-for-the-second-project.pdf"
- Alloy Specification: found on beep and google

- Past Project Examples taken from beep
- IEEE Recommended Practice for Software Requirements Specifications.
- RASD Document that is also part of the project

1.6 Document Structure

This document will be structured into five different sections:

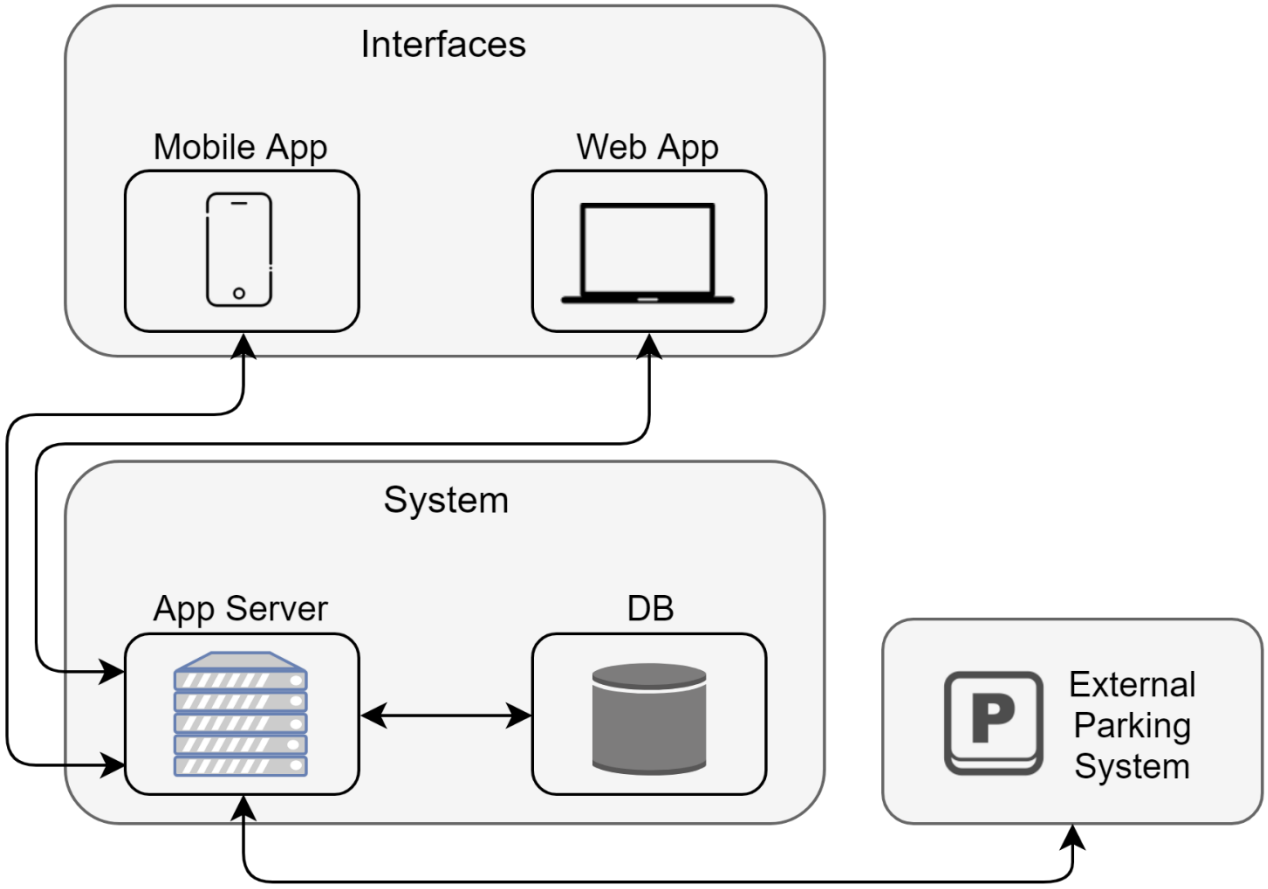
- Section 1: The introduction, a brief description of the project, the terms that will be used and some useful knowledge.
- Section 2: Architectural Design, it shows the main components of the system their design and how they interact. It includes the patterns and other choices made regarding the architecture.
- Section 3: Algorithm Design, a description of the most interesting parts of the algorithms needed to implement the system.
- Section 4: User Interface Design, a demonstration of how the user interface is going to look when completed, through mockups and UX diagrams.
- Section 5: Requirements Traceability, an explanation of how the decisions that have already been took in the RASD document now reflects to design elements included in this document (Design Document).
- Section 6: Implementation, Integration and Test Plan, aiming at identifying the planned order for the implementation and integration of the system components. Also including the testing of them.
- Section 7: Effort Spent + References, containing all the information used while writing the document and the time spent doing it.

2 – Architectural Design

2.1 Overview

This chapter focuses on the architectural structure, components will be described explaining how they interact with each other. The system will be illustrated both physically and logically

As a start a simple high-level representation of a part of the SmartCityAdvisor system that shows the interaction with the external parking system could be as follows:



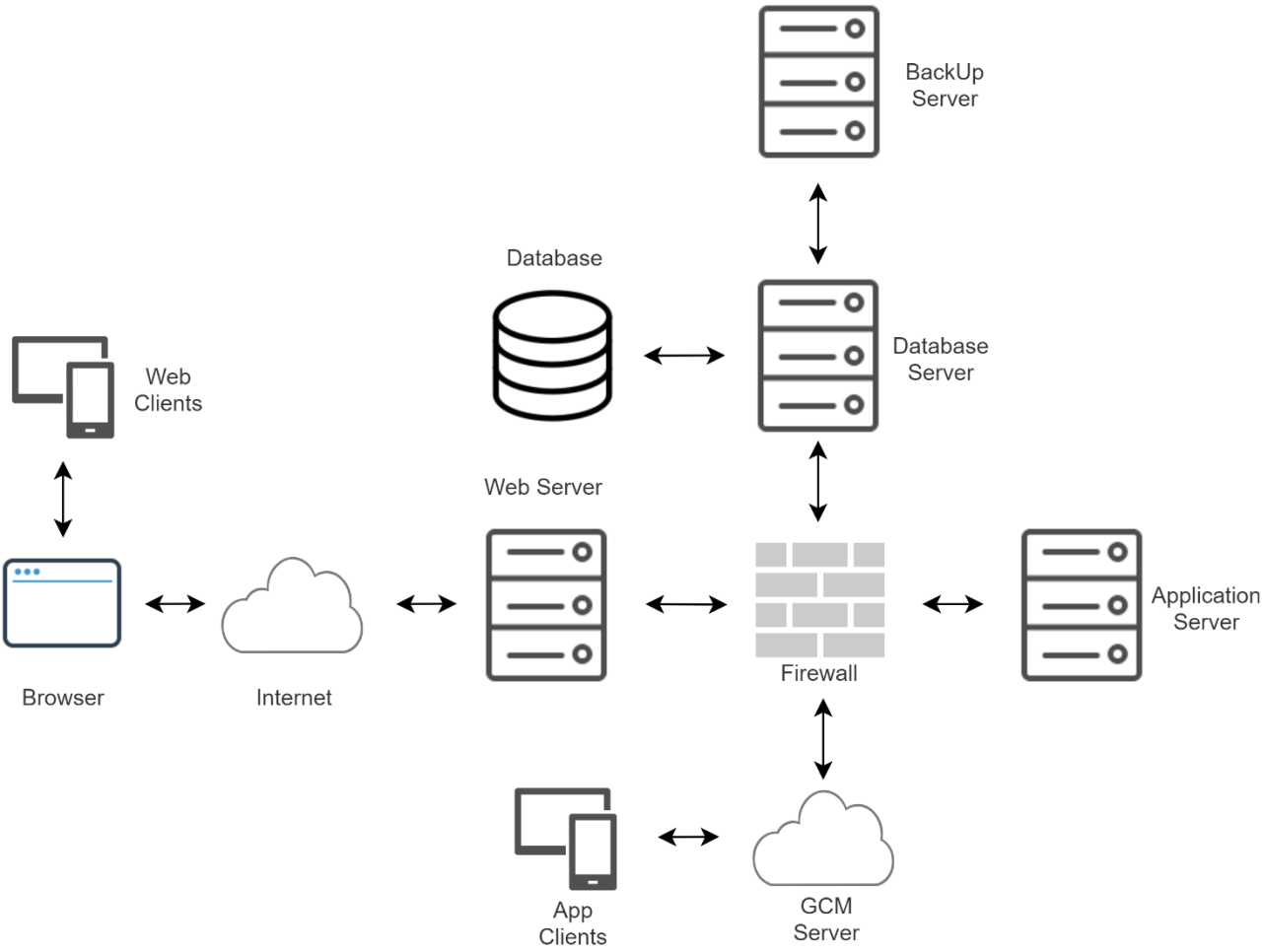
- **Interfaces:** Client-side part of the application, it's installed on user devices or loaded through a browser, it offers services that need the communication with the system.
- **Application Server:** Contains the server-side logic of the system, implements useful APIs and handle the client requests.
- **Database:** Is responsible of storing and retrieving asked data, it doesn't implement any logic and it is only used for data storing. DB guarantees the ACID properties.

The given simple overview is not detailed enough to describe accurately the whole system and its specifics will be addressed carefully in the following paragraphs.

2.2 Component View

First, we need to provide details about all the physical components of the system and this can be achieved thanks to the physical architecture diagram:

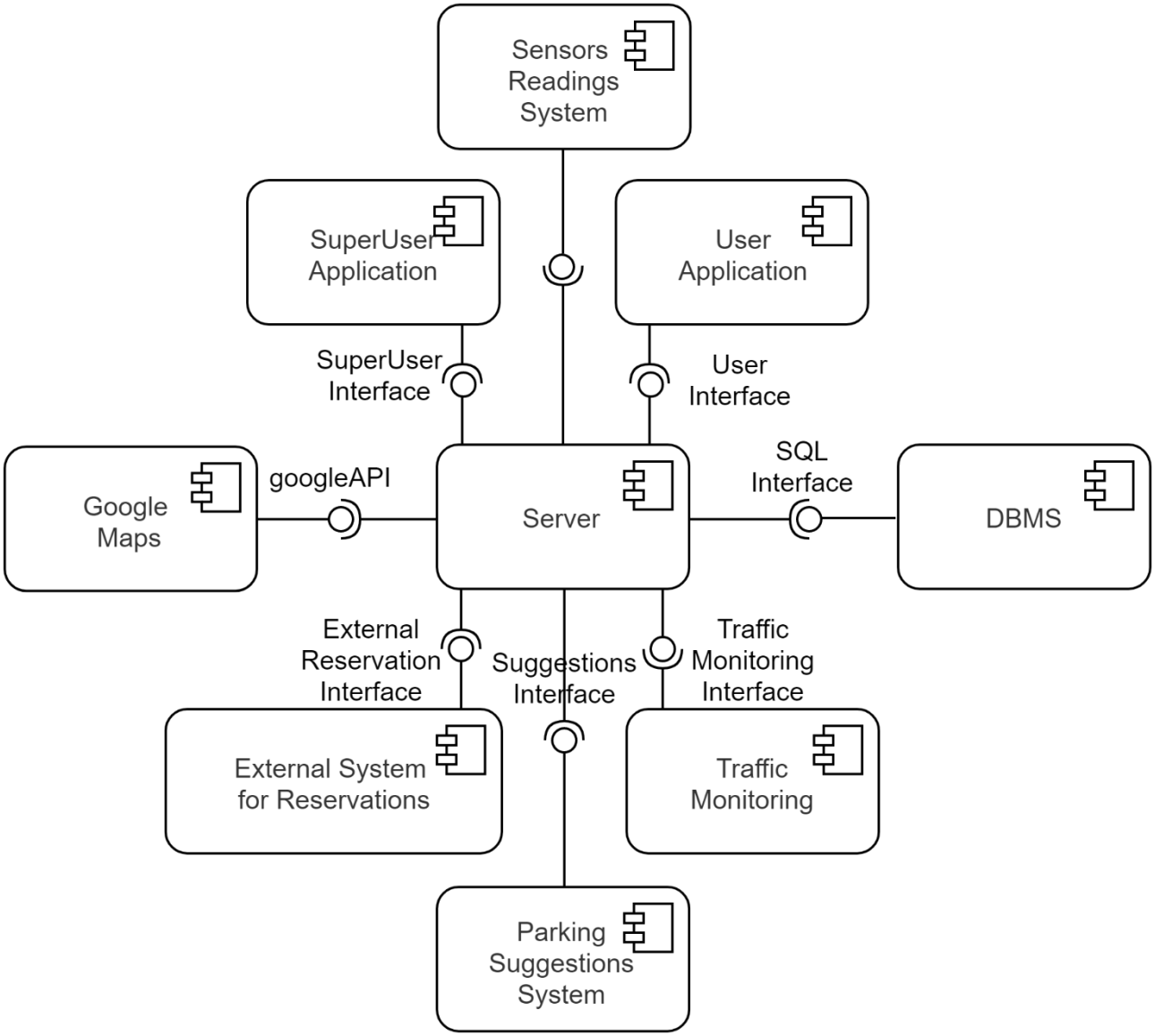
2.2.1 Physical Architecture Diagram



As shown in the diagram we need to introduce more components to our system:

- **Web Server**: A software running on a server that is able to communicate with the web application clients using the HTTP protocol.
- **Browser**: An application that retrieves web pages and presents them to the users.
- **GCM Server**: Google servers that take messages from an application server and sends them to a client app that is running on user's devices. Useful to handle app services like the notifications.
- **Firewall:** Software that protects the system from unauthorized access
- **Web/App Clients:** Two different clients that will interact with our system through a User Interface

Focusing on the interfaces provided by the server we obtain the following diagram that shows a representation of the logical architecture.



The obtained system includes different components that should be integrated:

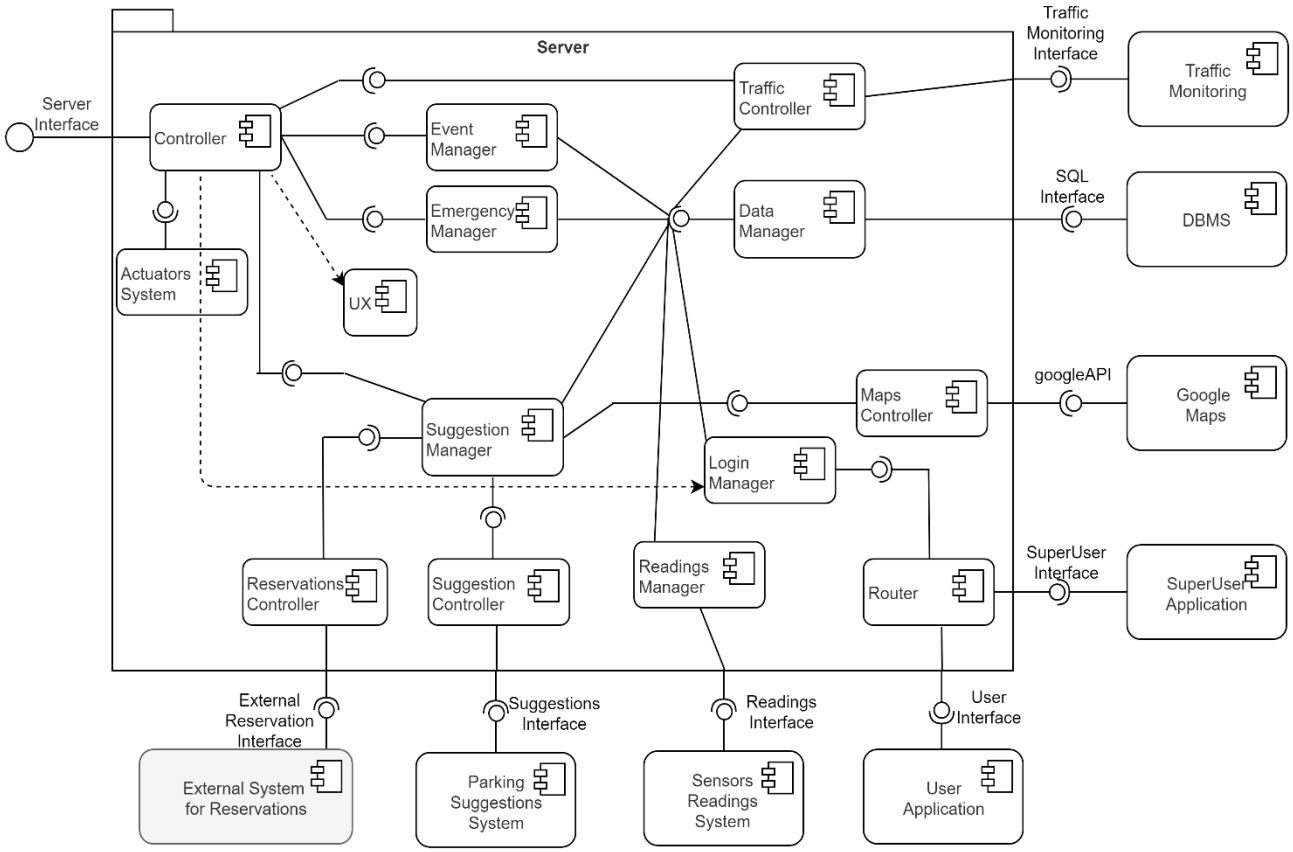
- Google Maps: A possible map system that is provided by Google and is accessible through google APIs. Useful to show to the user maps linked to the suggestions of parking lots.
- Database Management System (DBMS): Used to access all the necessary data in an efficient way.
- User/SuperUser Application: A component that takes care of handling all the functionalities that are available to the different types of users.
- Traffic Monitoring System: A component that models the traffic monitoring system parameters, allowing changes of system status.

- External System: A system not handled directly that allows our system to request reservations in parking lots available in Milan.
- Parking Suggestions System: A component that handle the suggestions when requested, the best possible choices should be elaborated.

Three logical level are still preserved (User Application / Server / DBMS).

2.2.2 Application Components Diagram

Lastly, we analyze the components that together form what we know as “server”.



Together with components previously already specified we can find few controllers and managers that handle all the information for the system to work properly.

The most relevant components that we just added are:

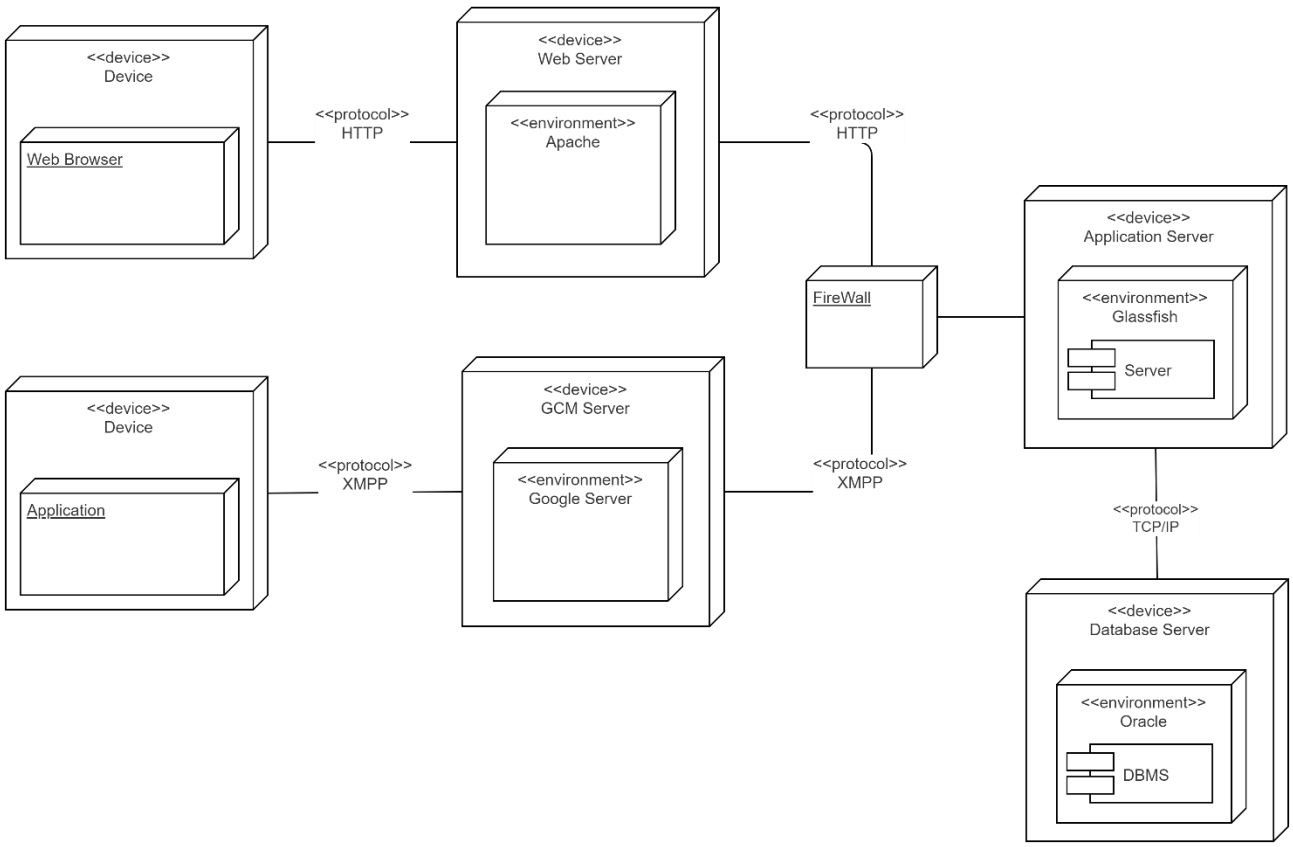
- A Data Manager: A component that provides methods for accessing the data of the system and care sending those data to the server so that they can be synchronized. It collects the data of other controllers.
- A Suggestion Manager: A component that has to interact with many others to collect and elaborate all the necessary information, allow a reservation only if the user has logged in.

- A Controller: A component that manages the User Interface providing all the necessary information.
- Router: A component that dispatches all the requests coming from the User/SuperUser application to a Login Manager that can handle them.
- Parking Suggestions System: A component that provides interfaces useful to the Suggestion Manager for it to work properly.

2.3 Deployment View

Next, we analyze how the system should be deployed, since the architecture should ensure the system security and maintainability Database and Application server run on two different machines, the Database Server also is covered by a backup server in case of any problem.

The GCM server interact with the application server and the application user devices using the XMPP protocol which can guarantee enhanced performances and is more flexible for future updates. The GCM server is also used to grant the system the possibility to send notifications to the app users.

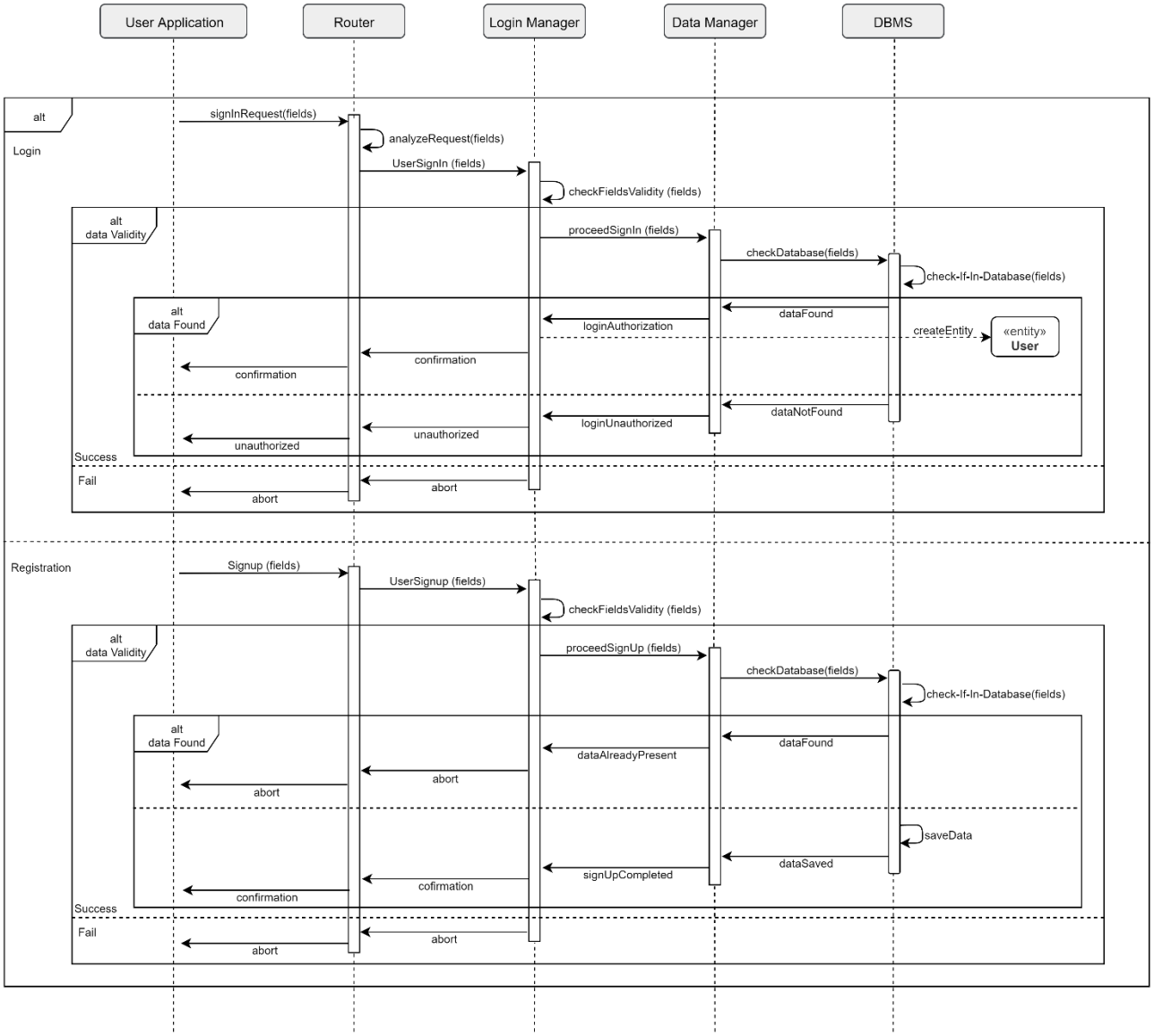


2.4 Runtime View

The most interesting dynamic behaviors instances of the system will be shown in this section. To show the most meaningful scenarios these instances may differ from the ones already analyzed in the RASD document.

2.4.1 User Registration and Login

First diagram shows the necessary flow of information after the actor (in this case a Guest) send a Sign-In request to the system. Such request contains some fields like the username, password, an extra parameter if the system has to remember the username and a hidden parameter that is automatically retrieved by the system.
The hidden parameter is a way that has the system to recognize if the username of the actor belongs to a normal User or a Superuser (superusers use special characters in the username).



In this case the request come from the User Application and is then handled by the router that classify it as a normal User request and then send it to the Login Manager.

The Login Manager need to reach the DB to check if the user information is present and correct so ask the Data Manager which is directly linked to the DBMS, once reached if the information is correct a confirmation is sent all the way back and the login is authorized, if information is not found then the login is rejected.

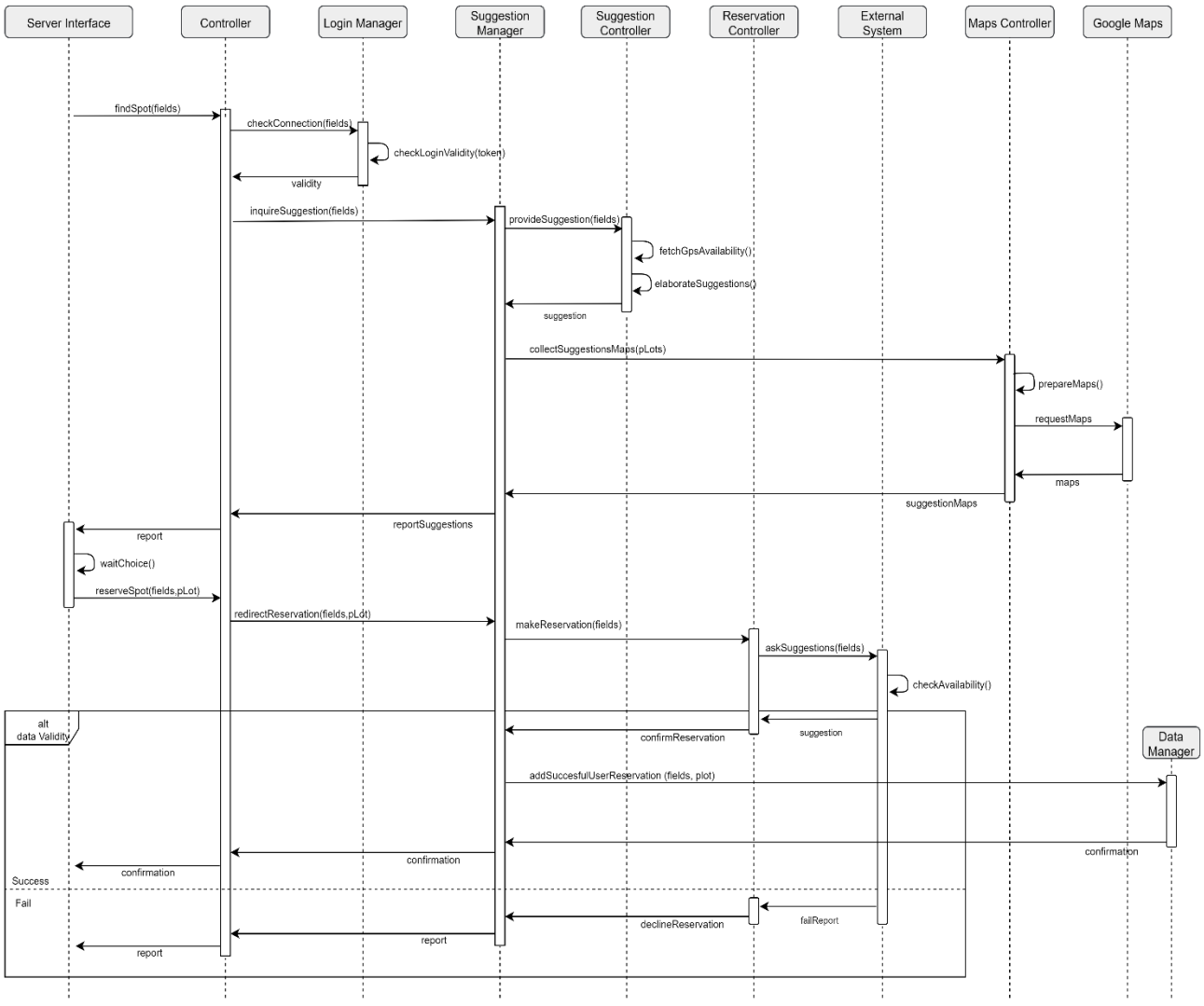
The Registration process is very similar to the login one but this time if the information is already in the DB the registration is rejected.

2.4.2 User Parking Spot Reservation

This scenario is the one that include most of the components of our system, here we start with a “findSpot” request that come from the server interface.

All the requests coming from here are sent to the Controller, which recognize the type of request and handled it to the correct manager, in this case we first need to check if the login authorization is still valid for the user, after the check is complete we hand in the request to the suggestion manager.

To not Overload the communication between the data manager and DBMS to write the data has not been repeated once more.



The Suggestion Manager component ask the Suggestion Controller to elaborate the possible suggestions and after the suggestions are retrieved ask the Maps Controller to associate the right maps so that the suggestions can be seen correctly by the user which will have to choose one of them.

After the choice has been made we need to ask the external system to reserve a spot in the chosen parking lot, if the external system confirm the reservation then the reservation is added to the user, if not then the reservation is rejected, and nothing happens.

2.4.3 Add/Remove a Special Event

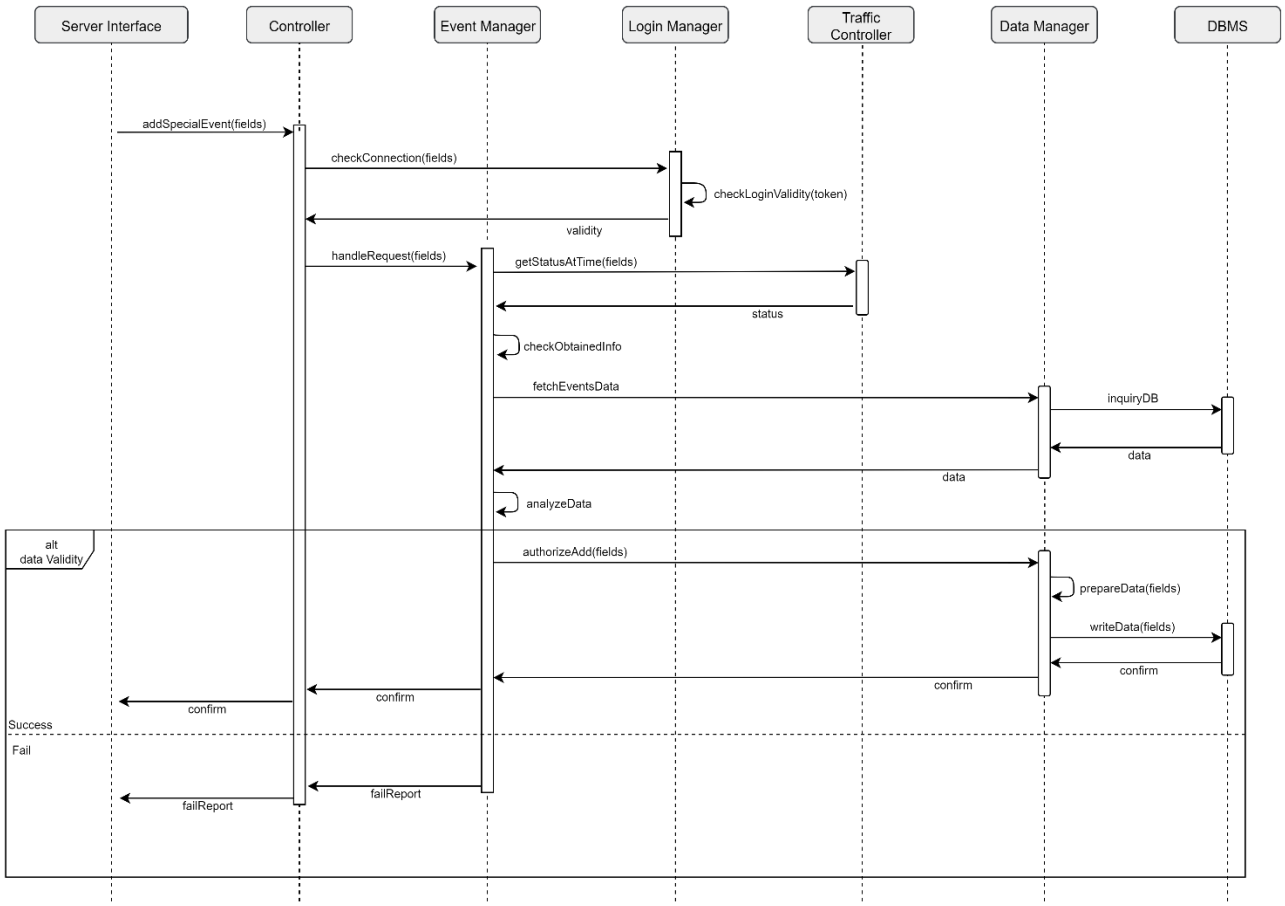
In this last scenario we want to add or remove a special event, this can be done by a superuser.

As an example, the addSpecialEvent request is handled here, this time the Controller send it to the Event Manager after the login manager validate the login status.

The Event Manager first check the traffic status of the given period to ensure other events that modify the traffic status are not present if everything it's ok it proceeds.

To do so both information coming from the traffic controller and data manger are needed, after that information are analyzed it we have two possible cases:

- The information is ok, and they can be added to the DB thanks to the Data Manager
- Something went wrong, and the operation need to be aborted, previous situation is restored.



2.5 Component Interfaces

Following will be explained the main methods that the system components implement, these methods will also be used in the algorithm design chapter.

The “fields” term represent some common parameters not worth of mentioning, for example a sign in request will definitely contain the username, the password, and the two included parameters already described in the previous section.

Event Manager

- Public boolean overlapping(event e1, event e2), check if two events overlap
- Public void editEvent(event e, field 1, .., field n), used to add an event
- Public boolean checkObtainedInfo(event e1), check if the fields inserted can be accepted
- Public boolean handleRequest(event e1, int type), handle the request based on the type

Emergency Manager

- Public void reportEmergency(emergency em), used to report an emergency
- Public void clearEmergencyState(), used to dismiss the emergency in the system

Traffic Controller

- Public Status getStatus(), used to fetch the current system status
- Public Status getStatusAtTime(Date t), gets the system status at the time t
- Public void updateStatus(Status s), changes the system status into s

Suggestion Manager

- Public Suggestion inquireSuggestion(User u), used to start the suggestion inquire process
- Public boolean redirectReservation(fields, parkingLot), used to start the reservation

Suggestion Controller

- Public void elaboratesSuggestion(User u), elaborates the possible suggestions for a given user
- Public suggestion provideSuggestion(), used to get the elaborated suggestions
- Public void fetchGpsAvailability(**token**), learns if the user is using and allowing the use of the GPS, uses the token to not query the DB

Reservations Controller

- Public void connectToSystem(), connects to the external system
- Public boolean makeReservation(), ask for a reservation to the system
- Public void registerReservation(), used internally after the confirmation is received

Maps Controller

- Public Map collectSuggestionMaps(parkingLot), gets the map for a suggestion
- Public void requestMaps(), request the maps to the external system (google maps)
- Public void prepareMaps(), set ups the received maps

Data Manager

- Public Event fetchEventsData(), used to inquire data about events
- Public void writeEvent(event e), used to write on the DB a new event
- Public void addUser(User u), used to write on the DB a new user
- Public void checkUser(string, string), user to authorize or not a registration
- Public boolean proceedSignIn(fields), redirects to proceed the sign in
- Public boolean proceedSignUp(fields), redirects to proceed the signup
- Public boolean authorizeAdd(event e), used to proceed with the write of the event
- Public void prepareData(), prepare the data to be written on the DB
- Public boolean addSuccesfulUserReservation(fields, User u, parkingLot pLot),

Router

- Public boolean signInRequest(fields), redirects to handle a sign in
- Public boolean registrationRequest(fields), redirects to handle a signup
- Public void analyzeRequest(fields), analyze the provided request

Login Manager

- Public boolean userSignIn(fields), proceed with sign in for a normal user
- Public boolean superUserSignIn(fields), proceed with sign in for a superuser
- Public boolean userSignUp(fields), proceed with signup for a normal user
- Public boolean superUserSignUp(fields), proceed with signup for a superuser
- Public void checkFieldsValidity(fields,**token**), check the fields of the request
- Public boolean checkConnection(User u), start a check validity process
- Public void checkLoginValidity(**token**), check the login validity using the token

DBMS:

- Public Request inquiryDB(Request r) [request represents many possible different data]
- Public void writeData(fields, User u)
- Public void deleteData(fields, User u)

2.6 Selected Architectural Styles and Patterns

The used network implements a **Client-Server model**, which is definitely one of the most used structure model from the network standpoint, it's been applied to our system in the following ways:

- The clients that use a web-application communicate with the application server through a web server.
- Mobile app clients communicate with the GCM connection server or directly with the application server, this to ensure the correct functioning of the system

For the communication a basic **request-response pattern** is used for the actors to interact with the server and for server components to interact together.

2.6.1 RESTful Structure

REST means Representational State Transfer and is an architecture style that has been implemented in our system in order to grant the interoperability between different systems on the Internet. It uses a stateless protocol and standard operations it can offer reliability and scalability to a system while being independent of the type of platform or languages.

In our system RESTful APIs are used for the front-end to communicate with the application server, where the APIs are wrapped.

When we are using a REST architecture an application can interact with resources just by knowing the identifier of the resource and the action to be performed on it. This token has already been used in the previous section by the Login Manager.

The used token has to be defined using the following rules:

- The token must be secure
- The token must include an ID to identify the user without having to query the database, including the type of user.
- The token will expire after a certain amount of time for security reasons

2.7 Other Design Decisions

As already explained briefly Google Cloud Messaging (GCM) servers are being used for the notifications to be received effortlessly by the app users. This means a use of Cloud Computing for our system instead than local data storage, at least for some of the data.

To be used it requires Google APIs and it can offer extra useful services like the Google Cloud Monitoring System to warn us of any problems or the Google Cloud Dataflow that will offer us insights of our data.

3 – Algorithm Design

In this chapter the most relevant scraps of algorithms that has to be implemented in the system will be presented, such algorithms will be written in a pseudocode based on java for simplicity.

Four different algorithms will be discussed as a demonstration given that our system lacks of complex yet interesting algorithms that other system may present.

First shown algorithm is a function of the Event Manager, such function helps to understand if two different events are overlap, this is not possible in our system, at most one event may take place at a given time and thus this short algorithm is necessary.

This function may be used before inserting a new event, and it should be checked for every other event in the system, at least for a given day.

```
1  public boolean overlapping(event e1, event e2){
2
3       int start1 = e1.start.hour;
4       int end1 = e1.end.hour;
5       int start2= e2.start.hour;
6       int end2 = e2.end.hour;
7
8       /* Events can't last more than a day! */
9       if (e1.start.month==e2.start.month)
10          if (e1.start.year==e2.start.year)
11          {
12              /* An event is at least one hour long! */
13              if(start1==start2) return true;
14              if(end1==end2) return true;
15              if(start1<start2 && end1>start2) return true;
16              if(start1>start2 && end2>start1) return true;
17          }
18
19      return false;
20  }
```

The second algorithm is just a simple check also present in the Event Manager, it checks that the inserted data are deemed correct (or at least make sense) and the request can proceed to its following step.

Both the first two algorithms work with a Boolean logic, return true if everything is ok or false if something went wrong.

```
public boolean checkObtainedInfo(event e1){

    /* Simple cascade checks to find if the fields
    have been filled properly*/

    /* The status of the system in the runtime view example
    has actually been fetched asking the Traffic Controller,
    here a getStatusAtTime method is used for simplicity */

    if (e1.title.length()>5)
        if(e1.desc.length()>20)
            if(e1.start.hour<e1.start.end)
                if(getStatusAtTime(e1.time)==true)
                    return true;

    return false;
}

```

The third algorithm is about the suggestions that are elaborated by the system after a user request, such suggestions need many information like the user position and gps availability, once those information are fetched a list of the best possible parking lots is obtained in the following way:

```
public parkingLot[] elaboratesSuggestion(User u){

    /* From the user we can get its position, its past reservations
    and understand if the gps is available and working*/
    Position pos = u.position;
    boolean gps = fetchGpsAvailability();
    Reservation res[] = u.reservations;

    /* We suppose that a function that returns all the parking lots
    in the system is available */
    parkingLot pLots[] = getParkingLotList();

    /* The parking lots are ordered by distance, removing the ones
    in which the user have already made a reservation, the ones that
    are currently full and then only the first three are returned */
    pLots = orderByDistance(pLots);

    pLots = discardFullLots(pLots)

    pLots =discardPastReservations(res,pLots);

    pLots = getTopThree(pLots);

    return pLots;

}

```

The last discussed algorithm is about the analyzeRequest function that the Router component offer. As input fields like username, password and two more Booleans are received, based on the content of those fields two different methods of the Login Manager may be called, if something is wrong the Router may choose to call an abort internal function.

Abort function is supposed to refuse the user login request in a non-problematic way such that the user may understand the mistake and retry.

```
public void analyzeRequest (String username, String password, boolean super, boolean remember){

    /* The method from Login Manager is called only if the fields are all correct
    and the right method is called based on the value of the "super" attribute
    which helps recognizing a superuser */

    if(username.length()>8 && password.length()>8)
        {
            if (super) loginManager.superUserSignIn(username,password,remember)
            else loginManager.superUserSignIn(username,password,remember)
        }

    /* In this case if the paremeters are not correct a possible "abort" method is called
    to block the execution correctly */

    else abort();
}

```

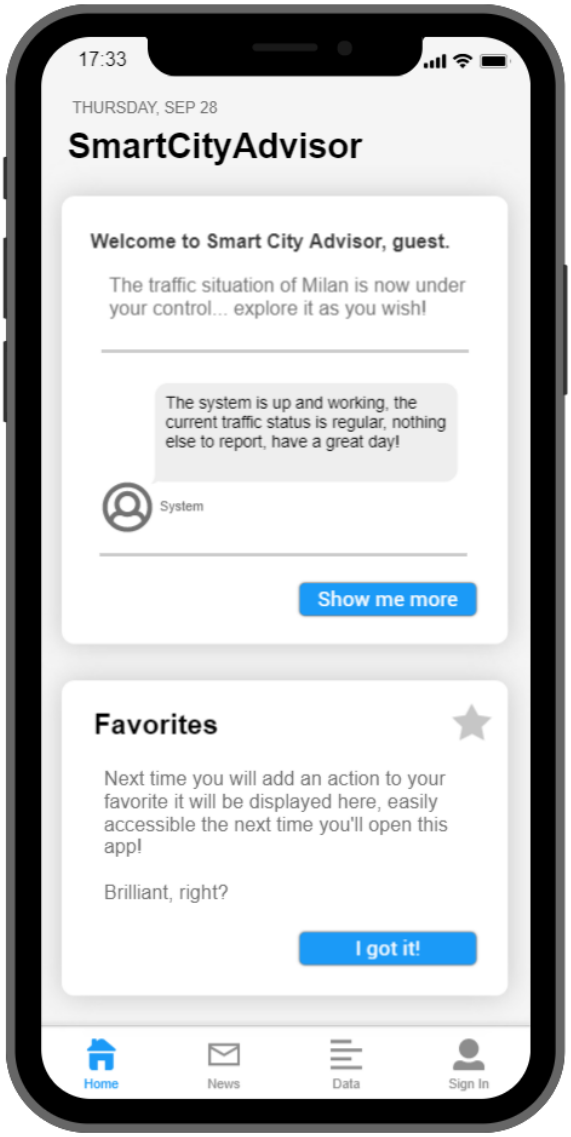
4 – User Interface Design

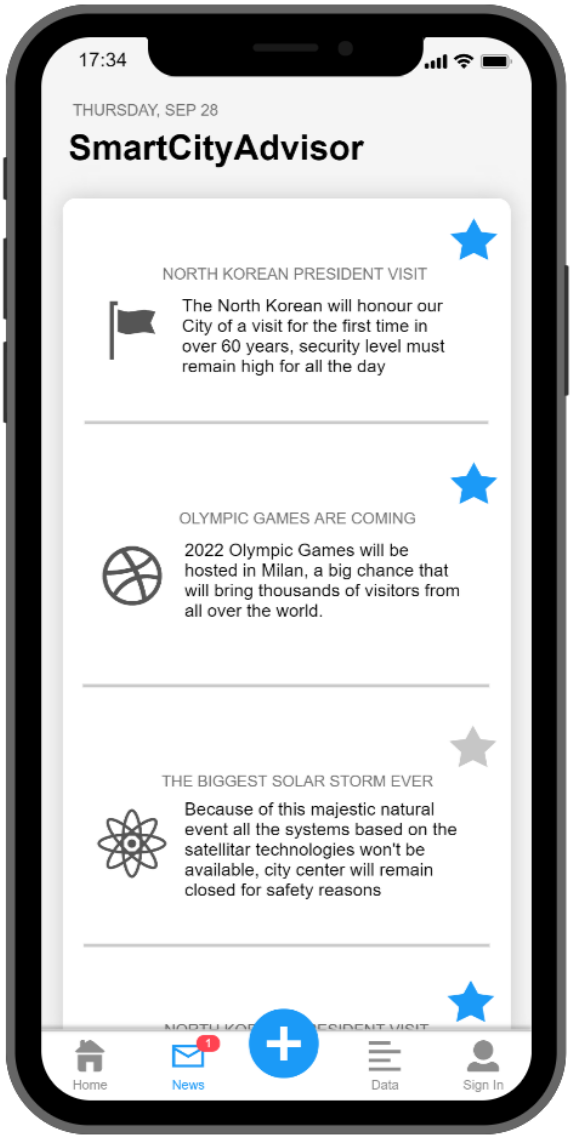
4.1.1 Mockups

Since the wireframes for both the web and mobile application have been already shown in the RASD document i would like to focus more on possible high fidelity mockups for the main pages of the mobile app only, which is undoubtedly the most interesting one to show.

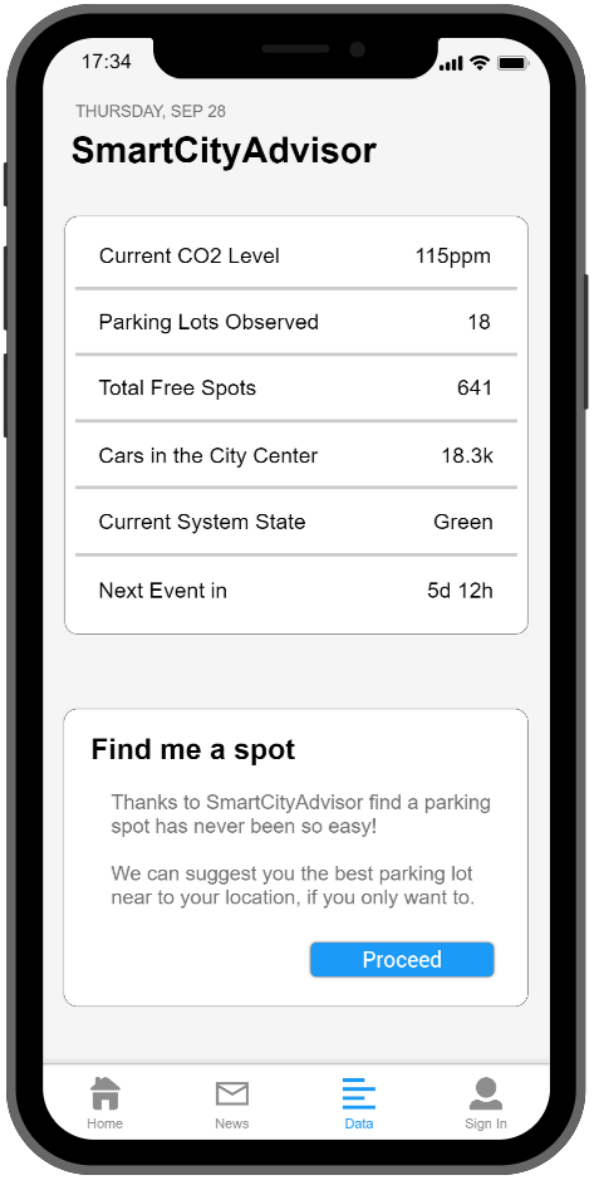
All these pages can be reached by any of the others, using the navigation bar at the bottom, the only exception is the one that regards the Reservation, which can only be reached by the “Data” page.

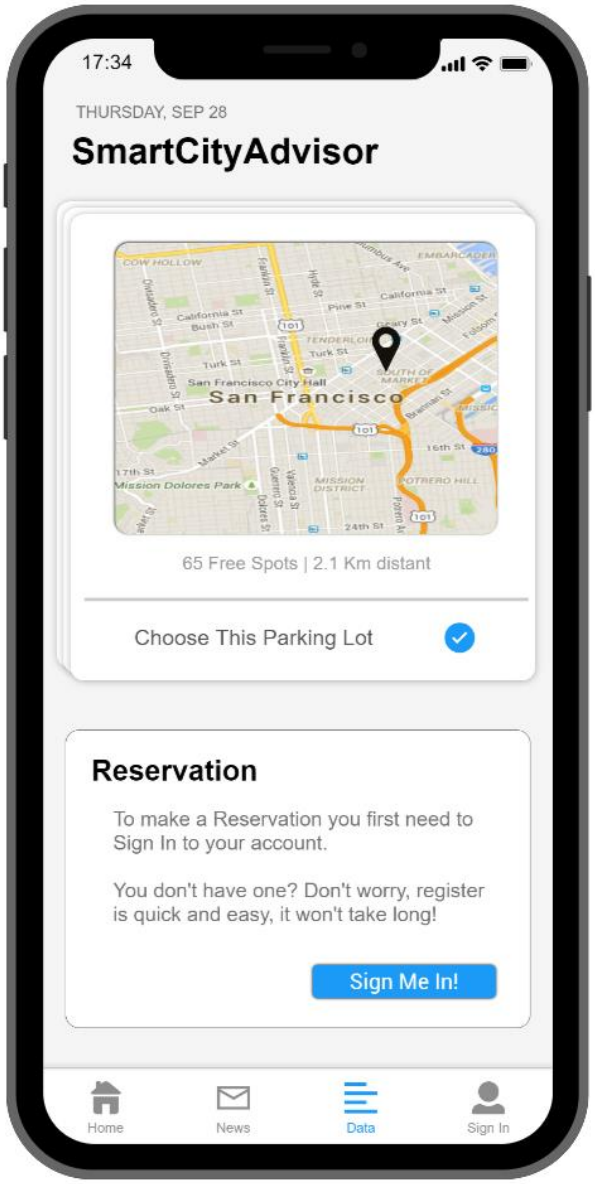
The first one is the homepage, where we can check the latest recap of the system and have a “favorites” function to make things easier to reach. The second one is the news section.



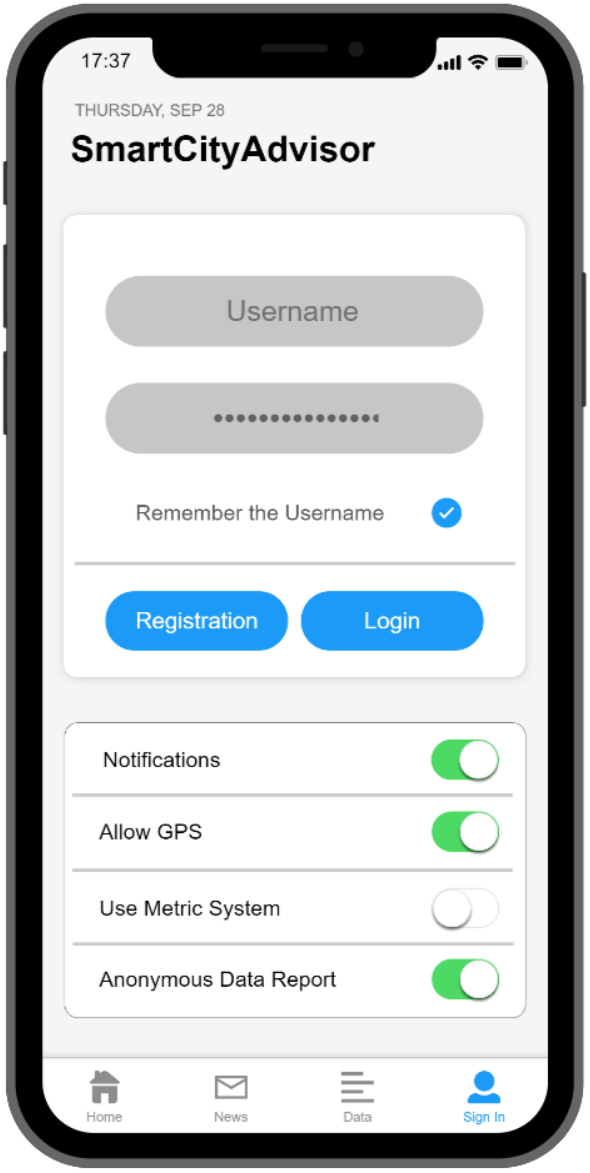


The third mockup is the Data page which includes all the latest data that comes from the sensors while in the fourth one possible suggestions of parking lots are presented. In the fourth one the page is as a guest would see it, and not a registered user.





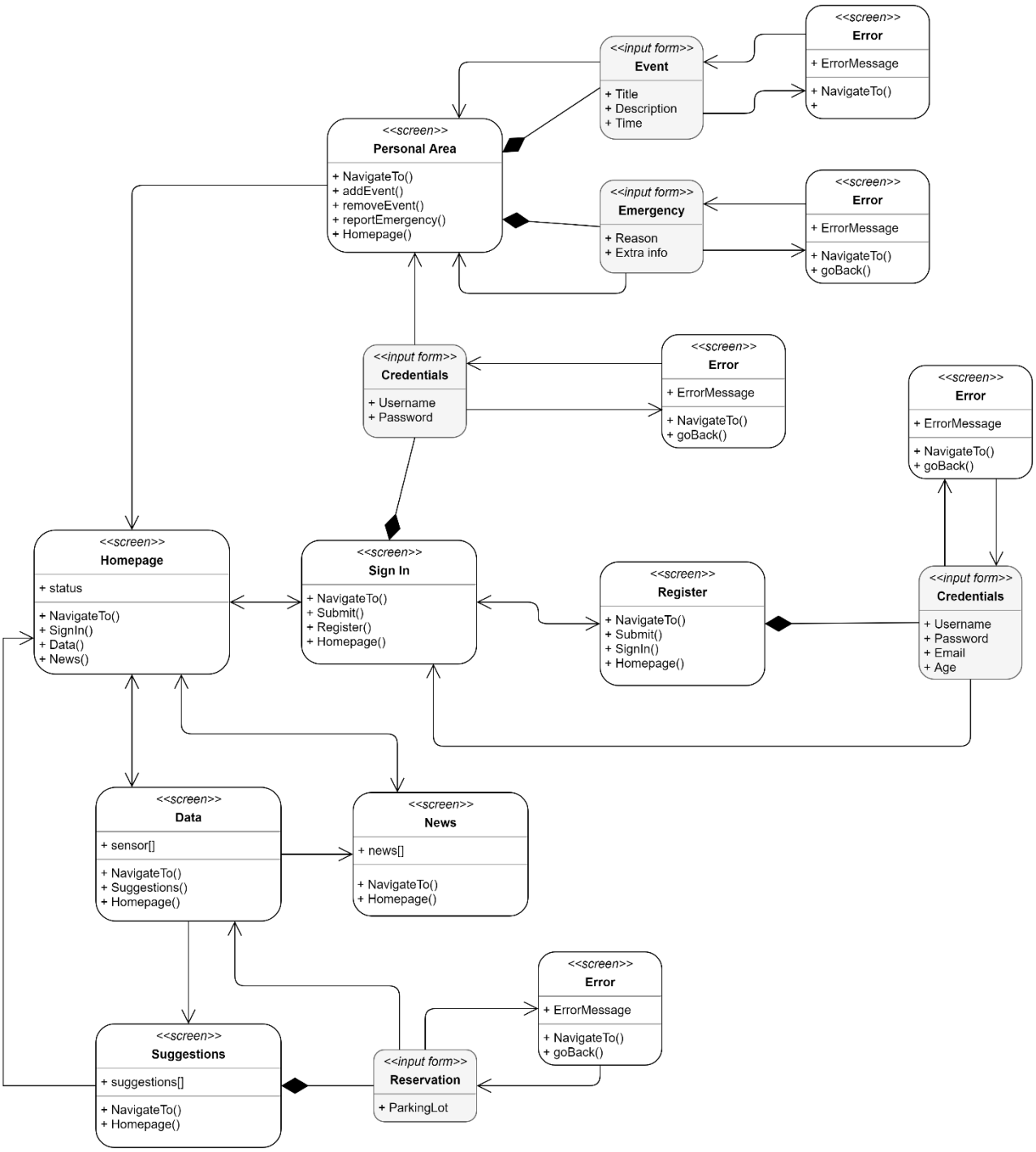
Lastly the login page is shown, which also includes some useful settings for the users and guests both. In this page we can also find a registration button that will sends to another page in the same “Sign In” category but with a different form.



4.1.2 UX Diagram

This diagram describes in detail all the pages of the application, how the application can be navigated including the screens, the input forms and the possible errors.

The navigation bar in the app allowed any page to move directly to any of the others (or almost), to not overload the diagram all the page can navigate to the homepage, that can move to the other pages in the navigation bar, basically we are adding an intermediate step to move from one page to another.



5 – Requirements Traceability

In this chapter all the requirements previously discussed in the RASD file will be matched to the components that help "solving" them. Only the main component is reported in the following table, but different components may take part in solving a single requirement.

Goals:

[G1] Limit the traffic accordingly to current CO_2 levels in Milan.

[G2] Warns the citizens of a change of the current status

[G3] Provide users with up-to-date information retrieved from the sensors

[G4] Manage the traffic accordingly to special events taking place

[G5] Simplify the process of finding and reserving an available parking spot

[G6] Providing Superusers the possibility to manage events and emergencies

Readings Manager:

- **[R1.1]** The system will automatically register the reports provided by the sensors
- **[R1.2]** The system will monitor the data and automatically make an action when moving after a threshold

Traffic Controller:

- **[R1.3]** The system may change its traffic status in any moment, in any possible status
- **[R2.1]** After a change in the system status citizens will be notified immediately and automatically by the system

Actuators System:

- **[R2.2]** The displays text will be changed and show relevant information

Data Manager:

- **[R3.1]** After data is received from the sensors it is categorized and added to a DB
- **[R3.2]** Relevant data is automatically loaded from the DB to the website every few hours

Login Manager:

- **[R4.1]** The user has to own superuser credentials
- **[R5.1]** User has to be logged in
- **[R5.3]** The system will send a link to the email provided to conclude the registration

- [R6.1] The Superuser has to be logged in
- [R6.2] The superuser has to reach it's personal area by navigating to it

Reservations Controller:

- [R5.5] The system will take the reservation and send it to the desired parking lot system
- [R5.6] The system will confirm the successful reservation by sending an email to the user

Suggestion Manager:

- [R5.4] The logged in user has to look for a parking spot and choose which one to reserve

Router:

- [R4.2] The user has to login on the website using the right credentials
- [R5.2] If the user is not logged in he needs to register by navigating to the right section and filling the forms

Event Manager:

- [R4.3] The superuser has to navigate into its personal area and add a special event
- [R4.4] Superuser must fill the form, the provided information will be then added to the DB
- [R4.5] The system will send an email to the superuser to confirm if the operation was successful
- [R6.3] Special forms to report emergencies, add/remove events can be found in the personal area

6 – Implementation, Integration, Tests

Overview:

In order to implement, integrate and test our project we choose to use a bottom up strategy. To do so we can divide the components of the system into different groups, every group will be implemented independently from the others. Every time a group of components has been implemented it will be integrated and tested.

Once that all the groups have been successfully implemented, integrated and tested separately the different groups will be put together and integrated and tested once again.

The third-party components are considered external to the system and the testing won't be done since they are already considered to be working correctly and reliably.

Implementation:

The implementation for the server and the clients can be done in parallel to make the development quicker since they can be considered independent from each other.

The implementation should also include the configuration that is needed to make the GCM server work correctly according to our system specifications.

Integration:

Since we are using well known protocols and interfaces for the communication of our components, the integration of them should result quite easy and not time consuming.

Components in a same group will be integrated in a specific order which can be found analyzing the dependencies of each components over the others, in this way everything will result less problematic.

Testing:

The testing of the components will happen in four phases:

1) Testing of the single components
2) Testing of subgroups of components in the same groups, according to dependencies
3) Testing of the groups of components
4) Testing of the entire system

To test the single components dotnet or xUnit tools could be used, while to test the web application we could use Jasmine or other open-sources tools.

While testing the entire system we'll make sure that all the possible functions will be working correctly thanks to a simulation made navigating the system on both the web and mobile app.

Particular care should be taken to improve the performance and reliability of the system to its maximum.

Load testing:

We definitely should remember that as our application grows the effort required to test it also grows exponentially, testing it before users can access it is way easier and should then be done extensively.

The server should also be tested to make sure it can handle the activity of thousands of users at the same time, calculating the computational power needed to complete all the requests can be challenging and to solve this problem we can use tools like Locust.

Thanks to load testing we can learn exactly how the system behaves under the simulated user pressure, how long does it take on average to respond and other useful data needed to make sure everything will work correctly, as we already said a high reliability is expected from this system.

7 – Effort Spent and References

This project has been developed by a single member because of a lack of other teammates, probably because not many other students are taking the second delivery of this year project.

The amount of time needed to complete this document is about 45 hours, many of them spent thinking which was the best way to display the information and making/modifying the needed diagrams.

Tools Used:

- The diagrams have been done using the draw.io free tool.
- The mockups have been realized using proto.io
- The wireframes have been realized using Balsamiq.
- Microsoft Word to write this document

The commits on GitHub are symbolical of the major steps of the work, as taking the project alone didn't really require updating the other members and share the work.