POLITECNICO
MILANO 1863

# DD

## Design Document

**Authors:**

**Antonio Pagliaroli**

**Filippo Pagliani**

**Davide Mangano**

Version:

Date:

Professor: Matteo Rossi

# Contents

# 1. Introduction

## 1.1 Purpose

The purpose of this document is to provide more technical and detailed information about the software discussed in the RASD document. It will represent a strong guide for the programmers that will develop the application considering its different parts: the basic service and the two advanced functions.

In this DD we present hardware and software architecture of the system in terms of components and interactions among those components. Furthermore, this document describes a set of design characteristics required for the implementation by introducing constraints and quality attributes. It also gives a detailed presentation of the implementation plan, integration plan and the testing plan.

In general, the main different features listed in this document are:

• The high-level architecture of the system

• Main components of the system

• Interfaces provided by the components

• Design patterns adopted

Stakeholders are invited to read this document in order to understand the characteristics of the project being aware of the choices that have been made to offer all the functionalities also satisfying the quality requirements.

## 1.2 Scope

CLup is a service that wants to help both the store and its clients prevent overcrowding inside and outside of the store trying to manage the flow in of the store.

The application will insert itself in a particular historical moment when avoiding overcrowding became more and more important since maintaining the social distance between people is a crucial issue in order to contain the spread of COVID-19. However it’s important to say that even if CLup reaches his maximum importance if it’s inserted in this particular situation, it should be very useful even on its own. Indeed the system can help the user saving a lot of time since it allows the client to take his place in the queue of a store virtually or even book your visit in advance avoiding the waste of time normally used waiting for your turn. Furthermore

even the store itself can benefit from CLup because thanks to its service it can optimize the flow in of the clients in order to maximize its number.

One more important purpose of the application is to be accessible by the largest possible user base since everyone should be able to shop in a store and in order to do that is CLup provides a fallback option for those who can' have access to the technology required. For this reason beside the possibility to line up or to book a visit, everyone can go to a store and physically ask to be added in the queue in order to let the shop take care of it.

## 1.3 Definitions, Acronyms, Abbreviations

### 1.3.1 Definitions

### 1.3.2 Acronyms

| | |
|---|---|
| **AS** | **Application Server** |
| **RASD** | **Requirements Analysis and Specification Documents** |
| **DD** | **Design Document** |
| **GPS** | **Global Position System** |
| **DBMS** | **Database Management System** |

### 1.3.3 Abbreviations

## 1.4 Revision History

## 1.5 Reference Documents

## 1.6 Document Structure

- **Chapter 1** describes the scope and purpose of the DD, including the structure of the document and the set of definitions, acronyms and abbreviations used.
- **Chapter 2** contains the architectural design choice, it includes all the components, the interfaces, the technologies (both hardware and software) used for the development of the application. It also includes the main functions of the interfaces and the processes in which they are utilised (Runtime view and component interfaces). Finally, there is the explanation of the architectural patterns chosen with the other design decisions.
- **Chapter 3** shows how the user interface should be on the mobile and web application.
- **Chapter 4** describes the connection between the RASD and the DD, showing the matching between the goals and requirements described previously with the elements which compose the architecture of the application.
- **Chapter 5** traces a plan for the development of components to maximize the efficiency of the developer team and the quality controls team. It is divided in two sections: implementation and integration. It also includes the testing strategy.
- **Chapter 6** shows the effort spent for each member of the group.
- **Chapter 7** includes the reference documents.

# 2. Architectural Design

## 2.1 Overview

The application should be done by using a three-layer structure:

• **Presentation level (P)** handles the interaction with users. It contains the interfaces able to communicate with them and it is responsible for rendering of the information. Its scope is to make understandable the functions of the application to the customers.

• **Business logic or Application layer (A)** takes care of the functions to be provided for the users. It also coordinates the work of the application, making logical decisions and moving data between the other two layers.

• **Data access layer (D)** cares for the management of the information, with the corresponding access to the databases. It picks up useful information for the users in the database and passes them along the other layers.

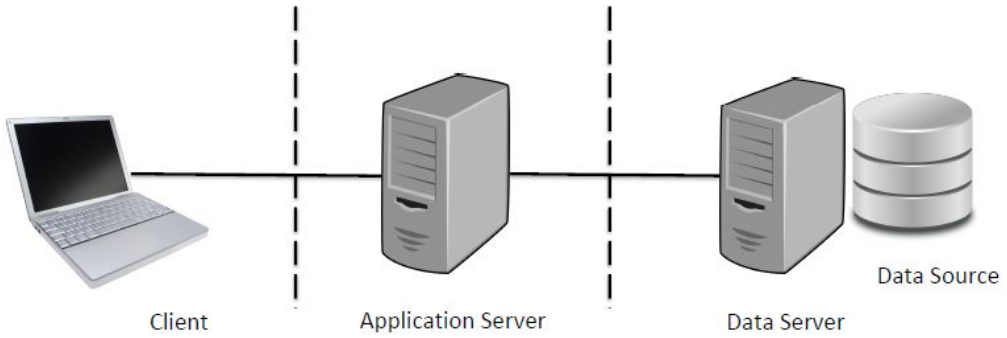
The architecture must be *client-server* style, because our structure is meant to have two actors in most of the possible actions. Client and server's communication takes place via other components and interfaces located in the middle of the structure (since they are being allocated into different physical machines), composed by hybrid modules.

The process always begins with the authentication of the clients (user or store) that can after ask for a certain service to the server which subsequently will elaborate the request and eventually query the DBMS if he needs some stored data based on which service has been requested.

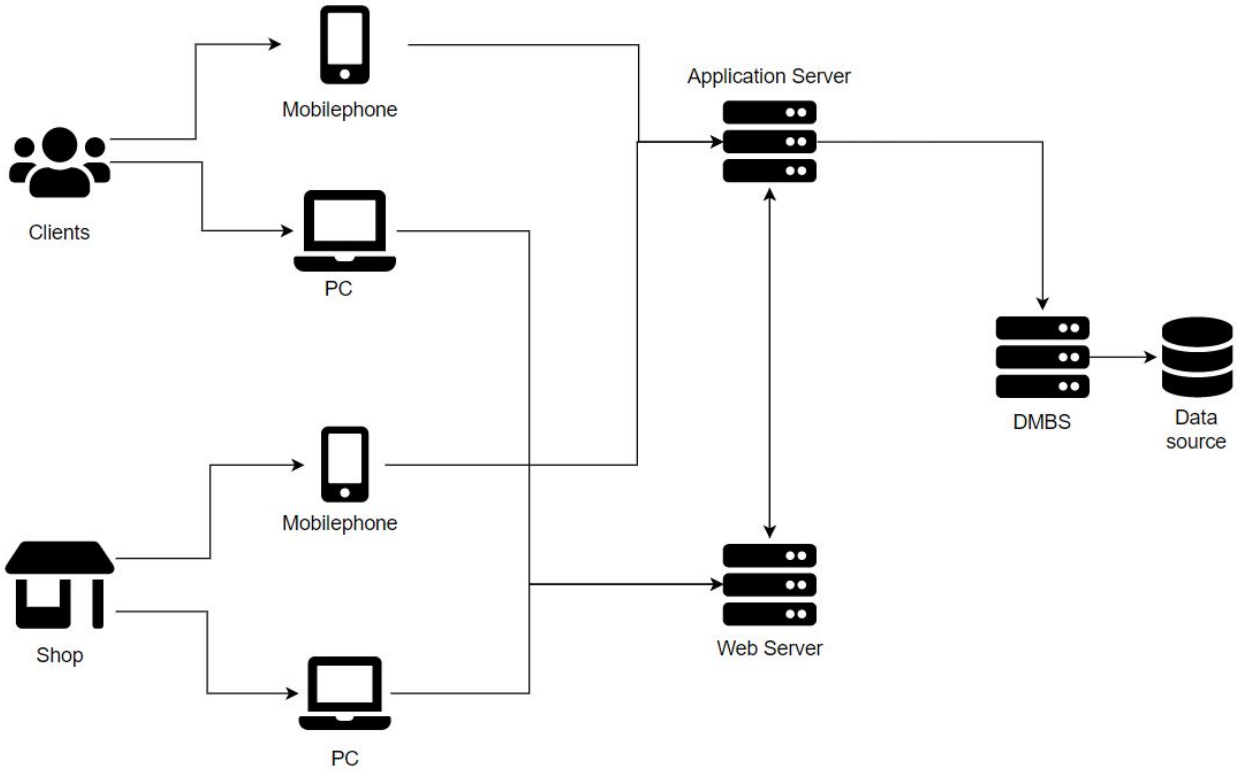
- **Lining up:** A user wants to take a place in the queue of a shop, to do so, once logged in, he has to select the store where he wants to go and then send the request to one server that will analyze it and, if the the store is actually open, will send back to the user a ticket providing him all the information that he needs including his place in line and the time he should leave the house in order to reach the store
- **Booking:** A user wants to book a future visit to a store, to do so, once logged in, he has to select the store where he wants to go in order to access its schedule and then select one or more free slots indicating when he wants to go. Subsequently the request will be sent to a server that will analyze it and if there are no errors will save the visit in the DB and then the server will send back a ticket providing the information needed to the user including his place in line and the time he should leave the house in order to reach the store.
- **Suggestion (chiedi per problema con architettura client server)**
- **Physically lining up:** A user wants to use the fallback option and go to a certain store to take his place in the queue asking the store for it. The shop will, through his account, add people in the queue so he simply send a request similar to the one of the normal line up with the difference that is coming by a store account. Once the system has processed the request, if there are no error, the schedule of the store will be updated in the DB and, after receiving a confirm from the server, the store can release to the user a ticket that indicates his place in line and the time he has to be at the entrance of the store.

### 2.1.1 High Level Components

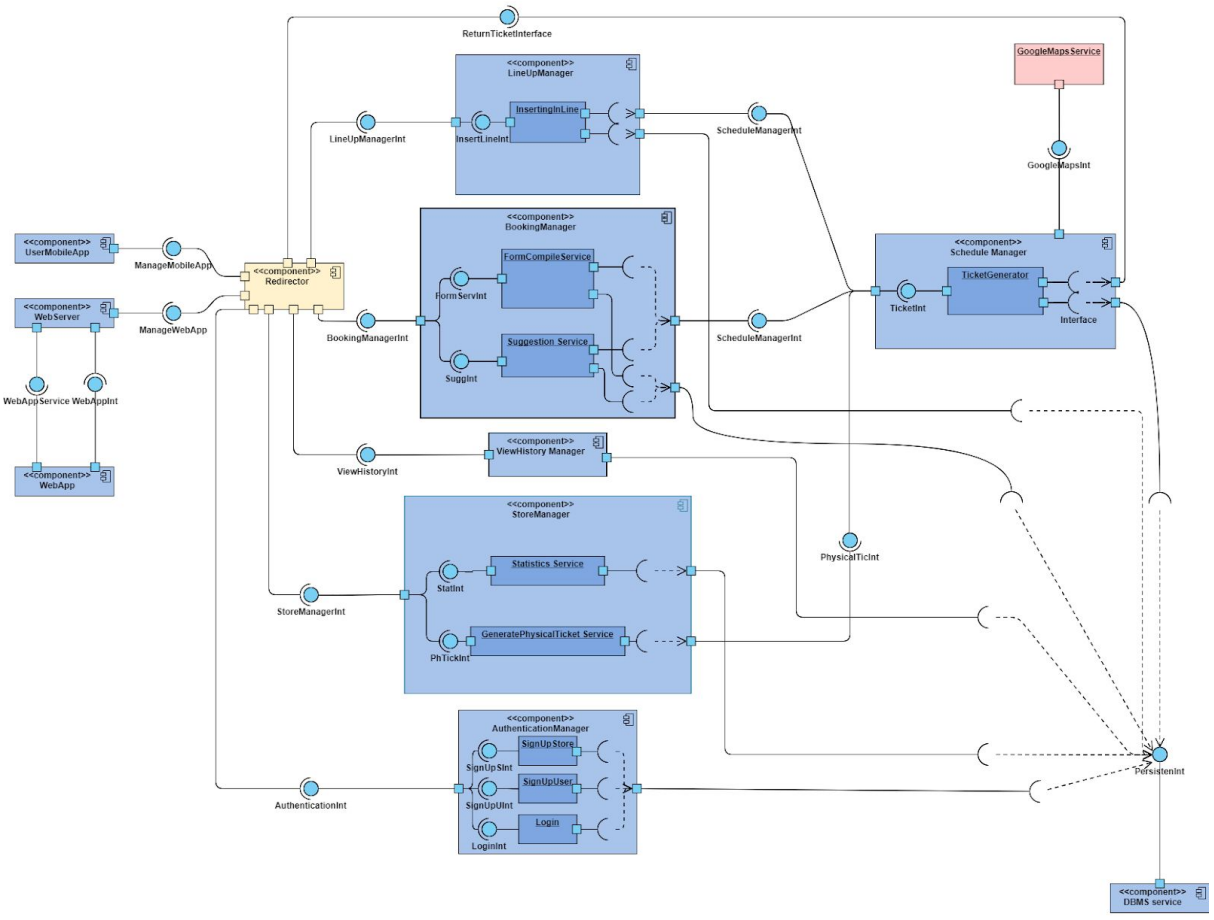
The hardware architecture chosen for the distributed application is Three-Tier. The three application layers are subdivided, therefore, among many dedicated machines, i.e.: a tier to interface with client **(P)**, a middle tier for the application level **(B)**, and another server for database management**(D)**. This approach is beneficial because the middle tier can maintain persistent connection with the DBMS, which is less expensive. Furthermore, having an intermediate machine between client and server can guarantee more security to the access control of the database from the users.




Both the user and the store can access to the service via a computer or a mobile phone but with obviously different pattern: if you are using a mobile phone your request will be sent to the Application Server, if instead you are using a computer your request will be first of all redirect to the Web Server that is designed to manage the HTTP request and then the server itself have the possibility to call some method from the Application Server if needed. The Application server is responsible even for the communication with the DataBase Server (DBMS) and for the use of the API used for the access to the map provided by Google Maps.


Mobilephone
Application Server
Clients
PC
DMBS
Data source
Mobilephone
Shop
Web Server
PC

## 2.2 Component View




The following component diagram gives a specific view of the system focusing on the representation of the internal structure of the application server, showing how its components interact. The application server contains the business logic of our software. Other elements in the diagram, besides the application server,have been depicted in a simpler way just to show how the communication is structured among these components and the application server.

- **StoreManager:** this component comprises three subparts:
    - **SubscriptionService**: it manages the subscription of a store to the system. To validate the subscription a form, including some documents needed to certify its identity, has to be forwarded.  Furthermore the possibility to be part of a chain is also given in this service.
    - **StatisticService**: it manages the request of each store to show an overview of the flow of clients, of the sold items and of the all visits.
    - **GeneratePhysicalTicketService**: it allows all clients without the app to visit the store. In practise each store has an external slot ( with a

person, or completely automatic) that print physical tickets in coordination with the system.

- **ViewHistoryManager:** it allows users to view all of past visits and booking in every store.

- **BookingManager:** this component comprises two subparts:
    - **FormCompileService:** this component allows the user to compile a form with specific fields (described in the RASD) to book a visit.
    - **SuggestionService:** it is a service complementary to the FormCompileService, because the user can accept a suggestion by the service in order to book a visit in a specific store and with specific slots.

- **LineUpManager:** this component comprises one subpart:
    - **InsertInLineService**:it allows the user to line up in the virtual queue. With the service the user can view the expected time to visit the store before the inserting in the queue and can choose the store that he wants to visit.

- **ScheduleManager:** this component comprises one subpart:
    - **TicketGenerator:** this is the core service of the application. It allows the system to generate tickets in a sequential way by coordinating the physical request, the lineup's request and the booking's request. It has an access to the DBMS service to check the number of the visitors in order to avoid overcrowded situations.

- **Redirector**: this component simply dispatches the requests and calls to methods from the users and the stores to the core of the application server. Every method is redirected to the proper component that can handle it. Also, responses and data sent back pass through this component to reach the applicant.

• **DBMSServices:** this is the component that allows every other component in the system to interact with the database. The Interface provided by this component contains all useful methods to store, retrieve, update data into the database from different actors. Every internal component of the application server uses some methods of its interface.

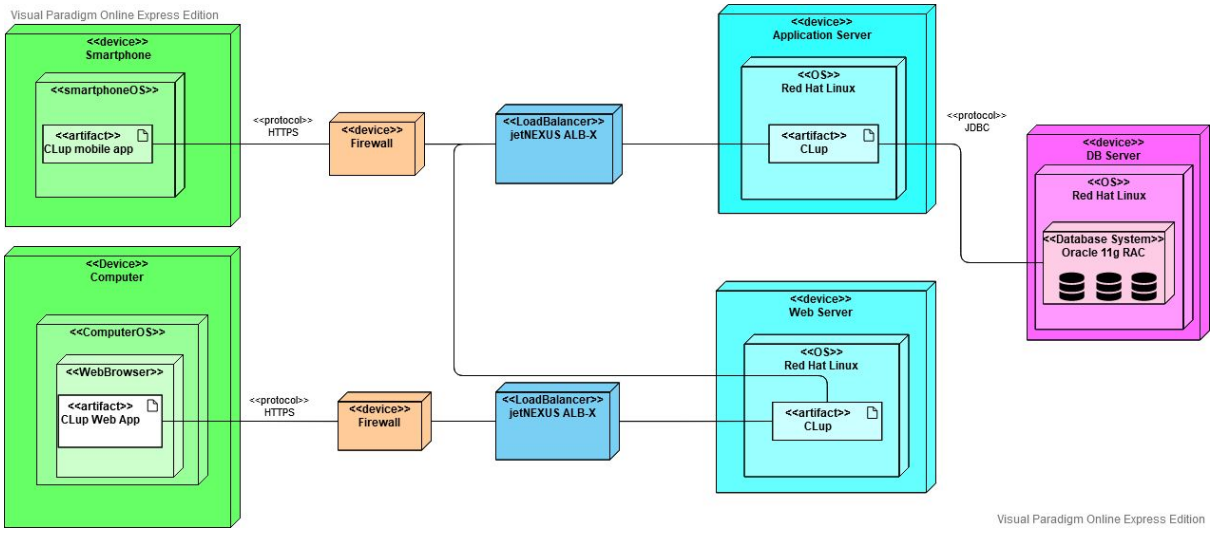
The external component of the system is also mentioned:

- **GoogleMapsService:** this is the component that provides a useful interface used to visualize the map of a specific store requested by the user. It is used by the component TicketGenerator to determine the waiting time and the departure time for each user that uses the app, and also by the store that prints physical tickets in order to calculate the same times based on the address given by the physical clients.

### 2.2.1 Additional Specification

//application and web server replicated 2 times

## 2.3 Deployment View




A concise topology of the hardware of the system is shown on this diagram. In particular:

- **Smartphone:** In general, a mobile device that can connect to the internet, and can download the CLup app from its AppStore. The user can manage his account from here: he can do bookings, enter a queue, see his chronology, and send these requests on HTTPS protocol.

- **Computer:** A device that can connect to the internet. With a browser it is possible to access the Web Page of CLup and to do everything it can be done on the App on mobile systems.

- **Firewall:** Intermediate between “internal” and “external” world, it helps to keep a closeted environment and deny attacks coming from outside. It’s the first rule of safety to have it, and with a sanitary emergency there is no space to joke around.

- **Load Balancer:** Keeps the amount of work distributed equally on the application servers. It is useful to avoid excessive workloads on one system that would otherwise be mortal in terms of performance, since the entire system could be blocked.

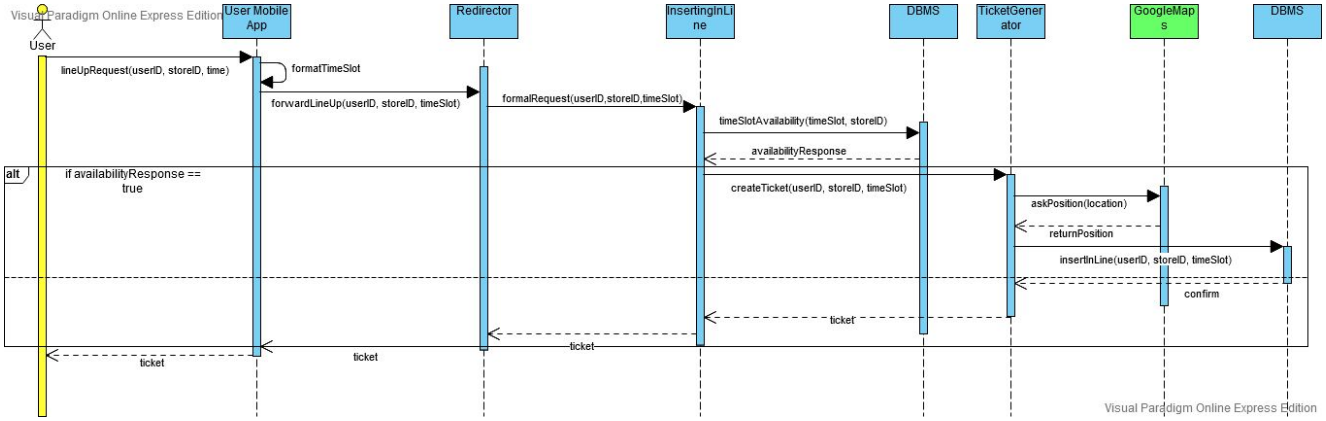
- **Application Server:** Here relies most of the logic of the entire process. Requests and answers are managed by forwarding them to the client (the App, in this case) or the DB.

- **Web Server:** The choice of using a different server for WebApp client was done because of the main objective of the App: in a sanitary emergency, availability is preferable to performance. In case of system shutdown, the stores wouldn't know how to properly handle queues nor people inside and it would end up being quite a mess, not only that, but also a futile danger for everybody. By separating the two clients, and giving them two servers for each, the hope is to never fall for excessive workloads that would shut down the process. Performance is obviously negatively influenced by this, since queries could wait a bit to reach the DB (since it has four servers linked to it), but performance is not considered a main goal for CLup.

- **DBMS:** In this case the choice of using Oracle was nearly mandatory, since it is the best choice for most applications. In our case, the presence of a peer to peer community that helps to solve all problems was a huge back up in case of failures. Also the main cons which are cost and space, are not a problem in our situation. Also Oracle database is secure and ensures that user data is not tampered with through prompt updates.

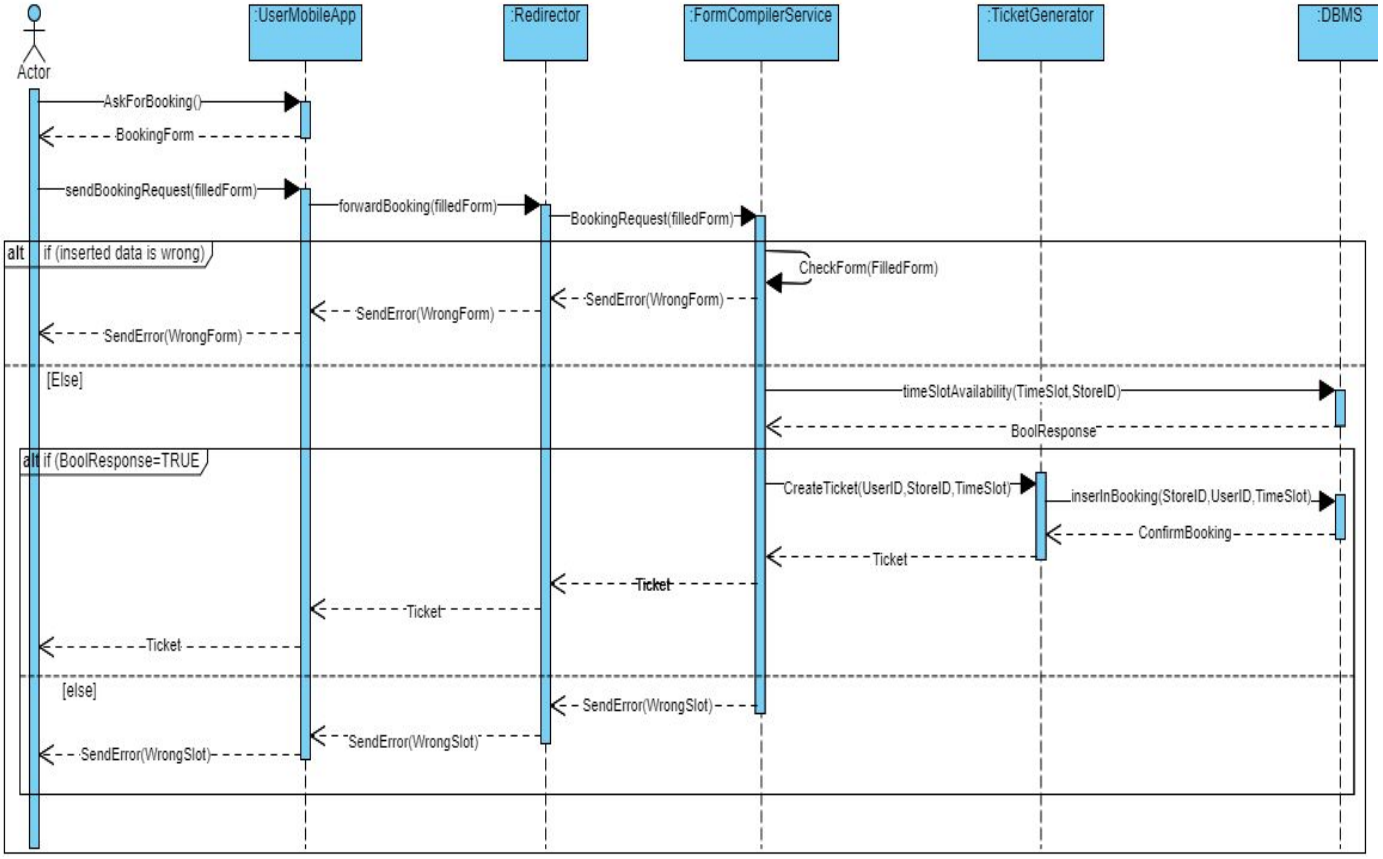
## 2.4 Runtime View

### 2.4.1 Line Up Request



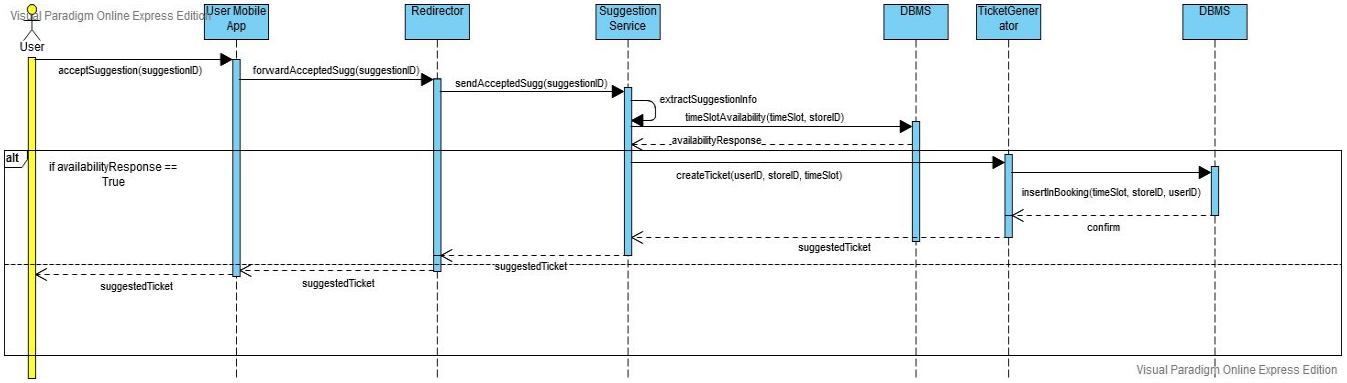

A brief presentation of a Line up request is shown above. The User, logged in through the User Mobile App, selects the store he wants to go to and he asks for a ticket. The system generates the time slot synchronized with the user's device system hour and formatting it as a time slot (so that it just shows the present time). Then the User Mobile App asks the Redirector to forward its information, and then all is sent to Inserting in Line. Here it is asked to the DBMS if the Time Slot is available: if the response is positive, the process continues. It is then asked to Ticket Generator to create a ticket by combining all the information it has, plus the position received by Google Maps services. All the data is inserted in the DB, so that the user is "officially" inserted in the queue. The ticket is then returned to User.

### 2.4.3 Booking



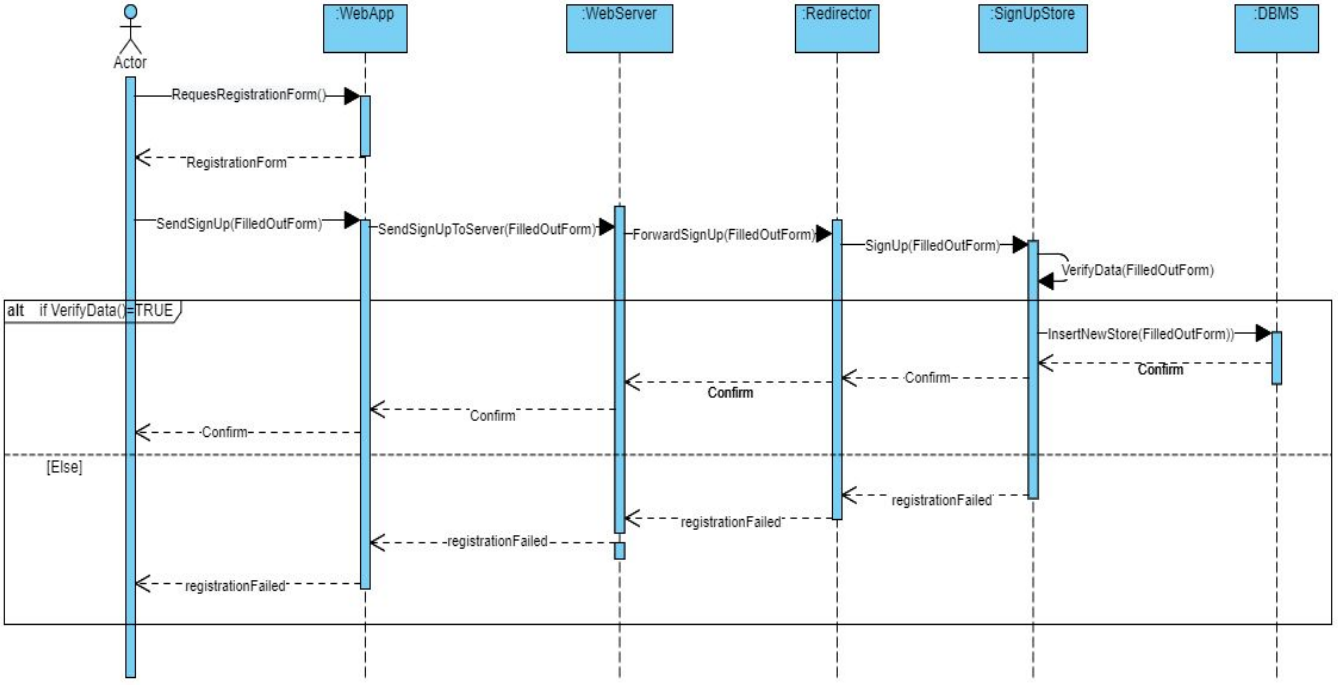

The sequence diagram above represents what actually happens when a user tries to book a visit inside the system. First of all the user ask for the form that indicate the data needed for the prenotation to the mobile app, once the form is delivered back the user can fill it with the required data and send it back then it will be redirected by the redirector to the FormCompilerService that has the duty to check the integrity of the data inserted by the user and if there is some type of error it has to report it to the user. If instead there are no mistakes the request is forward to the TicketGeneretor. The task of this last component is to control the availability of the required slot in the store's schedule. If the check is done without any issue a confirmation of the booking is sent back to the user, if instead there is a mistake this one is sent back to the user.

### 2.4.4 Accept Suggestion



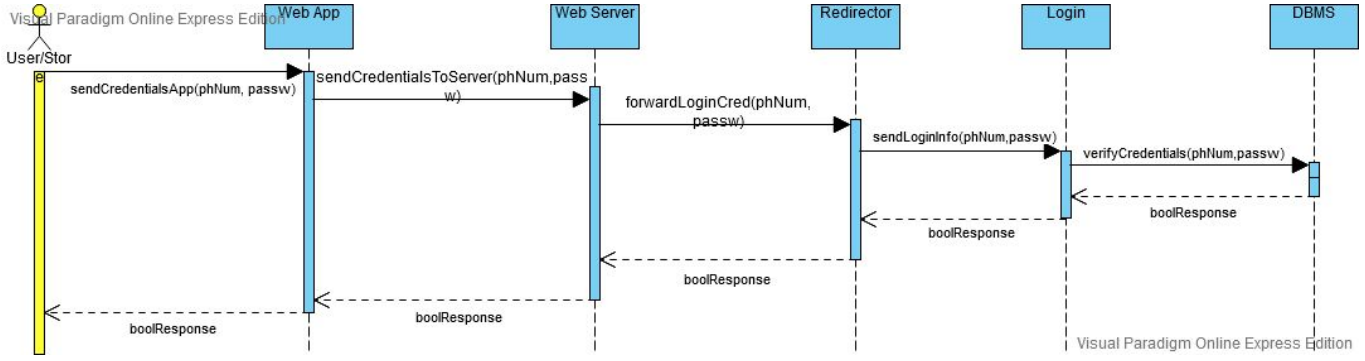

Here a brief explanation of how the User (logged in in the User Mobile App) can accept a suggestion sent by CLup. The User accepts it on the App, and a unique ID corresponding to the suggestion is then sent to the Redirector. Its task is to forward it to the Suggestion Service, which confronts the suggestion ID with the ones present in the system memory, and then proceeds to extract userID, storeID and timeSlot corresponding to that suggestion. The process that follows is, in fact, a booking; the request of timeSlot availability is sent to the DB (in case something changed since the forwarding of the suggestion); in case of positive response, the ticket is generated and the User is booked to the store, as shown to him in the suggestion. The ticket is then sent back to the User.

### 2.4.5 Registration (Store)



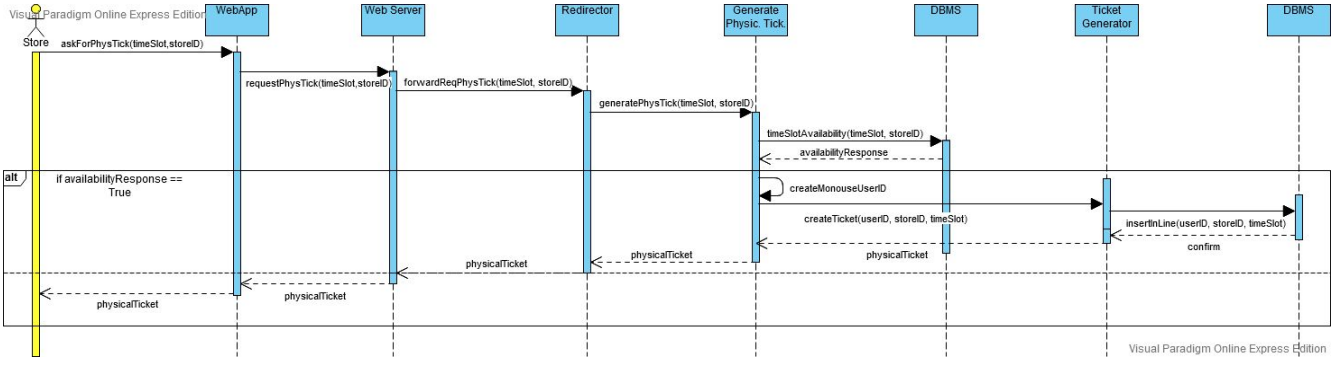
The diagram above shows how a store can register itself on CLup. The first step requires that the store ask for the form where can be inserted all the data needed for the registration from the WebApp. Then the store has to fill the form and send it back to the WebApp that will redirect it to the SignUpStore through the WebServer first and the Redirector right after. The signUpStore is responsible of checking the integrity of the provided data, if this one present an error the system report the error back to the user, if they are correct instead, the next step is to insert the new Store and all its data in the DB thanks to the DBMS and then send a confirmation of the registration.

### 2.4.6 Login (both User and Store)



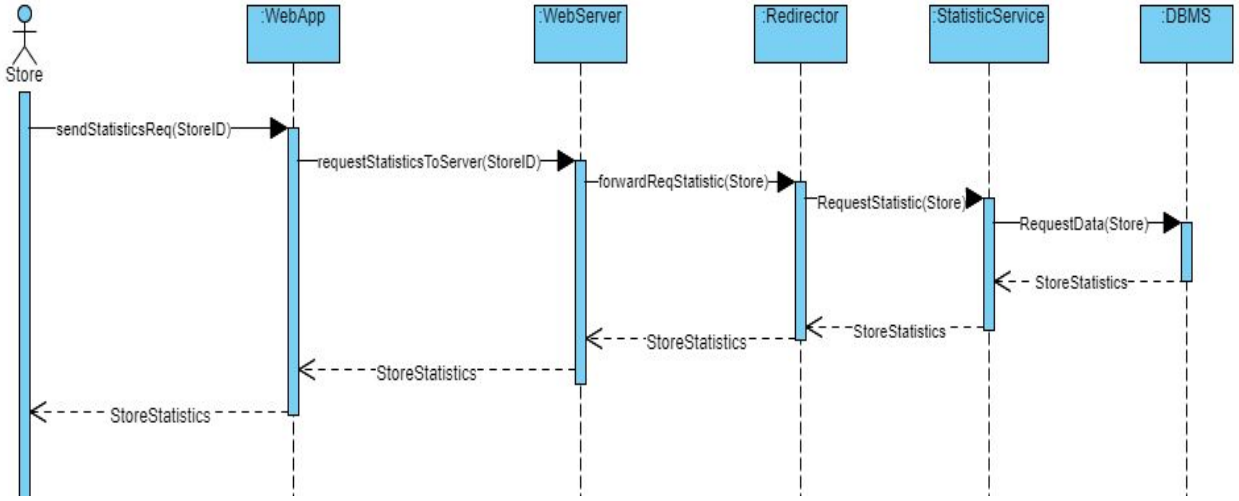

In the diagram above, a practical example of a User (or Store) Login to Clup is shown. In our application, as demonstrated in the Component Diagram before, there is no difference between User or Store Login via WebApp and for this reason there is only one Sequence Diagram to show both. For sake of conciseness, I will only present the User one: he inserts his credentials, previously saved in the DB when signing up, and sends them to the Web Server. After that, they are sent to the Redirector where they are forwarded to Login and then the query for the DB is ready, asking if there exists an User with such credentials. It is then sent back the boolean response, showing if the Login has successfully been completed.

### 2.4.7 Physical Ticket Request



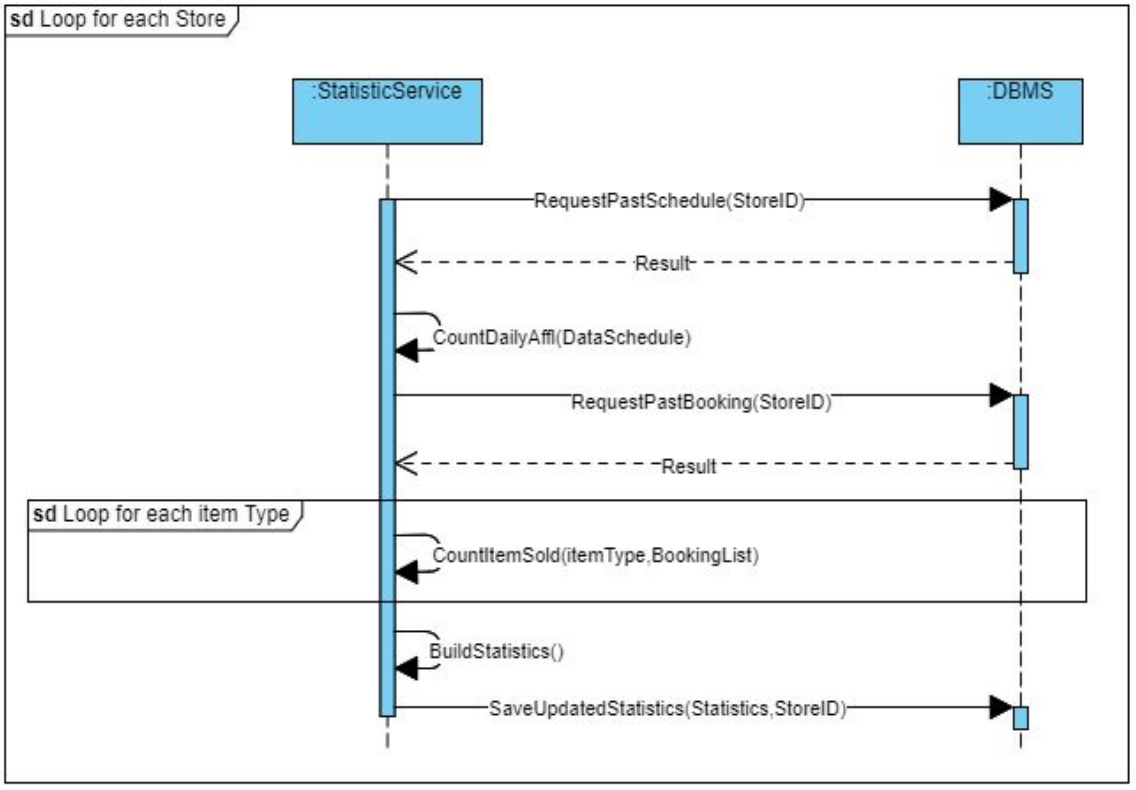
Above is shown how an unknown User can access the Store even without having a smartphone or PC to access CLup; this shows the accessibility of our application. In fact, an unknown User can ask the Store to generate a physical ticket for him; what happens is that the store does everything for him, by formally asking the system to line up. It is asked to the DB if the current time slot is available, if the response is positive then the physical ticket is generated by just printing the time slot and a system-generated monouse userID. In the DB, the unknown User is inserted in line and the physical ticket is then returned to the store.

### 2.4.8 Get statistics




The sequence diagram shown above is meant to describe how a store can have access to its statistics. First of all the store has to require the statistics to the WebApp that will forward the request to the StatisticsService passing through the WebServer first and then through the Redirector. The statistics server once received the request has to query the DBMS for the needed data and then it can build the statistics crossing the data sent from the DBMS. When the statistics are built and complete the only thing left to do is sent back all to the store.

### 2.4.9 Build Statistics

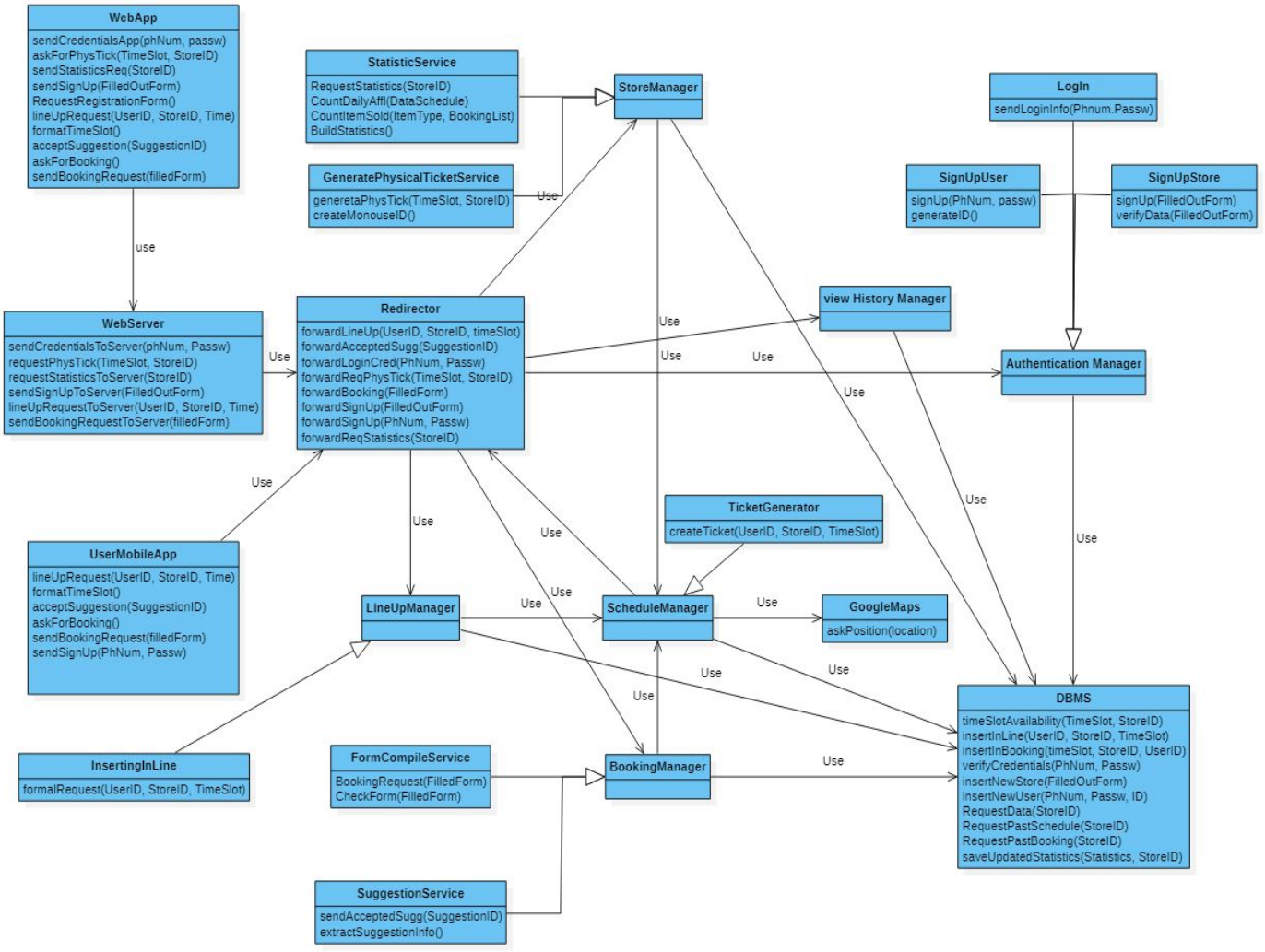



In this last diagram we can see how the system should periodically build the store's statistics. Once a week all the statistics are updated with the most recent data, to do it first of all the StatisticService will query the DBMS for its past schedule in order to collect the data on the affluence at the store. After that the next data required are the booking registered in the past days relative to that precise store from which can be extracted the data relative to which item was sold and when for every item type. Now that the StatisticService has all the needed data he can build the updated statistics in the right format and save them in the DB thanks to a request to the DBMS. Obviously all this process has to be done once for every store.

## 2.5 Component Interfaces

In the next diagram are described the main methods which can be invoked on the interfaces and their interactions, referring to the most important processes reported in the runtime view section.

One aspect is fundamental to be pointed out: in general, methods written in the Component Interfaces diagram are not to be intended exactly as the methods that the developers will write, but they are a logical representation of what component interfaces have to offer. They will be adapted facing the various aspects that will come out during the implementation of the code.




## 2.6 Selected Architectural Styles and Patterns

The selected architectural style is *client-server*; this was the choice since, as previously mentioned in the Overview, this is a perfect example of client-server structure: the interaction always involves two parts in which one has the role of the asker and the other one the replier.

The option of implementing an *event-based* application was also taken in consideration, in a way which considers taking a ticket as an event which will be broadcasted to the store as the only “subscriber” to the queue. At the same time, options like the store asking to close the queue because of a (mainly sanitary, but could be whatever) emergency would be broadcasted to the members of the queue as the “subscribers” to the queue. But thinking about the problem’s main possible

situations, which implied a more classical client-server structure, made it more clear that the event-based implementation was a stretch.

Of course the relationship between the Application Server and the DBMS is of type Client-Server too, in the sense that the AS sends queries to the DB.

Communication between components are always kept secure thanks to HTTPS, supported by the TLS protocol (SSL is outdated).

**Software Pattern: MVC**

Our main goal is to have a really slim and transparent component which stores the data. In the MVC pattern, it is sufficient to implement a slim Model by just implementing the data structures, while all the logic is reserved to the Controller. The View part also contains no logic too, to keep everything in each own compartment.

In the Client side programming, it will be considered the case of violation (with a view to risk possibilities): this will deny to insert important parts that could seriously compromise the entire system.

**Observer Pattern**

This pattern will be used in the case the store wants to stop the queue because of an emergency. The Observed object is the store which, when it gives the stopQueue signal, alerts the Observer which then sends a notification to all the queue and booked members.

## 2.7 Other Design Decisions

In precedence it was shown that the DBMS used was Oracle RAC, which is a relational database.

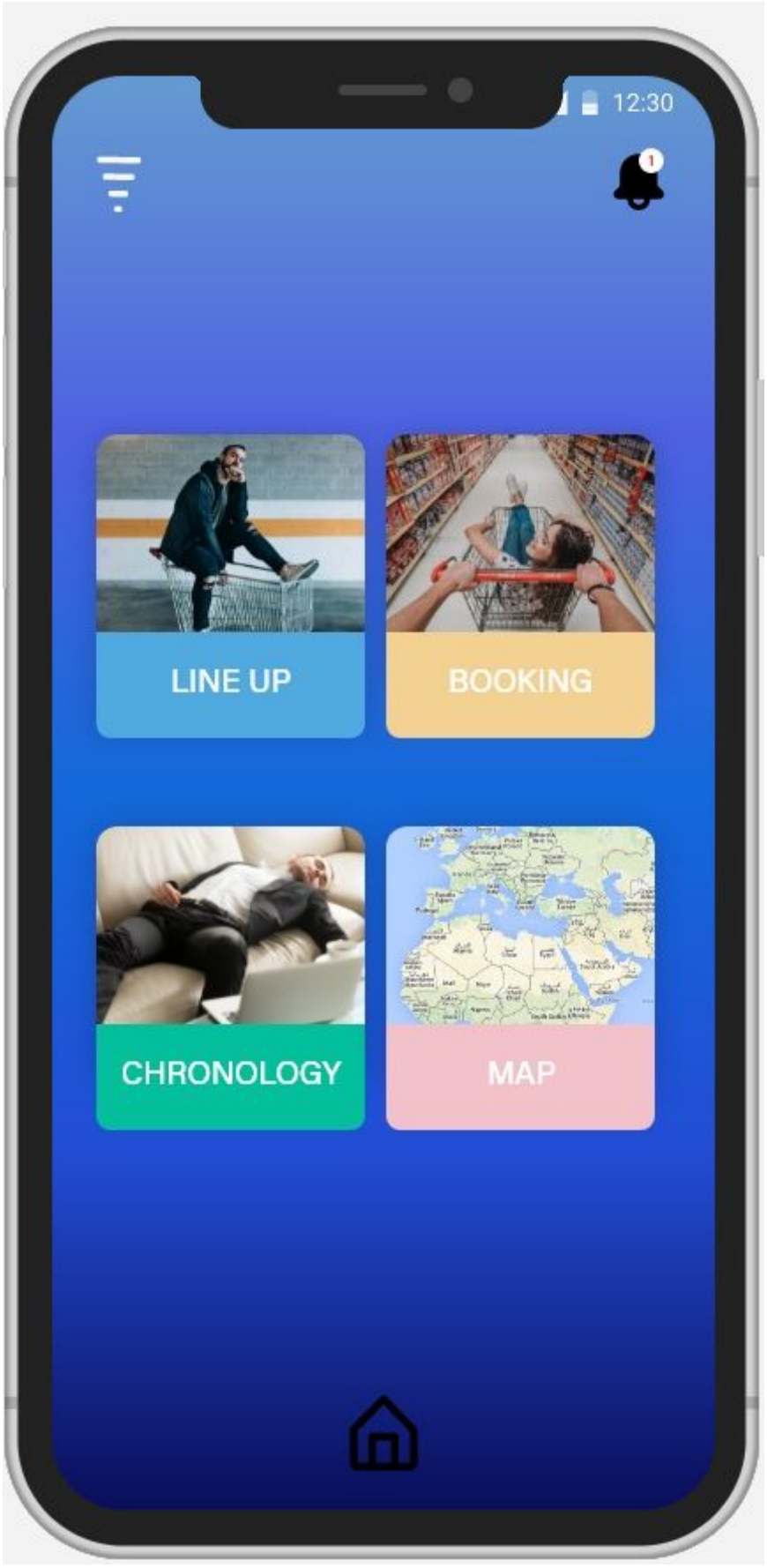
This was done since some of the key advantages of relational databases are that relational models structure the data in table structure, a format which is very familiar to work with. This *simplicity* also helps to retrieve certain data more easily by querying through any table, combining related tables to filter out unwanted data and allowing them to retrieve the data easier than navigating a pathway through a tree or hierarchy.

Another advantage of a relational DB is *flexibility*, which allows it to meet changing requirements and data without affecting the rest of the database. Tables and columns in the database can easily be added, removed and modified to meet requirements.

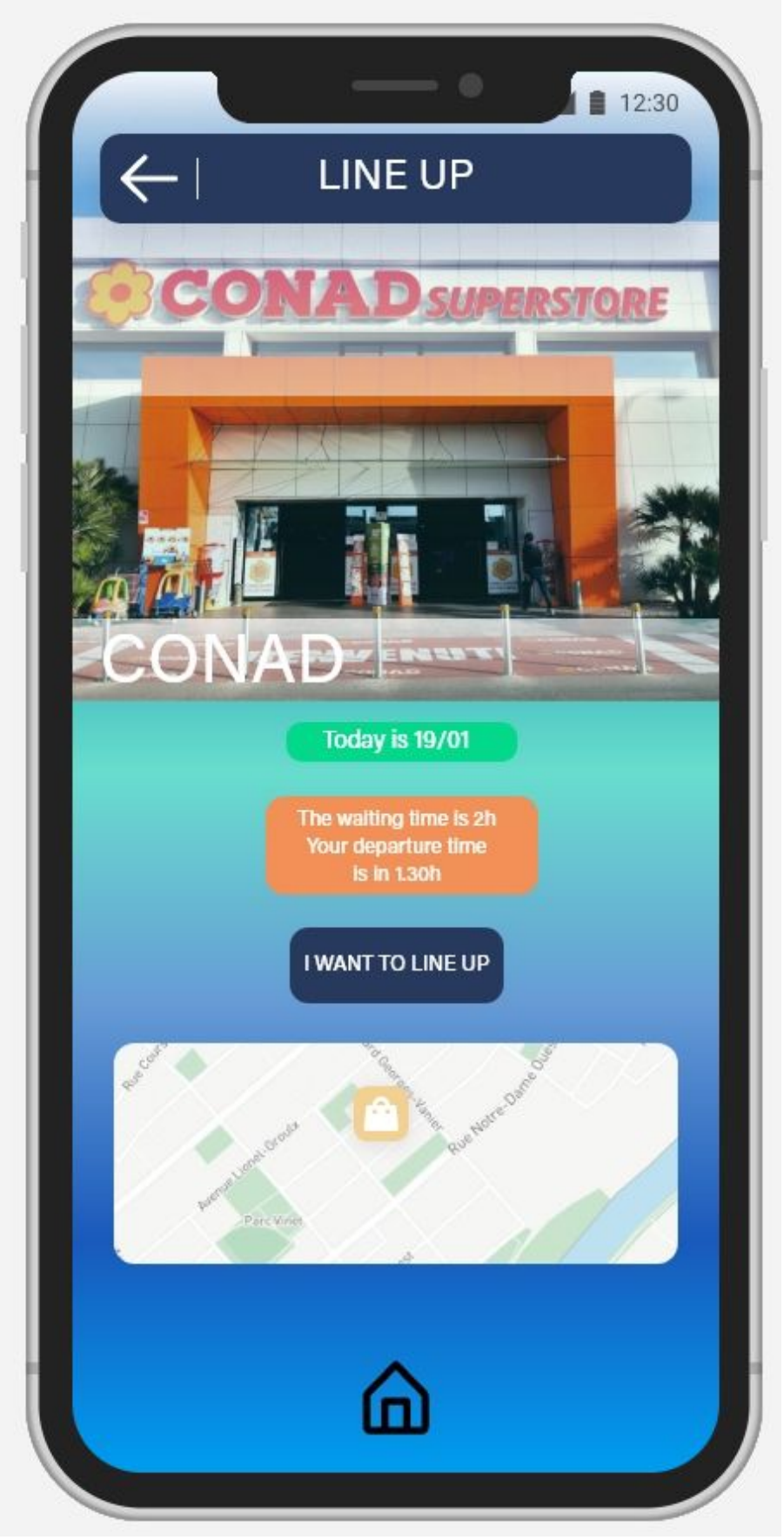
# 3. User Interface Design

## 3.1 Mockups

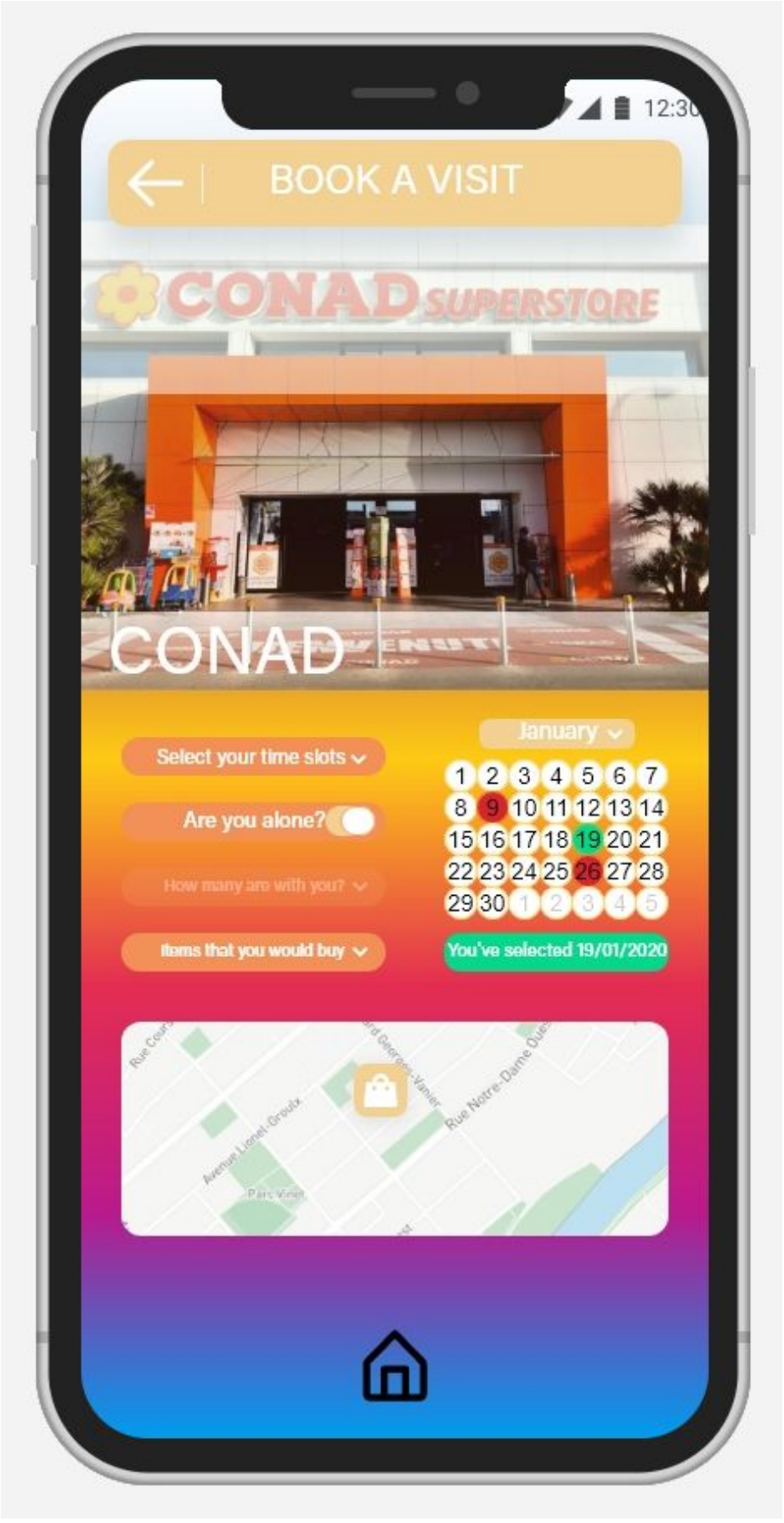
1-Login Page



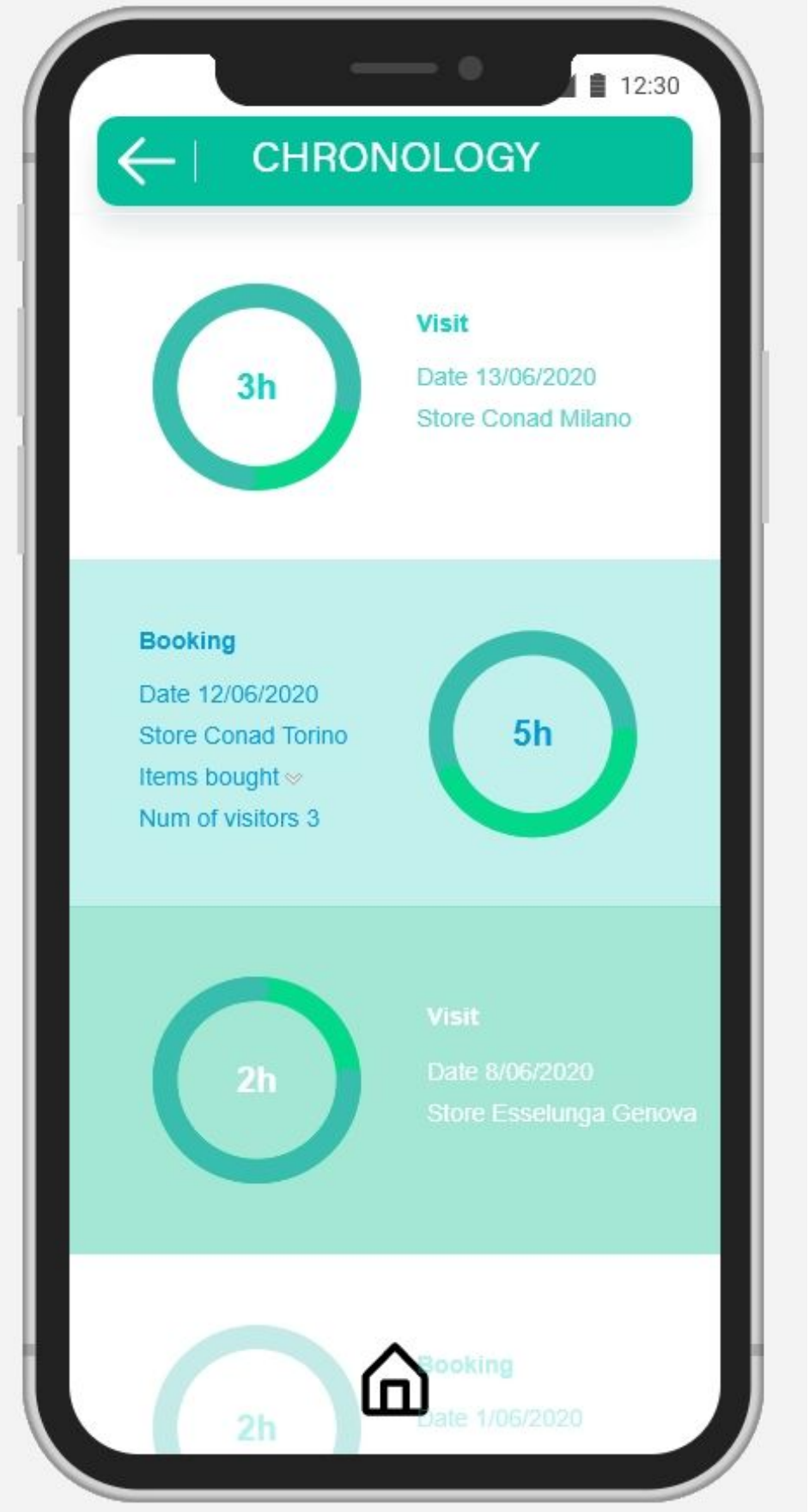

2-Home Page




3-Line Up page

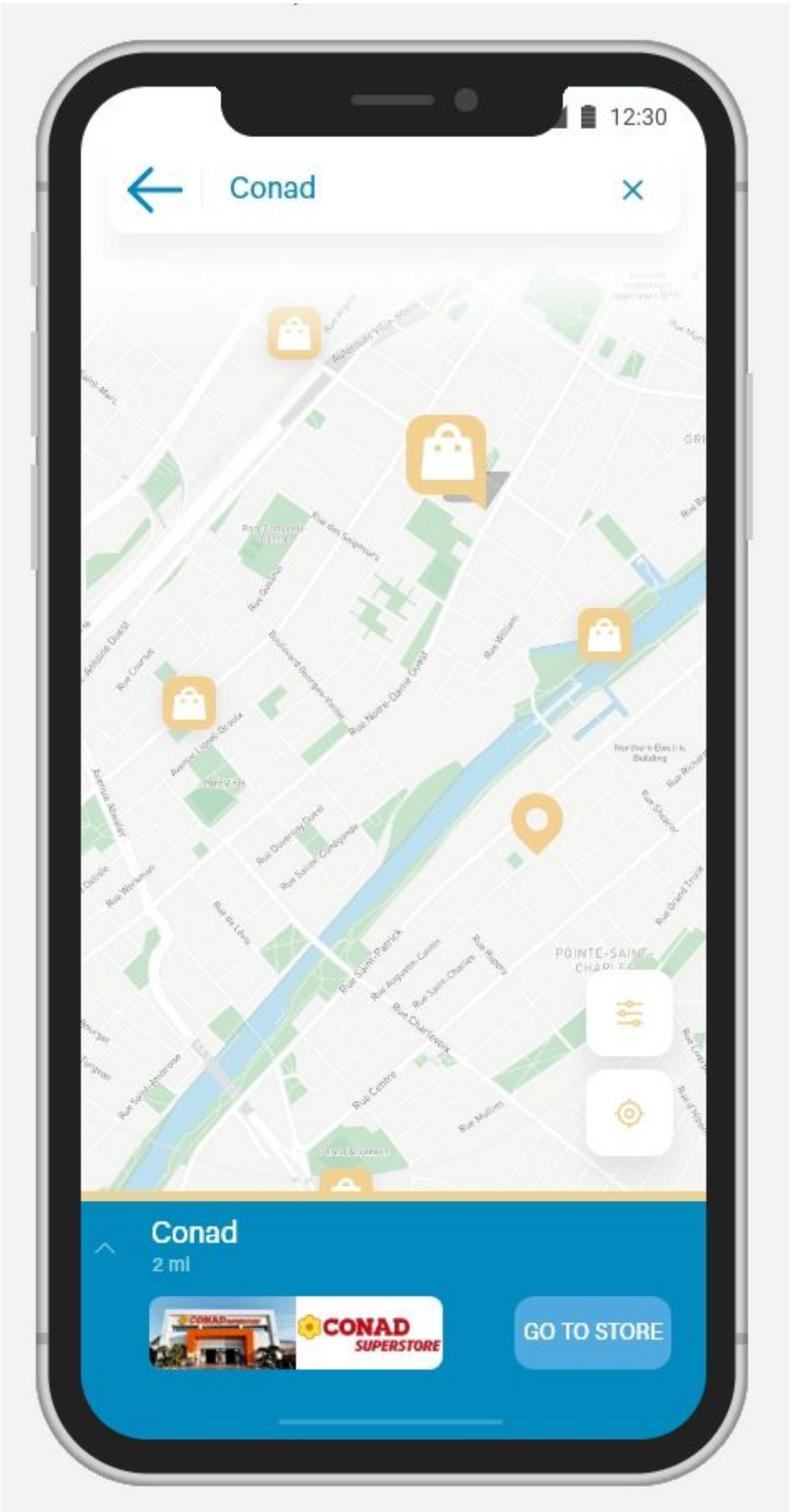



4-Booking Page

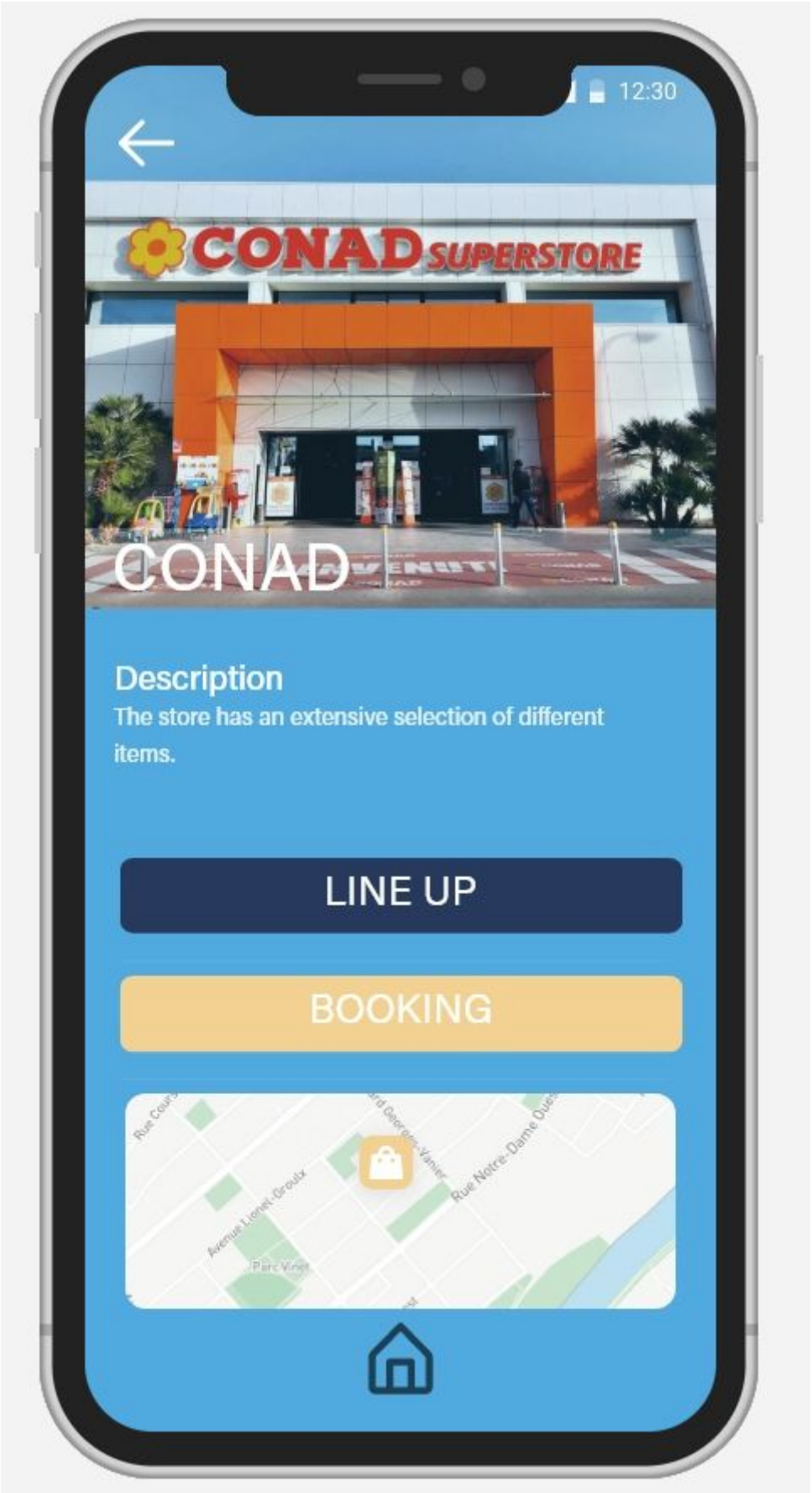



5-Chronology Page

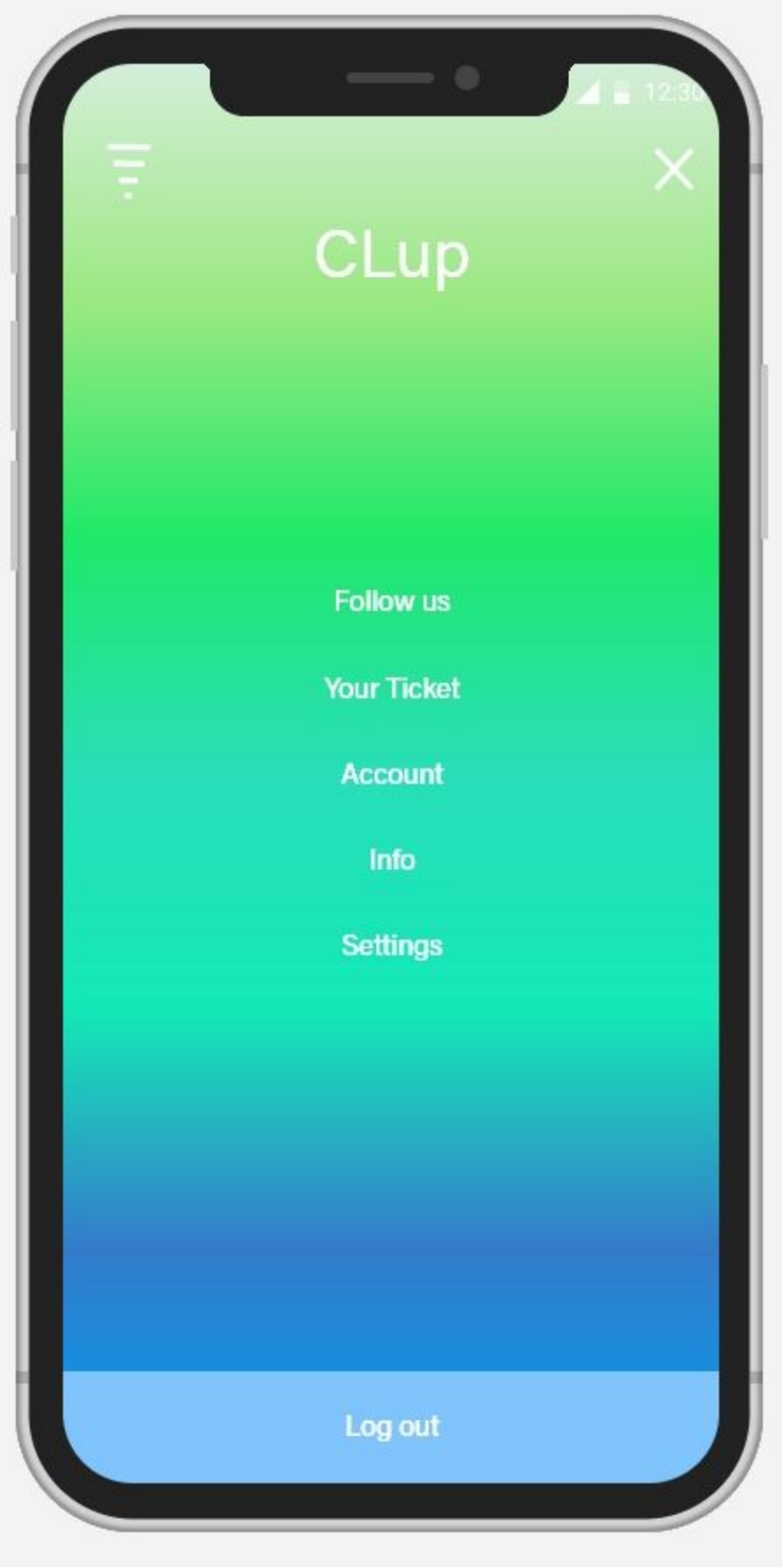

6-Suggestion Page




7-Map of stores

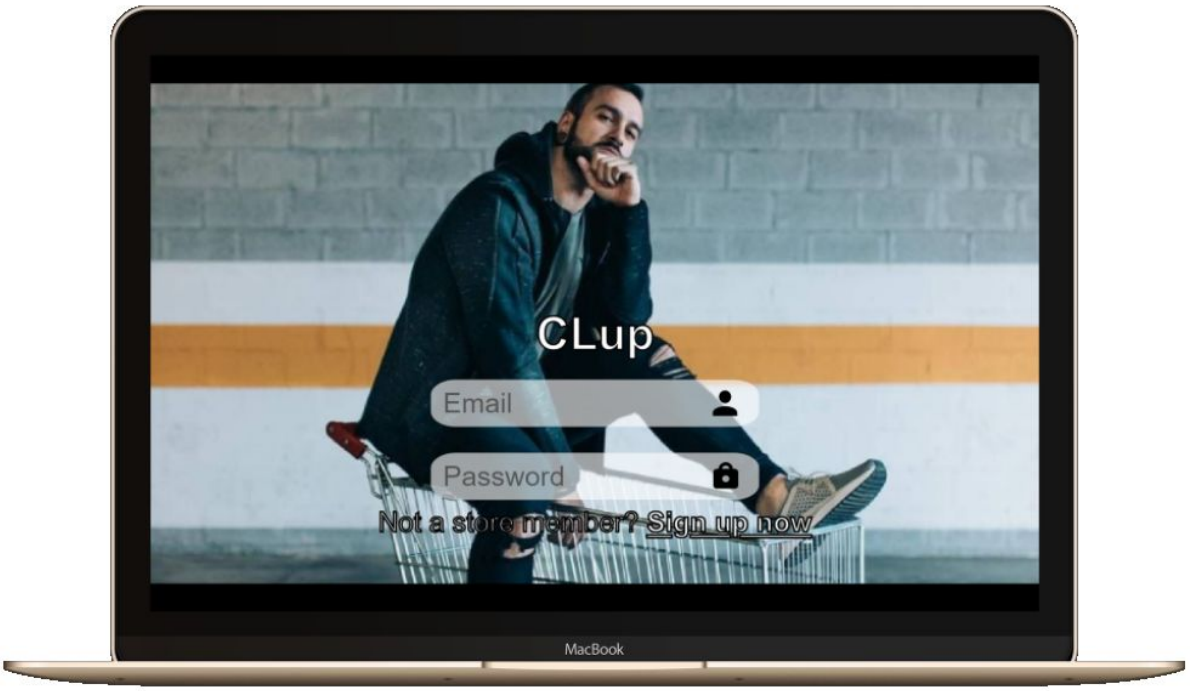

12:30
CONAD SUPERSTORE
CONAD
Description
The store has an extensive selection of different items.
LINE UP
BOOKING

8-Store Page

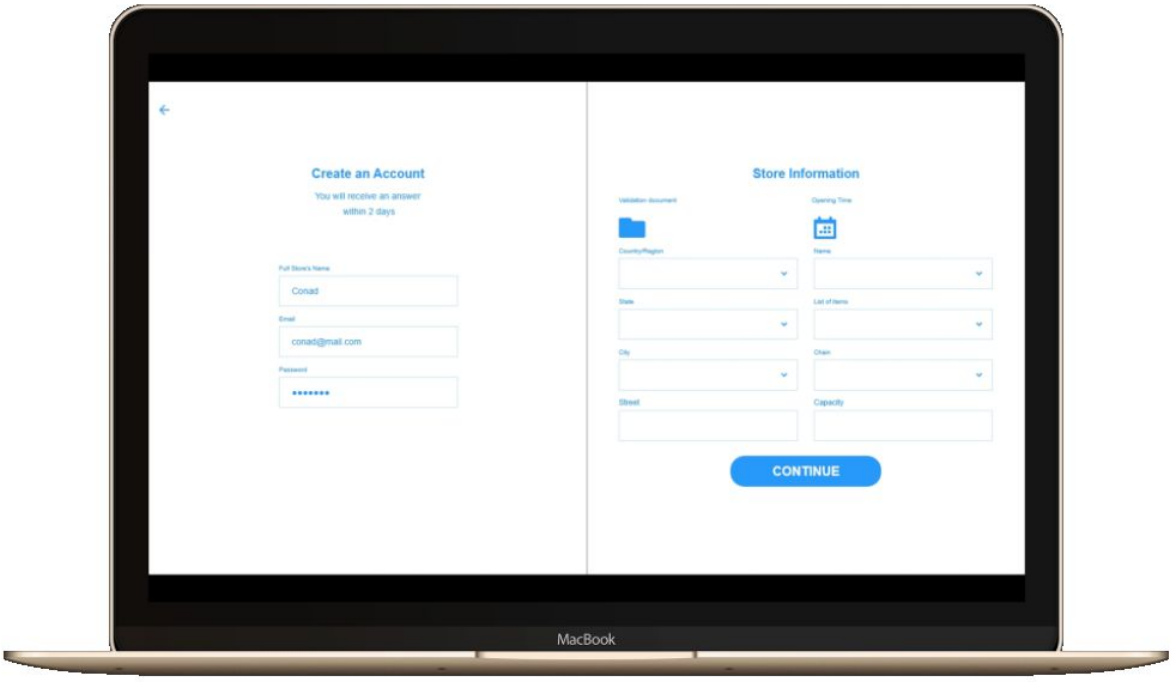



9-Menu

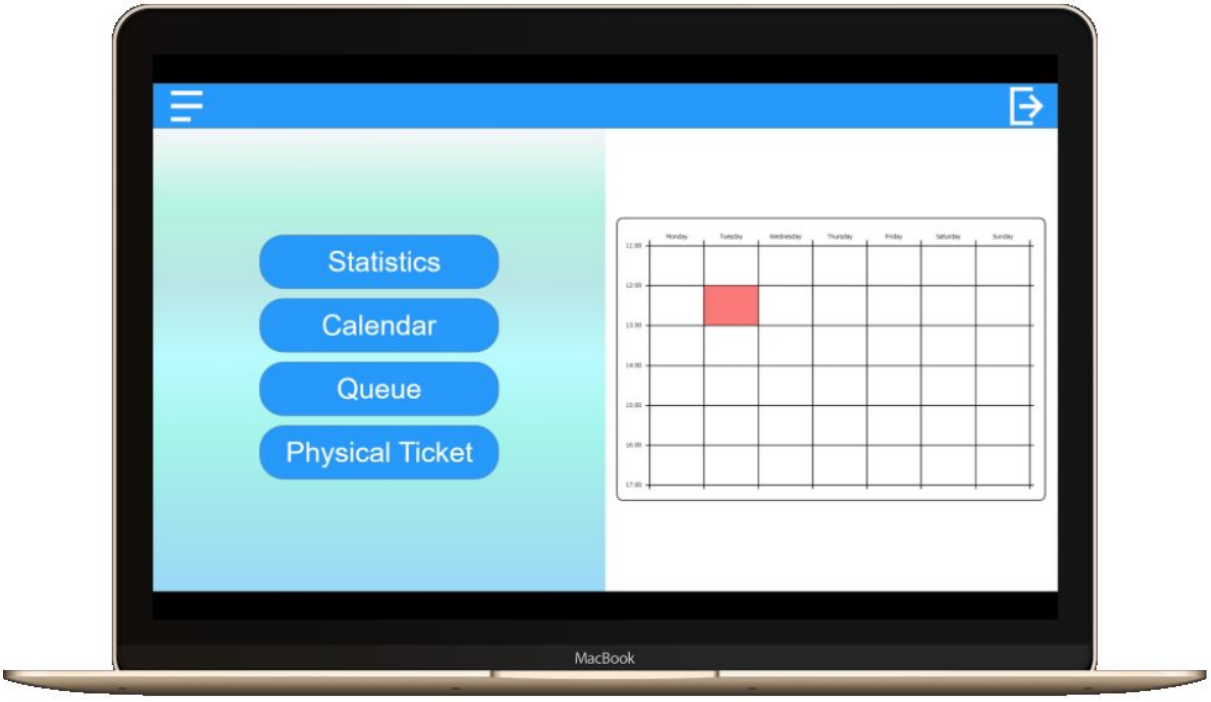

10-Ticket (QR is available)




11-LogIn Web




12-Store Registration




13-Store’s Home

# 4. Requirements Traceability

| | |
|---|---|
| **G1** | Allows store managers to regulate the influx of people in the building |
| | R2, R3, R7, R8, R12, R13, R16, R18, R19, R24 |
| | Components:<br>● UserMobileApp<br>● WebApp<br>● WebServer<br>● LineUpManager<br>● BookingManager<br>● ScheduleManager<br>● StoreManager[GeneratePhysicalTicket]<br>● AuthenticationManager<br>● DBMSServices |
| **G2** | Allows people to avoid hazards and wasting time lining up outside of the store |
| | R7, R9, R10, R12, R14, R15 |
| | Components:<br>● UserMobileApp<br>● WebApp<br>● WebServer<br>● LineUpManager<br>● BookingManager<br>● ScheduleManager<br>● AuthenticationManager<br>● DBMSServices |
| **G3** | Makes it easier to respect social distancing inside the store |
| | R9, R11, R14, R16, R19, R20, R21 |
| | Components:<br>● UserMobileApp<br>● WebApp<br>● WebServer<br>● StoreManager<br>● ScheduleManager<br>● DBMSServices |

| | |
|---|---|
| **G4** | Allows every demographic the possibility to use the service easily |
| | R1, R13, R24 |
| | Components:<br>● UserMobileApp<br>● WebApp<br>● WebServer<br>● StoreManager[GeneratePhysicalTicket]<br>● DBMSServices |
| **G5** | Notifies the user of the right time he needs to leave the house and go to the store by keeping track of the distance (between his house and the supermarket) and his travelling habits (by car, on foot, exc.) |
| | R4, R7, R9, R11 |
| | Components:<br>● UserMobileApp<br>● ScheduleManager<br>● GoogleMapsService |
| **G6** | Gives the store the possibility to hand out physical ticket if the user has not access to the needed technology |
| | R13, R24 |
| | Components:<br>● WebApp<br>● WebServer<br>● StoreManager[GeneratePhysicalTicket]<br>● ScheduleManager<br>● DBMSServices |

| | |
|---|---|
| **G7** | Allows people to book their visit in a future moment |
| | R1, R2, R5, , R6, R7, R10, R14, R15, R16, R17, R18 |
| | Components:<br>● UserMobileApp<br>● WebApp<br>● WebServer<br>● BookingManager<br>● ScheduleManager<br>● DBMSServices |
| **G8** | Stores the data concerning the visits of an user |
| | R2, R4, R7, R8, R9, R11, R17, R23 |
| | Components:<br>● ViewHistoryManager<br>● AuthenticationManager<br>● StoreManager[Statistics Service]<br>● DBMSServices |
| **G9** | Uses the stored data to find the optimal way to plan the visits |
| | R5, R9, R11, R12, R14, R22, R23 |
| | Components:<br>● ViewHistoryManager<br>● AuthenticationManager<br>● StoreManager[Statistics Service]<br>● ScheduleManager<br>● DBMSServices |
| **G10** | Gives the users the possibility to always find the nearest slot to their necessity, even if in other stores |
| | R2, R6, R7, R8, R9, R11, R19, R20, R21, R22 |
| | Components:<br>● ScheduleManager<br>● GoogleMapsService<br>● DBMSServices |

| | |
|---|---|
| **R1** | **User can require a ticket or booking a visit only with his telephone number** |
| **R2** | **Shops are certified with an authentication** |
| **R3** | **Shops should register to the application by filling the form with mandatory fields*** |
| **R4** | **Each store receives data only about its clients** |
| **R5** | **Each booking or request to line up must be certificated by a code sent by SMS** |
| **R6** | **Users shall authorise the system to send messages** |
| **R7** | **Users shall authorise the system to use their position** |
| **R8** | **The system must have the access to the data of the store** |
| **R9** | **Users shall authorise the system to acquire their data about purchasing habits** |
| **R10** | **A request of booking is also validated by the system before being confirmed** |
| **R11** | **Shops has no restrict to access the data of their customers** |
| **R12** | **Shops and users have to choose from a list of predetermined items (this is done for users to filter stores, in order to search for the ones providing desired items)** |
| **R13** | **The system has to coordinate physical slot with the app in order to have sequential tickets** |
| **R14** | **Users can choose the type of items from a predefined checklist when booking. They can select “no preference”** |
| **R15** | **Users that want to book a visit should compile a form in all its mandatory fields** |
| **R16** | **The booking system and the line system must be coordinated to ensure social distancing** |
| **R17** | **User can view in a registry the bookings already done** |
| **R18** | **User has to wait for the booking to be completed before booking again** |
| **R19** | **The system suggests possible alternatives to the clients** |
| **R20** | **The system must suggest other shops in the same municipality** |
| **R21** | **The system must suggest other time slots up to one week later** |
| **R22** | **The notification system is based on the client's habits** |
| **R23** | **The system can build statistics about client’s habits by considering precedent visits** |
| **R24** | **Each ticket and each booking have a QR code** |

# 5. Implementation, Integration and Test Plan

## 5.1 Overview

The first rule of survival in developing complex software is: never trust yourself 100%. Errors can be made and it's fundamental to test the code from the beginning, with Unit Tests, and not just in the final stages with a so-called Big Bang integration and testing. Stubs and drivers must be used to compose a realistic bottom up/top down testing environment too.

## 5.2 Implementation Plan

Our plan is a particular mixture of both bottom-up and top-down approach, to get the best of both worlds. With top-down, it is easier to get a "bigger picture", adn to model a better looking app, but it is difficult to meet granularity. With bottom-up instead, granularity is maximised, but with the tradeoff of losing system's unity and complexity.

Before describing the implementation plan, it is important to clarify two facts:

- The Redirector will be the last component to be implemented; this happens because its scope is to forward messages in the various components, and this can happen only after everything has been unit tested. With the Redirector, of course, come along also User Mobile App, Web App and Web Server, which are before the Redirector in the Component Diagram above.
- Google Maps will not be taken in consideration; this clearly happens because it is a trusted service from a world-famous company, and thus it is very unlikely to fail.

The order of Application Server components testing is the following:

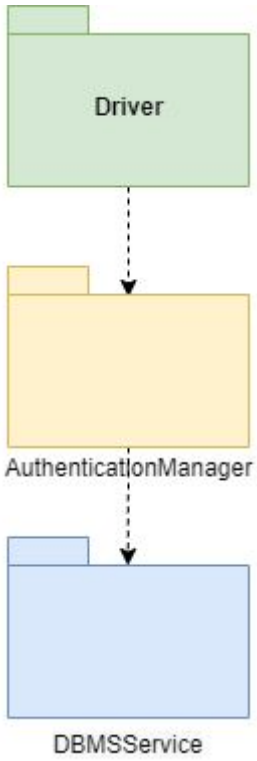
1. DBMS Service
2. Authentication Manager
3. Schedule Manager
4. Line Up Manager
5. Store Manager //physical ticket uguale a lineUp
6. Booking Manager
7. View History

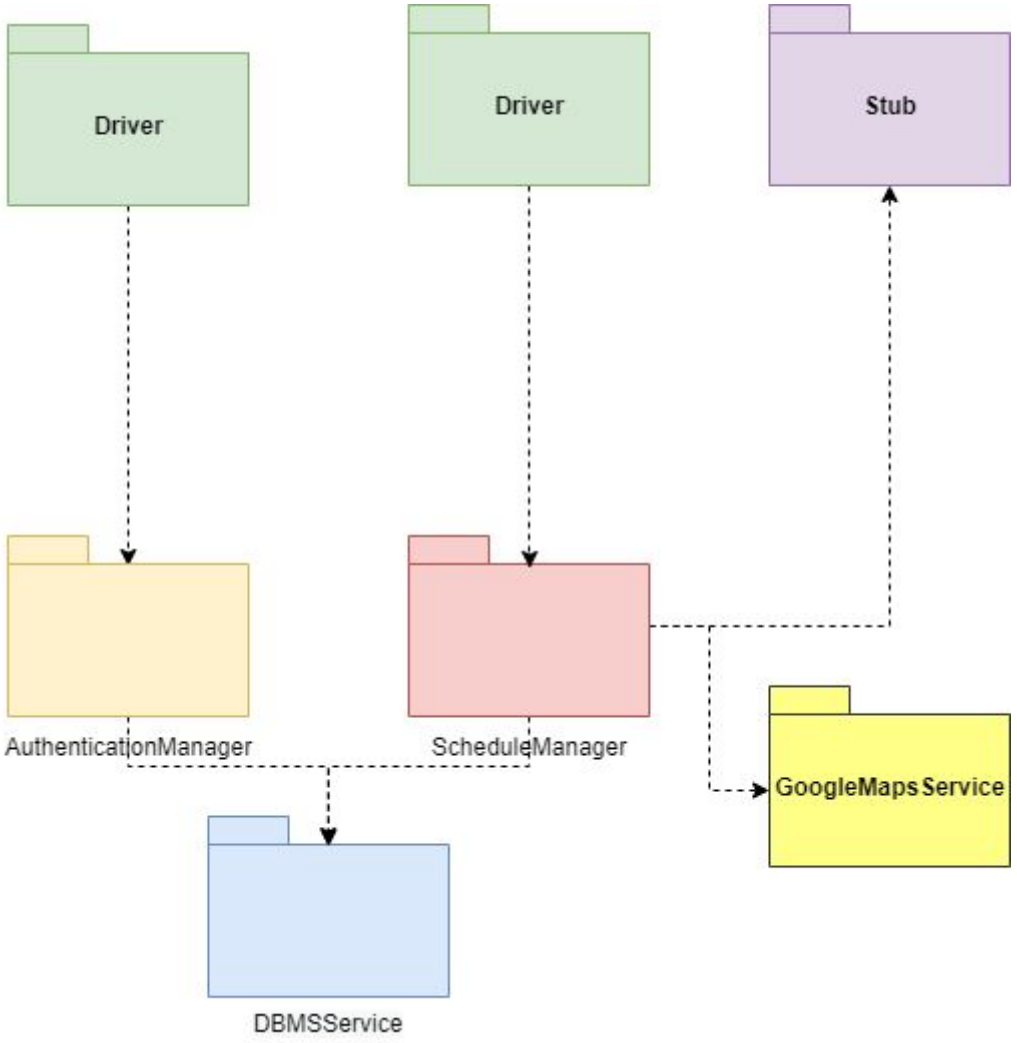
Web App and Web Server Testing can be done at any point, since with drivers usage is always possible to replace them and they do not take a fundamental part in the logic of CLup.

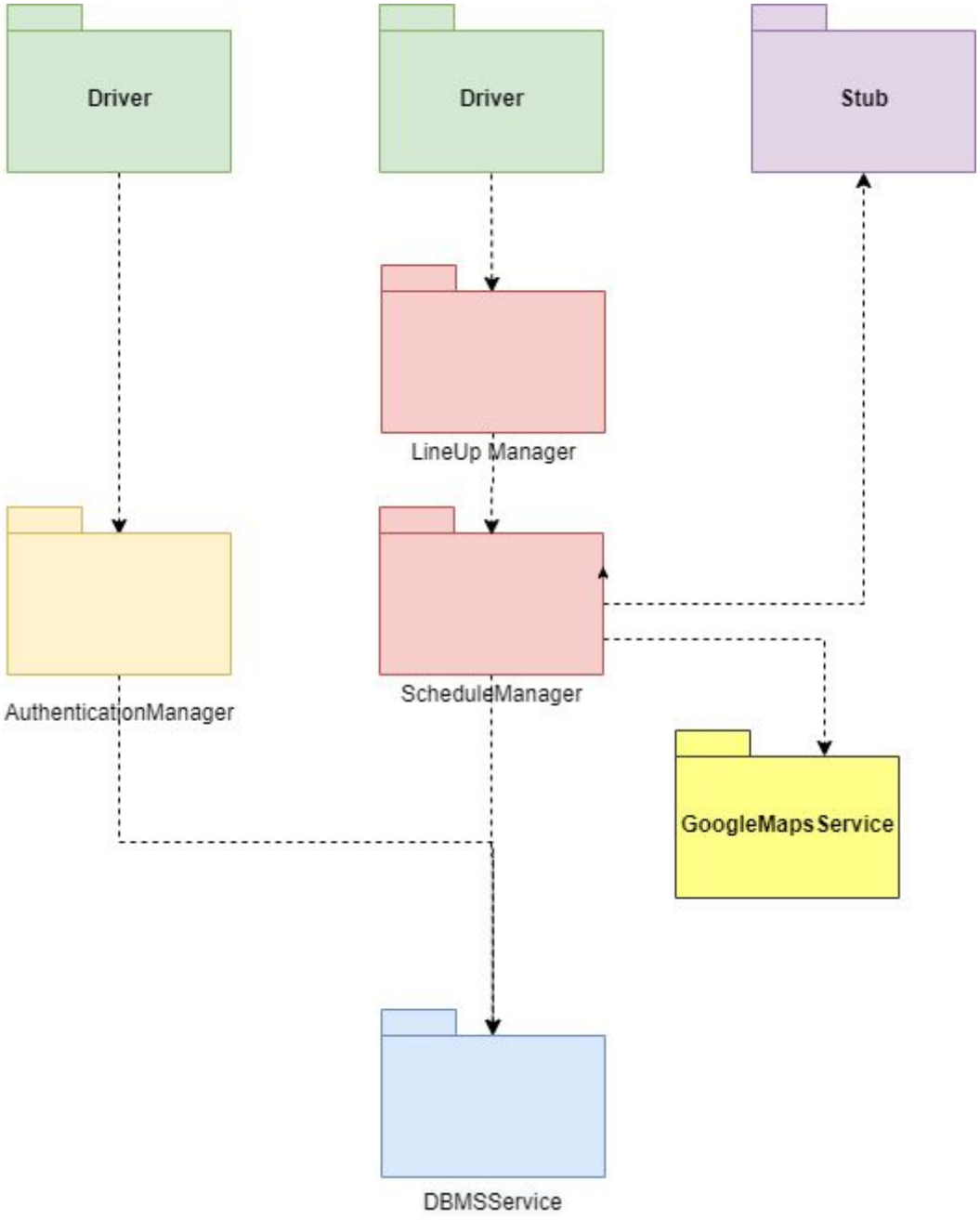
The order is a typical bottom-up testing: in case of “ties”, such as in Store Manager and Booking Manager Testing, they can be done in parallel, but some are preferred: in this case, testing Store Manager before is better since the GeneratePhysicalTicket service is nearly the same as LineUp.

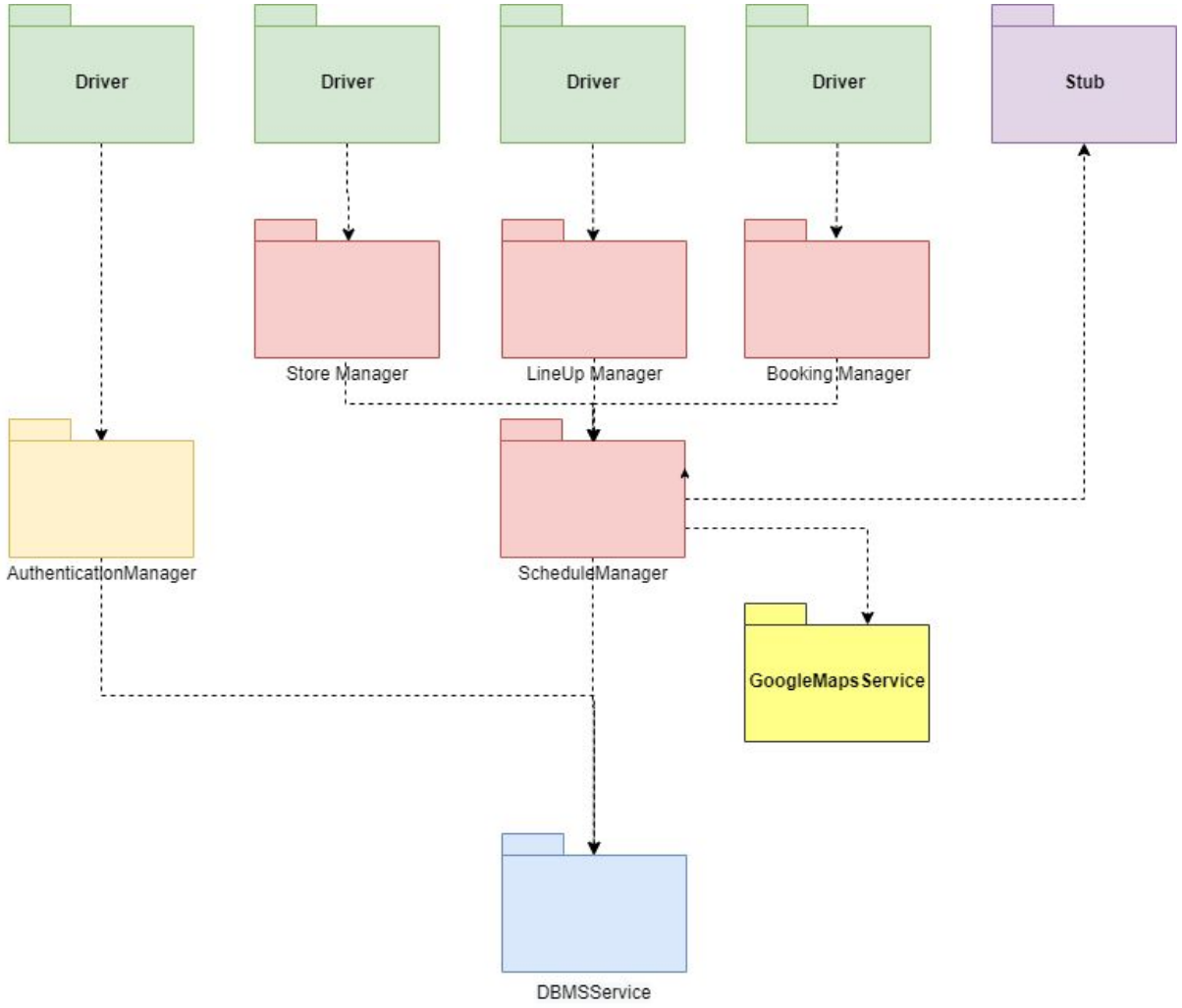
## 5.3 Integration Strategy

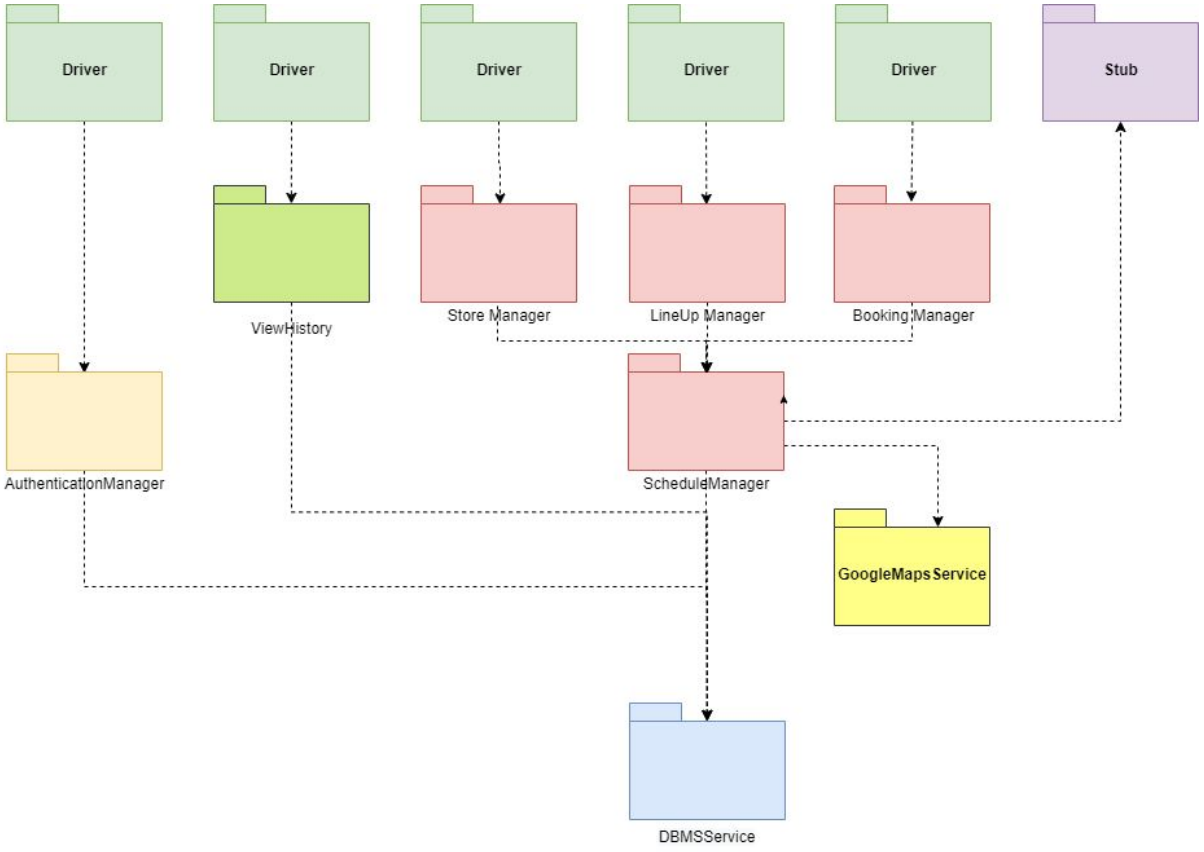
To implement and test the different functionalities of the system a hybrid approach has been used. The following diagrams describe how the process of implementation and integration testing takes place, according to a hybrid approach.

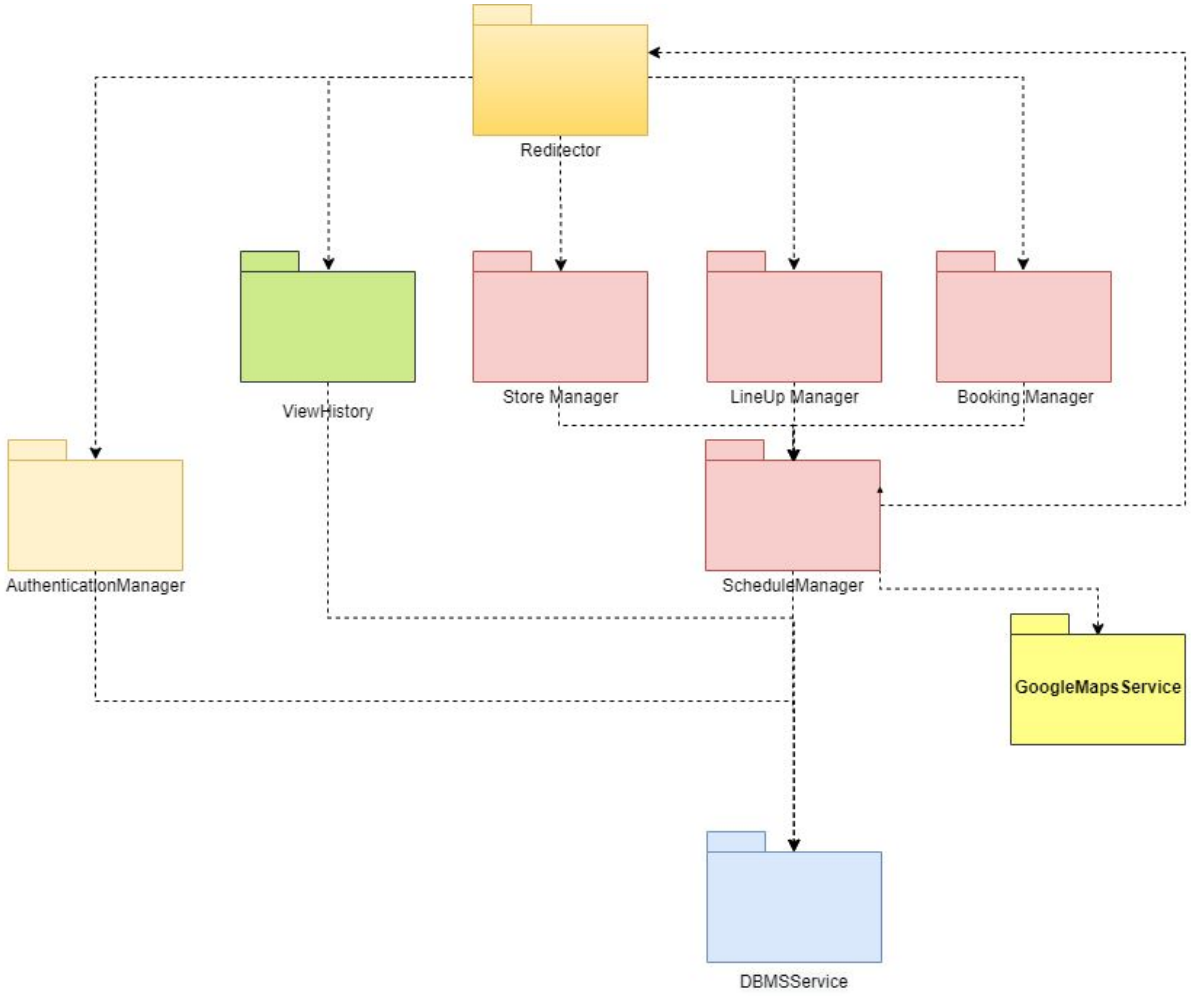



Driver
Driver
Stub
AuthenticationManager
ScheduleManager
GoogleMaps Service
DBMSService

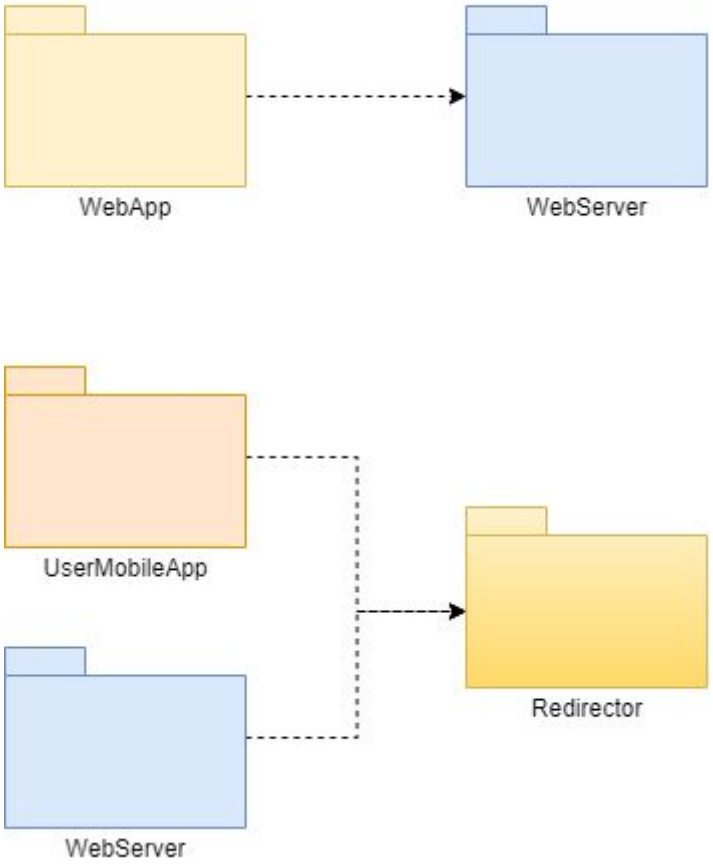

Driver
Driver
Stub
LineUp Manager
AuthenticationManager
ScheduleManager
GoogleMaps Service
DBMSService


Driver
Driver
Driver
Driver
Stub
Store Manager
LineUp Manager
Booking Manager
AuthenticationManager
ScheduleManager
GoogleMaps Service
DBMSService


Driver
Driver
Driver
Driver
Driver
Stub
ViewHistory
Store Manager
LineUp Manager
Booking Manager
AuthenticationManager
ScheduleManager
GoogleMaps Service
DBMSService


Redirector
ViewHistory
Store Manager
LineUp Manager
Booking Manager
AuthenticationManager
ScheduleManager
GoogleMaps Service
DBMSService

7) Finally, the remaining parts characterizing the client side must be implemented, unit-tested and integrated into the system. This process happens only when components of both sides have been implemented and tested.Once every component has been tested and put into the entire application, system testing can be performed.




## 5.4 System Testing

As said before, this part should ONLY come after Unit Testing. In this way, different components have already been evaluated singularly, without a more complex environment to debug.

First of all, **Performance Testing** is one of the most useful tests for assessing the user experience, since no one wants a big slow App in his phone. In this part bottlenecks are found and solved thanks to time-evaluating tools, while

benchmarking compares different phases of the project and controls the most efficient ones.

Of course, **Stress Testing** should be done: having the system's limitations, such as a predefined maximum number of HTTPS requests, the network could be tested by asking the double of such numbers, and seeing how the system reacts.

Finally, **Load Testing** should be done as well: some components, for example the Suggestion Service, hold in memory the SuggestionID in order to keep information about the User who received it. It is the only way this critical component can translate an incoming accepted Suggestion from a client into useful information to send to the Ticket Generator. Since each User can have multiple suggestions, and the Users are assumed to be many, denying memory leaks is a key factor; this can be seen in this part of the system testing, since, once reached the threshold and with useful memory-leak-controlling tools, memory problems should be evident.

## 5.5 Additional Specification on Testing

After having explained the "dynamic" part of the testing, such as Unit and System Testing, here the "static" testing, which will be executed on CLup, will be presented.

In the Inspection, external actors can observe the code written and discuss it. No more than 2x2 hours a day, a team composed of: the CLup developers and inspectors (who read the code the everyone) and a moderator (always external from the project), searches for bugs and system failures, without actually executing the program (could be done rapidly to check fine aspects that would be too slow to observe without running the program). Developers are not badly evaluated in this part; this is done to discourage hiding bugs.

After the Inspection, comes the Rework fase: in it developers try to fix problems found before and are badly evaluated for bugs, since in this part they are supposed to fix and not add problems.

Of course these two parts, Inspection and Rework can be repeated: this is useful every time the program is modified in more than 5% of its code.

# 6. Effort Spent

**Mangano Davide**

| Topic | Hours |
|---|---|

| | |
|---|---|
| General discussion | 2h |
| Deployment Diagram | 3h |
| Runtime View | 4h |
| Architectural style | 2h |
| Other Design Decisions | 1h |
| Overview, Implementation Plan | 2h |
| System testing, Additional Specification on Testing | 2h |
| Revision | |

# 7. References

- The UML diagram was made with StarUML
- The other diagrams were made with Visual Paradigm Online - Suite of Powerful Tools
- The mockups were made with Strumento di UI/UX design e collaborazione
- This document was made with Google Docs