POLITECNICO
MILANO 1863

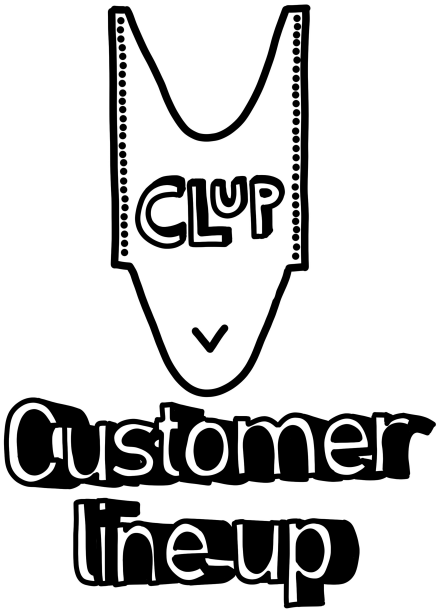
SOFTWARE ENGINEERING 2

Design Document

January 10, 2021

Authors:
Federico Mainetti Gambera, 10589654
Ludovica Lerma, 10522723

Professors:
Prof. Matteo Rossi
Prof.ssa Elisabetta Di Nitto



Contents

List of Figures

List of Tables

1 Introduction

1.1 Purpose

This document is mainly addressed to the developement software team. Its purpose, indeed, is to provide the reader of an overview of the architecture and design choices of the software product.

1.2 Scope

Clup is a digital application which will provide the following services to the clients:

- Browse of registered supermarkets
- Join a virtual queue of registered supermarkets
- Book up a shopping session in registered supermarkets
- Browse of previous and future shopping sessions

Clup is a digital application which will provide the following services to the shop-managers:

- Registration of their shop on the system
- Management of the registered shops

1.3 Definitions, acronyms and abbreviations

With the term **queue**, a disambiguation is needed: here and in every other part of this document, we indicate a series of tickets in temporal order, either queue tickets or visit tickets.

However, from time to time, the terms *queue, dequeue, enqueuement, dequement* will be referred only to users who decide to line up in the **virtual queue**, intended as the one composed only of queue tickets.

Even though it may seem confusing now, we have retained that the context will be enough of a disclaimer for the user to have clearly in mind which kind of queue we are talking about.

The following table [1] shows all the acronyms used in the document.

Acronyms	
API	Application Programming Interface
DD	Design DOcument
JPA	a grocery shop owner or administrator who hasn't already adhered to CLup
BCE	Business Controller Entity
AJP	Apache JServ Protocol
JPA	Java Persistence API
RASD	Requirements Analysis and Specifications Document
UX	User Experience
JS	JavaScript
JDBC	Java DataBase connection
FIFO	First in, first out
DMZ	Demilitarized zone
IaaS	Infrastracture as a Service
ELB	Elastic Load Balancer

Table 1: Definitions

2 Architectural design

2.1 Overview: High-level components and their interaction

The architecture of the system can be divided in three logical layers: **presentation layer**, **business layer** and **data layer**.

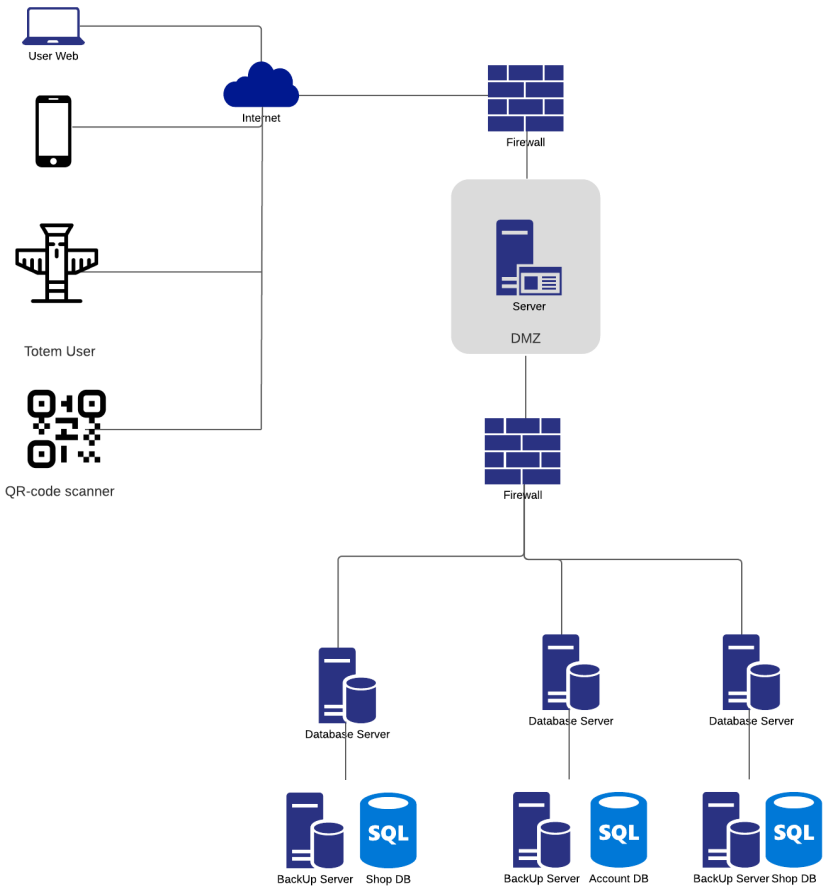


Figure 1: Physical Diagram

The **presentation layer** is the part of the application that manages the interface through which the user will actually interact with our application.
This layer will be managed either on client side, for what regards the offline functionalities and the static content display, and on server side, for what regards the dynamic content display.
It has been chosen to develop a cross-platform application: in order to do so, the front-end of the application will be developed according with the framework of Redux in combination with React and React-Native.
This means that the front-end will be supported either by IOs and Android Devices, as well as Web browsers. Indeed, in the figure [1] we see that our users can connect with the server-side of the application via mobile devices, laptop devices, Totem devices and the Qr-code Scanner.
Mobile devices will be able to connect either via their respective native application or the browser-based application.
For what concerns the laptop users and the totem users, they will be able to connect to our application

using only the browser-based application.
Qr-code scanner application is a little more complicated and deserve a deeper explanation. It will be composed of two applications which will cooperate in order to correctly scan the Qr-code on the tickets and consequentially unlock the turnstills. One of the application, which will be implemented by us, will be just composed of a basic native UI and a QR-code scanner, while the underlaying application will provide the unlock and lock of the turnstills.
The latter won't be discussed in this document any further since it's not under our supervision.

The **Business layer** is the core of the application and connect the presentation layer and the data layer. It allows users to retrieve desired data and to have access to dedicated functionalities. We decided to implement our business layer using the Micro-Services architecture. This architecture divides the application in multiple tinier applications, each of which is mainly dedicated to a specific function. All of those micro-services coadiuvate in order to offer to the user the entire set of features offered by the application. This is the best way to implement an application which must be scalable, as CLup, which is a young start-up hoping to grow a lot. Furthermore, we decided to implement a cloud computing based application: through the joined use of Kubernetes and Docker it is possible to scale the resources needed dynamically and according to the incoming requests, so that, with a IaaS approach, we will be able to reduce the costs to the essential.

By the way, each of our micro-services will need, in order to fulfill its duty, to store and retrieve several information. This information will be kept in dedicated SQL Data Bases, which represents, jointed with the SQL DBMS, the **Data Layer**. The data layer is the memory of our application and a significant part of it, for this reason for each database a Backup database will be implemented either.

2.2 Component view

The following diagrams [2] show the main components of the system and the interfaces through which they interact. The coloured ones are the one we are going to implement:

- The client side is made of four components, which refer to the qr-code scanner application, to the web users and to the mobile application users.
- The server side is divided in three subsystems, each of them implements a specific service of the application. For the sake of clarity three different diagram has been modelled. The subsystems are:
  - the **Queue Services Subsystem**: provides the interfaces needed either by managers and users in order to accesses to the functions correlated to the queue associated to a shop.
  - the **Shop Services Subsystem**: provides the interfaces needed by managers and users in order to access the functions correlated to a shop.
  - the **Account Manager Services**: provides the interfaces needed to manage the accounts and also implements some security features.

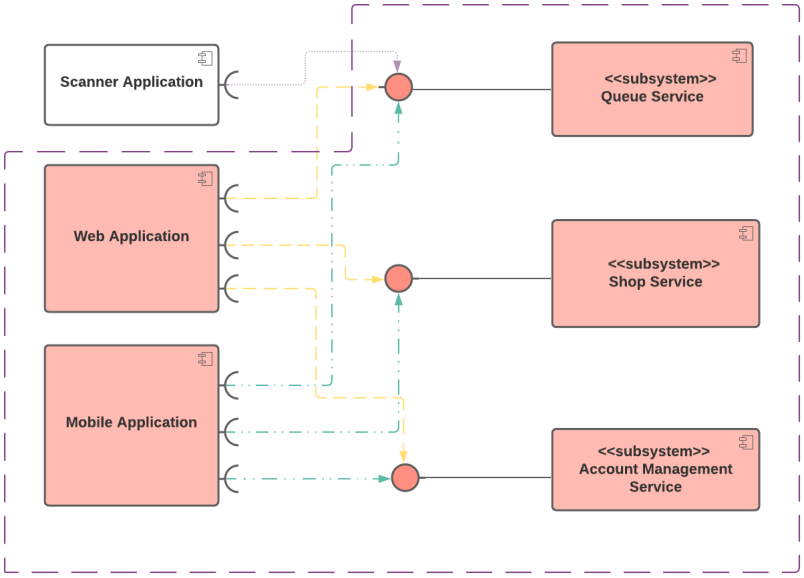


Figure 2: General component view

2.2.1 Queue Service Subsystem Projection

This service is divided in several components: *Qr-code scanner component, Visit component, LineUp Component, Queue Info Component, Analytics Component, Ticket Generator Component, Notificate User Component and Data Manager Component*.

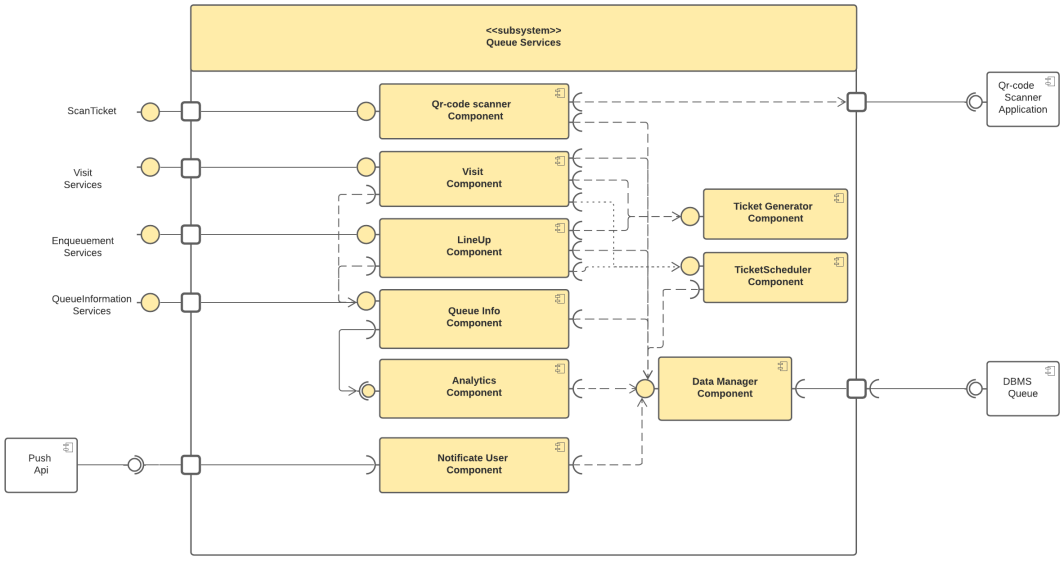


Figure 3: Queue Service Component

The **Qr-code scanner Component** is a component dedicated to update the status of the ticket once they are scanned. It communicates with the QR-code scanner, offering it an interface through which it will be able to call it, in order to correctly fulfill its function.

The **Visit Component** is a component dedicated to the management of the visit tickets. It generate

a ticket through the *Ticket Generator Component* and decorate it as a visit ticket. It offers to the client a set of functionalities, such as book a visit or cancel a previous booked one.

The **LineUp Component** is a component dedicated to the management of the queue tickets. It generates a ticket through the *Ticket Generator Component* and decorate it as a queue ticket. It offers to the client a set of functionalities, such as the possibility to enqueue or to dequeue in the virtual queue.

The **TicketScheduler Component** is a component dedicated to the update of the queue. If offers an interface to the *LineUp Component* and to the *Visit Component* and interact with the *DataManager Component*. Indeed, it offers a dedicated interface to the clients.

The **Queue Info Component** is a component dedicated to the retrievement of the information about the queue of a shop. Indeed, it offers a dedicated interface to the clients. It may have the need to handle immediate information or analytical data. The latter, are calculated thanks to the *Analytics Component*.

The **Analytics Component** is a component dedicated to the computation of analytical data about the queue.

The **Notificate User Component** is a component dedicated to the task of sending push notifications to clients. In order to so, it communicates with a Push Api.

Finally, the **Data Manager Component** is a component dedicated to interact with the Queue DBMS and offers an interface to all of the other component in the service in order to have access to it.

2.2.2 Shop Service Subsystem Projection

This service is divided in three components: **Manage Shop Component, Shop Info Module, Data Manager Module**.

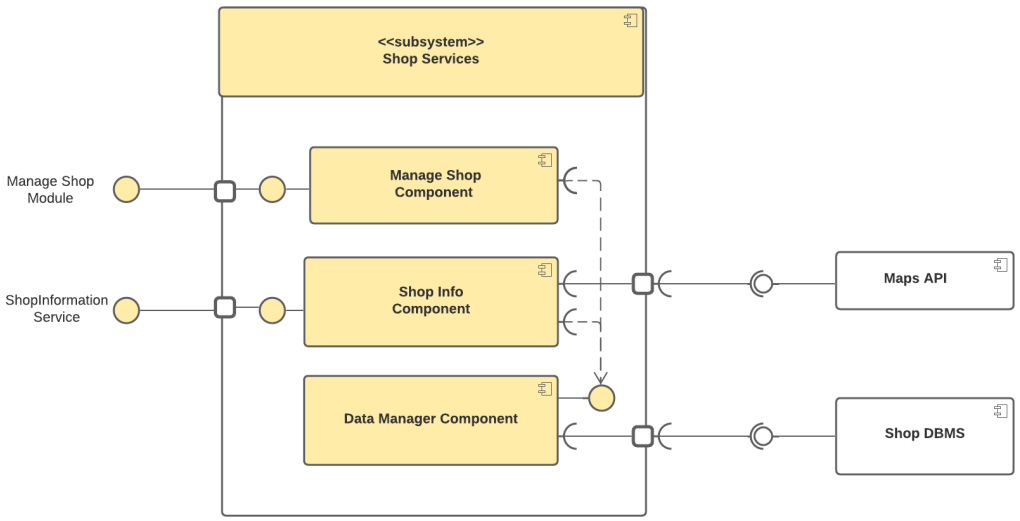


Figure 4: Shop Services Component

The **Manage Shop Component** is a component dedicated to implement features exploited mainly by the managers. It implements an interface which will allow a manager to add a shop and to update the information of a shop already registered in the system.

The **Shop Info Component** is a component which implements an interface that provides functions to retrieve information about a shop. This component, for some of its features, needs to communicate with the Maps Api.

Finally, the **Data Manager Component** is a component dedicated to interact with the Shop DBMS and offers an interface to all the other component in the service in order to have access to it.

2.2.3 Account Manager Subsystem Projection

This service is divided in three components: **Account Manager Component, Authentication and Authorization Engine Component and the Data Manager Component**.

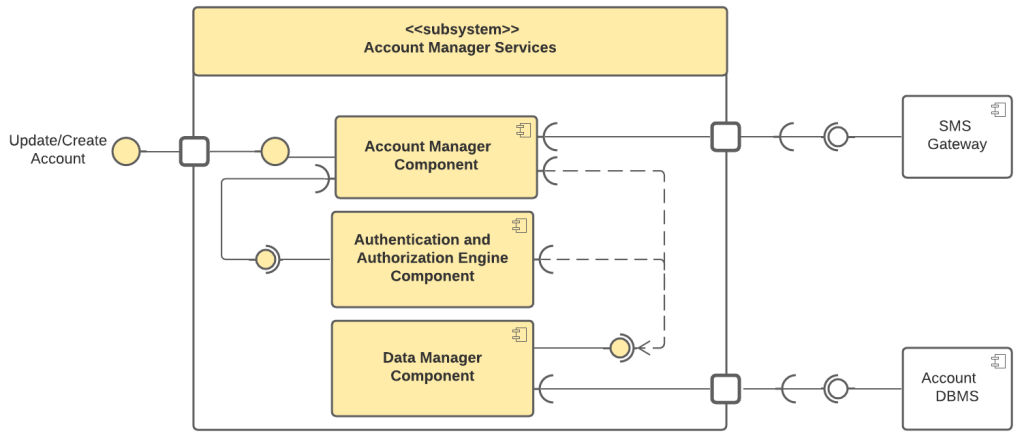


Figure 5: Account Manager Services Component

The **Account Manager Component** is a component dedicated to manage the users' accounts. It offers the users an interface through which they can, for example, sign on, sign in or update their profile.

It communicates with the **Authentication and Authorization Engine Component** which provides an interface with the functions needed in order to identify and authenticate a user. It will need to communicate with the SMS gateway from time to time in order to accomplish its functions.

The **Data Manager Component** is a component dedicated to interact with the Account DBMS and offers an interface to all of the other component in the service in order to have access to it.

2.3 Data Base structure

To show the database structure we have summarized all the main data with which the CLup system will interact and we have organized them in a schema that represents the structure of the entire database.

Furthermore, since our application is divided into three main services, the database will also reflect this division, as can be seen from the different colors in the image [6].

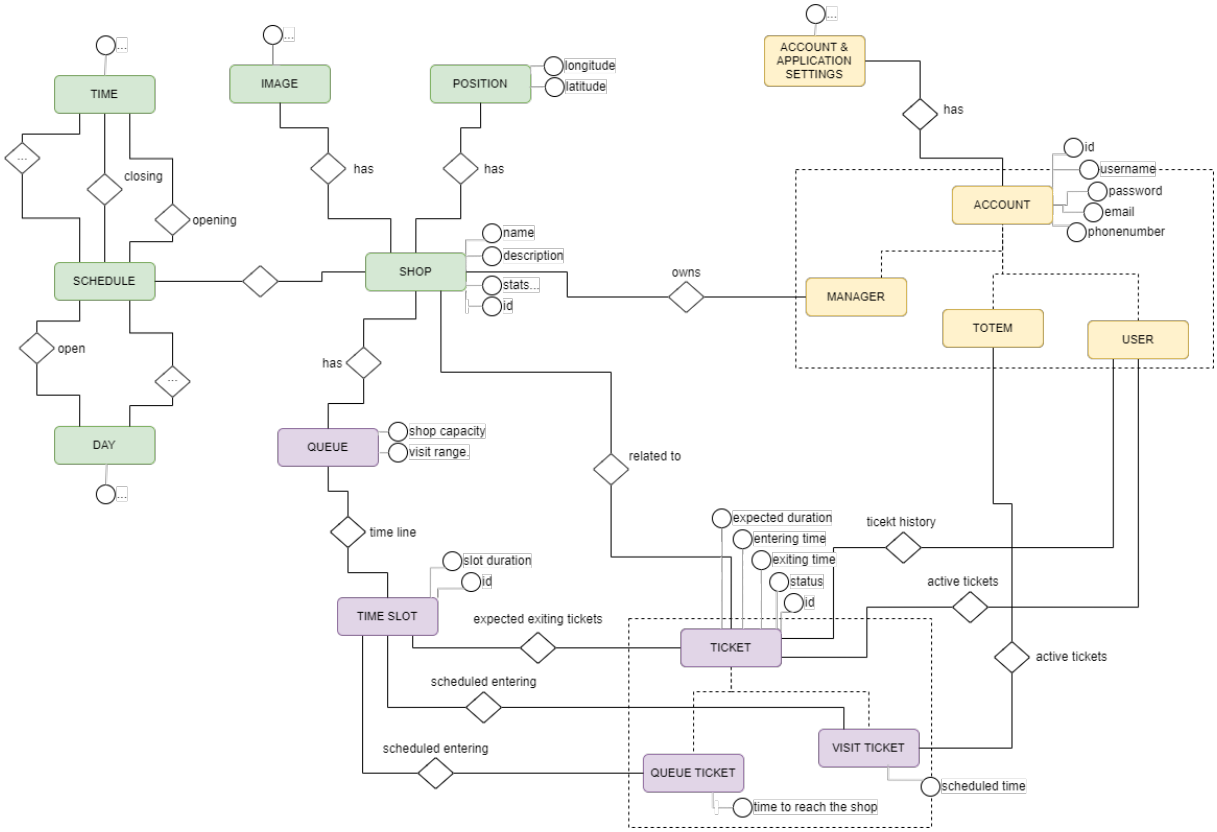


Figure 6: Data Base structure

2.4 Deployment view

The system architecture is a 3-Tier architecture, it is based on the JEE framework and it's typical of a Service Oriented Application:

- **TIER 1** is the client tier, it is composed of the mobile applications, the totems and the users browsers and the qr-code scanner application. All of them will communicate with the web server via the HTTPS protocol.
- **TIER 2** is composed of the application server. This tier hosts the OS system CentOs, which will welcome the platform Kubernetes which, with the joint effort of Docker, will orchestrate the ideal enviroment execution for our microservices application. Kubernetes, among its numerous services, offers an ELB api, through which will successfully dispatch correctly the incoming HTTP requests to the correct service. Each service will be deployed as a Docker Image directly by Kubernetes through a Docker File and will be hosted in a Docker container.
- **TIER 3** is the Data Tier. We decided to implement 3 different Databases, one for each offered microservice. The connection beetween Tier 2 and 3 is performed via JDBC connector which uses jdbc protocol.

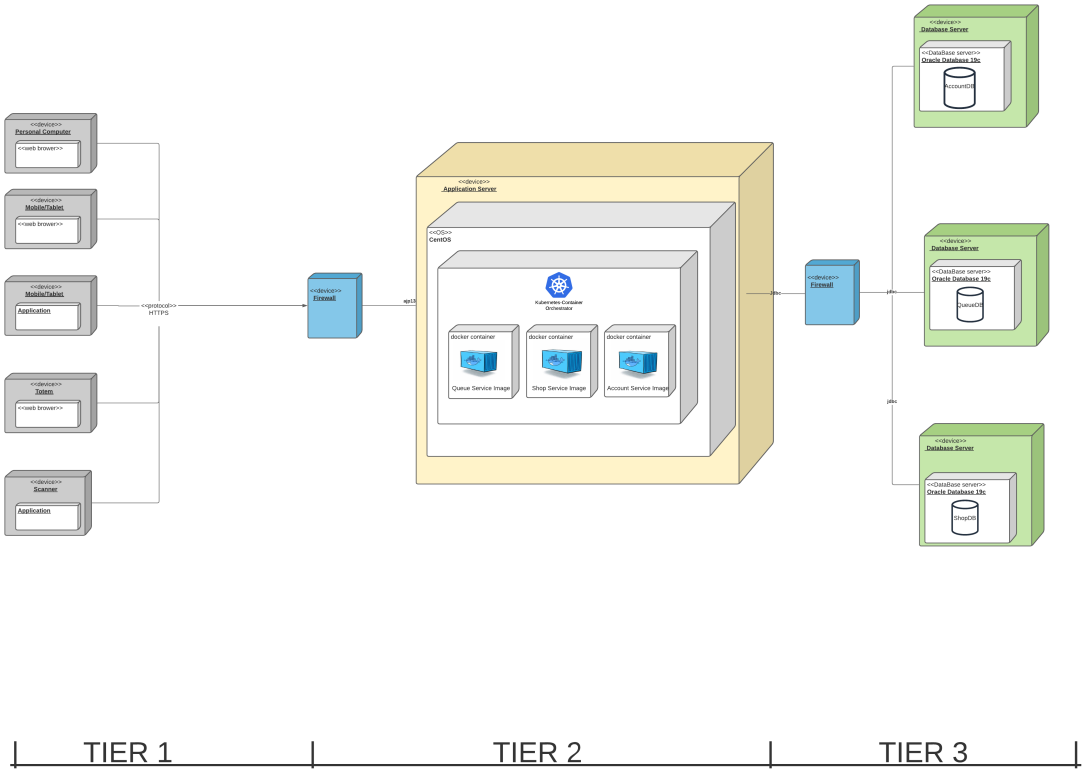


Figure 7: 3-Tier Architecture

2.5 Runtime view

The following sequence diagrams describe the interaction between the main components of the system in some of the most common use case scenarios. It is very important to highlight that these diagrams have no intention of specifying the detailed behavior of our product, but they have just an illustrative purpose. Indeed, their aim is to clarify with approximate sketches some of the interactions that may occur between the user and the system and, inside the latter, between the components. For this reason, some of the trivial thus fundamental behavior of the system have been omitted.

Between these, we want to specify in these instance the main two significant ones that the reader may find strange not to find in the diagrams.

First of all, we reassure the reader that all the data inserted in the forms compiled at client side will be verified also at server side, even if these checks aren't always represented.
In second instance all the features that will affect the queue of a shop- some of them in sequence diagrams 2,3,4- TicketScheduler Component in Queue Service Subsytem will be called and will perform the dedicated algorithm in order to update the queue. This will also trigger the push notifications to all the users involved through the Push Api.

With these premises, we shall let the reader continue their inspection of the following diagrams.

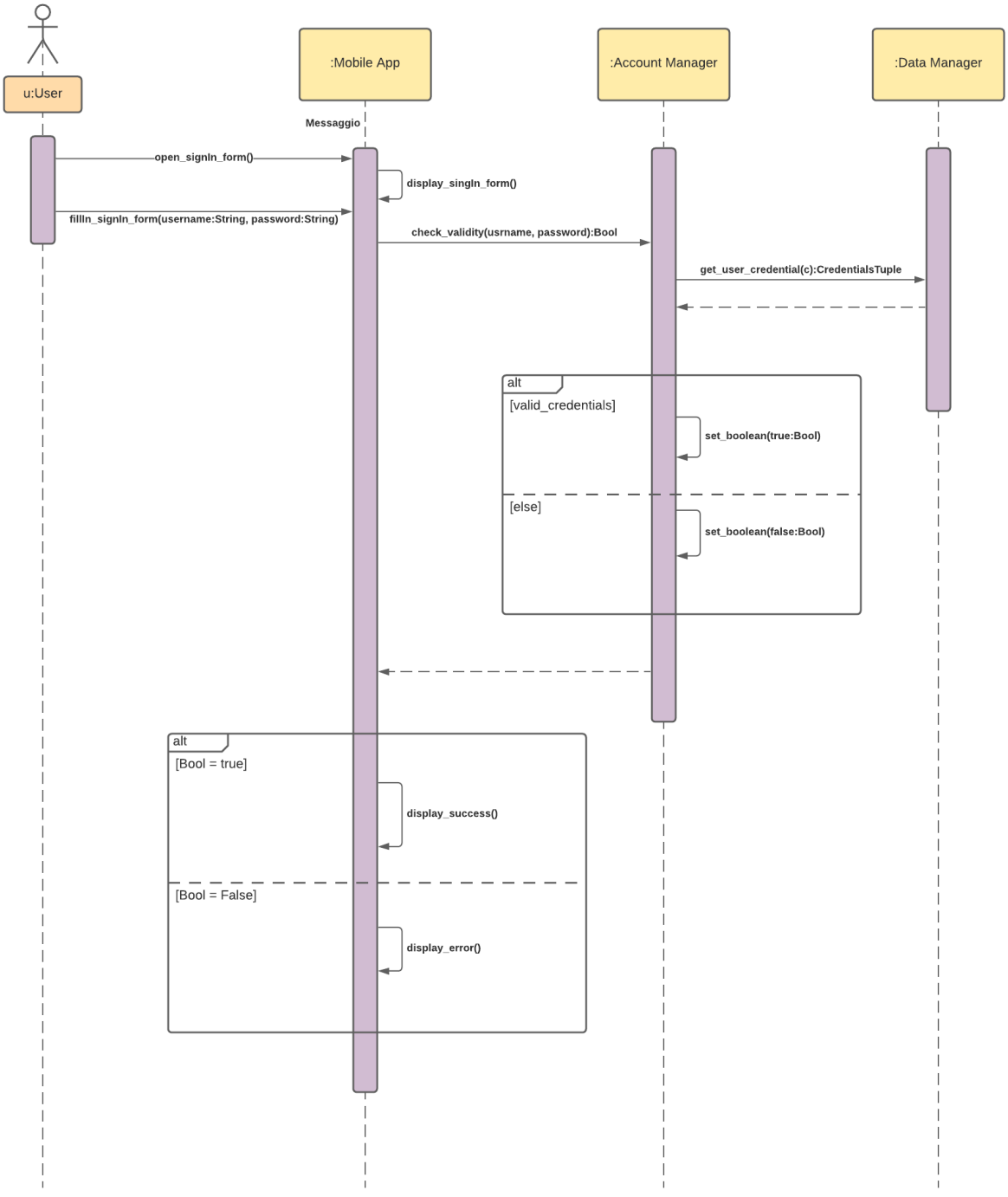


Figure 8: 1.User Logs In

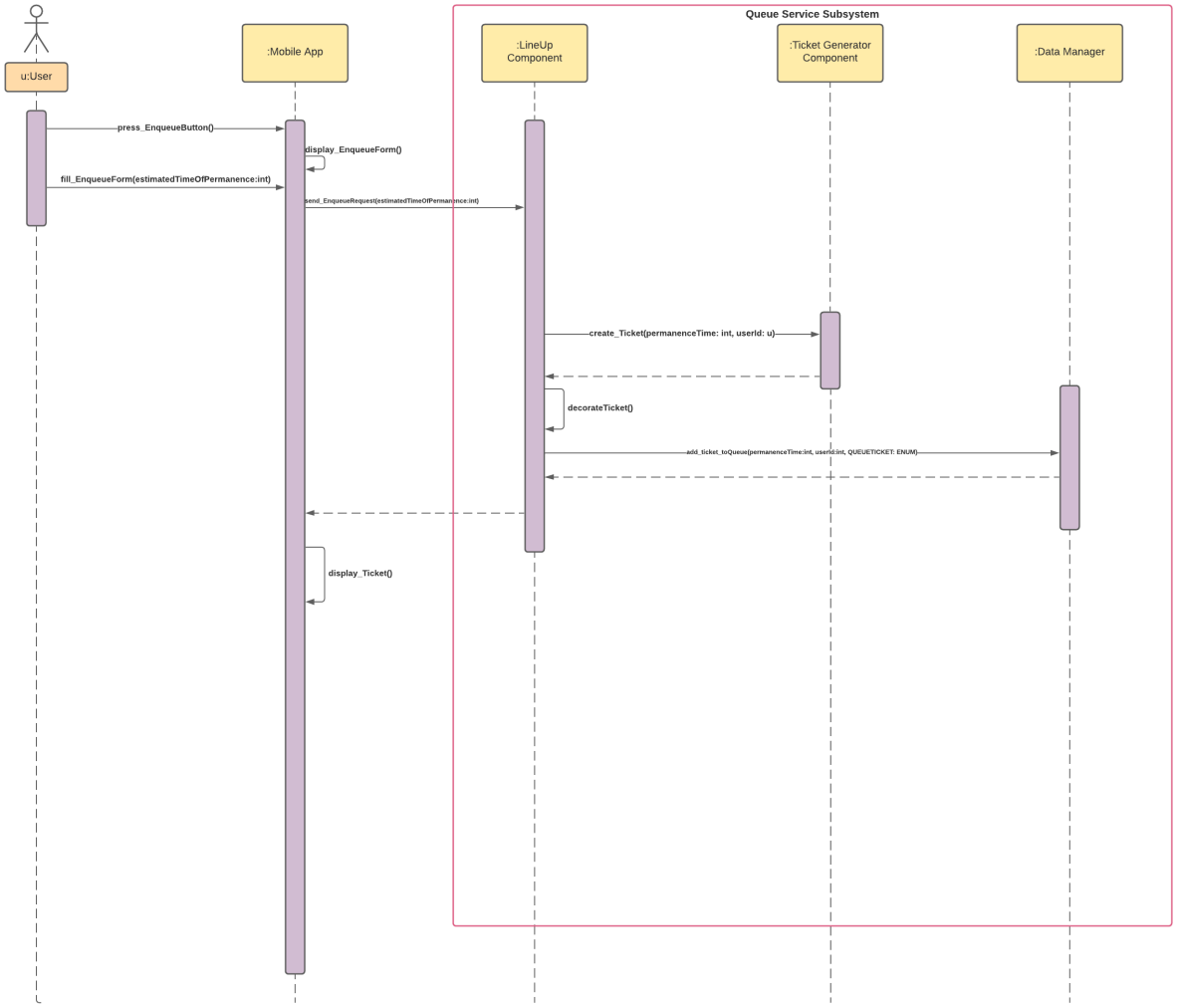


Figure 9: 2.Customer Enqueues

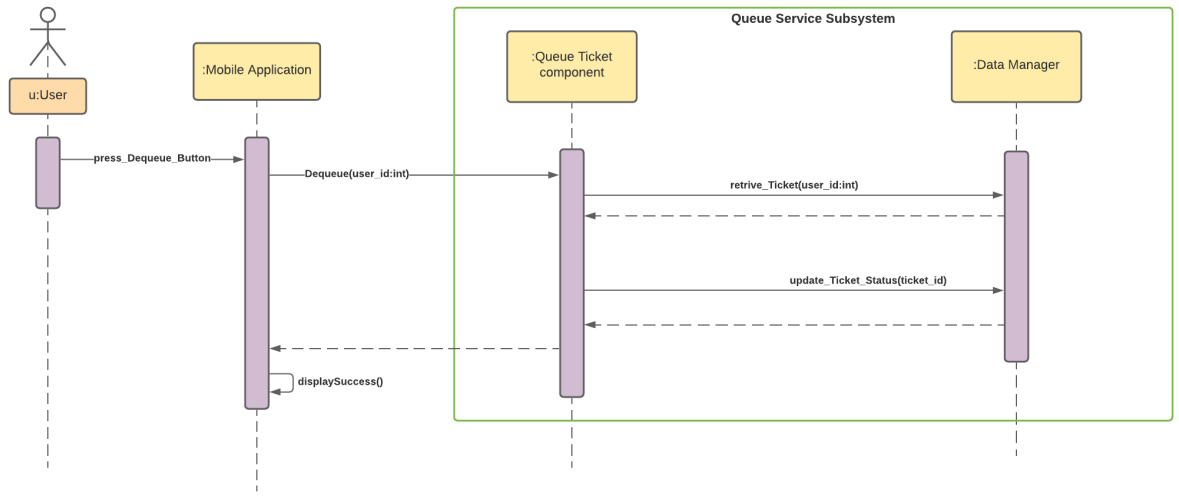


Figure 10: 3.Customer Dequeues

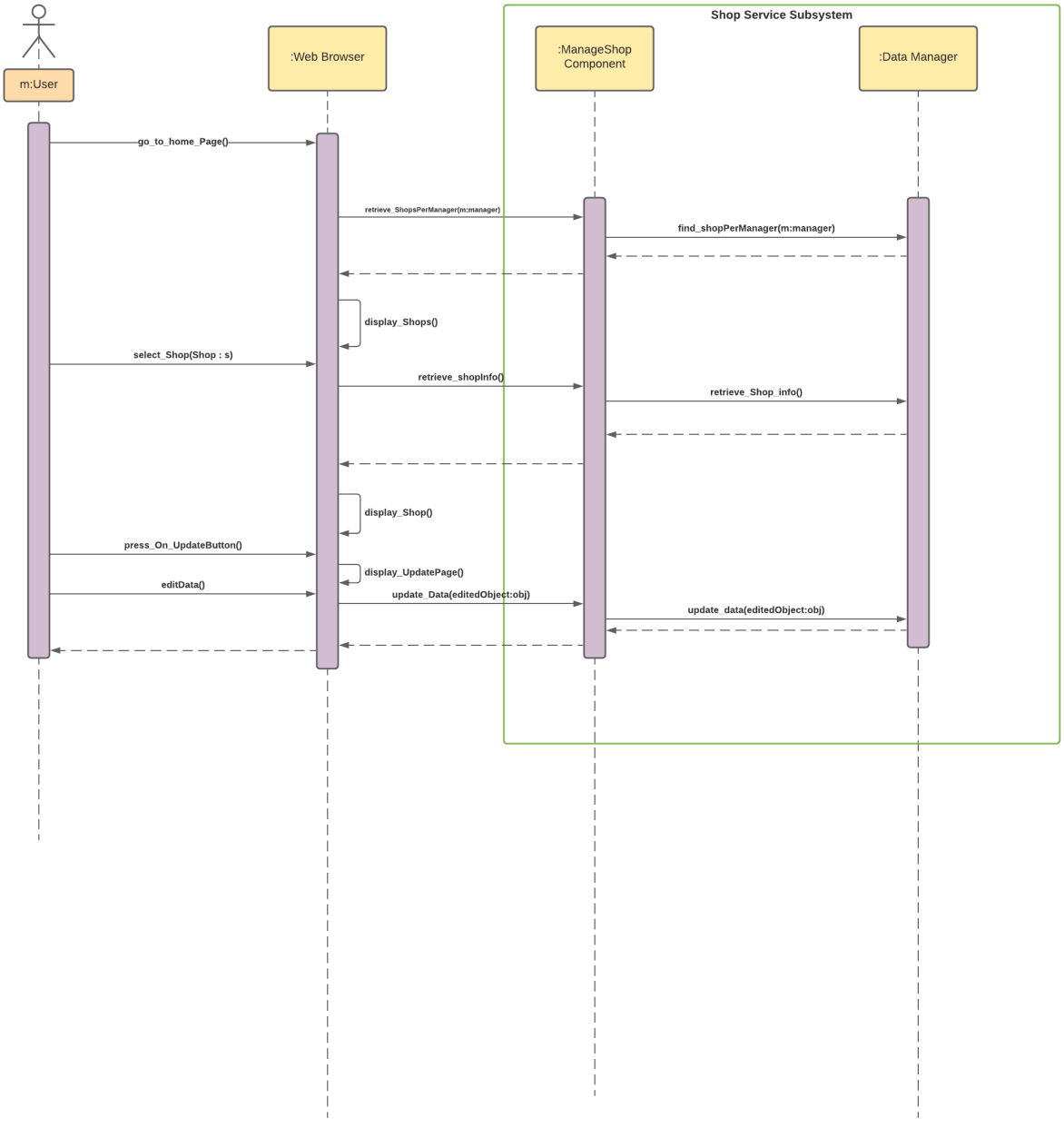


Figure 11: 4.Manager Updates Some Shop Information

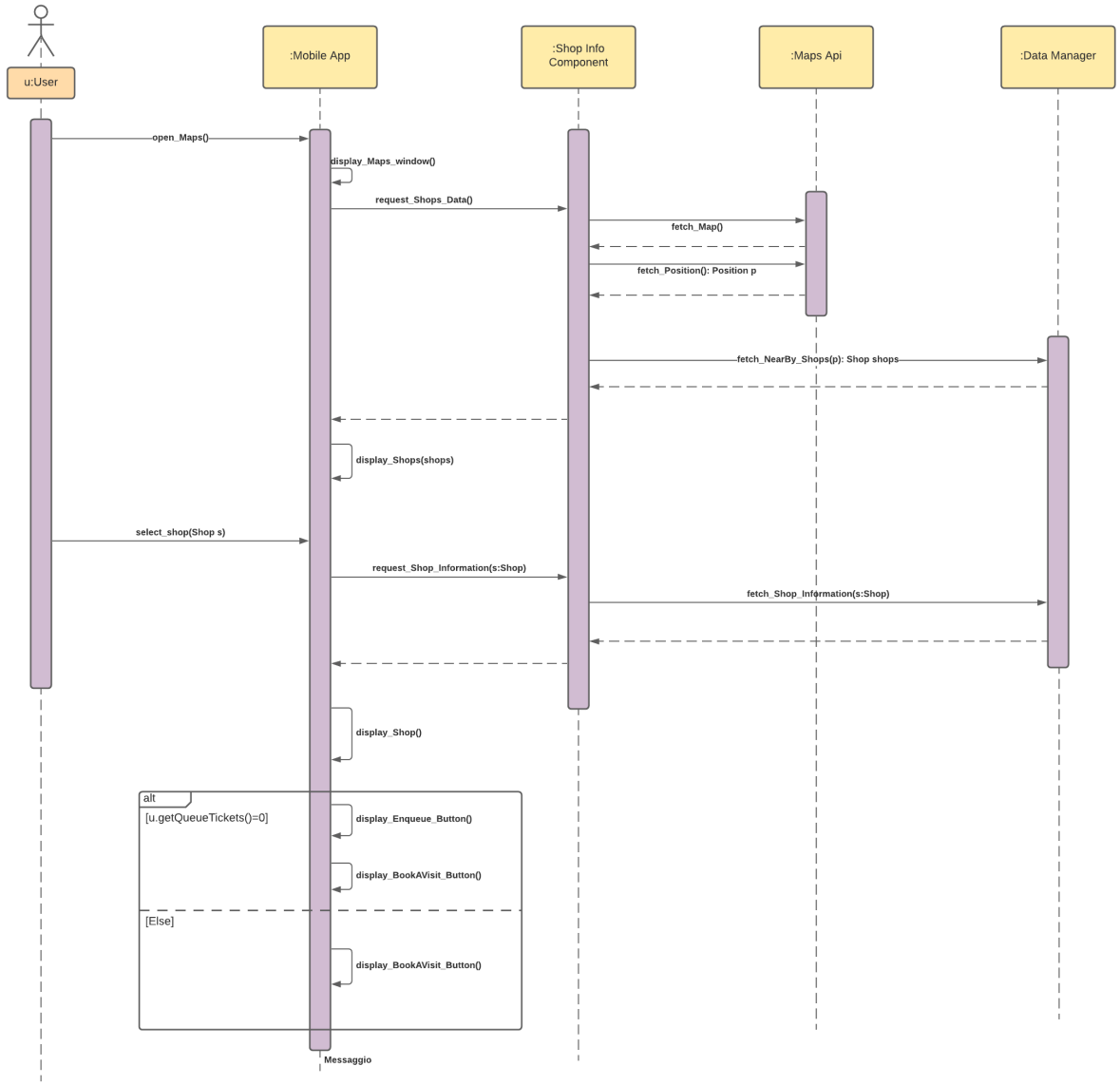


Figure 12: 5.User searches a Shop

2.6 Component interfaces

In this section we illustrate the main methods belonging to the interfaces of the components.

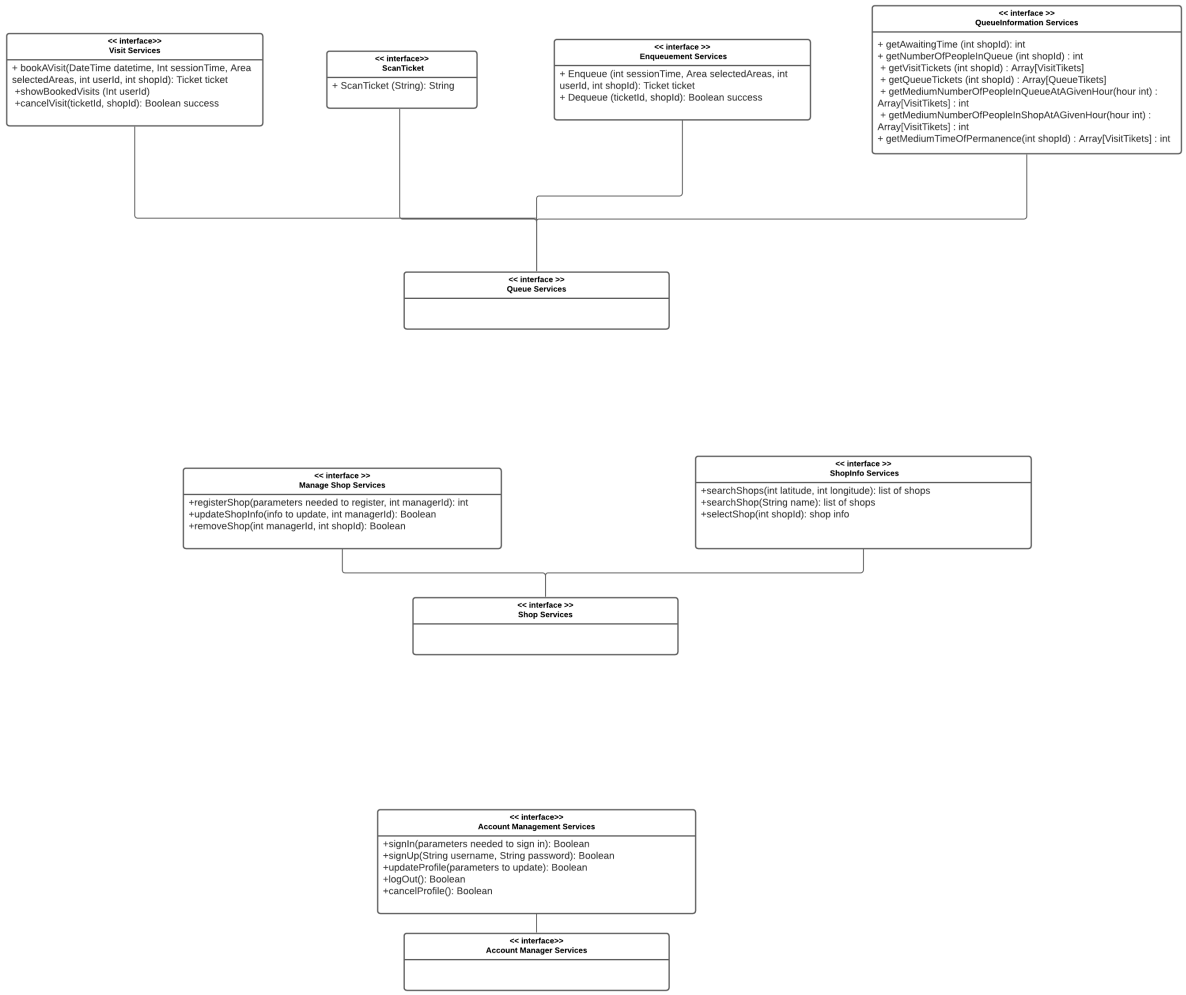


Figure 13: Interfaces Diagram

The **Queue Service Interface** gathers all of the functionalities needed to interact with the queue of a shop and the tickets that compose it.
Indeed, it offers the users the functions needed to manage enquements, visits, to retrieve information about the queue status, about their tickets and to inspect analytical data.

The **Shop Service Interface** gathers all of the features regarding the shops. Thus, it is the proxy through which a manager can register their shops on our system and to update them later on. It also offers methods that allow to retrieve all of the desired information about a shop, such as its position or the items that it sells.

The **Account Manager Interface** as clearly as its name may suggest- may I add, once again- this interface offers to the user all of the features related to their accounts. Examples of methods offered by this interface are "Sign In", "Log in" and "Update Profile".

2.7 Selected architectural styles and patterns

In the following, we have individuated a series of desing and structural pattern which may be selected in a future implementation of CLup.

- **Selected Design Patterns**
  - **State Pattern**: allows an object to behave different according to the circumnstances. It is used to implement the status of the tickets.
  - **Facade Pattern**: an objects which, through an easier interface, allows the access to several subsystems offering more complex and variagate interfaces.
  - **Adapter Pattern**: allows interoperability within different interfaces. This will be used for the interaction of the Qr-code application implemented by us with the underlaying system, since the latter isnt known *a priori* and can differen accordingly with the preferences of the managers.
  - **Factory Pattern** In class-based programming, the factory method pattern is a creational pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created. This can be in the optic of extending the range of tickets to be offered to the clients in the future.
  - **Decorate Pattern** In class-based programming, allows to add functionalities to an exisiting object at runtime. In a optic of growth of the application, it can be a wise choice to add new functionalities to the tickets, such as readapt them as coupons or as fidelty cards.
  - **Chain of Responsability Pattern** In this design pattern a request from a client can be handled by a number of handler or receiver objects. So basically, the client request is passed down each handler in the chain until a handler that can process the request is found. Servlet filters are a classic example of the Chain of Responsibility pattern. Spring has a class called HandlerInterceptorAdapter which is also an example of the Chain of Responsibility pattern. It has a method called preHandle, which follows the chain of responsibility pattern. Both servlet filters as well as Spring interceptors can be used to house code that is common for all requests like logging or authentication and that is separate from the business logic. We chose this to boost security aspects of CLup.
  - **Observer Pattern** is a pattern according to which an object, named the subject, mantains a lists of its dependents, called observers and notifies them automatically at any state change. It will be used for implement the push notifications and along with the MVC pattern explained right after this.
- **Selected Structural Patterns**
  - **Model View Controller**: is a very common pattern which separate the presentation logic of the application from the business logic. It is a very popular and effective way to decouple the different parts of the system.

3 Queue algorithm

3.1 General overview

The *management of shop entry turns* is a fundamental objective of the system in order to have it functioning properly, because most of the actions a user performs is related in some way to the stores queues.

Before we start discussing the queue algorithm, we need to define what we mean by the term "*queue*" in this section: a **queue** is a data structure representing a time line that contains all the necessary information regarding the entry times of all those customers who have queued or booked a shopping session.

Queues are objects that change over time based on unpredictable events and uncertain data. Our aim is to create queues and estimates that are as reliable as possible, to have solution for edge cases, and to be efficient not wasting the store and client's time.

3.1.1 Input

To create a queue the only inputs we need are listed in the following table [2].

Input name	Description
Store capacity	Parameter that indicates the maximum amount of people that the store accepts at the same time inside it in a instant of time.
People inside the shop	A list of all the tickets that are currently in the store.
Queued tickets	A list of all the tickets of those that have queued in the shop, and haven't entered yet.
Visit tickets	A list of all the tickets of those that have booked a visit in the shop, and haven't entered yet.
Search radius for visits	Parameter that indicates the amount of time within which to hold a free entrance to the shop in view of a person who has booked a visit.

Table 2: Queue algorithm input

For the algorithm to work properly, we must make sure that the data provided are coherent and correct, the algorithm does not perform any type of checks or validation.

3.1.2 Expected behavior

To understand the algorithm it is necessary to explain what are the most important rules on which the queues are based:

- When possible, the algorithm will use a *FIFO policy*.
- Each store has a *maximum capacity* of people that it can accommodate inside at the same time.

- *For each person who leaves the store, one can enter it*. This is the most important principle for achieving a constant and optimized flow of people within a store. To fully understand the logic behind this algorithm, it is important to highlight the fact that there must always be a **one-to-one correspondence** between a ticket leaving the store and a ticket entering the store.
  More formally we can say that there are two options available, if at time t a ticket leaves the store, a *one-to-one correspondence* will be created with:
  1. a visit ticket, if there is one whose entry is scheduled within the time $t+range$ (with "$range$" defined in the input table [2]);
  2. a queue ticket, that will be scheduled at time $t + 1$;
- Customers who book a visit have *priority* over customers who queue in the shop. However, giving maximum priority to visit tickets can result in the creation of inefficient queues, so the key to managing this aspect is to find the right balance between priority and efficiency, with a correct value for the input parameter *Search radius for visits*.
- The algorithm need to make sure that every person with a queue ticket is not scheduled ahead of the *time* it takes *to react the store*.

With these rules the algorithm will cover the main functionalities, but more features can easily be added, an example can be the possibility to set a maximum amount of people that can enter a slot in a single time slot.

3.1.3 Output

Output name	Description
Queue	A timeline divided into discrete time slots, each containing information regarding the tickets that are expected to enter the store in that time slot, and the tickets that are expected to leave the store in that time slot.
Estimates	General queue statistics, such as the estimated queue length, will be provided so that they are easy to access.

Table 3: Queue algorithm output

3.2 Integration of the algorithm with the system

In our system it is quite common for a process to use the data related to a store queue, for this reason the result of the algorithm will be saved in the database in order to make it available to any process that needs it. Every time the queue must be modified, the algorithm will build a new queue from scratch and override the old one in the database. This decision is in line with the application workflow, since most events will only need to read the queue data, and not to modify them, and therefore we expect to not overload the servers.

To have a functioning system, however, the queue must be always updated. To accomplish this, we have designed a life cycle for the algorithm based on the reactions to all those events that can modify the queue:

1. An event that could change the queue occurs, and the relative data in the database are updated.
2. The currently existing queue in the database is discarded.
3. The algorithm runs with the new updated data and produce a new queue.
4. The new queue is saved in the database, replacing the old one.

The following is a list of all the events that may cause a change in the queue:

- someone lines up;
- someone books a visit;
- someone scans a QR-code to enter or exit a store;
- someone cancels a visit or an enqueuement;

It is important to point out that there is a particular event, which, however, was not mentioned in the list above: *someone doesn't show up*. Due to the way the algorithm life cycle has been structured, this event resolves itself automatically. In fact, when someone does not show up at the expected time, the information can be ignored: when the next event happens, a new queue will be built and the person who did not show up previously will not be included among the inputs of the algorithm, and consequently will no longer be part of the queue.

3.3 The algorithm

In this section we want to show how the algorithm works describing it step-by-step in natural language and to show a possible implementation in Java code.

3.3.1 Natural language

The goal of this section is to explain the logical steps of the queue algorithm while ignoring the technical details, so that we can focus on the overall structure.
Following there are explained all the steps needed to obtain a queue that meets the requirements of the system:

1. Create an empty time line, made up of discrete time slots.
2. If the shop isn't full, the amount of people that can enter the shop in in the initial time is set to the number of available entrances.
3. Loops through the list of all the people inside the store, and for each one of them set the estimated exit time in the timeline.
4. For each visit ticket, set its entrance in the time slot in which the ticket is scheduled for, and set its exit in the time slot in which its exit is estimated.
5. Now we need to find a solution to create a *one-to-one correspondence* for entering tickets with exiting tickets.
   Loops through all slots of the timeline and for each person who exits:

- search within the established "range" (see algorithm input table [2]) if there is a scheduled visit ticket whose one-to-one correspondence hasn't already been solved.
  If a visit ticket is found, then we consider the ticket solved, and we set it in one-to-one correspondence with the person leaving the shop.
- Else, if it is possible, we mark the person leaving the store in one-to-one correspondence with the first queue ticket that manages to show up in time, using the FIFO rule to decide which to solve.

An important aspect of this algorithm is that more features can be easily added without changing the main logic steps.

3.3.2 Java code

This section shows a possible implementation of the algorithm in Java.

The following implementation is not to be considered a final nor complete version, its only meant to be a guide for future developers to better understand the logic. Furthermore, the code is structured to be easy to read, excessively commented, inefficient, and does not respect good coding conventions.

The following code is divided into three Java classes: *Queue, Ticket, TimeSlot*.

The TimeSlot and Ticket classes do not contain any algorithm logic, they are purely used to contain data in an orderly and sensible way. The **TimeSlot** class is useful for building the timeline, while the **Ticket** class is used to represent people queued, or who have booked a visit, or who are already inside the store.

The output of the algorithm is contained within the **Queue** class, which also contains all the logic within the Queue.buildQueue () method.

TimeSlot.java

```
1  public class TimeSlot{
2
3      private Integer id;
4
5      //queue tickets scheduled to enter in this time slot
6      private ArrayList<Ticket> queueTickets = new ArrayList<Ticket>();
7
8      //visit tickets scheduled to enter in this time slot
9      private ArrayList<Ticket> visitTickets = new ArrayList<Ticket>();
10
11     //tickets that are expected to exit the shop during this time slot
12     private ArrayList<Ticket> expectedExitingTickets = new ArrayList<Ticket>();
13
14     public TimeSlot(Integer id){
15         this.id = id;
16     }
17
18     //getters and setters...
19 }
```


Ticket.java

For ease of exposure and to reduce the amount of code within this document, this class is used to represent both queue tickets and visit tickets, although they should have different properties and methods. A more sensible hierarchy is expected in the actual application.

```
1  public class Ticket{
2
3      //expected duration of the permanence, expressed in quantity of TimeSlots
4      private Integer expectedDuration;
5
6      //indicates the time slot for which the ticket is booked (only for visit tickets)
7      private Integer scheduledTimeSlot;
8
9      //null if the ticket has never entered the store
10     //a negative value means that the ticket has entered the store
11     //a positive value means that the ticket hasn't entered the store
12     private Integer enteringTimeSlot;
13
14     //all types of tickets
15     enum TicketType {
16         VISIT,
17         QUEUE
18     }
19     private TicketType ticketType;
20
21     //ONE-TO-ONE correspondence:
22     //in order to let a ticket enter a shop, there must be an exiting ticket, this is a
           reference to the corresponding previous exiting ticket
23     private Ticket matchingPreviousTicket;
24     //in order to let a ticket enter a shop, there must be an exiting ticket, this is a
           reference to the corresponding following entering ticket
25     private Ticket matchingFollowingTicket;
26
27     //time needed to reach the shop expressed in quantity of TimeSlots, only for queue
           tickets
28     private Integer timeToReachTheShop;
29
30     //constructor for tickets (queue tickets or visit tickets) that have entered the
           store
31     public Ticket(Integer enteringTime, Integer expectedDuration){
32         this.expectedDuration = expectedDuration;
33         this.enteringTime = enteringTime;
34         this.ticketType = null; //not relevant
35         this.scheduledTimeSlot = null;
36         this.matchingPreviousTicket = null;
37         this.timeToReachTheShop = null;
38     }
39
40     //constructor for queue tickets that haven't already entered the store
41     public Ticket(Integer expectedDuration, Integer timeToReachTheShop){
42         this.enteringTimeSlot = null;
43         this.expectedDuration = expectedDuration;
44         this.ticketType = QUEUE;
45         this.scheduledTimeSlot = null;
46         this.matchingPreviousTicket = null;
47         this.timeToReachTheShop = timeToReachTheShop;
48     }
49
50     //constructor for visit tickets that haven't already entered the store
51     public Ticket(Integer expectedDuration, Integer scheduledTimeSlot){
52         this.enteringTimeSlot = null;
53         this.expectedDuration = expectedDuration;
54         this.ticketType = VISIT;
55         this.scheduledTimeSlot = scheduledTimeSlot;
56         this.matchingPreviousTicket = null;
57         this.timeToReachTheShop = null;
```



```
58      }
59
60      //constructor for "fake" (placeholders) tickets, all the properties are set to null
61      public Ticket(){
62          this.expecteDuration = null;
63          this.enteringTimeSlot = null;
64          this.ticketType = null;
65          this.scheduledTimeSlot = null;
66          this.matchingPreviousTicket = null;
67          this.timeToReachTheShop = null;
68      }
69
70      //getters and setters
71  }
```

Queue.java

This class takes care of the logic of the algorithm thanks to the .buildQueue () method, of storing the output, and of providing methods that allow easy access to the most important estimates.

```
1   public class Queue{
2
3       //output of the algorithm
4       private ArrayList<TimeSlot> timeLine = new ArrayList<TimeSlot>();
5
6       //quantity of tickets that can be inside the shop at the same time
7       private Integer shopCapacity;
8
9       //In this range of time slots the algorithm will try to find a one-to-one
            correspondence with a visit ticket and an exiting ticket
10      private Integer visitRange;
11
12      public Queue(Integer shopCapacity, Integer visitRange, Arraylist<Ticket>
            InShopTickets,
13                  ArrayList<Ticket> visitTickets, ArrayList<Ticket> queueTickets){
14          this.shopCapacity = shopCapacity;
15          this.visitRange = visitRange;
16
17          this.buildQueue(InShopTickets, visitTickets, queueTickets);
18      }
19
20      //auxiliary methods
21      private Integer existTimeSlot(Integer id){
22          //if it exists, returns the position of the time slot in the ArrayList of the
                time line, returns -1 if the time slot doesn't exist.
23      }
24      private TimeSlot getTimeSlot(Integer id){
25          //returns the time slot with the corresponding id from the timeLine;
26          //if the time slot doesn't exist, create one, add it to the timeLine, and returns
                it.
27      }
28
29      //QUEUE ALGORITHM
30      private void buildQueue(Arraylist<Ticket> InShopTickets, ArrayList<Ticket>
            visitTickets, ArrayList<Ticket> queueTickets){
31
32          //represents the beginning of the time line
33          TimeSlot firstTimeSlot = this.getTimeSlot(0);
34
35          //if the shop is not full we place an (placehoder) exiting ticket in the first
                time slot for each empty entry
36          Integer freeShopEntries = this.shopCapacity - InShopTickets.size();
37          ArrayList<Ticket> fakeExitingTickets = new ArrayList<Ticket>();
38          for (int i = 0; i < freeShopEntries; i++){
39              fakeExitingTickets.add(new Ticket());
40          }
41          firstTimeSlot.addExpectedExitingTickets(fakeExitingTickets);
```



```
42
43          //for each ticket inside the shop, calculate the expected exit time and mark the
                corresponding time slot
44          for(Ticket inShopTicket: InShopTickets){
45              Integer expectedExitTimeSlotID = inShopTicket.getEnteringTimeSlot +
                    inShopTicket.getExpectedDuration;
46              this.getTimeSlot(expectedExitTimeSlotID).addExpectedExitingTicket(
                    inShopTicket);
47          }
48
49          //for each visit ticket
50          for(Ticket visitTicket: visitTickets){
51              //add the visit ticket entry time in the corresponding scheduled time slot
52              this.getTimeSlot(visitTicket.getScheduledTimeSlot()).addVisitTicket(
                    visitTicket);
53
54              //calculate and add the visit ticket expected exiting time in the
                    corresponding scheduled time slot
55              this.getTimeSlot(visitTicket.getScheduledTimeSlot() + visitTicket.
                    getExpectedDuration()).addExpectedExitingTicket(visitTicket);
56          }
57
58          //loop throught all the time slots in the time line, and,
59          //reserve each exiting ticket for a visit ticket in range
60          //if there is any, otherwise, if possible, insert an entering queue ticket
61          for(int k = 0; k< timeSlot.size(); k++){
62
63              TimeSlot timeSlot = this.timeLine.get(k);
64
65              //all the ticket exiting the shop during this time slot
66              ArrayList<Ticket> expectedExitingtickets = timeSlot.getExpectedExitingTicket
                    ();
67
68              //for each expected exiting ticket check if there is in range a slot with a
                    visit ticket scheduled that hasn't already been handled
69              outerLoop:
70              for (Ticket expectedExitingTicket: expectedExitingTickets){
71
72                  //loops through the next slots till the visitRange, to find a visit
                        ticket to pair with the exiting ticket
73                  for(int j = 1; j <= this.visitRange; j++;){
74                      if (this.existTimeSlot(timeSlot.getId+j)){
75                          //for each visit ticket in the time slot
76                          for(Ticket visitTicket: this.getTimeSlot(timeSlot.getId+j).
                                getVisitTickets()){
77                              //if the visit ticket has not been handled before, meaning
                                    that there is not a matching previous ticket
78                              if(visitTicket.getMatchingPreviousTicket() == null){
79                                  //pair the exiting ticket with the visit ticket
80                                  expectedExitingTicket.setMatchingFollowingTicket(
                                        visitTicket);
81                                  visitTicket.setMatchingPreviousTicket(
                                        expectedExiticgTicket);
82                                  //jump to the next iteration of the outer loop, since we
                                        are done analyzing the visit ticket (refactor in
                                        multiple methods is recommended, here is like this
                                        for readability)
83                                  continue outerLoop;
84                              }
85                          }
86                      }
87                  }
88
89                  //if the following code is running it means that there was no visit
                        ticket in visitRange, so we can handle a queue ticket
90
91                  //loops through all the queue tickets to find one to place in the
                        following time slot and to pair with the exiting ticket we are
                        analyzing,
92                  //also the choosen queue ticket to place must be able to reach the shop
                        before his turn arrives
93                  Ticket choosenQueueTicket = null;
```



```
94                  for(Ticket queueTicket: queueTickets){
95                      if(queueTicket.getTimeToReachTheShop() <= (timeSlot.getId()+1)){
96                          choosenQueueTicket = queueTicket;
97                          break;
98                      }
99                  }
100
101                 if(choosenQueueTicket != null){
102                     //update the time line with the entering time of the queue ticket
103                     this.getTimeSlot(timeSlot.getId+1).addQueueTicket(choosenQueueTicket)
                            ;
104
105                     //update the time line with the expected exiting time of the queue
                            ticket
106                     this.getTimeSlot(timeSlot.getId + 1 + choosenQueueTicket.
                            getExpectedDuration())
107
108                     //pair the exiting ticket with the queue ticket
109                     choosenQueueTicket.setMatchingPreviousTicket(expectedExitingTicket);
110                     expectedExitingTicket.setMatchingFollowingTicket(choosenQueueTicket);
111
112                     //delete the queue ticket from the list of queue tickets, since it
                            has been correctly placed in the time line
113                     queueTickets.remove(choosenQueueTicket);
114                 }
115
116             }
117
118         }
119     }
120
121     //getters and setters...
122
123     //methods to calculate estimates and usefull information about the time line
124 }
```


4 User interface design

4.1 User experience

4.1.1 Key principles

Meet the user needs

The CLup system has relatively few services to offer to the customer, and it is therefore essential that the interaction with the views make them stand out. To offer an easy user experience, we want the user to always be able to understand what to do to achieve their goal.

To obtain a result of this type it is essential to identify a hierarchy that is easy to understand.

The pages are divided into three levels:

- Login and account management Pages;
- Home page;
- Sub pages: all those pages whose purpose is to guide a user to accomplish a task;

The functions of the application, and consequently the paths that a user can take, must also be clearly organized in main flows and sub flows, highlighting and making the most important ones easy to access.

Consistency

To facilitate the user experience, consistency is another key factor to consider.

Examples of consistency that we will adopt will be:

- Colors: to highlight the application brand and make it recognizable and to guide the user.
- Interactive colors: to make the user understand that he may or may not interact with a certain element of a graphic interface.
- Position of elements: elements with similar goals must always be positioned in key points that characterize them.
- Terminology.
- Overall view looks.

Less is more

The "less is more" principle does not mean hiding elements in views, but simply trying not to overload users with information. In general, try to keep the interfaces simple.

This principle is to be particularly kept in mind for pages like the "Home", which has a lot to communicate: it is a gateway for the user to all the main functions, it must be a source of information important events of interest to the user, it must entertain all those users who have no real purpose in mind.

We also use this principle for another very important aspect: the amount of interactions that the user must have with the application. Any page of the application must be easily accessible in a few clicks. The only time we require extra interaction from users is after making a relevant decision: *confirm before commit*.

Accessibility and demography

Since the public with which CLup interacts is vast and wide, we try to make stylistic choices as neutral as possible. We do not want to exclude or show particular interest in any demographic group.

In order not to exclude anyone, it is also important to keep in mind possible users with visual impairments.

4.1.2 Flow diagram

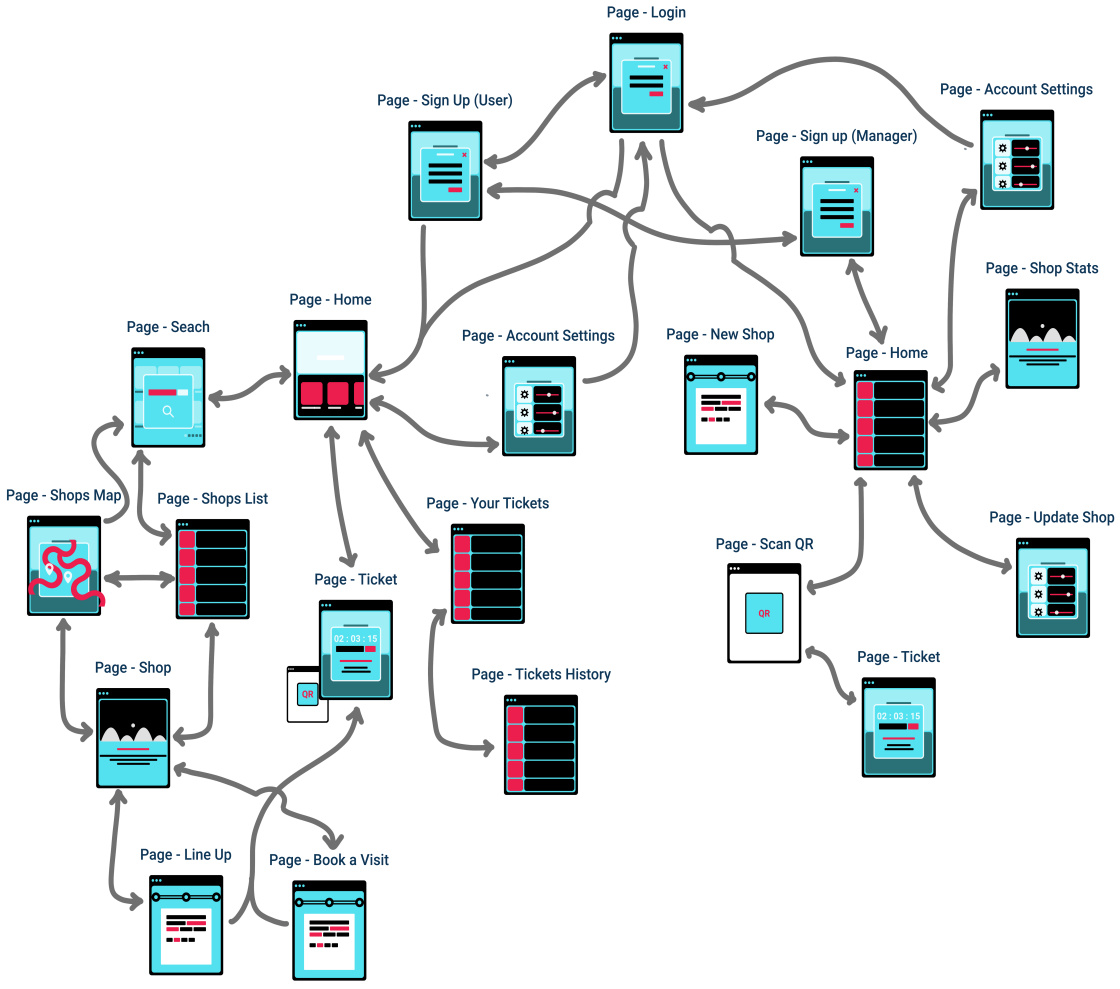


Figure 14: Interfaces Diagram

In this section we want to show the flow diagram that represents all the pages in which the application is divided, connected with arrows that indicate how they are reached.

The home page is always the Login Page if the user is not authenticated, otherwise it is the home

page.

The image [24] represents the flow diagram of the mobile application, it should be read taking into consideration that all the pages on the left are for users, while those on the right are for managers. For the web version, oriented towards PC users, you can use the same diagram, or at most merge some pages together since the space available to work with is greater. The version for the Totem, on the other hand, is a simplified version as many features are not needed.

4.1.3 Views

In this section we present some tables that serve to understand the main information of all the pages shown in the image [24].

What we want to focus on are the main and secondary functions of each page, as well as some extra informations about the actors that can have access to the pages and some useful notes.

What we leave out in these tables, however, are the aesthetic details and obvious functionalities, such as error reports, or the elements that allow you to go back to previous views, etc.

Also, it should be noted that each of these pages can be expanded into multiple pages if it can help the user experience.

Login Page	
Main Functions	Log in
Secondary content	-
Actors	Users and Managers
Notes	-

Table 4: Login Page

Sign Up Page	
Main Functions	Register
Secondary content	-
Actors	Users and Managers
Notes	Both users and managers have their own page, here they have been merged for simplicity of exposure. Also the page can be split into multiple pages.

Table 5: Sign Up Page

Account Settings Page	
Main Functions	Manage account setting
Secondary content	Log out
Actors	Users and Managers
Notes	Both users and managers have their own page, here they have been merged for simplicity of exposure. Also the page can be split into multiple pages.

Table 6: Account Settings Page

Home Page (Manager)	
Main Functions	Introduce the manager to all possible actions he can take
Secondary content	-
Actors	Manager
Notes	-

Table 7: Home Page (Manager)

Shop Status and Statistics Page	
Main Functions	Visualize the current status and the statistics of a shop
Secondary content	-
Actors	Manager
Notes	The page can be split into multiple pages.

Table 8: Shop Status and Statistics Page

New Shop Page	
Main Functions	Add a new shop to the system
Secondary content	-
Actors	Manager
Notes	The page can be split into multiple pages.

Table 9: New Shop Page

Update Shop Page	
Main Functions	Update shop informations
Secondary content	-
Actors	Managers
Notes	The page can be split into multiple pages.

Table 10: Update Shop Page

Scan QR-code Page	
Main Functions	Identify a ticket easily
Secondary content	-
Actors	Managers
Notes	-

Table 11: Scan QR-code Page

Ticket Page (Manager)	
Main Functions	Cancel a ticket from an owned shop queue
Secondary content	Visualize ticket informations
Actors	Managers
Notes	-

Table 12: Ticket Page (Manager)

New Shop Page	
Main Functions	Add a shop to the system
Secondary content	-
Actors	Managers
Notes	The page can be split into multiple pages.

Table 13: New Shop Page

Home Page (User)	
Main Functions	Introduce the user to all possible actions he can take
Secondary content	Display important events, entertain all those users who have no real purpose in mind.
Actors	Users
Notes	-

Table 14: Home Page (User)

Search Page	
Main Functions	Search bar
Secondary content	Use the device position as the search input
Actors	Users
Notes	-

Table 15: Search Page

Shops List Page	
Main Functions	Visualize shops in a list
Secondary content	Choose a shop, sort the list, filter the list, display basic information for each shop
Actors	Users
Notes	-

Table 16: Shops List Page

Shops Map Page	
Main Functions	Visualize shops in a map
Secondary content	Choose a shop, display basic information for each shop
Actors	Users
Notes	-

Table 17: Shops Map Page

Shop Page	
Main Functions	Visualize all the information of a shop
Secondary content	Start the line up and book a visit process
Actors	Users
Notes	The page can be split into multiple pages.

Table 18: Shop Page

Line Up Page	
Main Functions	Line up in a queue
Secondary content	-
Actors	Users
Notes	The page can be split into multiple pages.

Table 19: Line Up Page

Book a visit Page	
Main Functions	Book a visit
Secondary content	-
Actors	Users
Notes	The page can be split into multiple pages.

Table 20: Book a visit Page

Ticket Page (User)	
Main Functions	Visualize information about a ticket
Secondary content	Get the QR-code of a ticket, cancel a ticket
Actors	Users
Notes	-

Table 21: Ticket Page (User)

Your Tickets Page	
Main Functions	Visualize all active ticket
Secondary content	-
Actors	Users
Notes	-

Table 22: Your Tickets Page

Ticket History Page	
Main Functions	Visualize all past tickets
Secondary content	-
Actors	Users
Notes	-

Table 23: Ticket History Page

4.2 User interfaces

4.2.1 Colors

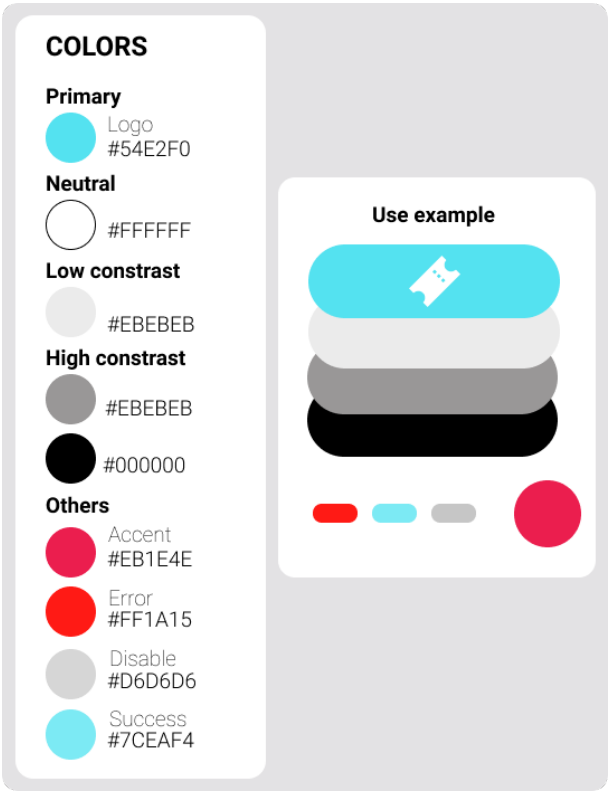


Figure 15: Colors palette

We present the chosen colors:

- Company color: #54E2F0. This color should be used throughout the application to make the CLup brand stand out and to make it recognized outside the system.
- Accent color: #EB1E4E. This color should be used throughout the application to communicate to the users what elements they can interact with.
- Error color: #FF1A15. This color should be used to highlight errors.
- Disable color: #D6D6D6. This color should be used to highlight disabled element.
- Success color: #7CEAF4. This color should be used to highlight a successful operation.

4.2.2 Mockups

In this section we show experimental mockups for some pages of the CLup mobile application.

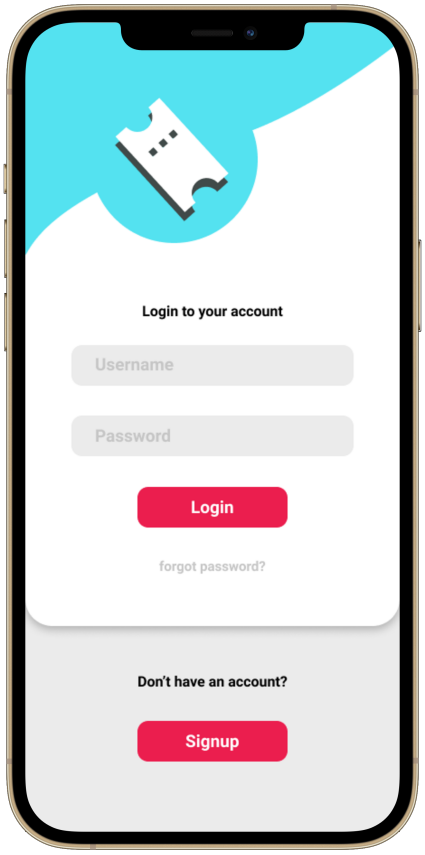


Figure 16: Login Page Mockup

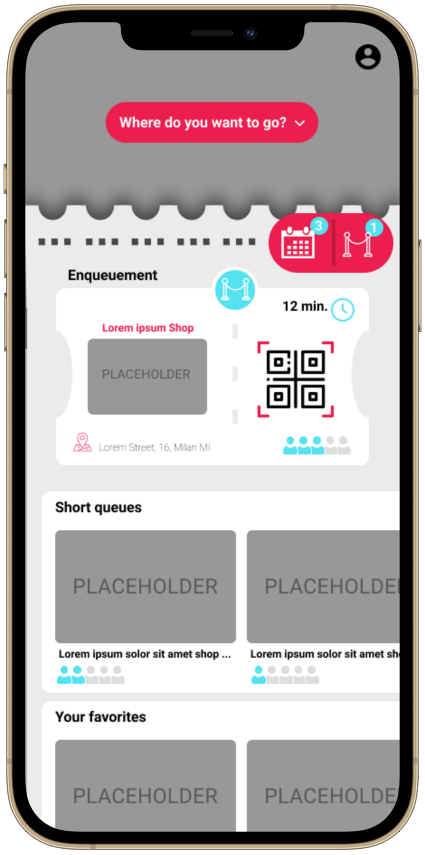


Figure 17: User Home Page Mockup

Figure 18: Search Page Mockup

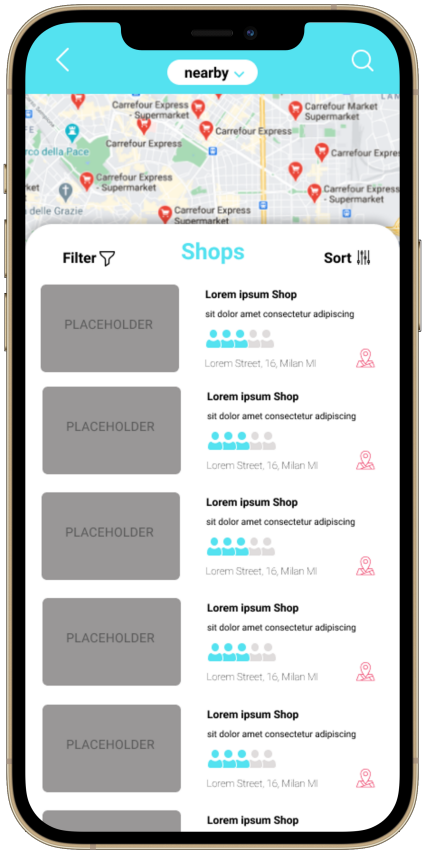


Figure 19: Shops Page 1 Mockup

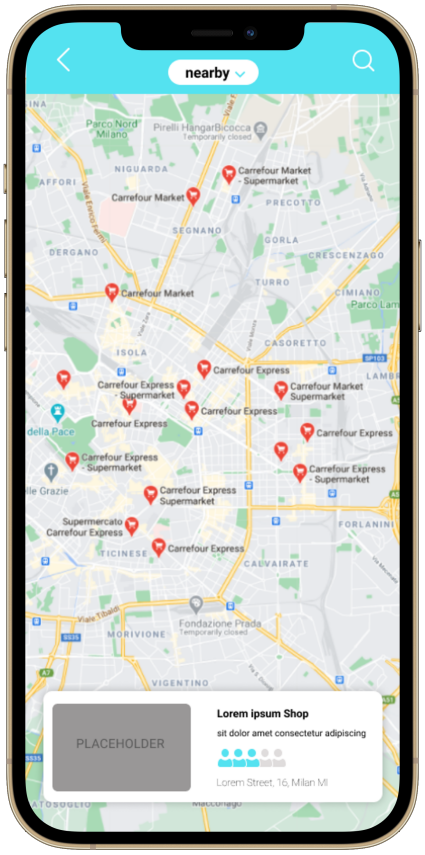


Figure 20: Shops Page 2 Mockup

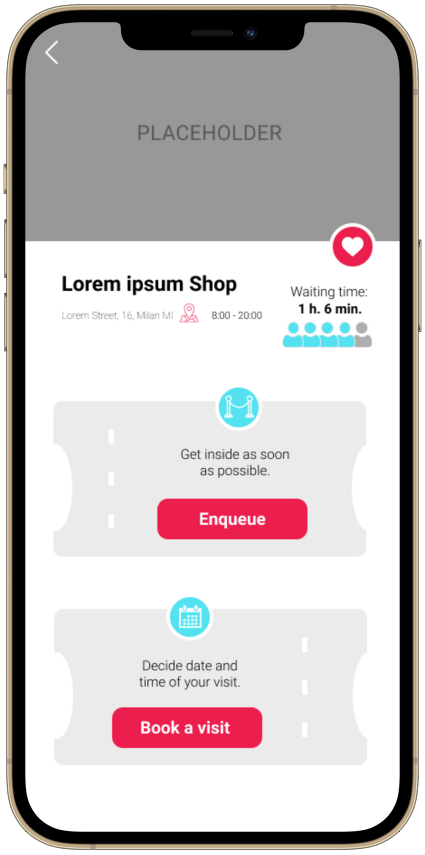


Figure 21: Shop Page Mockup

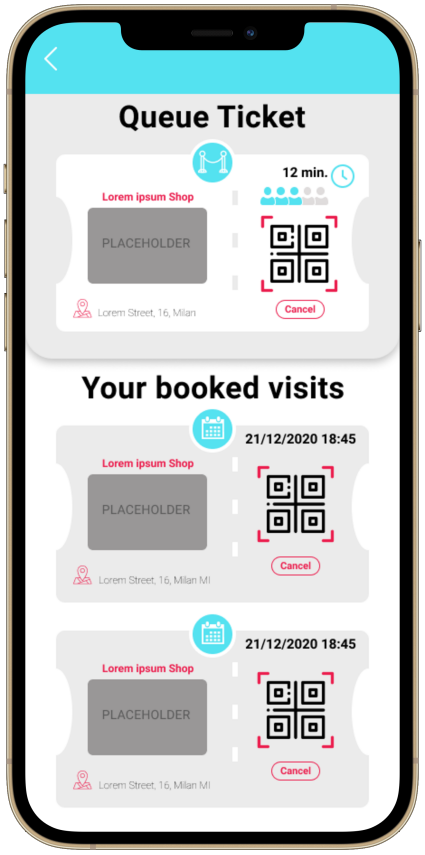


Figure 22: Your Ticket Page Mockup

Figure 23: QR code Page Mockup

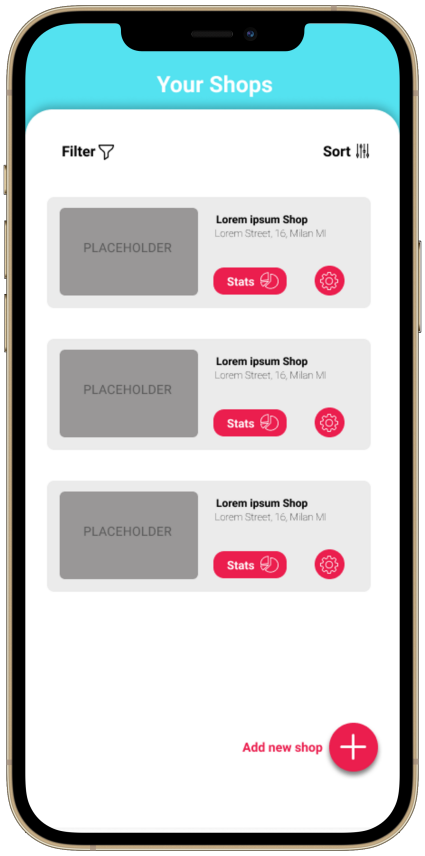


Figure 24: Manager Home Page Mockup

5 Requirements traceability

- **Account Management Service**
  - **Authentication and Authorization Engine Component**
    - R3, R4, R5, R6, R10, R12, R43, R44, R45, R48, R49, R51,
  - **Account Management Component**
    - R7, R46
- **Subsystem Queue Service**
  - **Analytics Component** and **Info Queue Component**
    - R28, R29, R30, R31, R32, R33, R34, R35, R60, R61, R64, R67, R68, R78, R79, R80, R81, R82, R84,
  - **LineUp Component**
    - R91, R92
  - **Visit Component**
    - R39, R40, R69, R74, R75, R78, R95, R96, R99, R100
  - **Push Notification Component**
    - R39, R40, R69, R74, R75, R78, R95, R96, R99, R100
  - **QR-code Scanner Component**
    - R88, R89, R90
  - **Ticket Generator Component**
    - R70, R71
- **Shop Service**
  - **Manage Shop Component**
    - R21, R25
    - **Shop Info Component**
      - R53, R55, R58

6 Implementation, integration and test plan

6.1 Entry Criteria

Components should be tested as soon as they are released. Though, some preliminary conditions have to be satisfied before the integration plan testing shall be put into pratice. Such preconditions are:

- all of the people involved in this process must come prepared to the meetings of the peer review. This means that they have previously and carefully read the RASD and this very same document.
- Some of the low level modules must be avaible in order to properly test some of the components of our system. Such low level modules consists in:
  - All of the DBMS must have been configured and the DBs - Shop Database, Queue Database and Account Database- possibly prefilled with fictitious data
  - For the integration test regarding the Shop Services Subsystem, and in particular, the Shop Info Component, the Maps API should be avaible and fully usable.
  - In order to test the part of the QR-code scanner application implemented by us, the underlaying application, the one that assures the correct unlock and lock of the turnstills, must be avaible and correctly functioning.
  - To proceed with the testing and integration of the Notificate User Component, which is part of the Queue Service Subsytem, the Push Api must be fully operative.
  - To test and integrate the Account Manager Services Subsystem, the SMS Gateway should be avaible and ready to use.
- In order to test a component and its interactions with other components, it must have reached a minimum level of satifistaction of its goals and functionalities. To be more precise, we will indicate the minium level of complection of each component in order for it to take part of the integration and testing plan:
  - 60 % of the Account Manager Component
  - 60 % of the Authorization and Authentication Engine Component
  - 80 % of the Ticket Generator Component
  - 90 % of the Ticket Scheduler Component
  - 70 % of the Visit Component
  - 70 % of the LineUp Component
  - 60 % of the Queue Info Component
  - 60 % of the Analytics Component
  - 80 % of the Shop Info Component
  - 70 % of the Manage Shop Component
  - 80 % of all of the different Data Manager Component present in all of the subsystem

  The different percentage of complection are due to the role of the component in the system. The most significant the features it offers are for the application, the higher the percentage is. Also the level of criticality of the algorithms implements by a component along with the level of its interconnection with other components have been considered for deciding the level of complection.

6.2 Elements To Be Integrated

The component that must be tested and integrated can be divideed in three main categories: -Front-end components: Qr-code Scanner Application, Web Application, Mobile Application -Back-end components: Queue Services Subsystem, Shop Services Subsystem, Account Management Subsystem -External Components: Qr-code underlaying application, SMS GateWay, DBMS, Push API, Maps API

There are two kind of integration to perform. The first integration is the integration between components that are part of the same category. In this category we can find the integration that have to be performed within the front-end components and the back-end components internally with respect to their subsystems. The second kind of integration concerns the integration externally to their subsystems.

Front-end components and external components are indipendent from each other, while back-end components must be integrated either with front-end components and external-components.

Due to this architecture, it is possible to partially test first each component of a category, integrate them with other components of their same category and finally integrate all of the system.

The main integration of the back-end components with the front-end components are:

- **Mobile Application, Queue Services**
- **Mobile Application, Shop Services**
- **Mobile Application, Account Manager Services**
- **Mobile Application, PUSH API**
- **Web Application, Queue Services**
- **Web Application, Shop Services**
- **Web Application, Account Manager Services**
- **QR-code scanner Application, Queue Services**

The main integration of the back-end components with the external components are:

- *Queue Services Subsystem*
  - **Queue Services, Queue DBMS**
  - **Queue Services, PUSH API**
  - **Queue Services, Qr-code scanner Application** (this is inted as external to the component....)
- *Shop Services Subsystem*
  - **Shop Services, Shop DBMS**
  - **Shop Services, MAPS API**
- *Shop Services Subsystem*
  - **Account Manager Services, Account DBMS**

6.3 Integration Testing Strategy

According to what previously stated and in agreement with the nature of the application we are going to develop, we decided to test our system following a bottom up approach in combination with critical modules.
Thus, we decided to implement in first instance the QueueScheduler Component, which is the core module of our application, and, indeed, the most critical and delicate one. After this component is fully functional and tested, it will be possible to proceed all of the other components that some one gravitates around it with a thread-based integration. In the thread-based integration, the external dependencies will be integrated before the others: the bottom-up approach is intended this way.
This way, it will be possible to parallelize the developement phase and, moreover, to have more of an immediate feedback also at UX level. Further more, using this approach, the effort spent in providing stubs and drivers will be minimal.

6.4 Sequence of Component Integration

In this section we will describe the order of integration of components and subsystems. As a notation, an arrow going from a A to B means that B needs A in order to be tested, so that A needs to have been previously implemented.

6.4.1 Software Integration sequence

Queue Services

- **External Dependencies** all of the components in this subsystem depends on the Data Managar, which, in turn, depends on the DBMS. Furthermoer, the *Notification User* component depends on the Push Api, while the *Qr-code scanner* component depends on the Qr-code scanner application.

Figure 25: Queue DBMS depencency

Figure 26: PUSH API depencency

Figure 27: Qr-code scanner application depencency

- **Internal progressive component integration** Here it is shown the progressive integration of the components internal to the subsystem. The order of integration has been chosen due to the dependencies intra components. the *Ticket Scheduler Component* and the *Ticket Generator Component* have been colored in red because a particular attention need to be paid while developing and testing them due to their critical role in the application.

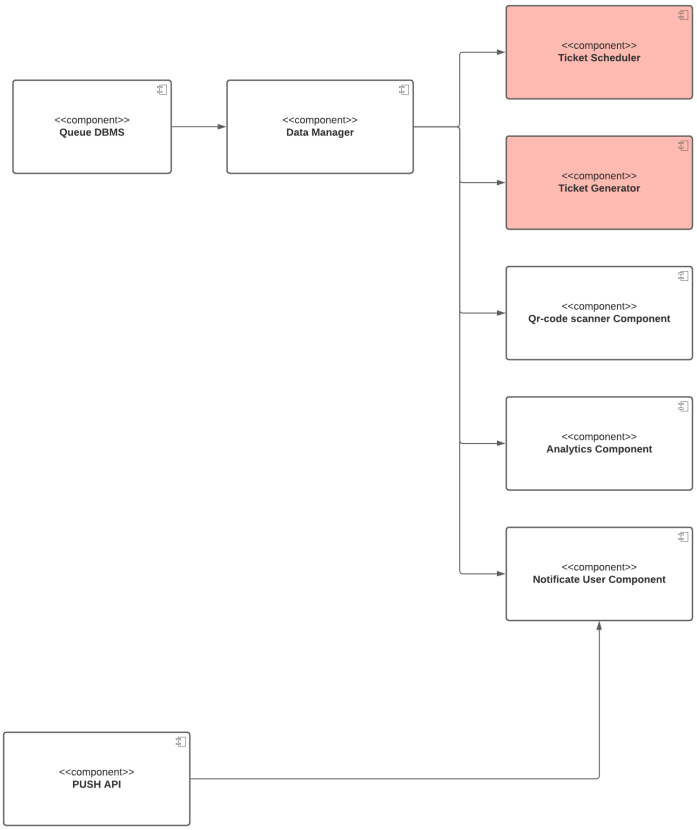


Figure 28: Progressive Integration 1

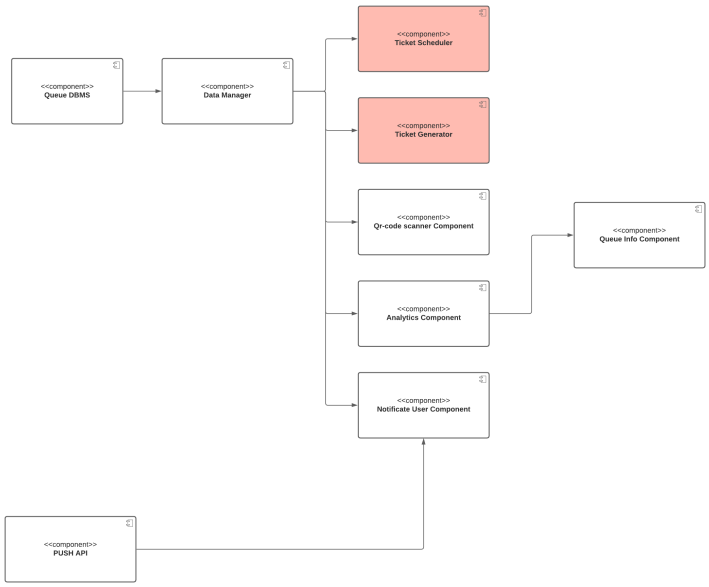


Figure 29: Progressive Integration 2

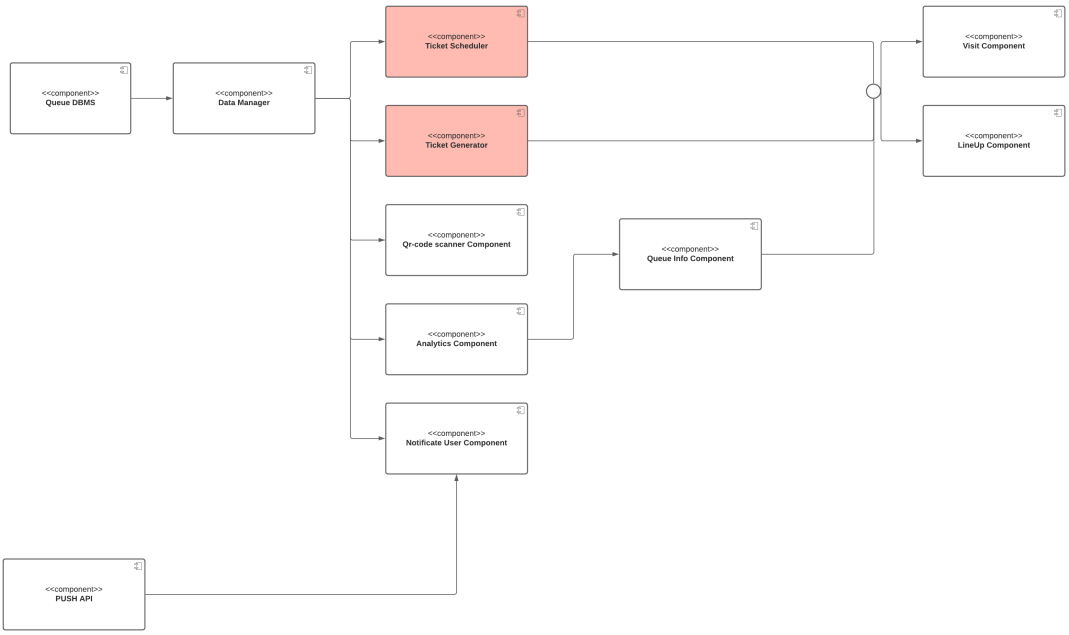


Figure 30: Progressive Integration 3

Shop Services

- **External Dependencies** all of the components in this subsystem depends on the Data Managar, which, in turn, depends on the DBMS. Furthermoer, the *Shop Info Component* depends on the Maps Api component.

Figure 31: Maps API depencency

Figure 32: Shop DBMS depencency

- **Internal progressive components integration** Here it is shown the progressive integration of the components internal to the subsystem.

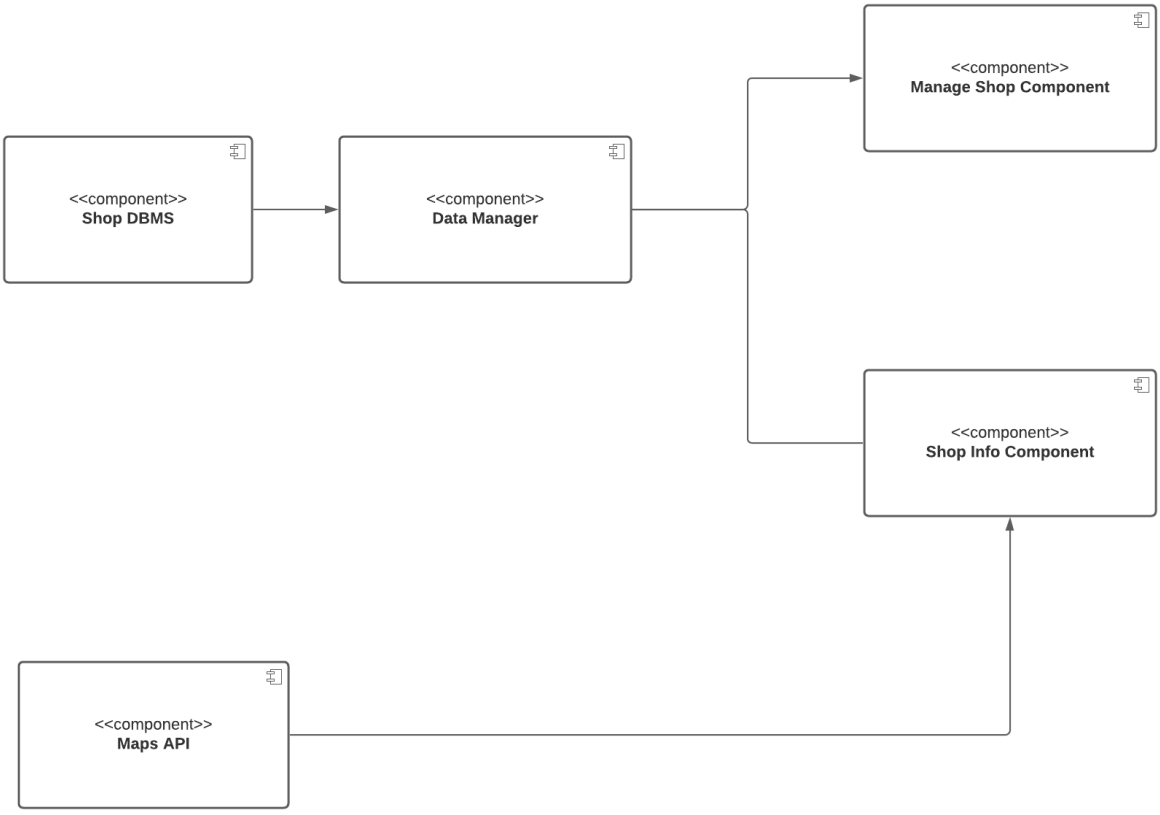


Figure 33: Shop Progressive 1

Account ManagerServices

- **External Dependencies** all of the components in this subsystem depends on the Data Managar, which, in turn, depends on the DBMS. Furthermoer, the *Account Manager Component* depends on the SMS Gateway component.

Figure 34: Account DBMS depencency

Figure 35: SMS Gateway depencency

Internal progressive components integration the following images shows the progressive integration of the components internal to the subsystem

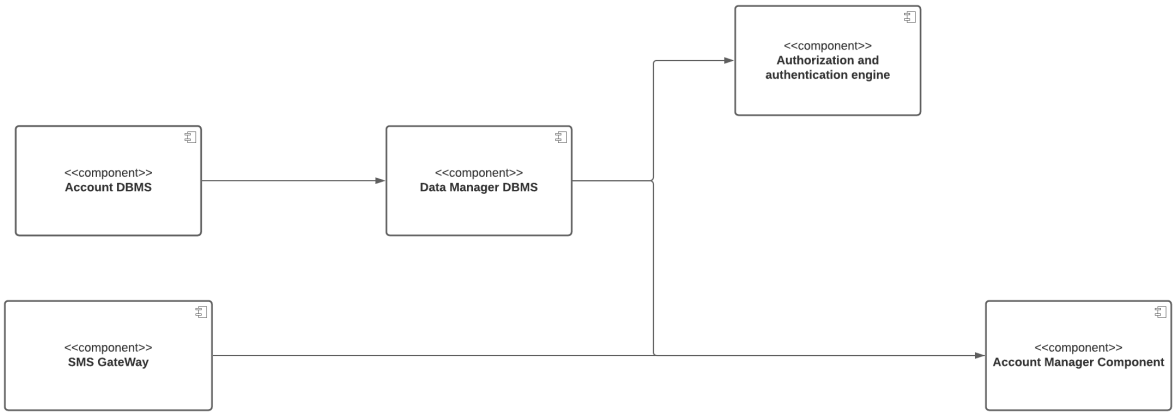


Figure 36: progressive sequence 1

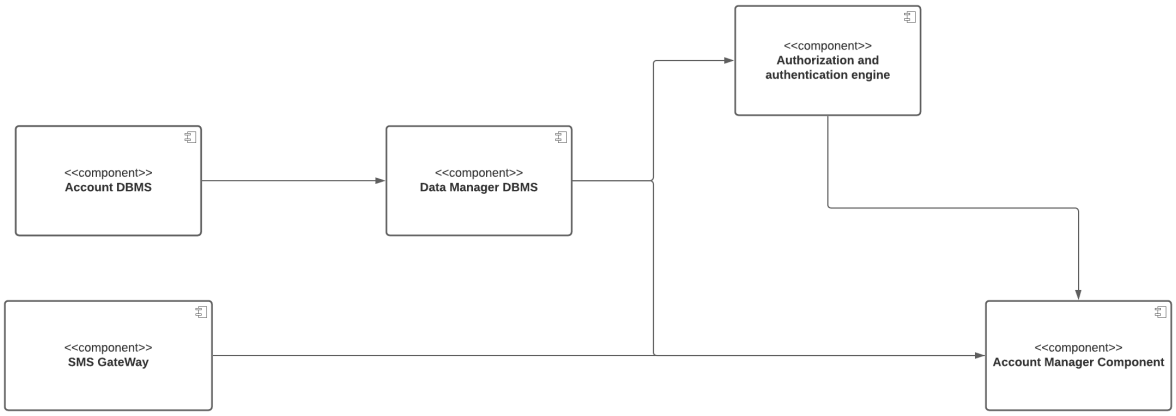


Figure 37: progressive sequence 2

6.4.2 Subsystem Integration Sequence

The following diagrams present the integration of the subsytems with the front-end components. In this diagram, it have been chosen, for the sake of simplicity, to represent the Front-end as a cross-platform subsystem.

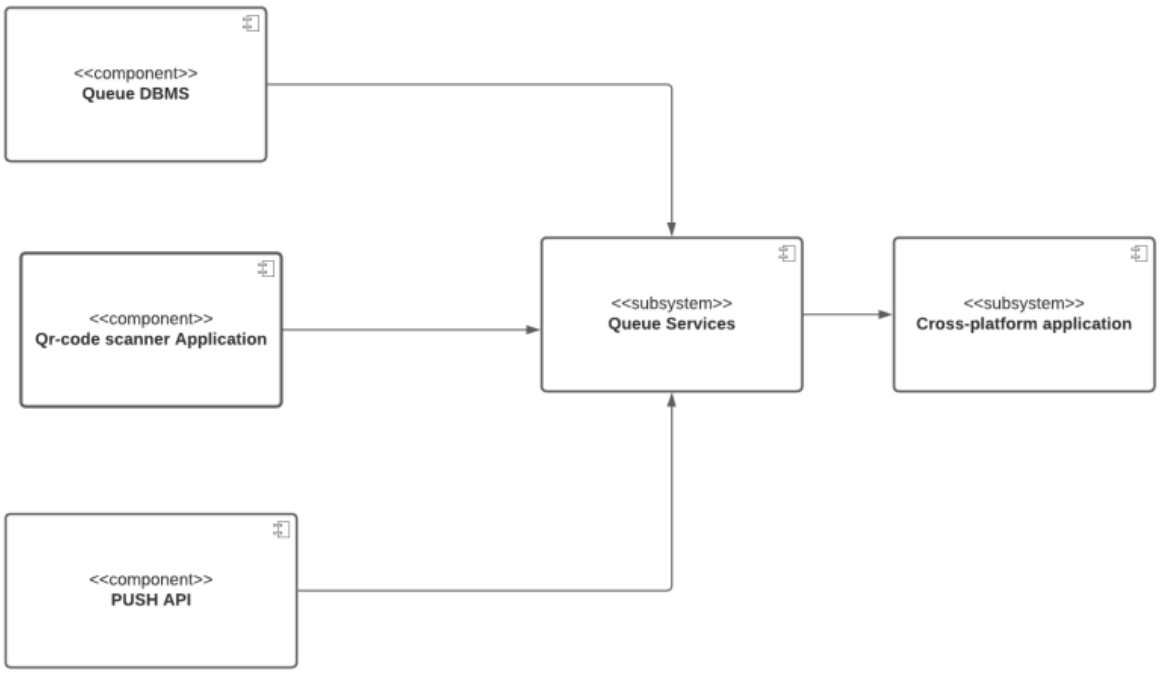


Figure 38: integration Queue Service Subsystem with front-end component

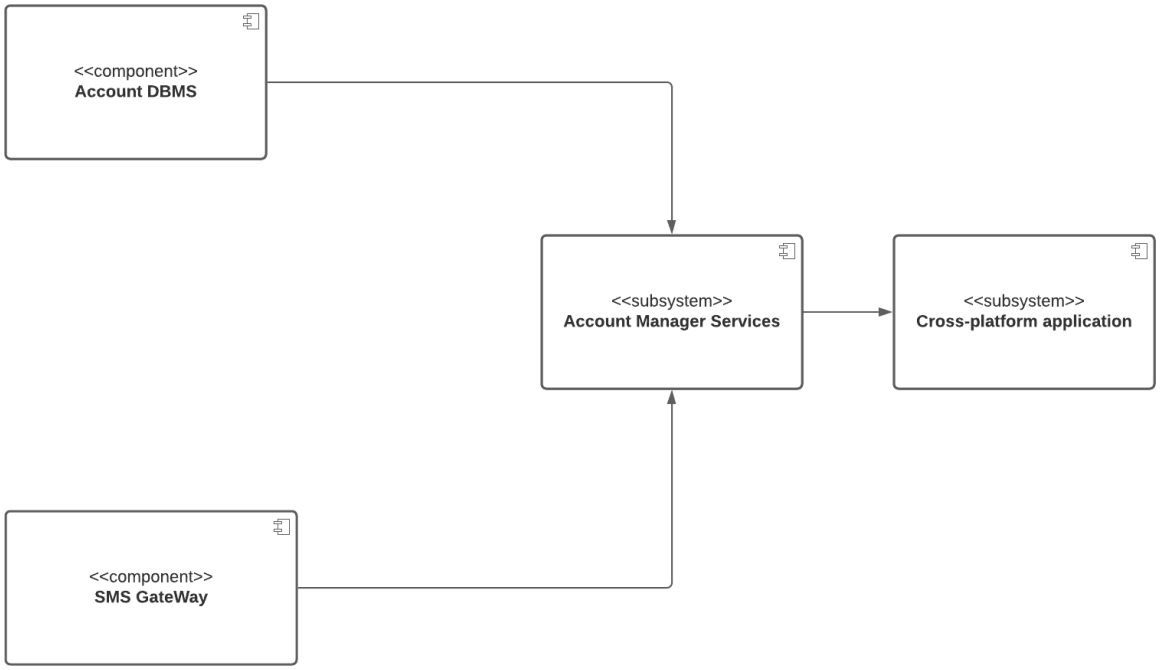


Figure 39: integration Account Manager Subsystem with front-end component

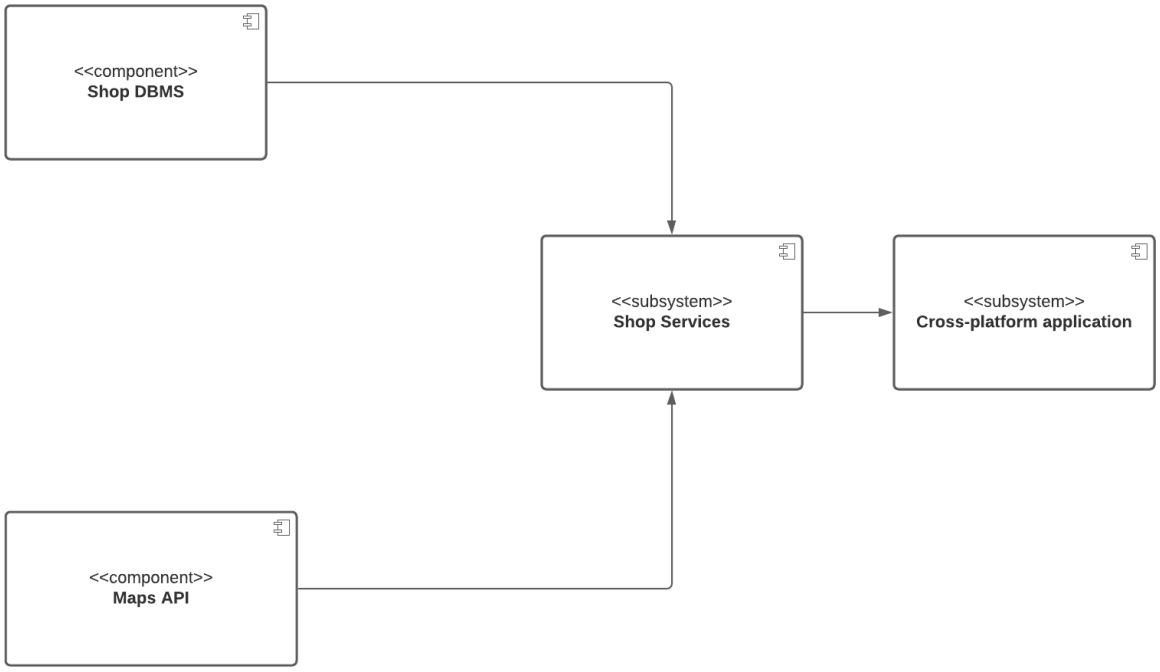


Figure 40: integration Shop Services Subsystem with front-end component

Finally, once all of the subsystems are fully integrated, we are ready to put together, ready for deployment, the Clup Application. Here we decided to split the front end subystem into its component, in order to have a complete representation of all of the components of the System.

application.png

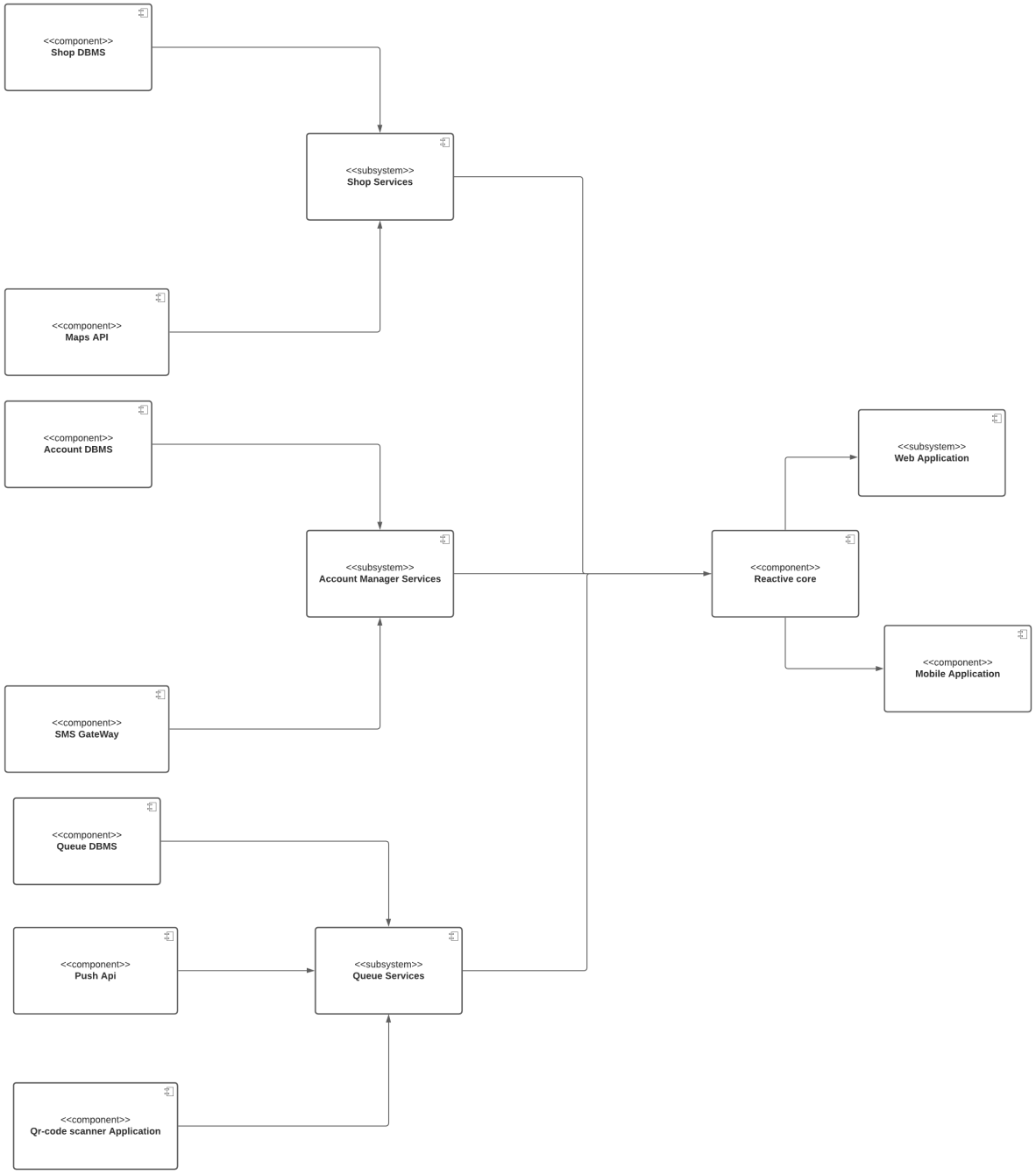


Figure 41: System integration

6.5 System Testing

The System testing will be conducted by indipendent teams in order for the functional and non functional requirements of our application to be verified. The type of tests we have selected in order test our applications are the following:

- **Performance Testing**: performance testing will be performed in order to indentify the bottlenecks of our system which may affect the usability of our application. Benchmarking will be used too:

this means that comparable data coming from similiar applications already present on the market will be retrieved and compared with our results.

- **Load Testing**: this kind of testing will help us identify the main problem relativly to the management of the avaible memory and to identify the components which needs upper limits constraints. This kind of testing will be performed gradually increasing the load on the system till a threshold is reached and also by loading the system with maximum load it can support for a long period of time.
- **Stress Testing**: this kind of testing is performed in order to be sure that the system is able recover quickly and correctly after a failure event. The way we are going to implements this kind of testing is by trying to break the system, for exemple by overwhelming it or by subtracting resources from it.

7 Effort spent

8 References

- Project goal, schedule and rules (Assignements AA. 2020/21)
- Requirements Analysis and Specifications Document v.2
- UML component and deployement diagram
  uml-diagrams.org
- Java client library for Google Maps API Web Services documentation
  github.com/googlemaps/google-maps-JavaGoogle Maps Distance Matrix API documentation

