



**POLITECNICO**  
MILANO 1863

**DD**

Design Document

**Authors:**

**David Leonardo Gomez 10787907**

**Martín Anselmo 10789796**

**Vinay Krishna Munnaluri 10780972**

**Document Version: 1.0**

**Date: 29 November 2020**

**Professor: Elisabetta Di Nitto**

# Contents

<b>1 Introduction</b>	<b>3</b>
1.1 Purpose	3
1.2 Scope	3
1.3 Definitions, Acronyms, Abbreviations	3
1.3.1 Definitions	3
1.3.2 Acronyms	4
1.3.3 Abbreviations	4
1.4 Revision history	4
1.5 Reference Documents	5
1.6 Document Structure	5
<b>2 Architectural Design</b>	<b>6</b>
2.1 Overview: High-level components and their interaction	6
2.1.1 High Level Components	7
2.2 Component view	9
2.2.1 Additional specifications	11
2.3 Deployment view	12
2.4 Runtime view	13
2.4.1 Login in	13
2.4.2 Request a ticket	13
2.4.3 Book a visit	14
2.4.4 Alternatives for the customer	15
2.4.5 Scan QR code	16
2.5 Component interfaces	16
2.6 Selected architectural styles and patterns	18
2.7 Algorithms	18
<b>3 User Interface Design</b>	<b>20</b>
3.1 Mobile Customer Wireframes	20
3.2 Shop Manager Wireframes	22
3.3 Physical Devices Wireframes	24
<b>4 Requirements Traceability</b>	<b>25</b>
<b>5 Implementation, Integration and Test Plan</b>	<b>26</b>
5.1 overview	26
5.2 Implementation	26
5.3 Integration Strategy	28
5.4 System Testing	31
<b>6 Effort Spent</b>	<b>32</b>
6.1 Hours of Work	32
<b>7 References</b>	<b>33</b>

# 1 Introduction

## 1.1 Purpose

This document will provide information about the technical implementation, architecture, design and plan of the software presented in the corresponding Requirement Analysis and Specification Document, CLup, a system designed to solve the queueing problem in stores during the COVID-19 pandemic.

The purpose of this document is to be a complementary document to the RASD, but more focused on the actual implementation of the system, as the intended readers are the programmers that are going to be developing it. For this, an implementation, integration and testing plan will be included.

## 1.2 Scope

CLup is a software designed to handle crowds and lines in stores during the COVID-19 pandemic. It consists in a digital queueing system so customers do not have to go and stay in line for long periods of time, putting their health in risk and wasting their time.

The system offers three main functionalities. *Lining up*, for requesting tickets to enter a store, *Booking a visit*, for planning ahead visits to the stores, and *alternatives for the customer*, where the system can offer the customer other shopping options so he can have a better experience. The reader must note that this is a summary of the system, and the full description is in the corresponding RASD.

All of the functionalities happen through the generation and scanning of QR codes, notifications through sms messages, and their actual implementations will be described later in this document.

## 1.3 Definitions, Acronyms, Abbreviations

### 1.3.1 Definitions

<b>Ticket</b>	The QR code generated for entering into a store
<b>Invalid Ticket/code</b>	A QR code which is not valid to enter a store in that moment

	(Because it is not the time to enter, or because it already expired)
<b>Manager</b>	The person in charge of a store
<b>Store</b>	A branch or location of a supermarket.

### 1.3.2 Acronyms

<b>REST API</b>	Representational State Transfer - Application Programming Interface
<b>CDN</b>	Content Delivery Network
<b>RASD</b>	Requirements Analysis and Specification Document

### 1.3.3 Abbreviations

<b>WPn</b>	World phenomenon number n
<b>SPn</b>	Shared phenomenon number n
<b>Gn</b>	Goal number n
<b>F1</b>	Functionality 1: <i>Lining up</i>
<b>F2</b>	Functionality 2: <i>Booking a visit</i>
<b>F3</b>	Functionality 3: <i>Alternatives for the customer</i>
<b>Rn</b>	Requirement number n

## 1.4 Revision history

Date	Modifications
------	---------------

28/12	Initial version (0.1) <ul style="list-style-type: none"> <li>• Setup of the document</li> </ul>
09/01	First version (1.0) <ul style="list-style-type: none"> <li>• First completed version of the whole document</li> </ul>

## 1.5 Reference Documents

- Assignment Document: "R&DD Assignment A.Y 2020-2021"

## 1.6 Document Structure

- **Chapter 1:** Contains an introduction to CLup, as well as its goals and objectives. It also includes a description of the document, its purpose and main goals as a whole.
- **Chapter 2:** Contains a detailed description of the architecture that will be used CLup. It also has a explanation of the logical components that will exist in the system, as well as how they will interact.
- **Chapter 3:** Contains all the UI designs and wireframes of CLup.
- **Chapter 4:** Contains the binding between the Goals, requirements and components of CLup.
- **Chapter 5:** Contains the description of the integration and test plan of the development process.
- **Chapter 6:** Contains the efforts spent by each member of the group.
- **Chapter 7:** Contains a reference to the used documents.

## 2 Architectural Design

### 2.1 Overview: High-level components and their interaction

The architecture of CLup is set up in a logical three layer structure, organized as a client-server architecture. Each layer represents one part of the interaction between the user and the system, and the connections happen subsequently.

- **Presentation layer:** This part of the system corresponds to the one that users (managers and customers) interact with. It is responsible for rendering the information that the business layer sends to, as the user interacts with it, and it is just responsible for redirecting the requests to the business layer, as no logic happens here.
- **Business layer:** Contains all the functions and logic that makes CLup what it is, but users do not interact with this layer directly, only through requests from the presentation layer. It is also responsible for passing data from the Data access layer to the presentation layer.
- **Data Access layer:** Is in charge of managing all the data that the system needs to work properly, as well as the database's access. Receives requests from the business layer and passes it along.

As in a client-server architecture, parts will be hosted on different machines, allowing easier replication, scalability, performance and other characteristics that will be explained in section 5. The communication between layers will happen through specific interfaces detailed below.

The following section will give a high level explanation of the communication process for each main functionality, and later will be detailed for each component of each tier.

- **Lining Up:**
  - The client (customer) wants to get a ticket for a store. Through the provided interfaces, the client makes a request to the system's backend (server).
  - The backend gets the information from the database, calculates the waiting time if any, then adds the client to the queue and returns a code, so the client can use it as a visible scannable QR code.

- **Book a visit**

- The client wants to book a visit to the store at a later date, and same as before, uses the web interface to make the request.
- The server receives the requests, and after transforming the request into the database format, sends it to the data access layer for it to be saved. It then returns the ticket data back to the client.

- **Alternatives for the customer**

- This happens in conjunction with the other two functionalities. As the server receives the requests and after getting data from the database, it performs some checks so the customer can get the best shopping experience. In case it detects a better alternative it notifies to the client so he can decide.

### 2.1.1 High Level Components

CLup will be organized as a client-server architecture, more specifically, a three-tiered one, as it has many benefits like security, scalability and reliability.

The **presentation tier** corresponds to the web app provided by the content delivery network (a static web page hosted there), which provides the user interface where the users will interact with the application (the users access it through a web browser). Then, we have the **business tier** which will host the application server. Here, all the logic from the application will live, and is also in charge of connecting to the data tier, to get the information needed to function. The **data tier** has the database server, where all the data is stored.

Figure 1 shows a high level component view, with each tier and its interactions, as well as where will each tier live. The users, which can access the system through a web browser from their phones or computers, request the web page from the CDN web server. Then, from the user's device, the web page will interact with the application server, which is connected and can communicate with the database server.

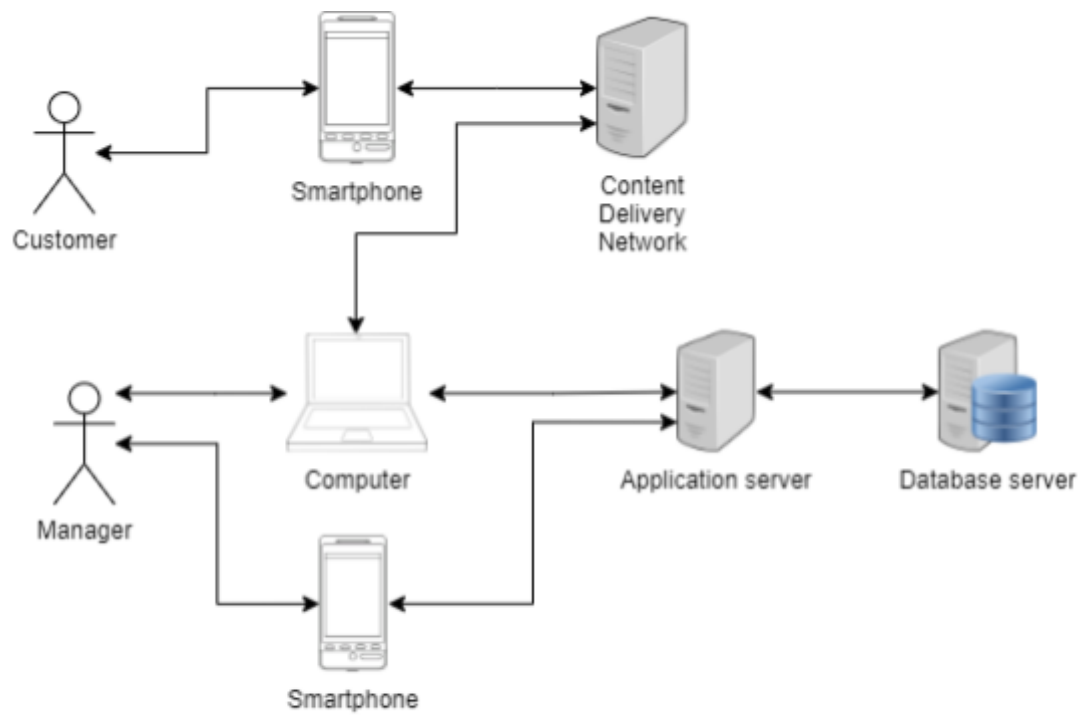


Figure 1: System Architecture



## 2.2 Component view

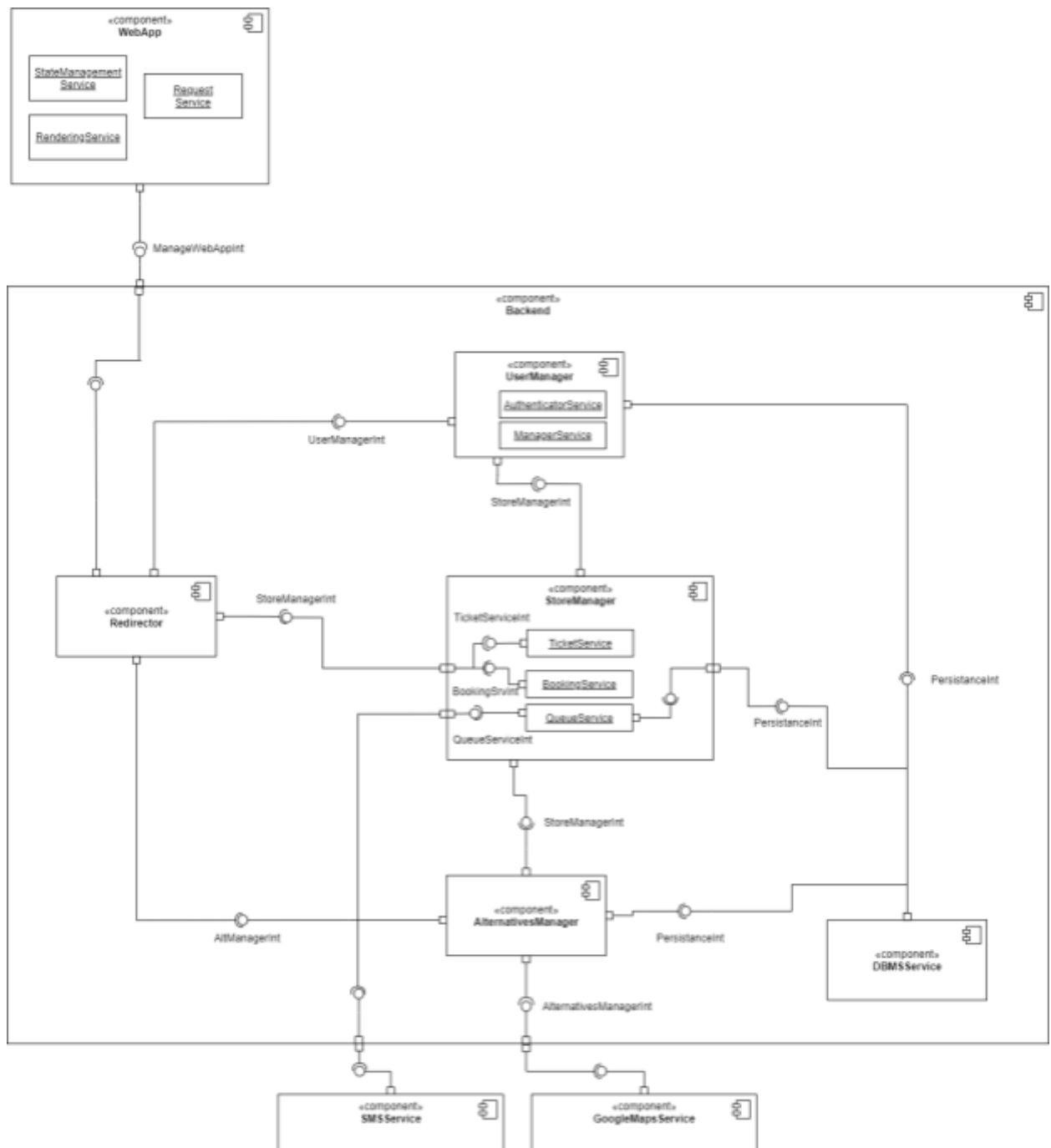


Figure 2: Component diagram

The component diagram on top shows the logical components and interactions that the system has. It is divided into three main parts, the web app, which is run on the client side. The backend or server, which is run on one or many machines and contains the logic and interaction with the

database. Last, there is a set of external services, which CLup needs to work better. The detail of each component and service is explained below.

- **Web app**

- Not shown in much detail in the diagram, as it is a classic static web app run on the browser. Has three main components for functioning correctly. A state management service, for keeping track of user sessions, a request service, for actually using the backend and a rendering service, for showing the user interface on the browser.

- **Backend**

- **Redirector**: This component is just used for sending each request to the specific component that will handle it. Please note that on the actual implementation this will be given by the backend framework, and will consist in a router for http requests.
- **UserManager**: It is used to handle requests that involve authenticated users. First, there is the **AuthenticationService**, for checking customers and managers credentials. Second, there is the **ManagerService**, which handles the scanning of tickets and checking stores.
- **StoreManager**: This component is in charge of managing all functionalities related to the store, this is the queue that lives in it and all tickets that will be used there. It contains three services. **TicketService**, which handles the creation of tickets, and using the **QueueService**, they get added to the queue of that store. Lastly, there is the **BookingService**, which using the other 2 services, handles the *booking* functionality of CLup.
- **AlternativesManager**: This component is in charge of checking the other stores with the QueueService in case a registered customer wants to book a visit or just get a ticket, and there is a long line for the requested store.
- **DBMSService**: This component is used by almost all the others, and is in charge of retrieving the data from the database. Contains a persistence unit interface that allows efficient queries and transactions.

- **External services:** The backend uses two external services for all of the functionalities of CLup
  - **SMS Service:** Used by the QueueService, it is used to notify the customers when their time to enter a store is near.
  - **GoogleMaps Service:** Used by the AlternativesManager, for giving customers suggestions on stores nearby based on their location.

### 2.2.1 Additional specifications

The **frontend** will be a static html page (generated by React, the frontend framework), which can be then hosted on a CDN and will communicate with the **backend** through http methods (the backend will provide an REST API interface). This configuration will give CLup many non functional benefits, such as:

1. Serving the frontend on a CDN allows for much faster response times, as the client does not have to wait for a server to generate the html that it sees. It also allows for caching data, so it doesn't have to be requested twice. Lastly, there will not be scaling issues with the frontend, as no server is being used.
2. On demand scaling for the backend will be allowed if needed through replication of the server with load balancers. The database will allow for multiple connections assuring ACID properties.

## 2.3 Deployment view

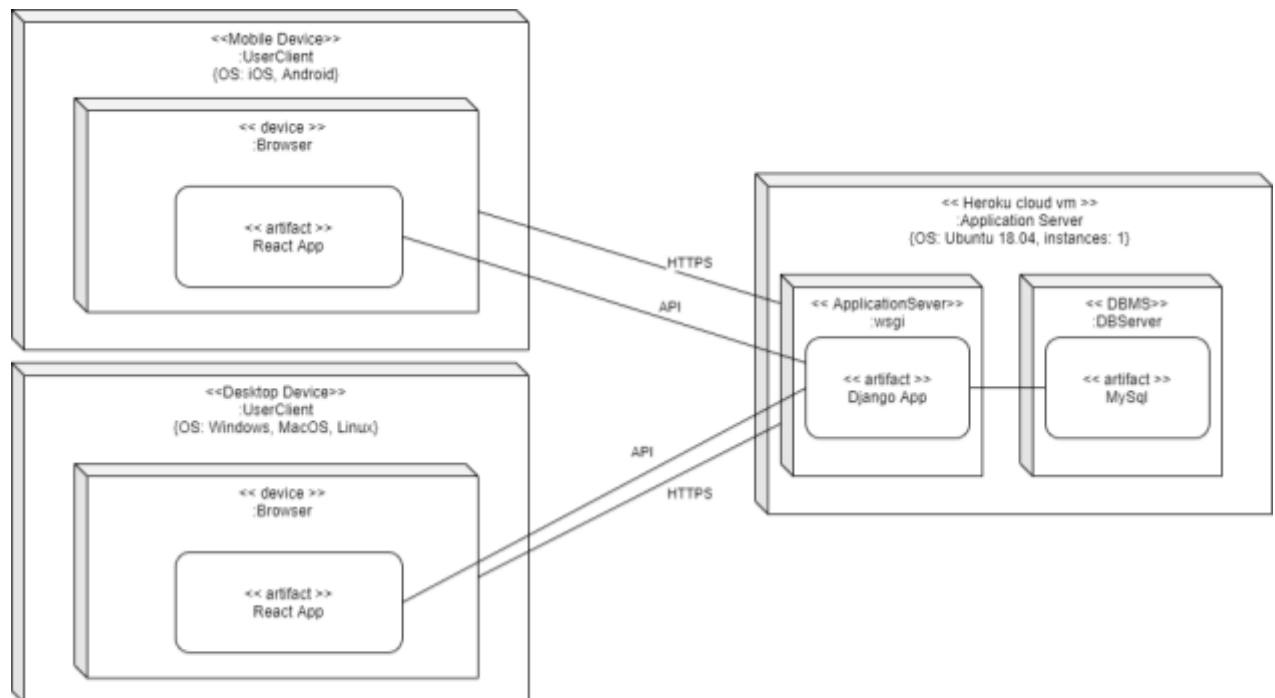


Figure 3: Deployment diagram

In the deployment diagram above, we can see how the system is organized. Note that the different tiers described in section 2.1 are well shown here. For the **presentation tier**, we have the mobile and desktop devices used by the end users, while the **business and data tiers** are represented by the application server, which hosts both of them. Each component is described in detail below:

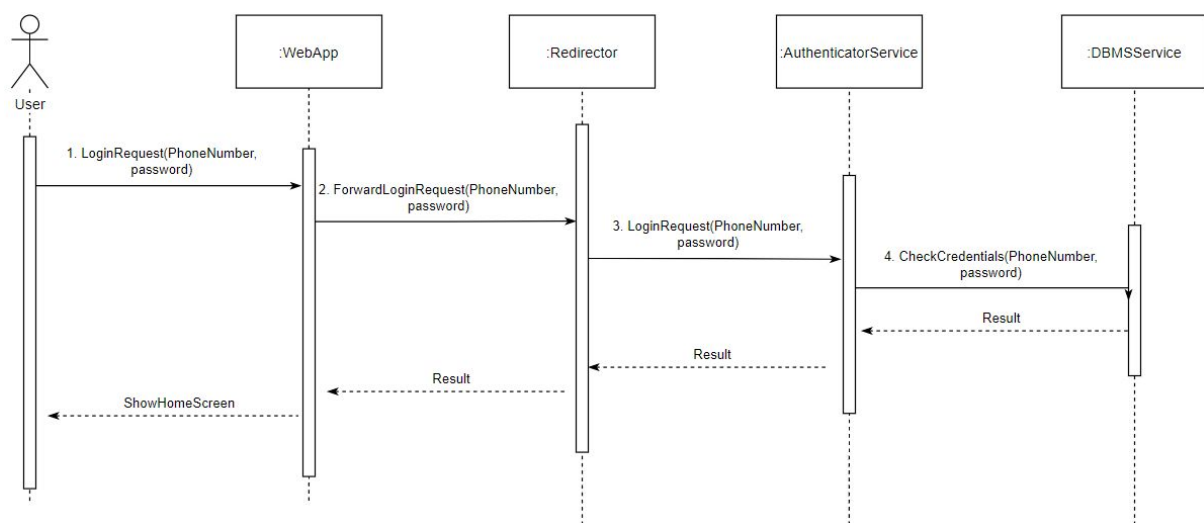
- **User client:** The end user (customers and managers) runs the web application on his/her devices through the browser, which is a static html page served by a CDN.
- **Application server:** Has two parts, the server itself, and the DBMS.
  - **Server:** A Django application, which exposes a REST API interface for the other components to interact with. All the logic which involves doing calculations and creating objects happens here, so that any client that connects with it can have the same version
  - **DBMS:** One MySQL instance hosted on the same machine as the application server. This will all be handled by Heroku, a Platform as a service which provides

many plugins so the data is secure, handled by a fast dbms and can be scaled down the road as needed.

## 2.4 Runtime view

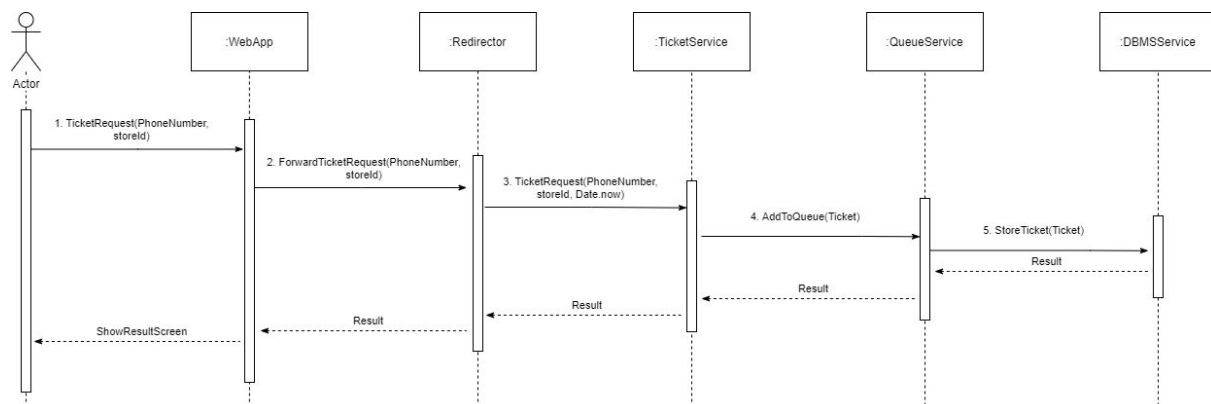
In this section, the interaction between the main components shown in the component view will be detailed, as well as the order in which this happens. This will be done through sequence diagrams, detailing the main activities that happen in CLup.

### 2.4.1 Login in



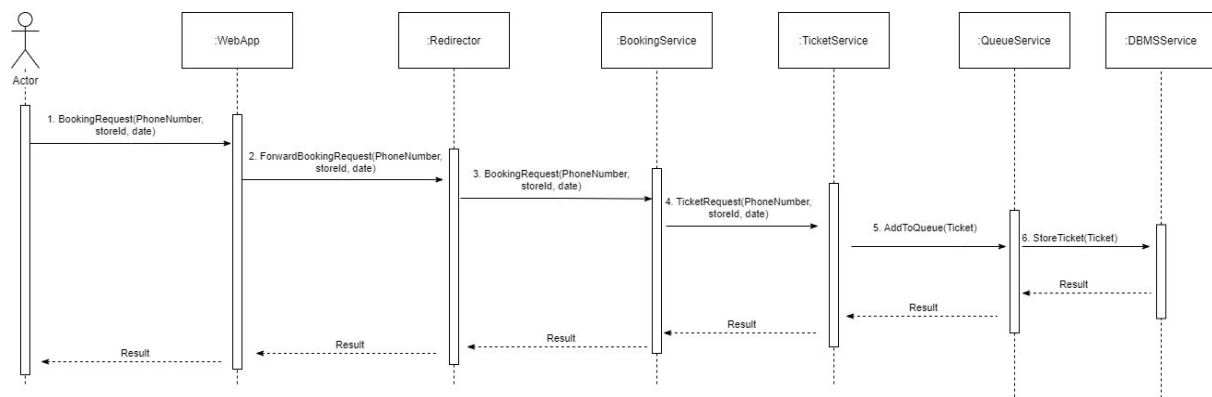
Here, the process of login in to CLup is shown. This is a general process, as both the customers and managers will do the same. The only difference between those two users, is that when a manager logs in, he will be redirected to the manager version of CLup, which has the functionalities to check stores and scan tickets.

### 2.4.2 Request a ticket



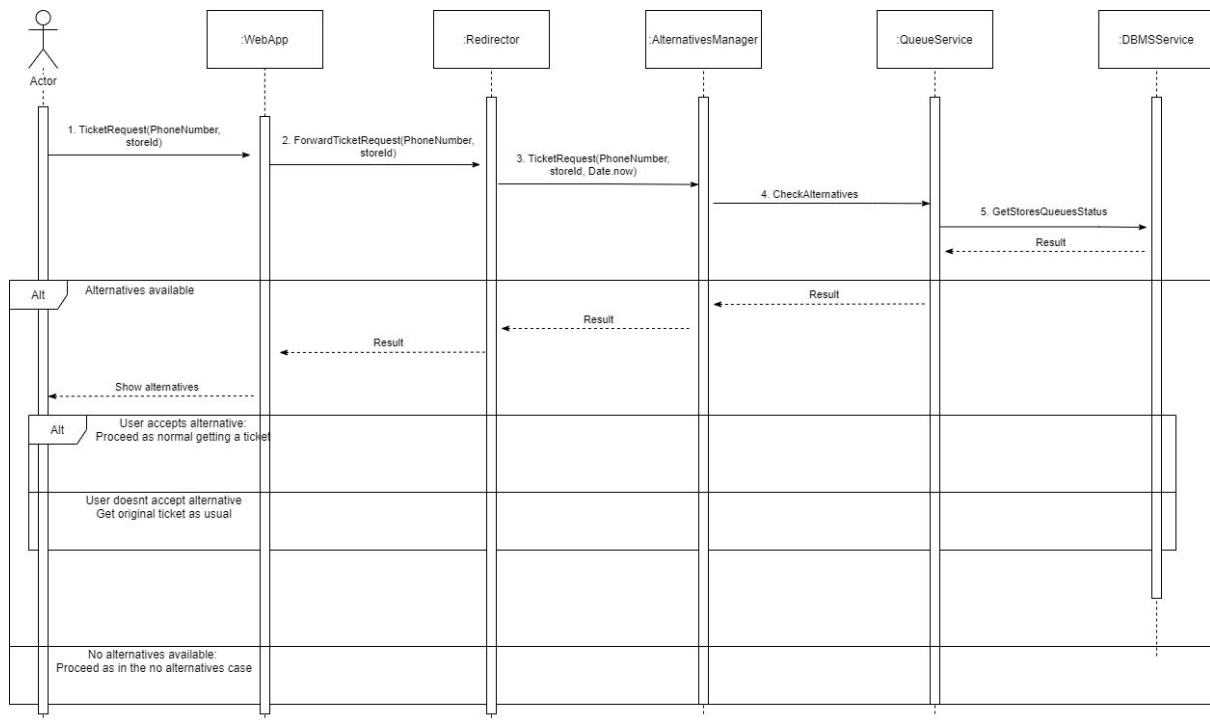
In this sequence diagram, the process in which a customer requests a ticket is detailed. This diagram is valid for non authenticated customers, as in the other case, the redirector passes the request to the alternativesService. Here, the redirector captures the user's request (in which the web app asked for the customer's phone number), and forwards it to the TicketService, which creates the Ticket object which is then saved in the queue and persisted through the QueueService and DBMSService respectively. If everything went ok, the result is given back to the user as a success screen.

### 2.4.3 Book a visit



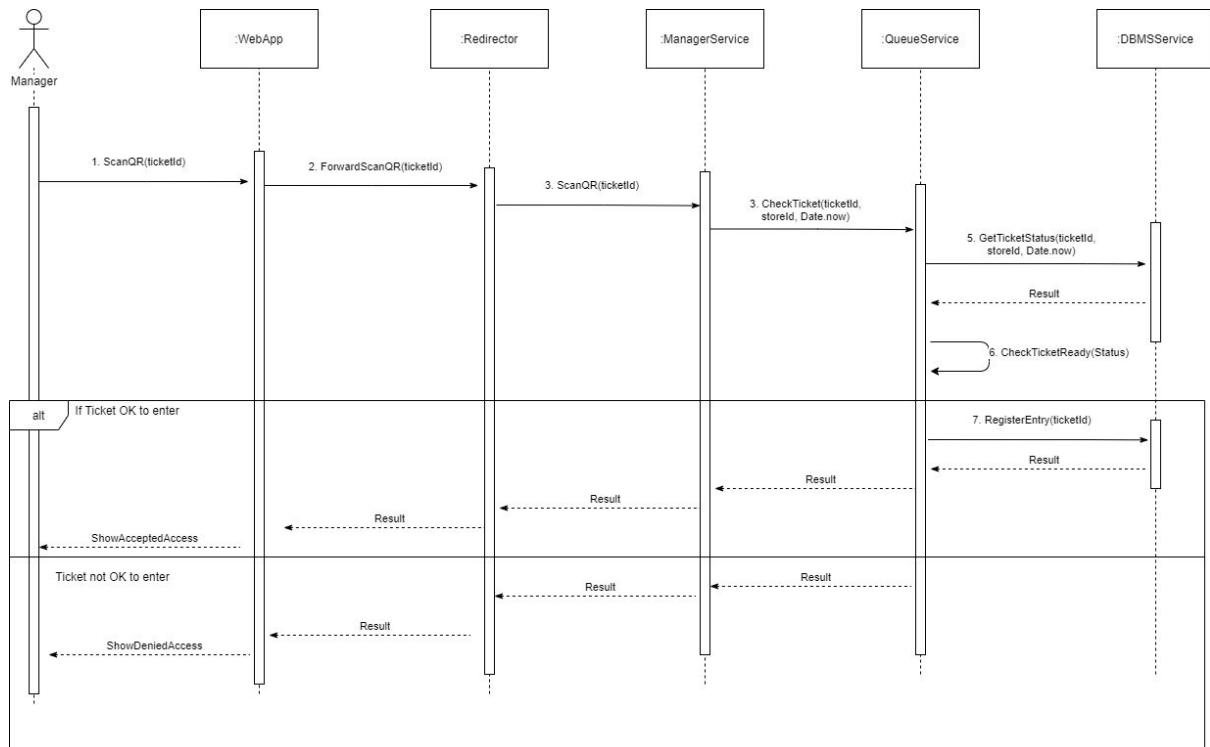
This process is very similar to 2.4.2, as the end goal is the same, generating a ticket. The only difference is that here, we go through the BookingService, which has a bit of logic to handle these requests, which are tickets with a different starting date and priority.

## 2.4.4 Alternatives for the customer



This sequence diagram shows a different version compared to the one in 2.4.2. The difference here is that the customer is authenticated, so the redirector takes the request to the alternativesManager first, which checks if there are any alternatives for the customer (better store, or better time later). The rest of the process is not shown, as it is the same to 2.4.2, in case the user accepts/rejects the alternative, or if there is not an alternative.

## 2.4.5 Scan QR code



This sequence diagram shows the interaction in which a manager scans a ticket from a customer, and the systems decides if he gets accepted into the store or not, based on an algorithm described in section 2.7 by the QueueService.

The sequence goes as follows. We start with the request (after a device has scanned the QR code), which is sent to the redirector. This component redirects to the managerService for authentication purposes, and finally the QueueService does the checks in the database to see if the customer is ready to enter or not, based on the state of the queue and the store.

## 2.5 Component interfaces

The following diagram represents the interfaces provided by the many logical components described in section 2. 2, and how they interact with each other as in the runtime view section.

The reader must note that this is a logical description of their interfaces, and do not represent in its entirety how these components will be programmed. Also, some methods are omitted for the sake of simplicity, which can be inferred from the context (registration methods, logging methods, etc)



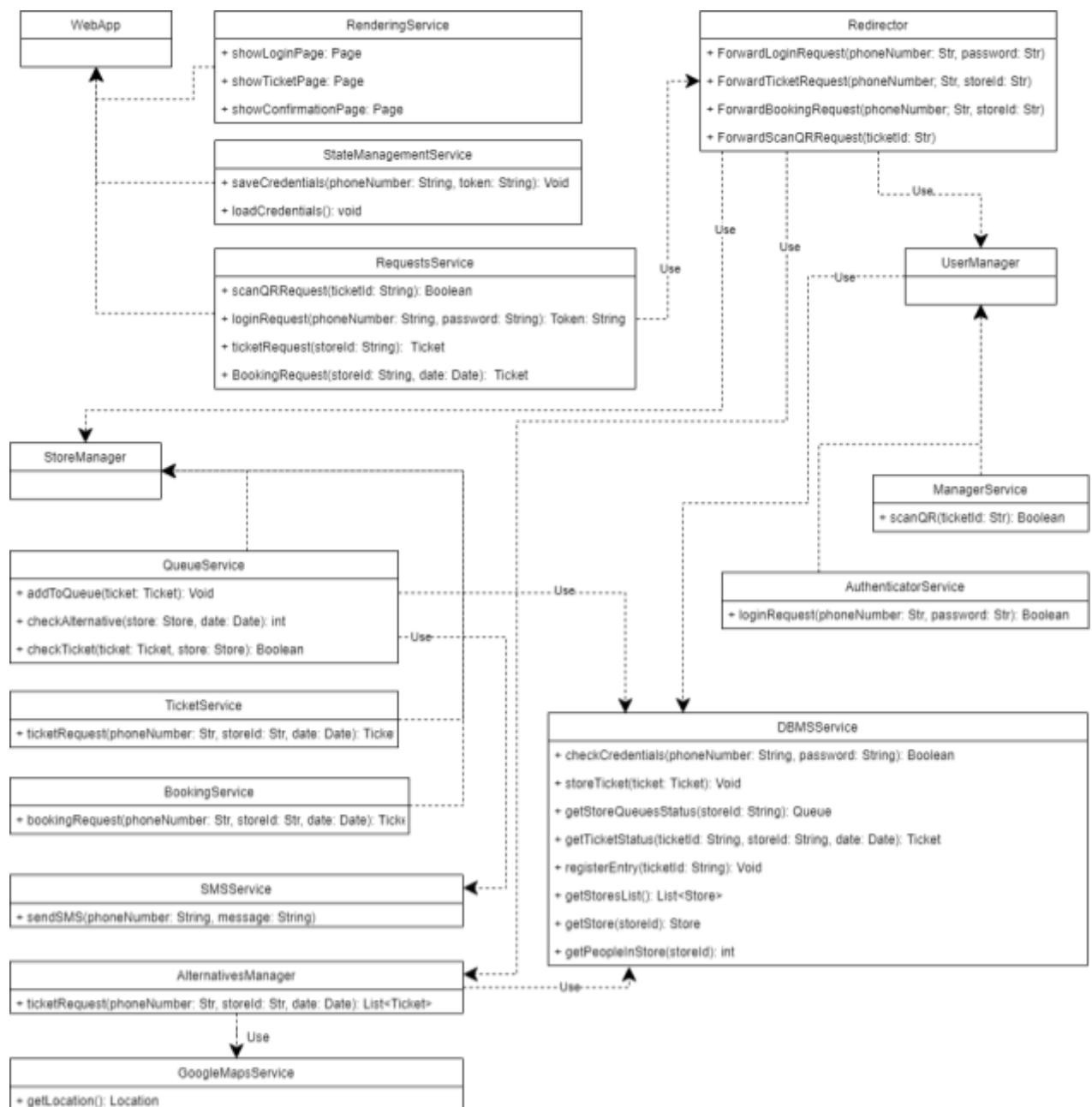


Figure 4: Interfaces diagram

## 2.6 Selected architectural styles and patterns

As mentioned before, the architecture for CLup is a pretty well established client-server one, with three well defined tiers. Having a clear separation of the client and the server gives many benefits, such as better scalability, less coupling of components, and it is easier for developers to actually work on it.

This separation is further improved by the use of **client side rendering**. This means that all clients receive the same web page to render, and later the data is fetched from the server, allowing faster load rates and less data being passed from one side to the other. All of this information will be passed through https methods, because the server will expose an REST API interface.

### REST API

The server, written in the django framework, will be exposing an REST API (*representational state transfer application programming interface*) so the client can make requests from it. This is a software architectural style for providing a uniform and stateless interface to interact with, it is widely used in the web and simplifies the passing of information. All data will be passed as json objects, which maps perfectly to the world of javascript, in which the client will be coded.

### Facade pattern

The redirector acts as a facade for the client to interact with the server. It simplifies the communication and provides one consistent and easy to use interface to the client. The implementation of this pattern is given in the framework chosen for the server, django, as the router which handles all the requests automatically delivers them as they are needed, based on the route that is being called.

### Relational Database

For the database, a relational one was selected for many reasons. The data in CLup will be highly structured and won't change over time, it is pretty simple to use and integrates very well with the framework chosen.

## 2.7 Algorithms

In this section, the algorithm used for checking who can get into a store based on a ticket, as mentioned on section 2.4.5, is described in detail through pseudo code.

```

Boolean scan_ticket(store: Store, ticket: Ticket)
{
    Queue queue = store.Queue

    int people_in_store = Store.get_people_in_store()
    int max_occupation = Store.max_occupation

    int spaces_available = max_occupation - people_in_store

    // This functions returns how many people ahead of
    // this particular ticket are currently in the queue
    int place_in_line = queue.get_position(ticket)

    if (spaces_available == 0)
    {
        // there is no space in the store
        // Nobody can get in until somebody Leaves
        return false
    } else
    {
        if (spaces_available >= place_in_line)
        {
            // The user can enter the store, as the space in the store
            // does not violate the restriction that there are people
            // ahead of him
            return true
        } else
        {
            // There is space in the store, but many people have priority
            // over this user
            return false
        }
    }
}

```

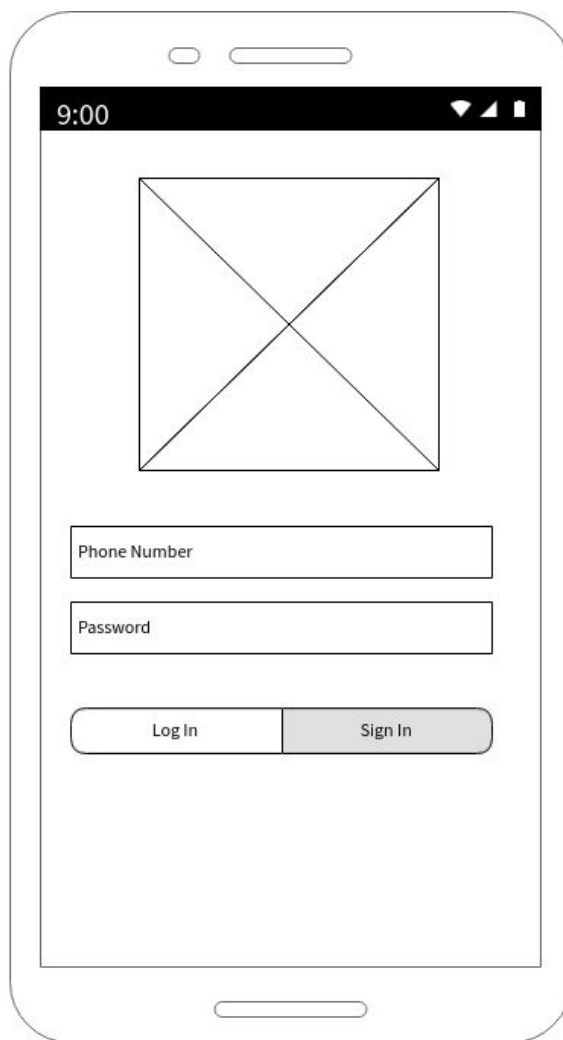
A very similar function will be used when the customer wants to know how much time he has left for entering, as the only difference will be to return the number of people in front of him minus the space left on the store, multiplied by some value that approximates the time that one customer takes in the store.

### 3 User Interface Design

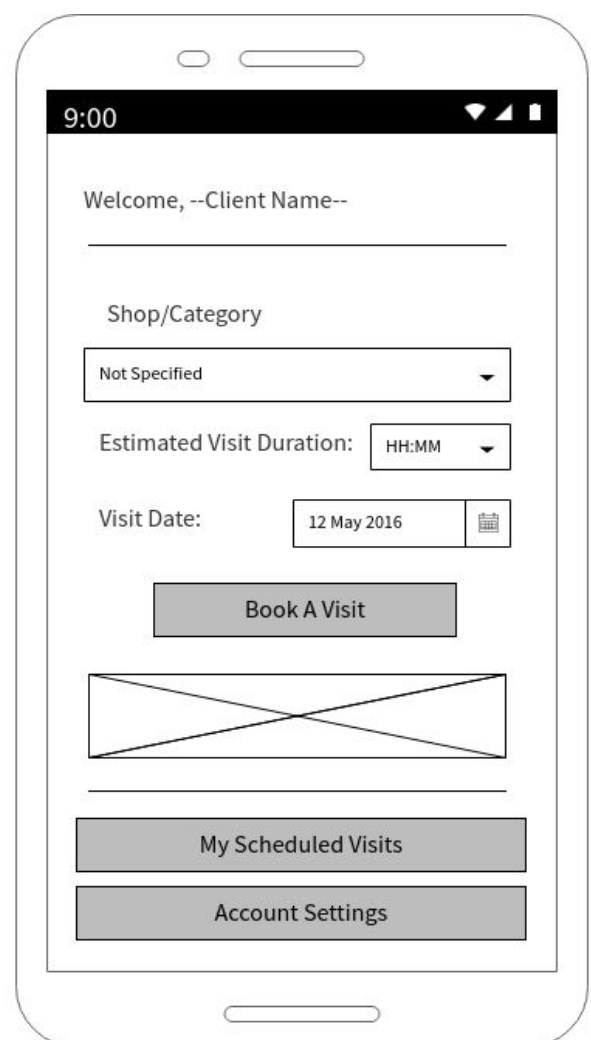
The following wireframes show how the web app will show to the customer users on mobile browser (subsection 3.1), to the shop manager and personnel (subsection 3.2) and the UI of the physical devices (subsection 3.3).

#### 3.1 Mobile Customer Wireframes

##### A. Customer Login Screen



##### B. Booking Screen

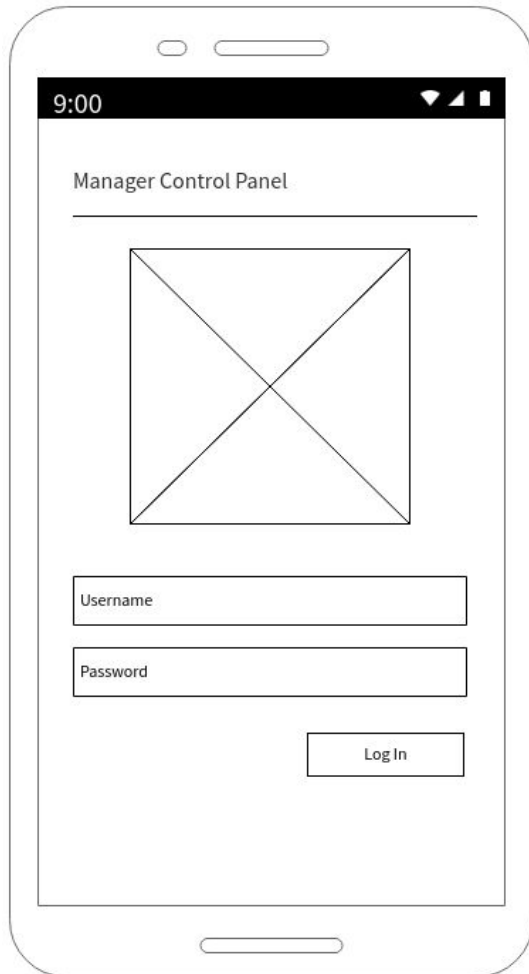


### C. QR Code Screen

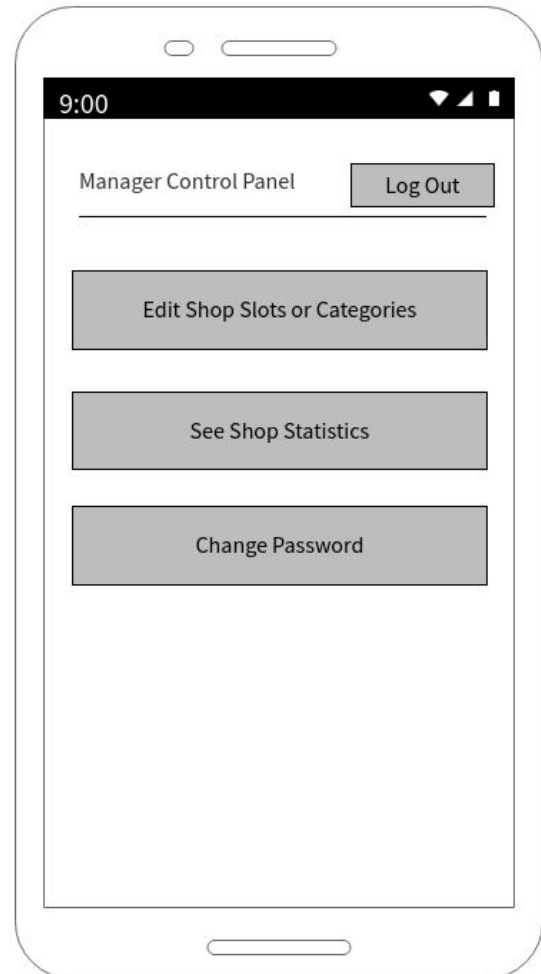


## 3.2 Shop Manager Wireframes

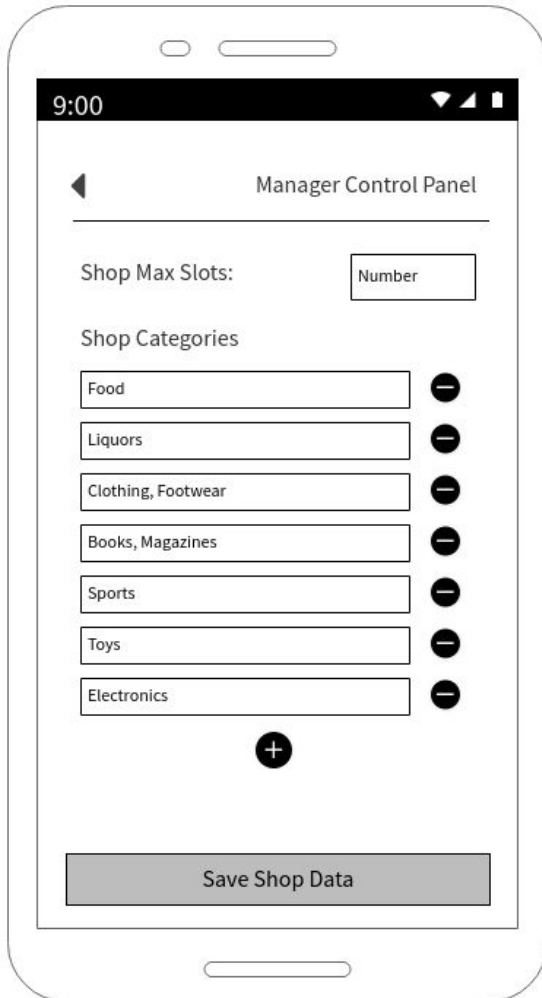
### A. Manager Log In Screen



### B. Admin Screen



### C. Shop Edit Screen



The Shop Edit Screen is a mobile application interface for managing shop data. It features a status bar at the top with the time 9:00 and standard Android navigation icons. Below the status bar is a header with a back arrow and the title "Manager Control Panel". The main content area is divided into two sections. The first section, "Shop Max Slots:", contains a text input field labeled "Number". The second section, "Shop Categories:", contains a list of categories: Food, Liquors, Clothing, Footwear, Books, Magazines, Sports, Toys, and Electronics. Each category is represented by a text input field with a minus sign button to its right. Below the list is a plus sign button. At the bottom of the screen is a large button labeled "Save Shop Data".

9:00

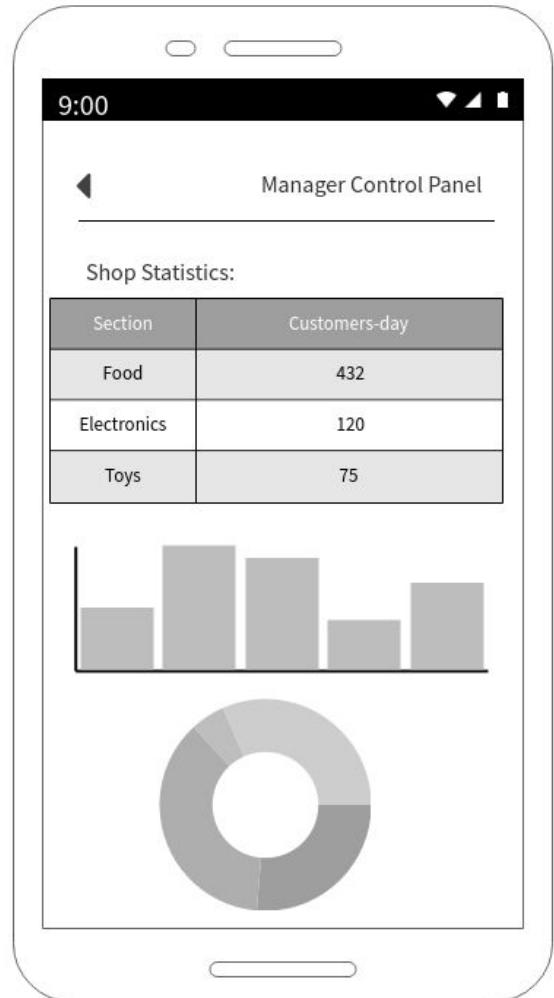
Manager Control Panel

Shop Max Slots:

Shop Categories


- Food
- Liquors
- Clothing, Footwear
- Books, Magazines
- Sports
- Toys
- Electronics

### D. Shop Statistics Screen



### 3.3 Physical Devices Wireframes

#### A. Ticket Dispenser




Welcome to #SHOP NAME#!  
Fill in the sections and then press the button to get a ticket

Estimated Visit Duration:      Phone Number: \_\_\_\_\_




30 Minutes	1	2	3
1 Hour	4	5	6
2 Hours	7	8	9
3 Hours	+	0	

PRINT TICKET

#### B. QR Scanner



Welcome to #SHOP NAME#!  
Place your QR Code in front of the scanner on the right



Scanner Camera

Dont Forget to sign your exit



## 4 Requirements Traceability

This section bind the goals defined on the RASD with design components

G1	Allow managers to regulate the influx of people inside supermarkets.
	Requirements: R2, R4, R6, R7, R8
	Components: <ul style="list-style-type: none"><li>• Web App</li><li>• Store Manager</li></ul>
G2	Save people from having to line up outside of stores for long times.
	Requirements: R1, R2, R4, R5, R6, R7
	Components: <ul style="list-style-type: none"><li>• Web App</li><li>• Store Manager</li></ul>
G3	Allow customers to have a safer shopping experience.
	Requirements: R2, R3, R5, R7, R8
	Components: <ul style="list-style-type: none"><li>• Web App</li><li>• User Manager</li><li>• Store Manager</li><li>• SMS Services</li></ul>
G4	Allow customers to have a more efficient shopping experience.
	Requirements: R1, R2, R3, R4, R5, R6, R7, R8
	Components: <ul style="list-style-type: none"><li>• Web App</li><li>• User Manager</li><li>• Store Manager</li><li>• Alternatives Manager</li></ul>
G5	Allow customers who don't have a smartphone to shop in this technological era.
	Requirements: R2.2, R4
	Components: <ul style="list-style-type: none"><li>• Store Manager</li><li>• SMS Service</li></ul>

## 5 Implementation, Integration and Test Plan

### 5.1 overview

To develop an error free software, verification and validation are keys to achieve success at this step. Bugs can be found even after testing of the application. So, the goal here is to identify and fix as many bugs as possible before the date of release of the application.

### 5.2 Implementation

Implementation of bottom-up approach along with other components for the entire system has to be done. A bottom-up approach in conjunction with the client-server architecture chosen, allows for simultaneous implementation on both client and server side along with testing. Moreover, the bottom-up approach is implemented in incremental fashion, which will turn out to be an easy way to construct with little amount of components along with integration. One of the major advantages of the bottom-up strategy is that it allows you to make decisions with a much wider pool of knowledge. At this stage it is important to note that a bottom-up approach will be implemented and testing of services that comprises different parts of the system.

Furthermore the client side **WebApp** component can be implemented at the start, its a component which manages all the user interface and client side functionalities like taking inputs from the user and displaying information to the user. This Frontend will be integrated with the logical backend, so then everything can come together in the form of a single page application.

For the **Backend**, lower level logical components will be implemented first, and with a gradual flow higher level components will then be included. There exists a dependency among other components. For instance, every part of the system and its components directly or indirectly communicate with the database. Implementing the **DBMSServices**, provides access to the Database to perform queries and various other operations. It plays a major role in extracting, transforming and loading the Database and is a continuous part of the system.

Followed by the implementation of the two major components present in the server, the **UserManager** and **StoreManager**: the first manages all the users information, authentication and validation of the user. We consider the manager (actual person interacting with the app, not a logical component) of the store also as a user but with different levels of access and functionality like a system administrator. The latter one, the StoreManager manages all the functionalities of the manager like updating the ticket status, booking a ticket, checking the status

of the queue. Each one of these components has subcomponents that will need to be implemented first.

Then, the **AlternativesManager** component can be implemented, which is used to provide users with choices available from the current choice with the help of data retrieved from the database. Using a bottom-up approach to test this right away is not so easy and possible because AlternativesManager depends on external service **GoogleMapsService** which provides the location of user and store and also to display the list of available stores. So this component can be fully tested only after the integration of **GoogleMapsService**. So, at the same level **StoreManger** also depends on **GoogleMapsService** to calculate the estimated time to reach the store and present the user with appropriate time of the ticket.

Thus in the previous section while implementing, the **StoreManager** partial testing can be done and after integrating the **GoogleMapsService** partial testing is done and **SMSService**, which is an external service also integrated and then final testing of the StoreManager can be done. SMSService manages the notification and timely updates that are needed to be sent to the user regarding details and time of the visit.

After implementing the above components we can start implementing the **Redirector** component which manages the connection between the components and routes all the requests to corresponding components where and when required. The chosen framework provides an easy to use and already tested interface which will help in this task, so only integration testing will need to be done here.

The order in which we will implement our components in the system are:

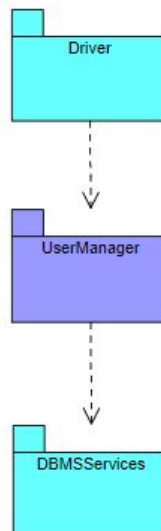
1. DBMSService,WebApp
2. UserManager
3. StoreManager
4. AlternativesManager
5. Redirector

The **GoogleMapsService** and **SMSService** are not included in the above list because they are external services and they only need to be integrated and not implemented from scratch. They will be integrated after implementing the **AlternativesManager** component and tested while performing the integration tests.

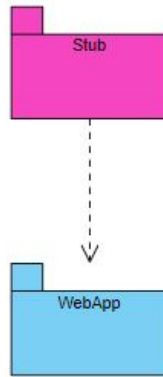
### 5.3 Integration Strategy

In order to implement and test the various functionalities of the system a bottom-up approach is used. The process of implementation and integration takes with the same approach mentioned above.

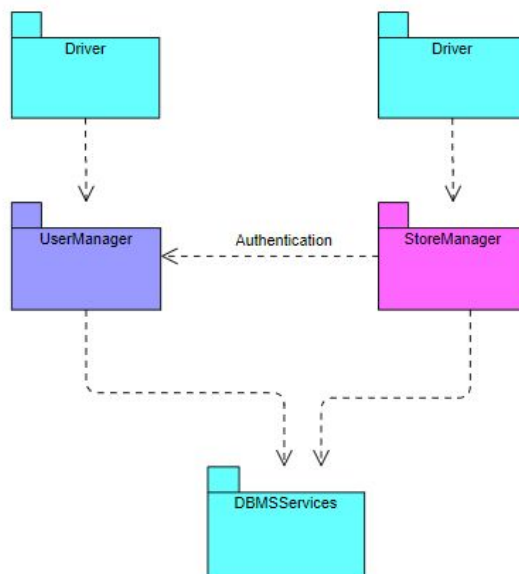
- (1) First, the **UserManager** is implemented and unit testing is performed using the components that are in the process of implementation. It is used to implement functionality of authenticating and validating the user. As it is not a complex component but different components depend on it to provide services. This is the reason behind the implementation and testing before others..



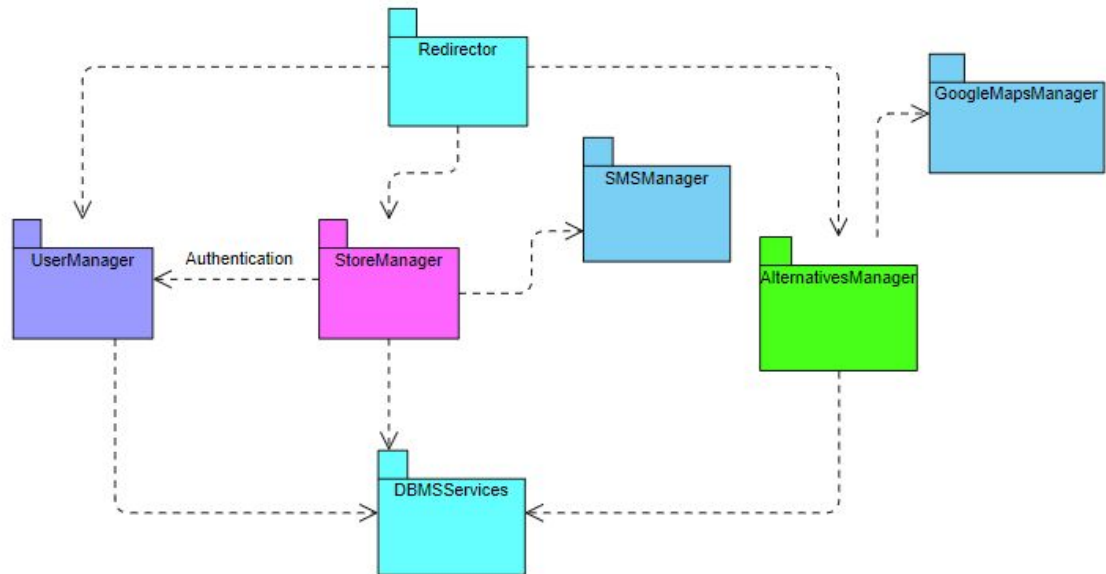
- (2) The **WebApp** is implemented parallelly alongside with other components. For testing purposes, we can use stubs mimicking the real components to complement the original calls and data. Once the remaining components are implemented then we integrate it with them and perform integration testing. This approach will greatly improve the implementation time, as 2 different teams can work in parallel. This is a direct benefit of the chosen architecture.



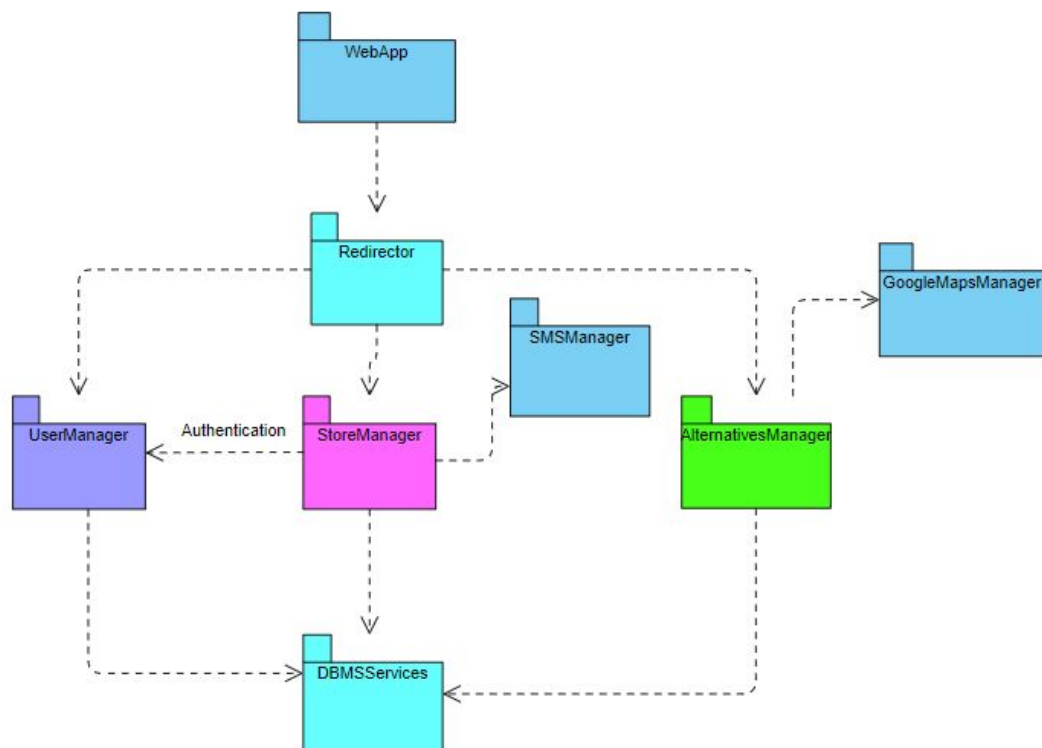
- (3) Then, we proceed with the **StoreManager** implementation which is used to track the number of tickets in the queue, booking of tickets, and related functionalities.



- (4) Then takes place the implementation of **AlternativesManager** which is used to provide alternate choice to the user. **Redirector** is used for mapping input requests to corresponding components.



- (5) Finally we integrate the **webApp** which is implemented parallelly to the backend with rest. Once it is integrated we can perform final integration testing.



Finally remaining parts characterizing the client side must be implemented, unit-tested and integrated into the system. This process happens only when components from both sides have been implemented and tested.

Once every component has been implemented and tested and put into the entire application, system testing can be performed.



## 5.4 System Testing

To validate and verify the functionalities of the system, as well as the non functional requirements mentioned in the RASD, the system should be completely integrated. To achieve this system testing is the only way. In Addition, the testing environment should be the closest possible component to the production environment.

The system testing can be divided in 4 types:

- **Functional testing:** Verifies whether the application satisfies the functional requirements mentioned in the RASD, this means, does the CLup complies with the functionalities set by management.
- **Performance testing:** This testing technique checks how well the system performs under normal conditions, measuring response time, utilization of resources, throughput and performance.
- **Load testing:** This is done to understand how CLup will behave under expected load, and to check if the non functional requirements are satisfied. It exposes bugs such as memory leaks, mismanagement of memory, buffer overflows and identifies upper limits of components.
- **Stress testing:** Is used to verify stability and reliability of the software and the system, by putting it under heavy load. It also makes sure that the system recovers gracefully after failure.

## 6 Effort Spent

### 6.1 Hours of Work

Topic	Hours
Initial setup of document	1hr
Section 1 - Introduction	3hr
Section 2 - Discussion on architecture of the app	2hr
Section 2 - Architectural design description	4hr
Section 2 - Logical components, interfaces and sequences - design and modeling	10hr
Section 3 - Web app wireframes	2hr
Section 4 - Component and Goals binding	1hr
Section 5 - Discussion and uml diagrams	5hr
Final review of document	3hr



## 7 References

- Diagrams made with <https://www.draw.io/>
- Testing and integration definitions from <https://www.guru99.com/>
- Three tier architecture description <https://www.ibm.com/cloud/learn/three-tier-architecture/>
- Recorded classes and documents from Software Engineering 2