POLITECNICO
MILANO 1863

- DD -

Design Document

Computer Science and Engineering

Software Engineering II

A.A. 2020/2021

Daniele Mammone - 10625264
Gianmarco Naro - 10610374
Massimo Parisi - 10583470

Contents

1 Introduction

1.1 Purpose

The main target of this document is to describe the **Customer Line-Up** (*CLup*) design from a more detailed point of view. This document follows faithfully what was defined in the Requirement Analysis and Specification Document and must be read carefully before starting the software implementation in order to understand the software design in detail. Moreover, we tried to maintain independence between *DD* document and *RASD* document in order to allow greater flexibility in case you want to reuse the design model.

1.2 Scope

CLup is a booking application and nowadays this type of applications are more and more widespread in people's everyday life and are becoming more and more indispensable. For this reason, intelligent design choices have been made that extrapolate the positive aspects of existing booking apps, in order to make CLup really "achievable" in real life.

The main purpose of CLup is to facilitate customers to access at a store in **security**, both allowing them to reserve a spot on the queue for entering the store through the app and to book a visit at the store at a determined time of a certain day, selected by the customer. Thanks to this, store managers can manage the **affluence** in their store more easily, and moreover can reduce the crowd in front of the store, that is one of the main purposes of the application. Another important feature deals with obtaining statistics from customers and generic information from stores in order to help customers during their reservation. For this, the system has to store a very large amount of data. This data are mined and used later, which is why it is very important to identify the user's role so that we can provide him with more accurate and suitable information. Indeed a customer wants to provide information about the stores while, for example, a store manager is more interested in information about customer permanence in the store.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

QR Code	Bi dimensional bar code that allows the user to check-in/check-out at the store entries/exits
Reservation	Indicates both booked visits and spots on the queue to enter the store as soon as possible
Customer	The clients of the store, that uses the system to get a reservation to access the store
Store manager	The app user that access to stores' bookings, occupancy and settings, in order to manage the flow of customers
QR Code Reader	Device used to scan customers' *QR Code*
Totem	Electronic device that allows customers to physically get a spot on the queue to enter the store a soon as possible; it allows to specify the same parameters that can be inserted through the app
QR Code Printer	Device used by totems to print *QR Code*
Department	Part of the store that contains the same category of products
Query	Synonym for request

1.3.2 Acronyms

RASD	Requirement Analysis and Specification Document
DD	Design Document
ETA	Estimated Time of Arrival
GPS	Global Positioning System
API	Application Programming Interface
UML	Unified Modeling Language
DBMS	DataBase Management Service
OS	Operative System
HTTPS	HyperText Transfer Protocol over Secure Socket Layer
TCP	Trasmission Control Protocol
IP	Internet Protocol

1.3.3 Abbreviations

IIT	Implementation, Integration and Testing
Rn	Requirement number n
ASAP	As soon as possible
POV	Point of view

1.4 Revision History

Version	Date	Changelog
1.0	24/12/2020	First version

1.5 Software and Tools

- LaTeX as software system for document preparation
- UMLet for the UML diagrams and other diagrams
- Photoshop for the mockups
- Git & Github as work space. The repository is here.

1.6 Reference Documents

- Specification Document
- Slides of the lectures

1.7 Document Structure

The structure of the document is thought with the intention of allowing simple navigation through it. Also, various abbreviations, highlighted in Abbreviations section, have been used to make the content smoother. Hence, the structure of the document is the following one:

- **Introduction**: this section gives a general view of the problem and describes the scope and purpose of the *DD*, including a set of definitions, acronyms and abbreviations used.
- **Architectural Design**: this section starts with a high level description of the architecture of the system and continues going into details, specifying the components and interfaces used.
- **User Interface Design**: this section presents the mockups of the application, describing how the clients can navigate the application, highlighting the actions they can do.
- **Requirements Traceability**: this section describes the connection between the RASD and the DD, identifying the relation between goals and requirements described previously and the components that allow to realize them.
- **Implementation, Integration and Test Plan**: this section establishes a plan for the development of components, identifying the conditions needed to be met before starting the development process and then maximizing the efficiency of the developer teams with a precise schedule.
- **Effort Spent**: the main focus of this section is to track the time spent to complete this project. In particular, is highlighted the subdivision of the working hours of the various sections.
- **References**: this section is dedicated to all references used in this project.

2 Architectural Design

The aim of this section is to give a look to the architectural aspect of the analyzed system. In doing this, a top-down approach will be followed, starting from a very general view of the system, then going into more detail through **Component Diagrams**, more and more detailed. This allows us to precisely describe each component of *CLup*, so that who will develop the system would know the behaviour of each component.

2.1 Overview

A first high level overview can be done through the system's **Composition Diagram**. In fact, here there is a first subdivision of the system's component, each one with a different task.

- **Communication Interfaces**: to work properly, the system needs to interface over the network with both remote applications, and external services. Because of this, there are some components devoted to interact with remote devices:
  - **UserHandler**: to interact with users, both managers and customers, the latter both at *totems*, and on app.
  - **EntranceManagement**: so that the store can process a scanned *QR Code* at entry/exit.
  - **NumberCallingInterface**: to notify the call of a certain number.

  and others designated to access external services, such as
  - **Map Component**: to access an external **Map Provider**.
  - **Notification Component**: to send notifications to customers through an external **Notification Service**.
  - **Mail Component**: to externally sending mails, avoiding building an internal **Mail Manager**.

- **Business Logic**: here there is the whole Business Logic of the system. Since each store may maintain a different logic from another, there is a Component for each store managed by the system. Each store, manages the following aspects:
  - **Reservation Manager**: manages all the things concerning reservations and their making. It's useful also to retrieve some important data, such as *ETA*s and available time intervals to enter the store.
  - **Statistic Manager**: it's the part used for the calculation of statistics about customers' shopping time, and about the average time spent in a single department. Since each store is different from the other, the statistics are maintained unique per store. This part is also used by **Reservation Manager** to calculates *ETA*s and time intervals;

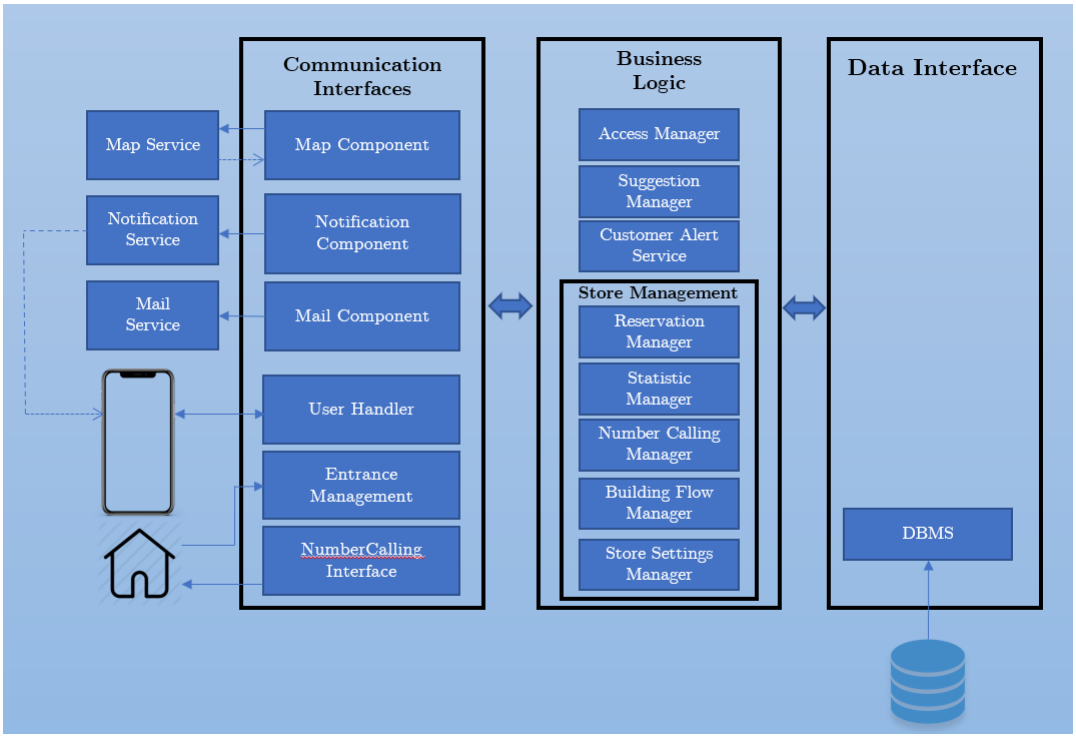


Figure 1: *Number calling system*

- **Number Calling System**: is the component dedicated to admit reserved customers inside the store.
- **Building Flow Manager**: it takes care of processing scanned *QR Codes* at the entry and the exit of the store.
- **Store Settings Manager**: manages all the things concerning stores' settings and parameters, such as working hours and capacity (also for each department).

This subdivision allows to separate better the logic of each store. Moreover, there are component common to each store, so the ones designated to allow users to request services to the system.

- **Access Manager**: manages users' registration and log-in;
- **Suggestion Manager**: manages the retrieve of suggestions when required, or necessary;
- **Customer Alert Service**: takes care of sending departing notifications to customers.

At the end, there is another block: the **Data Interface**: it represents the *Data Tier* of the system and separates the Business Logic from the data. It interacts with the *DBMS* to store and load informations.

2.2 Component View

The aim of this section is to give a look to the architectural aspect of the analyzed system. In doing this, a top-down approach will be followed, starting from a very general view of the system, then going into more detail through **Component Diagrams**, more and more detailed. This allows us to precisely describe each component of *CLup*, so that who will develop the system would know the behaviour of each component.

- **MobileApp**: it is the component relative to the mobile application used by store managers and customers; in order to access *CLup* services. It uses three interfaces: *CustomerInterface*, *StoreInterface*, used to access functionalities respectively for customers and store managers, and *AccessInterface*, that allows users to log-in and register. Moreover, uses *MapsAPI* to access to cartographic services, and *NotificationAPI* to send local notifications.
- **CLupServer**: it is the component designated to manage users' access, and customers requests.
- **StoreComponent**: it is the component that principally manages the business logic of the system, since takes care of the functionalities associated with a single store. Each store will have its own **Store Component** deployed in a dedicated container. The aim of this is to separate the critical functionalities of each store, to reduce the risk of a complete block of the system and to make maintenance easier in case of dysfunctionality of a single store.
- **TotemApp**: it is the component that represents the interface used by a physical totem installed at stores in order to request the issue of a ticket. The totem interface is separated from the Mobile App's one, since a totem can request tickets only to the store where it's installed, directly communicating with it.
- **ScannerApp**: it is the component that represents *QR Scanners*', that asks the associated store to process the scanned *QR Codes*.
- **NumberCallingApp**: it is the component designated to notify at stores that a number has been called. It's used by the associated store to notify that some customers are admitted to enter the store.

Moreover, to exploit some features, *CLup* relies on some external services, using them through their *API*s:

- **MapsAPI**: exploits the *API*s of an external map services, necessary to compute percurrency times and sort stores depending on the position.
- **NotificationAPI**: it is the interface used to send notifications to customers' devices. Since each mobile *OS* already has its own notification system, *CLup* exploits this in sending notifications, without the need of building a new in-house notification service.
- **EmailAPI**: it is the interface used to communicate with a mail provider service, to send emails to customers.
- **DBMSAPI**: the interface used to access the Data Logic of *CLup*.

It has been chosen to not develop these features, since there isn't a real necessity of making them from scratch: the existing ones work well, are already tested and complete (for example, developing an internal map service will require inserting the whole world cartography, while a non *OS*-integrated notification service must deal with aggressive *OS*es' power management). Moreover, this outsourcing allows to reduce the time for developing the system, and the maintenance costs, since it's externally done by the service provider.

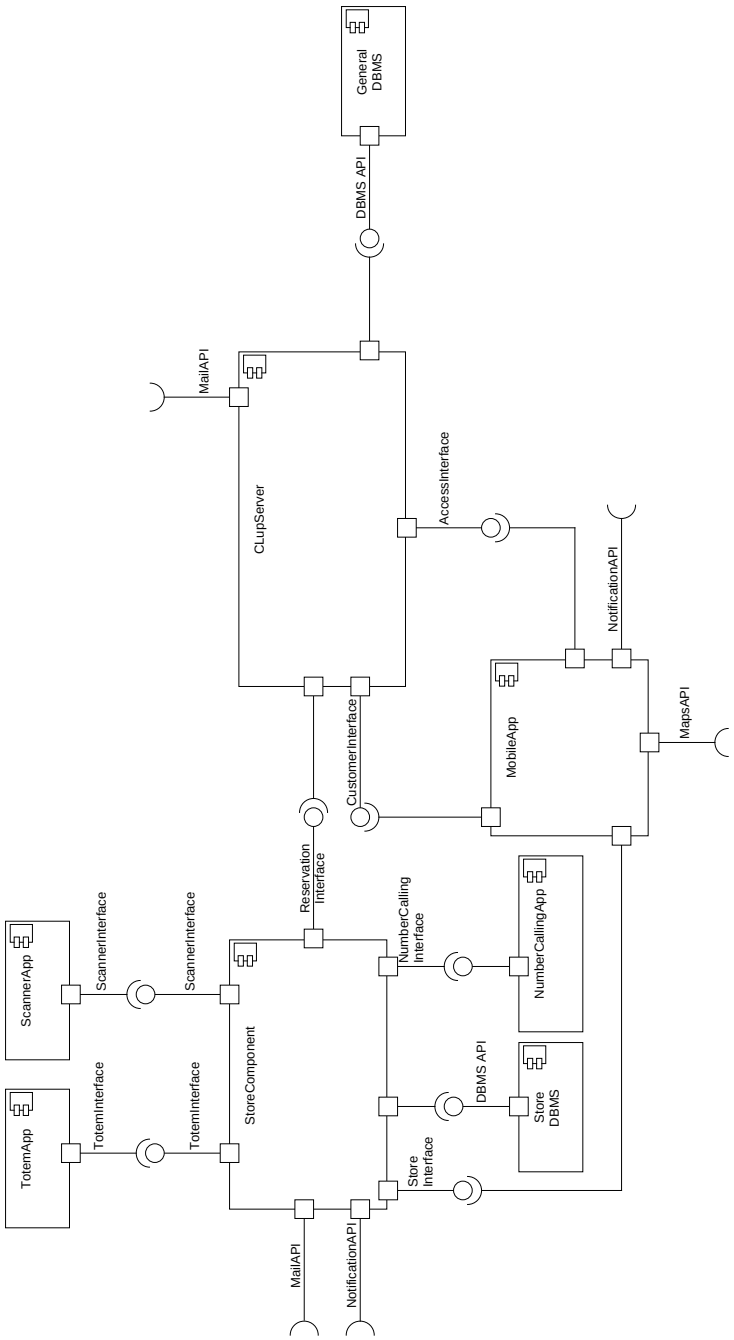


Figure 2: *High Level Component Diagram*

2.2.1 Adapters

The system uses some "Adapters" to access the external services; no *API*s are directly exposed to any component but the adapters. So, it is sufficient to change the corresponding adapter to migrate to another service, without the need of rewriting code of other components. Moreover, if required in some circumstances, it's possible to use different external services: it is sufficient to use the correct adapter in order to access the desired service.

- **MapManager**: it is the component that takes requests from other components, and redirects them to the chosen external **Map Service**.
- **NotificationComponent**: since each mobile *OS* uses its own notification service, this adapter takes in requests of notification sending, and redirects them to the correct service; the latter will take care of delivering the notification to the customer.
- **MailComponent**: an adapter used to send emails; as the previous ones, allows to change the used mail service painlessly.
- **DataManager**: probably, the most important adapter of the system, since it is the one connecting the other components to the system's data logic.

Here, there is described the adapter for *DBMS*, since it's the most articulated among those.

2.2.1.1 Data Manager Component The Data Manager is the component that manages the requests for data manipulation. The component handles two aspects:

- Data Manipulation
- Query Execution

The components' services are requested through the *DataInterface*, while it access to the *DBMS* through its *API*s. Data can be accessed or written to the *DBMS*. This partition is very important because it allows to separate the query for obtaining data from the ones to manipulate the databse.

- **DataRequestor**: it is the component that manages the data reading. Indeed if *DataInterface* asks for a data, this component allows to satisfy this request.
- **DataStorer**: it is the component that manages the data writing. Indeed if *DataInterface* asks to manipulate the *DBMS*, this component allows to satisfy this request.
- **QueryExecutor**: it is the component that manages the connection with the *DBMS*. It uses the *DBMSAPI* and both *DataRequestor* and *DataStorer* rely on him to interface directly with the database.

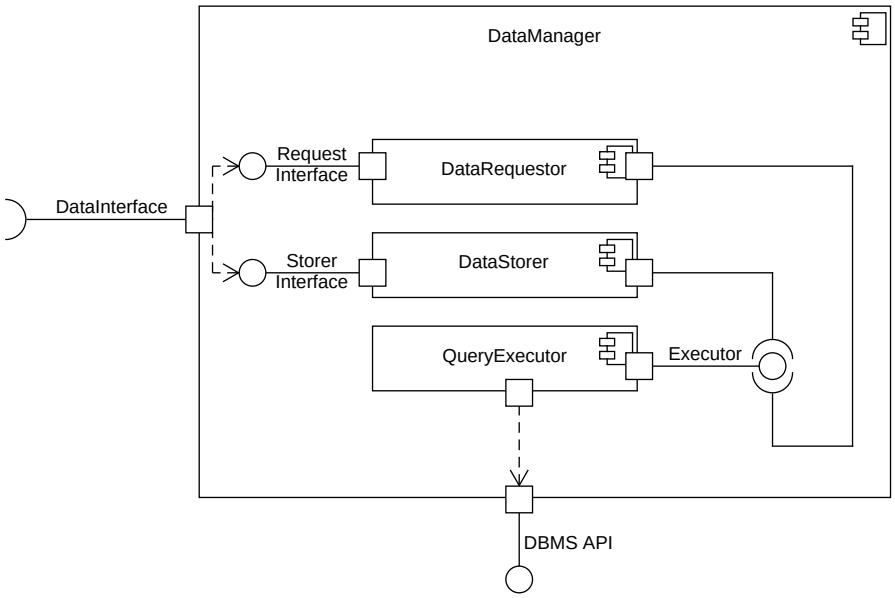


Figure 3: *Data Manager Component View*

2.2.2 CLup Server

As said before, it manages customers' requests and access operations, through the two interfaces it expose:

- **CustomerInterface**: expose all the methods relative to services a customer can access, such as making a reservation and retrieving already made reservations.
- **AccessManager**: the interface designated to manage login and registration of users. After a login, will send to the user the correct set of operations the user can do, depending on their role: customers or store managers.

In the following paragraphs, there is a description of the most relevant component contained in **CLup Server**.

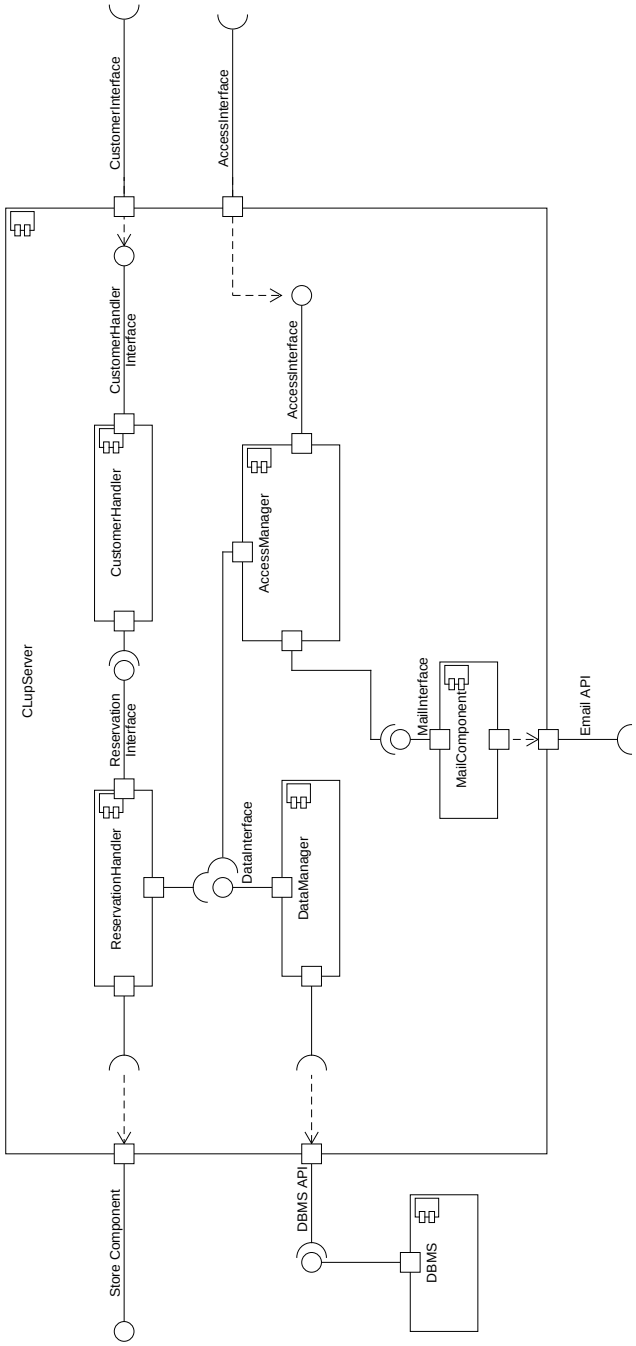


Figure 4: *Server Component View*

2.2.2.1 Customer Handler Component The CustomerHandler is the component that manages all the actions that a customer can make within the application. The component analyzes the aspects concerning reservations.

This component is used by invoking methods on the *CustomerHandlerInterface*. This component is divided in some subcomponents:

- **ASAPManager**: it is the component that manages the *ASAP* reservations and it is formed from two sub-component: **DeleteRes** and **AddRes**. The first one deals with deleting an *ASAP* reservation while the second one deals with creating a new *ASAP* reservation.
- **BookingManager**: it is the component that manages the booking via app and it is formed from three sub-component: **DeleteRes**, **ModifyRes** and **AddRes**. The first one deals with deleting a reservation, the second one deals with modifying a reservation while the third one deals with adding new reservations.
- **RetrieveReservation**: it is the component that is used to retrieve and update all user's reservations.

All of these components uses the *ReservationInterface* to complete their tasks.

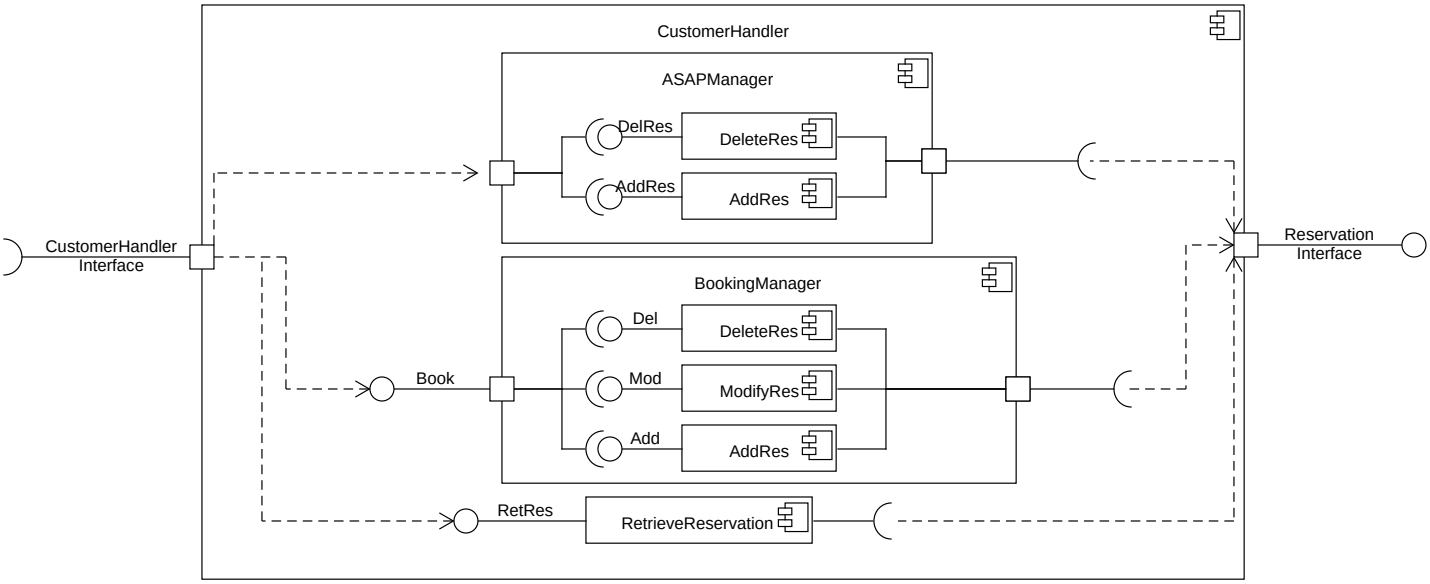


Figure 5: *Customer Handler Component View*

2.2.2.2 Access Manager Component The Access Manager is the component that manages all the issues concerning the access of the users. Indeed, as mentioned above, it is very important to recognize the role of the users in the app in order to provide them correct information, and only the functionalities/-data they are authorized to use. The component analyzes two main aspects:

- Login
- Registration

The component uses two very important interfaces: *EmailAPI* and *DataInterface*. The first one is used for the registration, in order to send confirmation messages, while the second one is used for storing registration data, and to check credentials for log-in operations. Furthermore, its services are requested through the *AccessInterface*, that implements the interfaces exposed by **Login Component**, and **Registration Component**. This component is invoked by *StoreManagerAccessInterface* and *CustomerAccessInterface* in case an user wants to log in or register

- **LoginManager**: it is the component that manages user's login. Indeed, if a user logs in, the component checks if the credentials are correct and then allows the user to use their personal area.
- **RegistrationManager**: it is the component that manages the user's registration. Indeed, if a user makes a registration, this component, at the beginning, checks through *DataInterface* if the data inserted by the user are valid (e.g. uniqueness of the email, or certification for store managers). Then, if the data are correct, the user's data are stored in the *DBMS*.
  - **RegistrationChecker**: it is the component that checks if the information is inserted during the registration, from the user, are valid. In case of customers, this component checks if the email is valid while in case of store manager checks if the store certification is valid and the registration *ID* is unique.
  - **RegistrationStorer**: it is the component that stores the new data in the *DBMS* after checking.

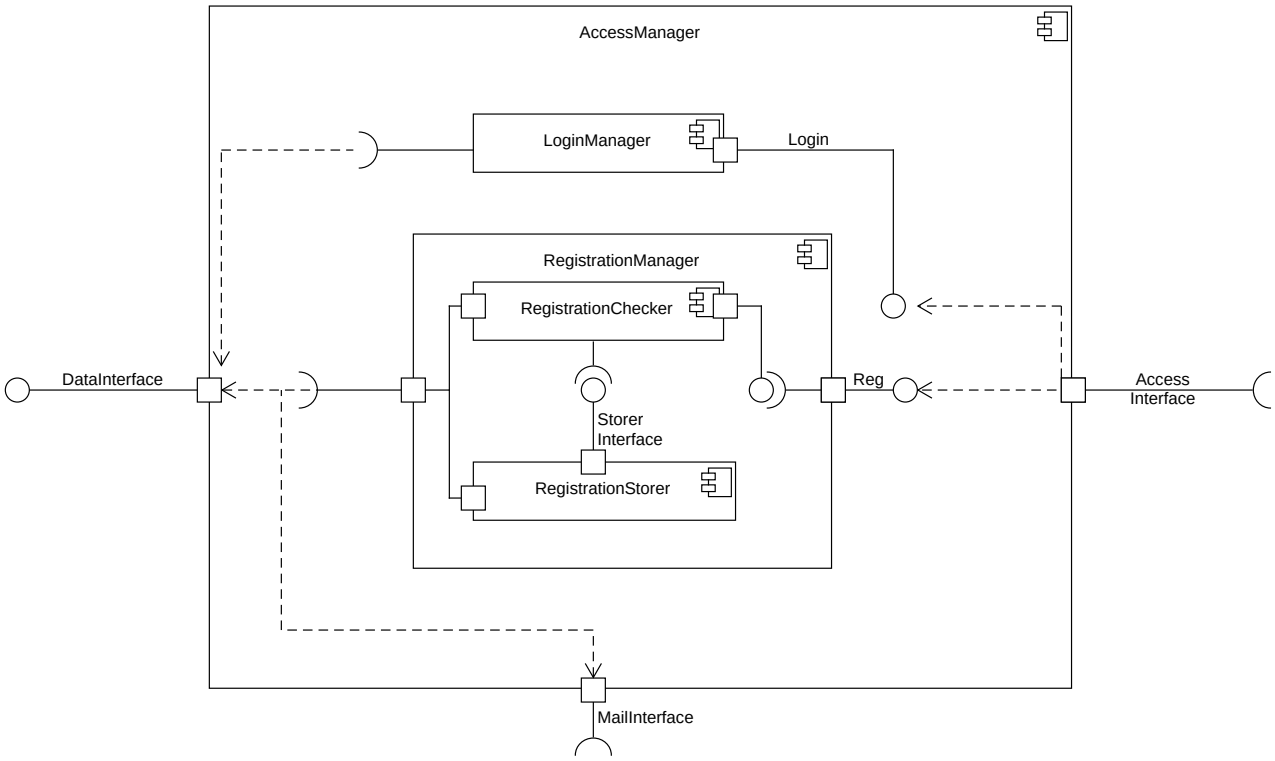


Figure 6: *Access Manager Component View*

2.2.2.3 Reservation Handler Component The Reservation Handler is the component that manages the customer's reservations. The component analyzes two main aspects:

- Suggestions
- Reservation Management

The component uses *ReservationInterface* (that implements *QueryInterface* and *SuggestionInterface*) to accept requests, and **Store Component**, used to make requests to a specific store. This component is invoked when the customer tries to make or modify a reservation, or tries to retrieve his ones. Each time a reservation is retrieved, the customer will receive the updated information (such us *ETA* to enter the store). Moreover, it uses the *Data Interface* to access to CLups' general DBMS, in order to retrieve informations on stores.

- **SuggestionComponent**: it is the component that manages the suggestions that are made available to the customer at the time of reservation.

It will query each store registered on the platform, to retrieve availability information, depending on the preferences inserted by the customer and the type of reservation they are making (so, the component will use the right **Query Component**); it's invoked by both the *ReservationInterface* when a customer asks for suggestions on bookings, or by **ASAPQueryComponent** while the *ETA* for entering the store *ASAP* is too high.

- **StoreQueryComponent**: it is the component that manages the queries that are made at a particular store, indeed this component uses the **Store Component**. The queries can be of two types:
  - **ASAPQueryComponent**: it is the component that manages the *ASAP* reservation, asks the server for waiting times for accessing the store, notifies the store if a reservation is made and manages the deletion of *ASAP* reservations.
  - **BookingQueryComponent**: it is the component that manages the booking via app, asking the server time slots to enter the store, in case a reservation is confirmed, notify the store to save it, and manages all the requests about modification and cancellation of booking reservations.

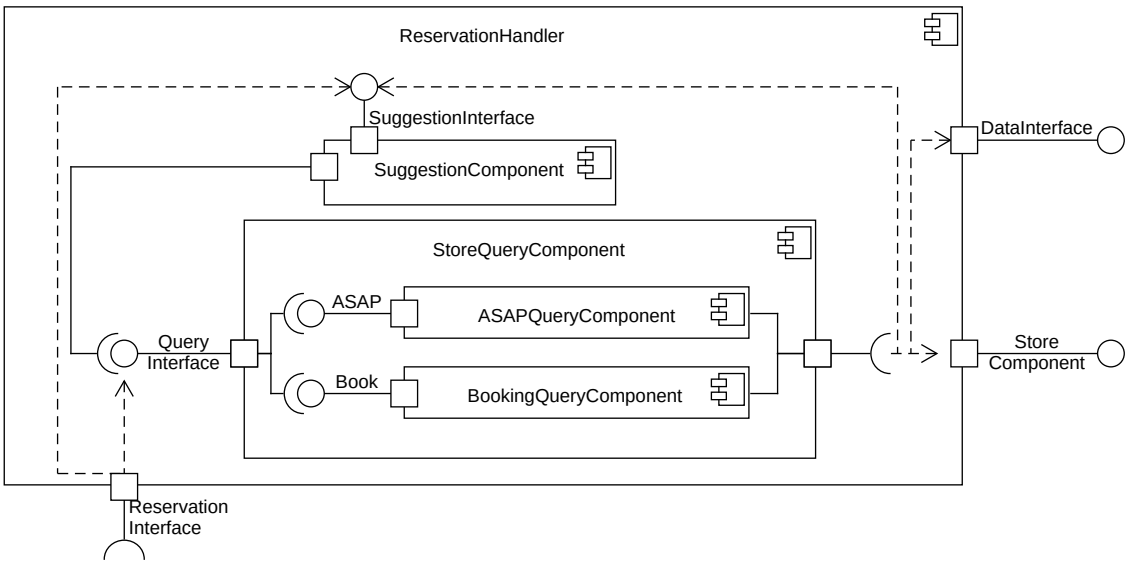


Figure 7: *Reservation Handler Component View*

2.2.3 Store Component

The Store is the component that manages the store from all points of view and it is the most important component of the whole architecture, since implements the whole business tier. The component analyzes many important aspects, but the most important are:

- Reservation
- Admit Reservations Inside
- Store Parameters
- Statistical Values

As said before, it's been preferred to separate the logic governing each store by designing a component for each different store. Also here, the store component have been designed in a modular way, to separate the different features offered by a single store; these are:

- **FlowComponent**: this component manages the functionalities that are requested when a *QR Code* is scanned at the store entry, and uses the **Calling Number Component** to check the scanned QR Code status, the **Department Component** to register the entrance/exit and the **StatisticComponent** to update visit statistics.
- **StatisticComponent**: it is the component dedicated to building customer statistics. It's used by:
  - **EntranceComponent**: in order to update visiting statistics at customers' exit
  - **ReservationComponent**: invokes the methods when, during a reservation process, it's necessary to infer the estimation of customers' shopping time.
- **RearrangeComponent**: it is the component that rearrange the reservations when a store manager modifies the capacity of some part of the store, or some reservations are changed, and the queue must be reschedules. Interacts with the **Calling Number Component** in order to rearrange the queue. In case of delays or cancellations, the customers will receive a notification.
- **StoreDataManager**: it is the adapter between the store and its dedicated *DBMS*. The choice of dedicating a *DBMS* for each store is due to efficiency and security reasons: if a *DBMS* breaks, it wouldn't affect the whole system, while a store can access only its data.
- **ParameterManager**: it is the component that manages the parameters of the store, the settings and the situation of each department of the store. It's used by:

  - **FlowComponent**: is used to update Departments' situation.
  - **StoreManagerHandler**: is used to update store parameters, and to access the real-time situation of the store.
  - **ReservationComponent**: is used to request store situation in order to make a reservation.

  Moreover, uses **RearrangeReservation** when a parameter is modified, to make coherent the store's status

- **ReservationComponent**: it is the component that is dedicated to the management of reservations. It's used to retrieve reservations made in that store, to retrieve waiting times, and to store customers' reservations. Uses the **Calling Number Component** to notify a modification in reservations list, and it's used by the latter to initialize the queue at store opening. It also handles modifications made by customers and store managers to reservations. Furthermore, a store manager uses it to contact customers about some specific reservation. Because of this, the components interact with **Notification Component** and **Mail Component** to interact with customers. To logically separate the management of the two types of reservations, the component is divided in two subcomponents: **BookingComponent**, and **ASAPComponent**.
- **CallingNumberComponent**: it is the component designated to call in reservations, and to notify customers they should depart for the store. It's divided in two subcomponents, each carrying out different functionalities:
  - **CallableReservationComponent**: checks whenever a reservation in the queue is callable, checking the store affluence, through the **Parameter Manager Component**;
  - **CallNumberComponent**: takes a queue of reservations to be called, and when it's possible, a reservation is called to enter. It's alerted by the **Entrance Component** that somebody exited the store.

The core store functionalities are requested through the *ReservationInterface* (dedicated to customers' features) and the *StoreInterfaces*, dedicated to store managers; their requests pass through an handler, here described.

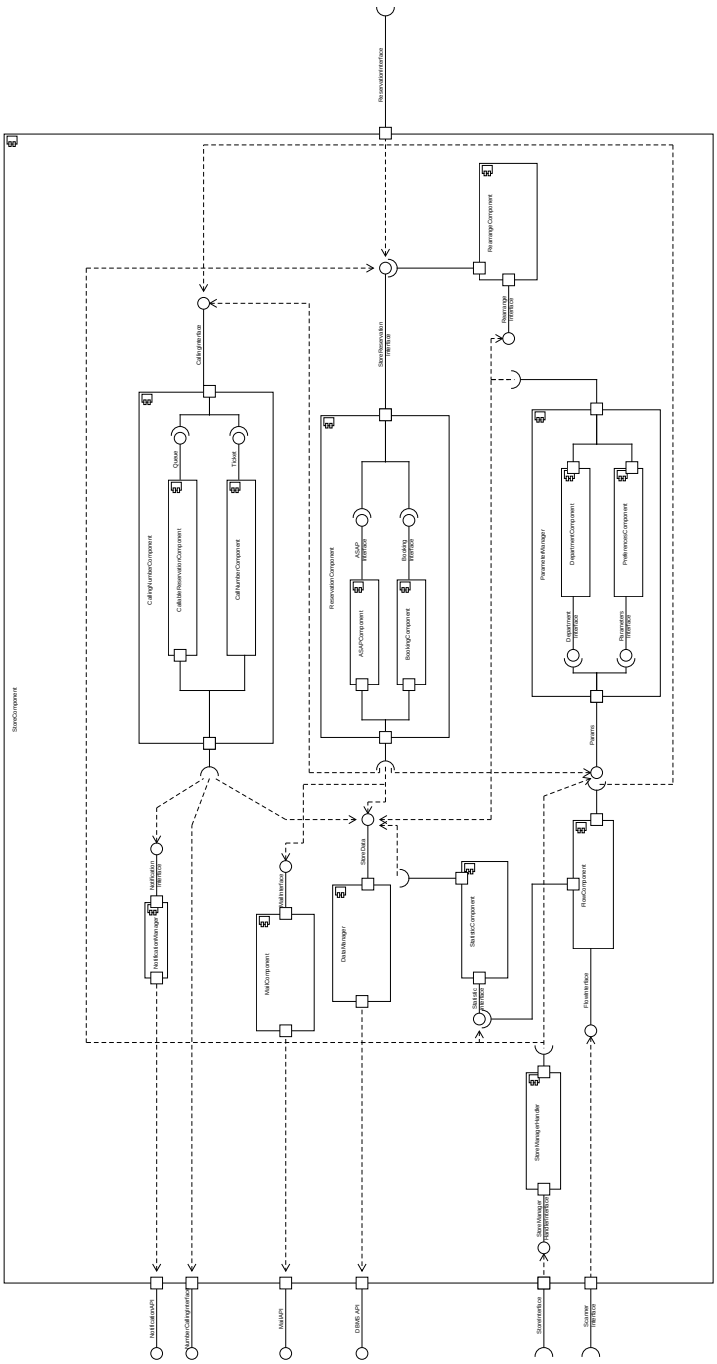


Figure 8: *Store Component View*

2.2.4 Store Manager Handler Component

The Store Manager Handler is the component that manages all the actions that a store manager can make within the application. The component analyzes various aspects:

- Reservations
- Modify Parameters

This component handles all the possible settable options of a store manager, it is invoked on the *StoreInterface*, from the mobile application, and it interfaces with *StoreComponentInterface*, that manages all information regarding parameters and reservations.

- **ManagerReservationHandler**: it is the component that manages the store reservations and it is divided in two sub-component:
  - **DeleteRes**: it is the component that manages the elimination of the reservations.
  - **ModifyRes**: it is the component that manages the modification of the reservations.
- **ModifyParameters**: it is the component that manages the store parameter. They are of two types:
  - **DepartmentManager**: it is the component that manages all the parameters which concern the individual departments. Indeed, the store manager can add or delete a new department and can modify the maximum capacity of each department.
  - **StoreSettingManager**: it is the component that manages all the parameters which concern the entire store. Indeed, the store manager, can change the maximum capacity of the people inside the store and can modify the opening and closing hours.
- **RetrieveReservation**: it is the component that manages all the reservation of a specific store.
- **SituationManager**: it is the component that manages the requests for the real time situation of the store.

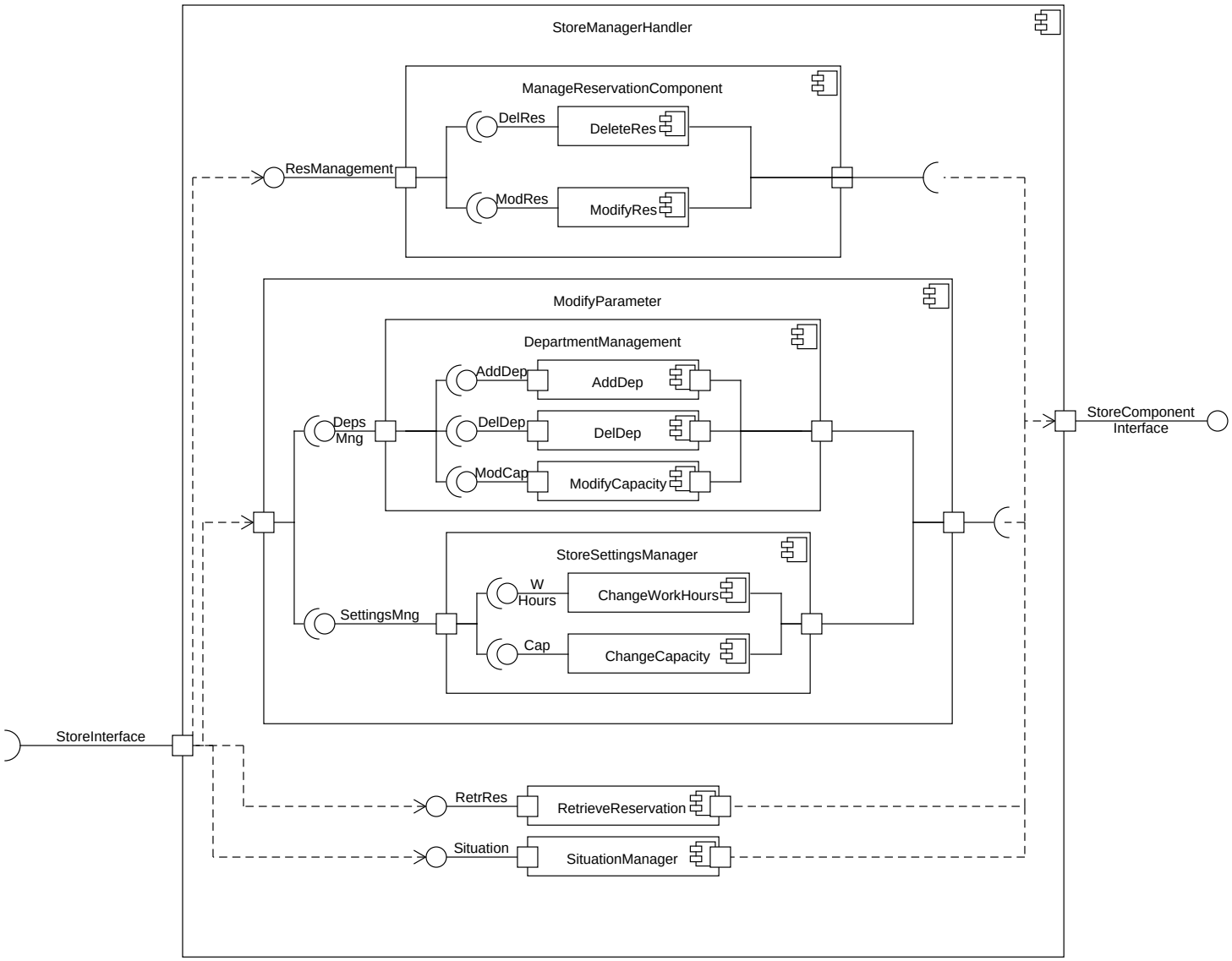


Figure 9: *Store Manager Handler Component View*

2.3 Deployment View

As said in the previous pages, the system have been developed in a very modular way, trying to handle all the required functionalities through dedicated components.
Since the application needs to remotely communicate with various actor, the architecture which best fit for *CLup* purposed is the **client-server** one. Moreover, this architecture has been implemented through the **three-tier** paradigm. This aims to separate the different logic part of the system: **presentation**, **business** and **data**.

2.3.1 Client-side

In *CLup*, there are many actors constituting the system's clients, since they are necessary for many *CLup's* features:

- **Mobile app**: it's the mobile application used by *customers* and *store managers* to request services to CLup. It also represents the **Presentation tier** of the system, developed following the *thin client* paradigm, leaving the whole business logic to the server, but for a thing: the notifications when a customer should depart for the store, and the map related functions. In fact, it's the mobile app that monitors when a it's time to go to the store. Even if a *thin client* shouldn't handle operations, this aims to reduce the work load of the server: it's useless to remotely request the customer's position, and then send it to the *Map Service*, while mobile devices can do it on their own (the same applies for distance and travel time);
- **QR Code Scanner**: it's role is to send server the scanned QR Code to process it;
- **Totem**: a client needed to allow making requests at any store;
- **Number Calling Device**: it's the device used to receive from the server the called reservations, so that a customer in front of the store knows when they can enter.

2.3.2 Server-side

CLup's server side implements both **business logic** and **data logic**.

- **Business-logic**: it's implemented by **CLupServer** and **StoreComponent**. The first is needed to manage users access to *CLup* services, and to manage reservations request, forwarding them to the correct store, or to generate suggestions querying all the store managed by the system. Instead, *StoreComponent* manages the business logic related to each store, including reservations, ticket calling, capacity ecc. This design choice is to separate the logic of each store, and to avoid disservices to the whole CLup system if something may fail.

- **Data-logic**: it's implemented through various **DBMS**s: one for the general server, and one for each store. Business components can access the DBMS though their related DataManager, the adapter between DBMS and other components.

Here, follows a diagram of the subdivision of the various tiers. The clients (implementing the presentation tier) are in red, the components related to the business logic in blue, while the data-related parts are in green.

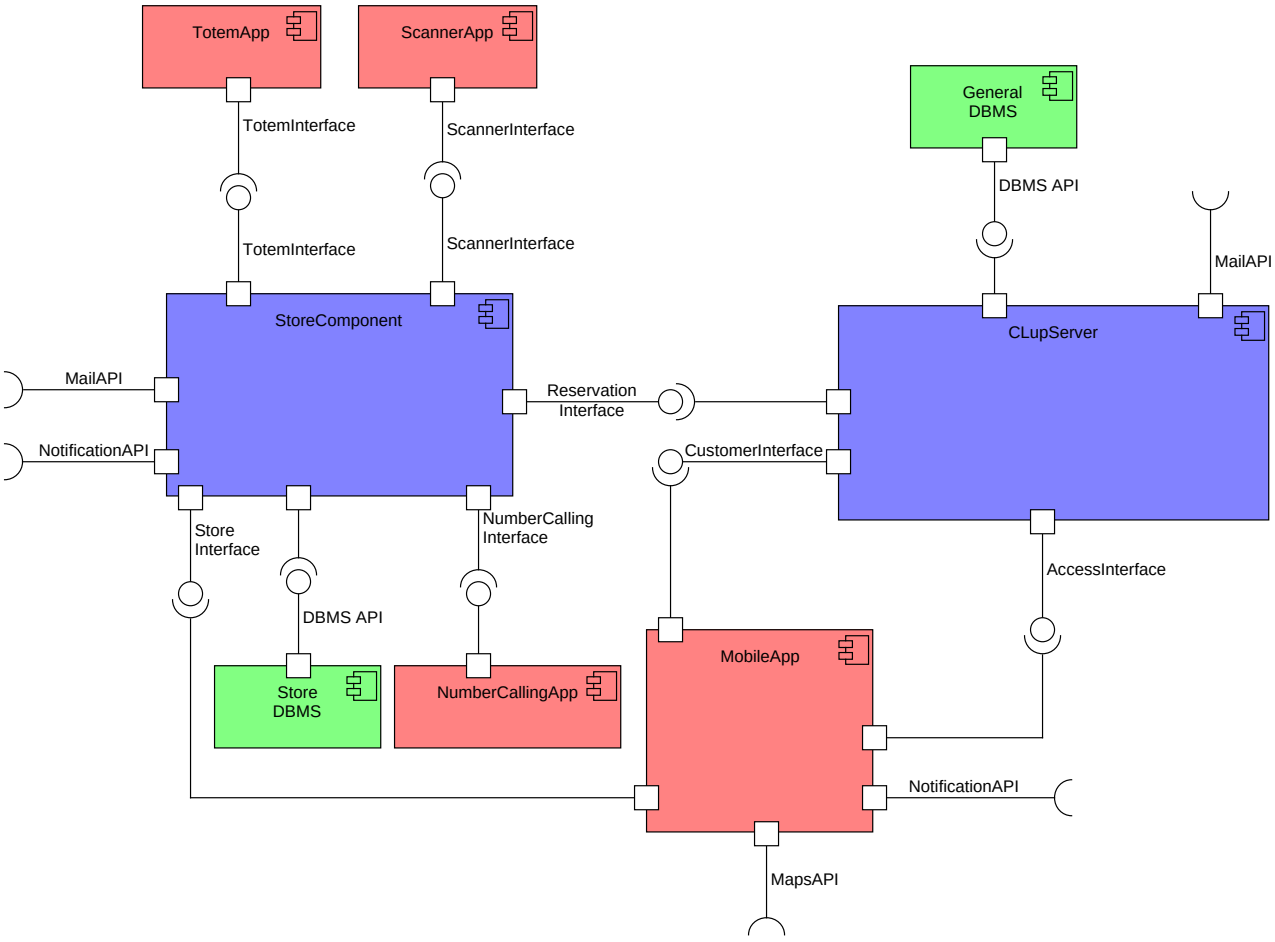


Figure 10: *Component mapping with tiers*

2.3.3 Deployment Diagram

Here there is the deployment diagram of the described system. As described in the *RASD*, all the communications are routed via the *TCP-IP* transmission protocol, using a secure transport protocol (such as *TLS*). For example, *HTTPS* may be used to connect client and server, and to access external services.

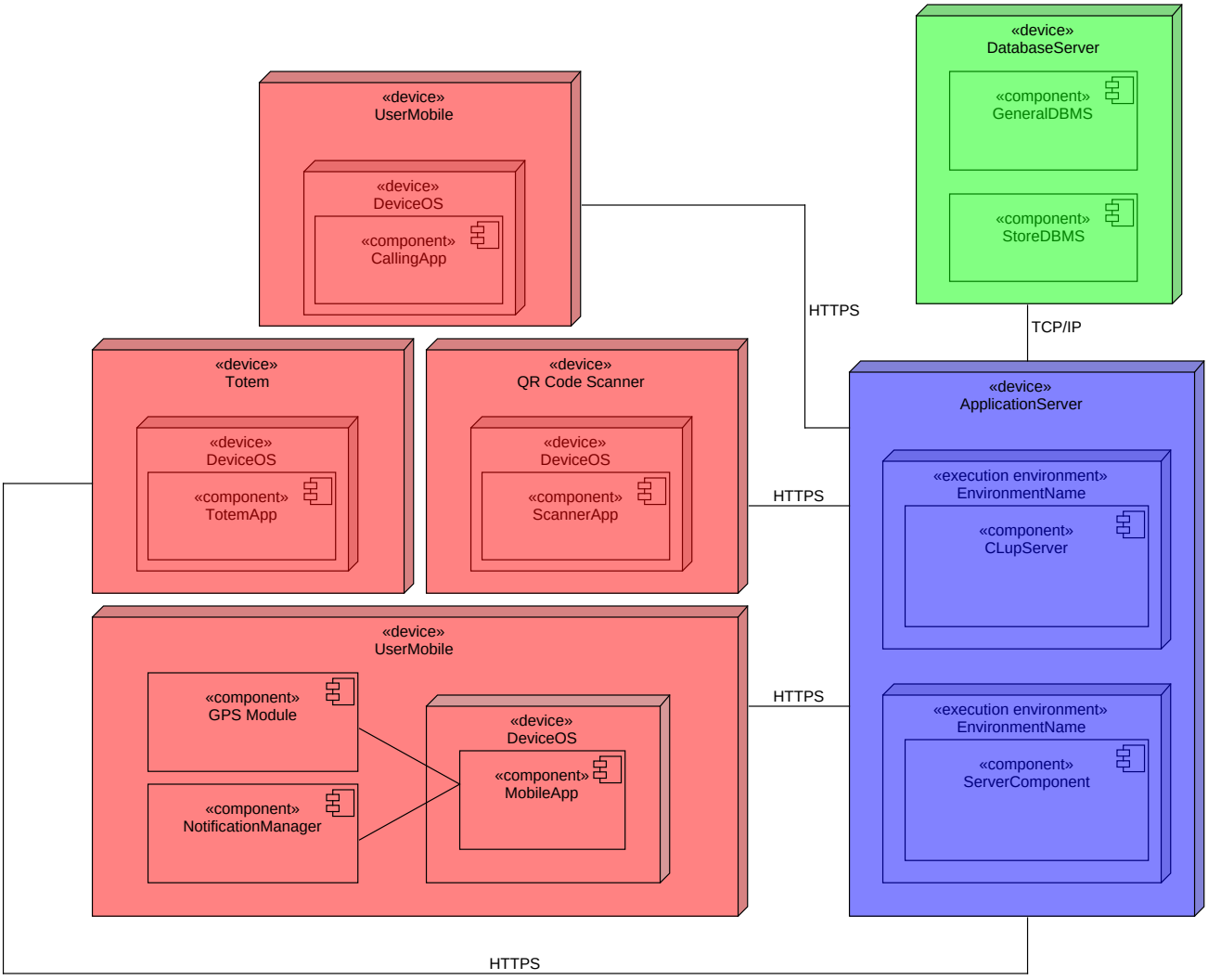


Figure 11: *Deployment Diagram*

2.4 Runtime View

In this section will be analysed the runtime view of the system, through some *Sequence Diagrams*. This should help to understand how the different components interact. Some aspects have been simplified to make the diagrams more readable; e.g., some interactions with the DBMS and between adapters and external services have been omitted, some operations with identical behaviour have been collapsed in generic requests, through a generic method, just to model the interaction. The *Component Interfaces diagram* will report all these methods in detail.

2.4.1 User login

Since the interactions for both *customers* and *managers* log-in are the same, here there is reported only the ones relative to customers.

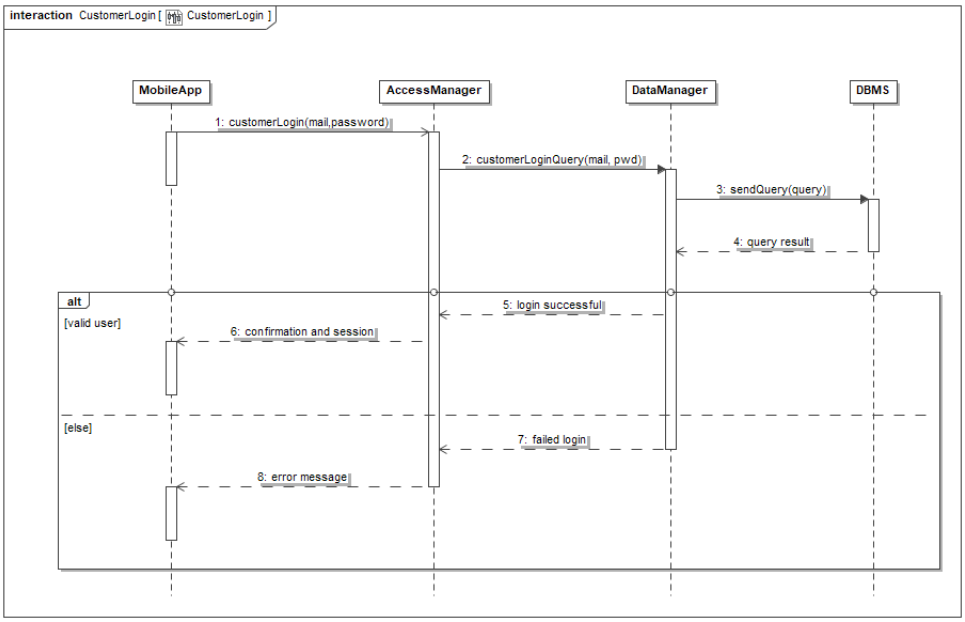


Figure 12: *Customer Login Sequence Diagram*

First of all, the **Mobile App** requests the **Access Manager** to check the credentials. This is done by asking the **Data Manager** to query the *DBMS* for the selected credentials. Once the operations on *DBMS* ended, the **Access Manager** will send an error message if the credentials are not valid, or a confirmation. Then, in the latter case, AccessManager forwards the user's session. If it's a manager to log-in, the **MobileApp** will use the *storeManagerLogin* method, passing as parameter the *StoreID* instead of the *email*.

2.4.2 Customer Registration

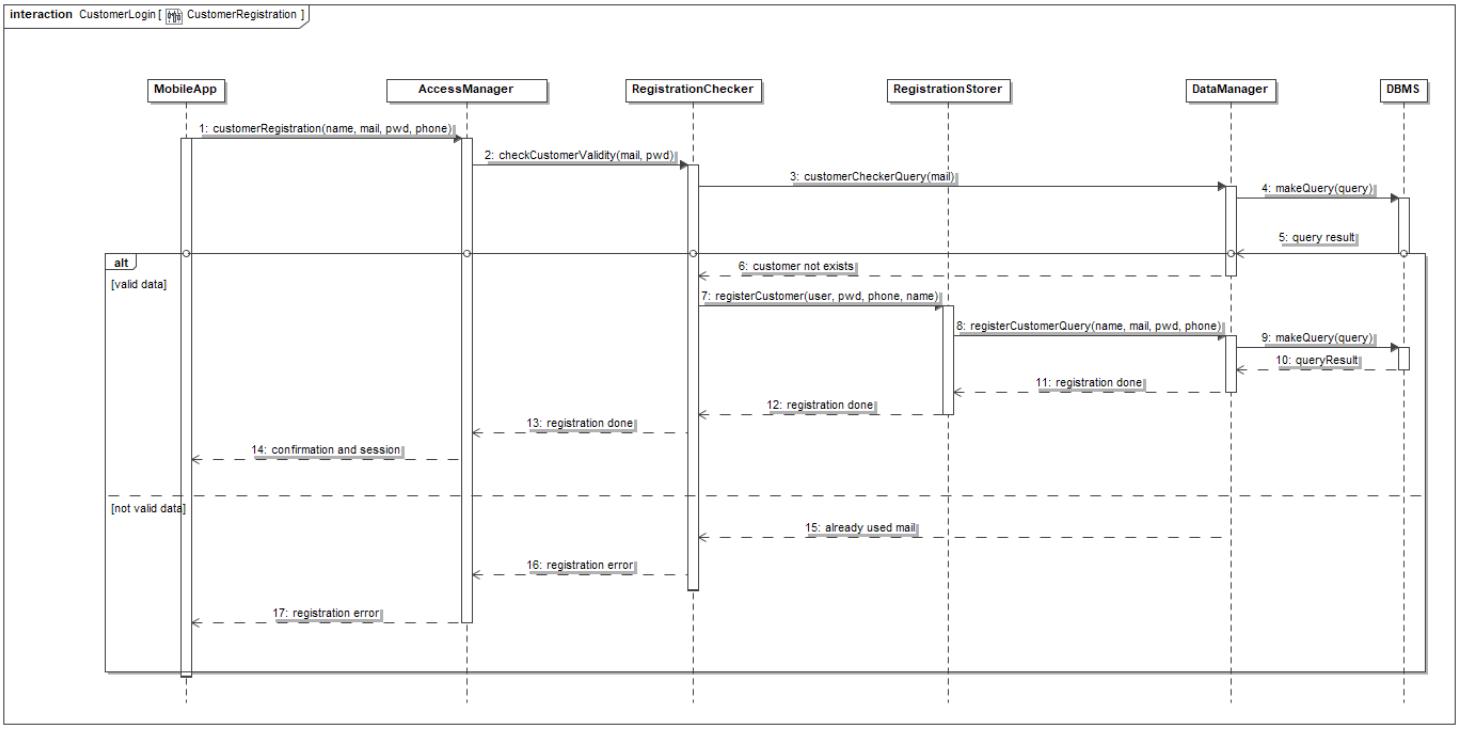


Figure 13: *Customer Registration Sequence Diagram*

The registration process for a customer is quite similar to the login one. In fact, the app requests the **AccessManager** to register the customer with the inserted data. Then a request to check the selected credentials is forwarded from the **RegistrationChecker** to the **DataManager** . If the selected email isn't already in the system, the registration process continues, and the user is stored in the DBMS; at the end, the *MobileApp* receives the confirmation and the session. If the credentials are not valid, an error message is sent.

2.4.3 Store Manager Registration

The registration process for a *Store Manager* is identical to *customers* one, but for the intermediate step of certification check. In fact, if the credentials are valid, before registering the user, the **RegistrationChecker** verify the validity of the uploaded certification. If valid, the process continue, else an error message is thrown to the **MobileApp**.

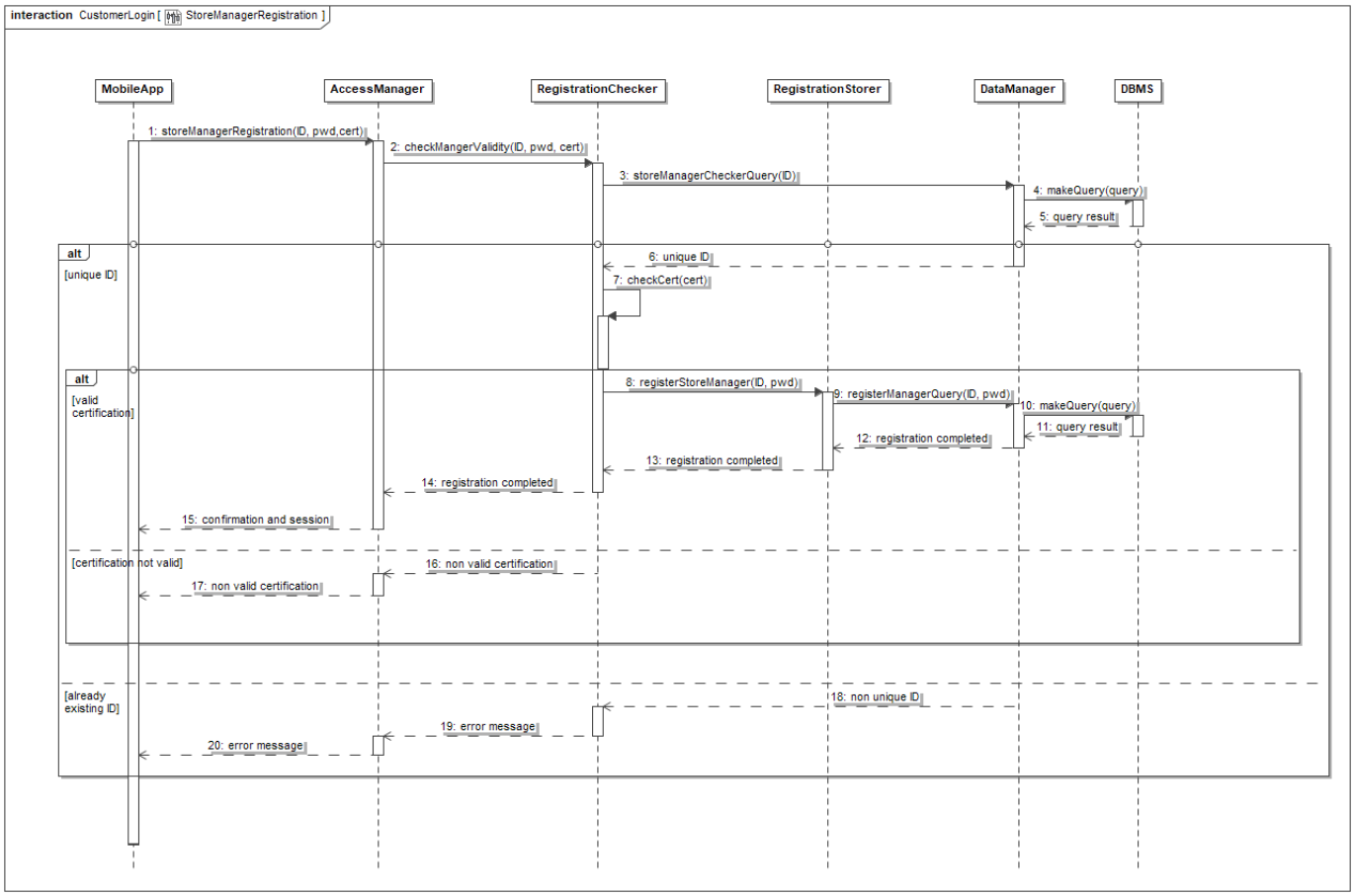


Figure 14: *Store Manager Registration Sequence Diagram*

2.4.4 Make a reservation

To make a *reservation*, it's necessary to retrieve the list of stores. So, through the **CustomerHandler**, the **MobileApp** requests to the **ReservationHandler** the list of stores. Once retrieved from the *DBMS*, this list is sent to the **MobileApp** and sorted by distance, through the *map service*. Then, after the customer select the store, the departments, and insert the shopping time estimation (here it's not treated the calculation by the CLup for the sake of simplicity), they can both make an *ASAP reservation*, or a *Booking*.

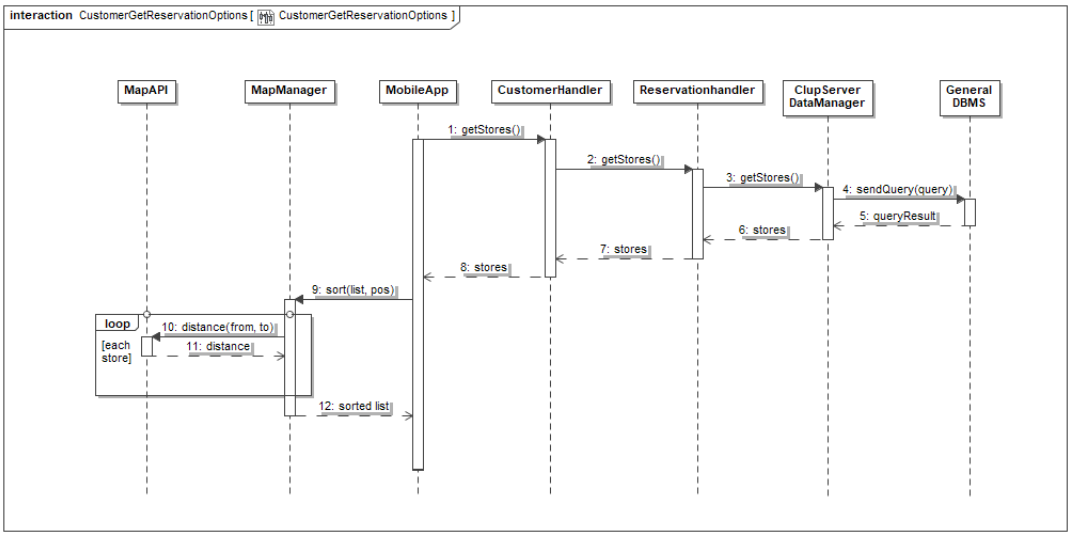


Figure 15: *Reservation Options Sequence Diagram*

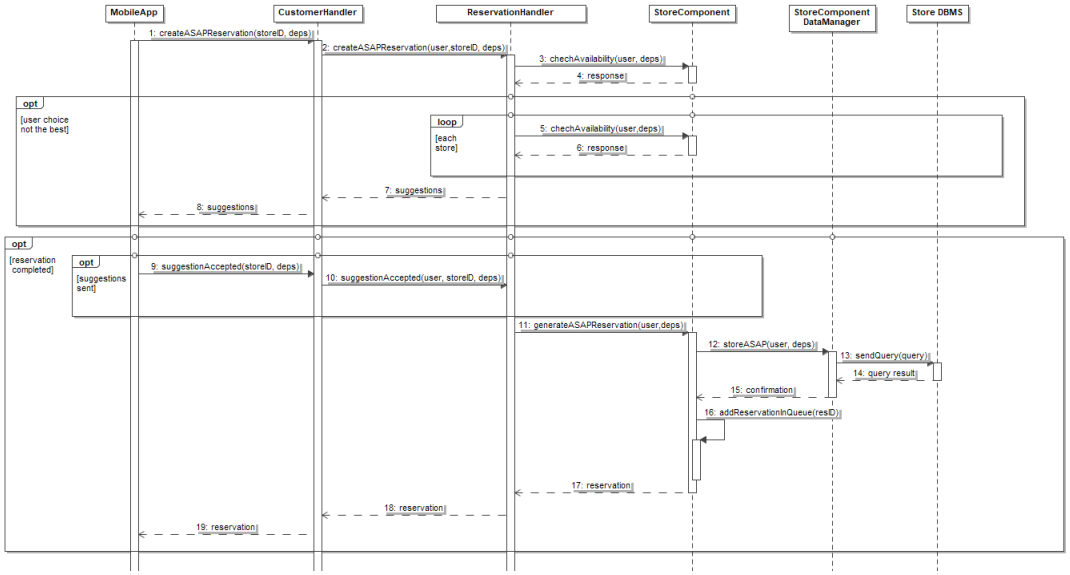


Figure 16: *ASAP Reservation Sequence Diagram*

In the case of an ASAP reservation, a request is sent to the **Reservation-Handler**, that asks to the component of the selected store if the reservation is possible, with the eventual waiting time. In the case the user choice is ok and

optimal, the **Store Component** receives the request to save the reservation, and after queueing it, the ticket is sent to the customer; else, some *suggestions* are generated and sent to the customer, who can choose among them. After an option have been selected, it starts the previously described process.

Instead, if the customer desire to make a *booking*, the **ReservationHandler** asks the selected store's **StoreComponent** when the customer can enter the store, and forward it to the **MobileApp**. Then, the time intervals are shown to the customer. If they're not satisfied with the proposed options, it's possible to ask for suggestion. So, the **ReservationHandler** queries all the stores for time intervals, and send them back to the MobileApp. So, the customer can select a store, and see its time intervals. When they are satisfied, select a time interval and confirm the selection. The request arrives at the **StoreComponent**, that, if the selection is correct, creates the *reservation*, puts it in queue and forward to the customer. If the selection is wrong, an error message is sent to the **MobileApp** and the time interval page is reloaded.

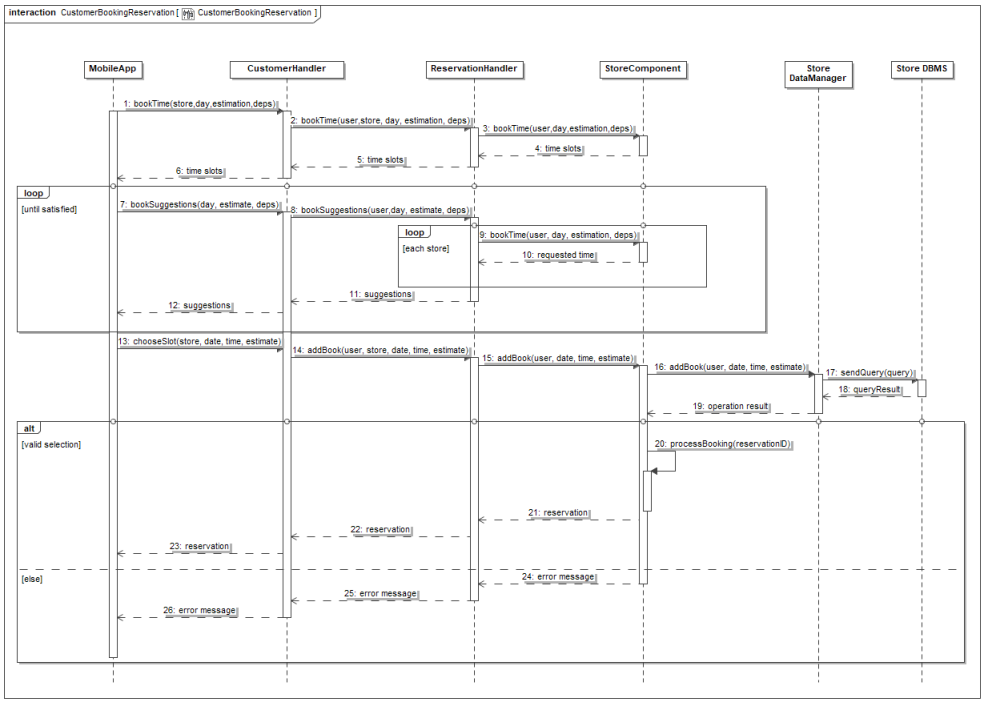


Figure 17: *Book Reservation Sequence Diagram*

2.4.5 Customer Reservations Handling

CLup allows customers to retrieve and manage their reservations from the *Mobile App*. When a reservation is opened from the personal area on the app, its details (such as ETA to enter the store and the needed time to get physically at the building) are updated.

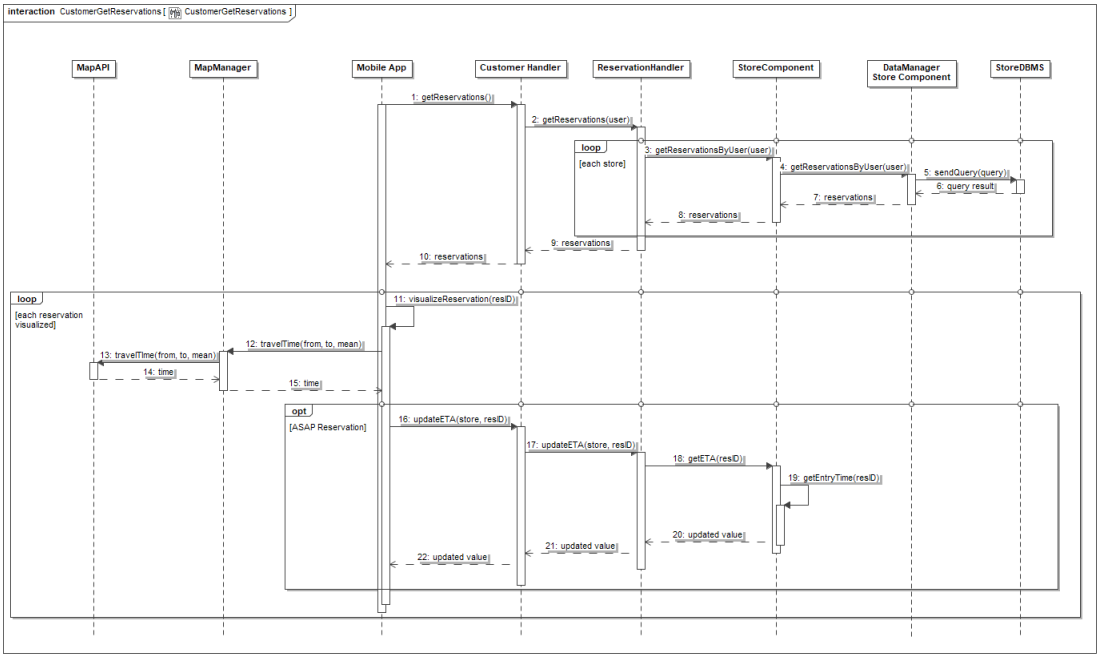


Figure 18: *Get Reservations Sequence Diagram*

When the customer open the app's area reserved for made reservations, and there is an internet connection, the Mobile App requests the server for all the reservations made by the logged customer; the process is mainly handled by the Reservation Handler, that queries all the Stores for the reservations. Once this process is completed, the customer have a list of reservations. When someone is opened, the app requests the travel time for the store to the MapManager, and, in case of an ASAP reservation, the new ETA to enter the store at the ReservationHandler (which forward the request to the correct store component).

In the following diagram there are modelled the interactions occurring when a reservation is modified; the diagram is simplified, since each specific operation (such as delete, reschedule ecc.) are substituted by a generic modifyReservations; that's not a problem, since the interactions are the same for all type of operation. When the related store's **ReservationComponent** receives the modification request, process it, register it in the DBMS and notify the **Call-**

ingNumberComponent of the modification, so that the queue can be accordingly updated.

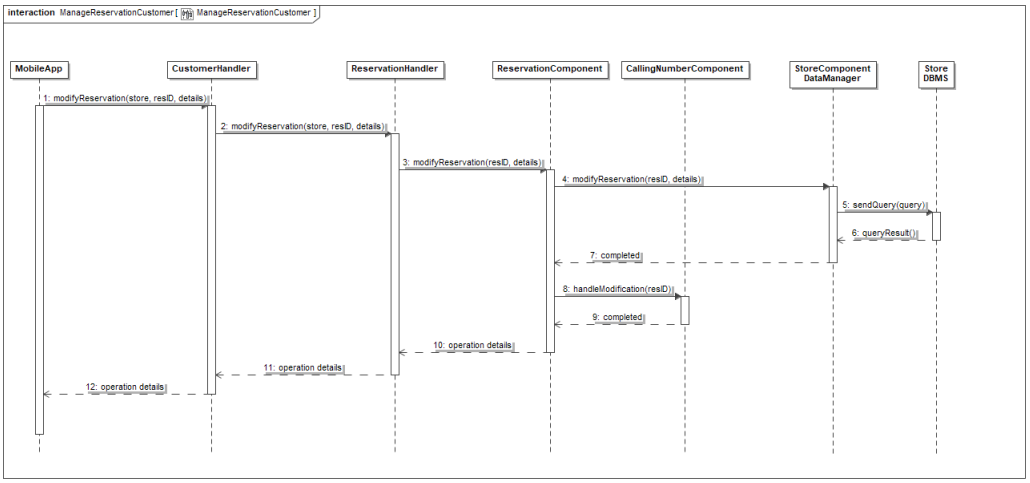


Figure 19: *Reservation Modification Sequence Diagram*

2.4.6 Store Management

As done for the customer side, here there is described how the different components dialogue to allow a *Store Manager* in managing their store.
A note on these sequence diagram: to make them more compact, the interactions with the **StoreManagerHandler** have been avoided, since it's only a redirector from the mobile app to the correct component of the **Store Component**.
First of all, all the reservations must be retrieved from the server. The request is received from the **ReservationComponent**, that queries the *DBMS* for all the made reservations. Once retrieved the store's reservation, a manger can both contact customers about a specific reservation, or modify one.

In the first case, after inserting a message, a mail send is requested to the **StoreComponent**'s **MailManager**.

In case of a modification, the request is sent to the **ReservationComponent**. After the changes are registered in the *DBMS*, the **ReservationComponent** asks the **CallingNumberComponent** to update the store's queue, and then a notification of the changes is sent to the customer.

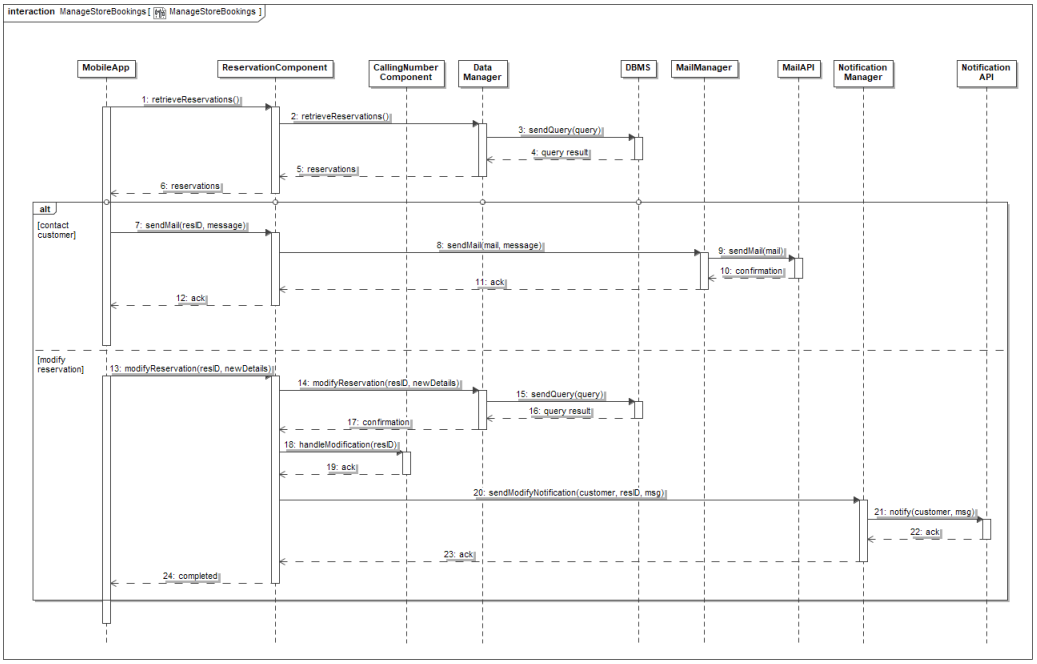


Figure 20: *Store Manager Reservation Handling Sequence Diagram*

A store manager may also see the situation inside their store, among visits statistics. To do this, a request to the **ParameterComponent** is made, that will query the *DBMS* for the actual situation, the **StatisticComponent** for the statistics, and finally send the data to the **MobileApp**, that will elaborate them.

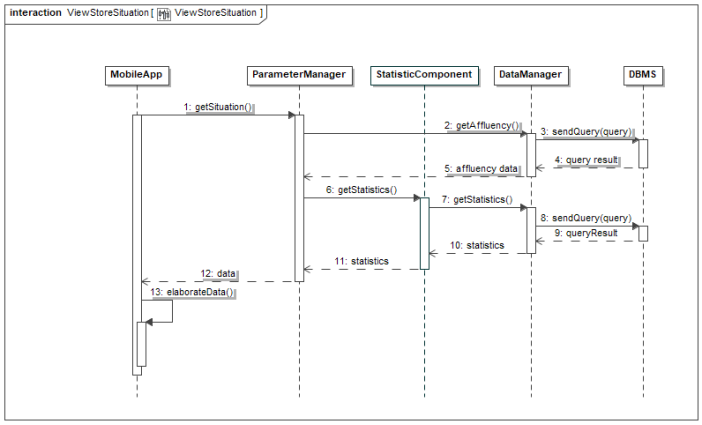


Figure 21: *Monitor Store Situation Sequence Diagram*

2.4.7 CLup Critical Features

In the previous sections, there are described the diagrams related to requests made by customers and store manager. But, CLup also need to handle on its own some processes, needed to manage the store's flow.

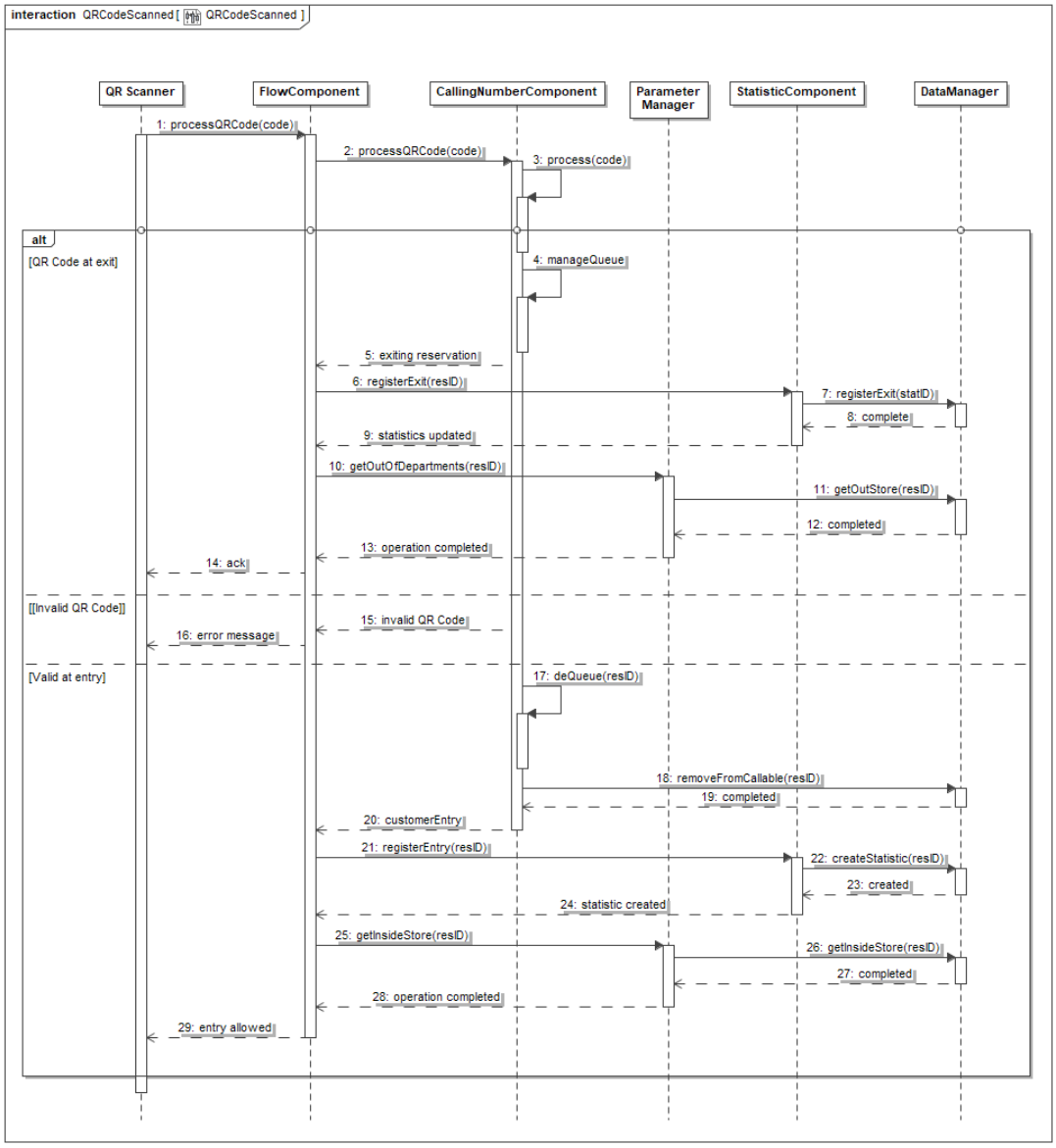


Figure 22: *QR Code processing Sequence Diagram*

In order to register customers' entry and exit, the scanned QR Coded must be processed. When a QR Code is scanned, a request of processing is forwarded to the FlowComponent, that asks the CallingNumberComponent to check the status of the scanned QR Code, and some case can happen:

- **A customer exits the store**: in this case, the visit's statistics are updated through the StatisticComponent (that registers the exit in the DBMS), and then the reservation is removed from the "inside" of the store, through a request to the ParameterManager (that updates the departments' status on the DBMS). But, before returning the QR status to the FlowComponent, the CallingNumberComponent starts the process to call a new number inside the store.
- **Invalid QR Code**: a wrong QR Code is scanned, and an error message sent to the store.
- **Valid QR Code at entry** in this case, the CallingNumberComponent dequeues the reservation and notifies the FlowComponent of the QR status. So, the visit statistics are generated and the customer put "inside the store", through the same components and dual requests used for the exit case. Finally, the authorization for entry is sent to the store.

2.4.8 Call A Number

Here there are described the actions made by the system to handle the process for calling a customer to get inside the store. This process is mainly triggered when the queue is modified in some way, such us someone exits the store, some reservation has been rescheduled or cancelled or someone didn't enter the store in time, and their reservation has been discarded.

First of all, the CallingNumberComponent process the status of the queue, getting the first reservation that has the right to be called. Then, the ParameterManager is asked for the current situation of the store, and the data are sent back. In case there is the possibility to allow the considered customer to enter the store, the reservation is set as called called, this status registered in the DBMS, and the call is notified to the NumberCallingApp, so that outside the store the number can be called. Finally, the system checks if the ETA for some future reservation decreased, and if so sends a notification to the related customer's mobile app, so that the calculation of the departure time is precise. Even if not described in other diagrams, this last process takes place each time a modification in the queue is done.

This process takes runs until the store is full and/or no other reservation can enter the store.

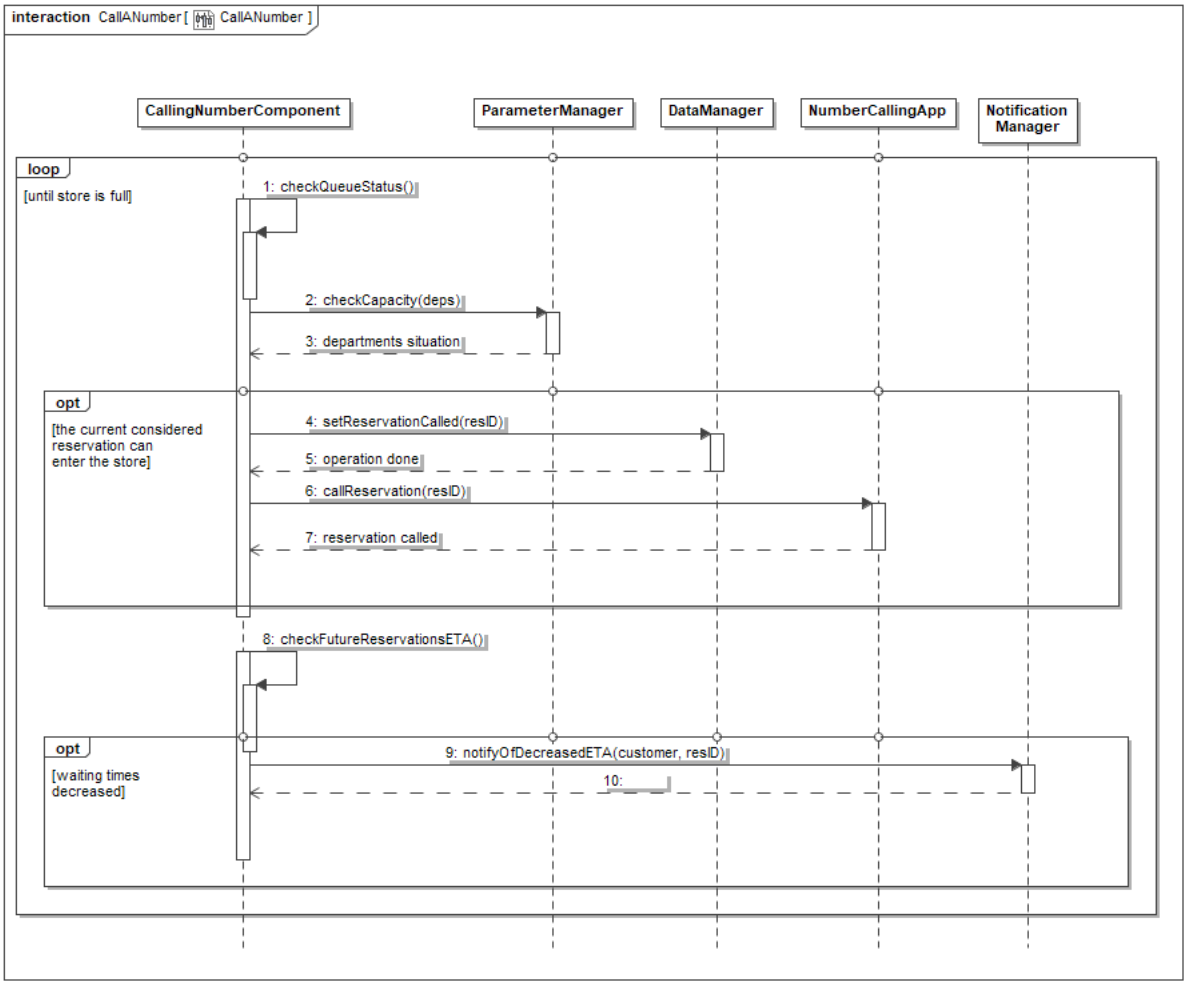


Figure 23: *Calling Number Sequence Diagram*

The last process to be analysed is the notification when a customer should depart to get in time, and not too late to the store. As described before, this operation is entirely handled by the MobileApp: if this is processed by the server, it would be necessary to continuously request to the client the GPS Position, forward it to Map Service to calculate the travel time, and if necessary send to the customer a notification. This may overload the server of useless traffic, since this operation can be handled by the MobileApp, directly accessing to the map external service and the *GPS Module* of the mobile device running it.

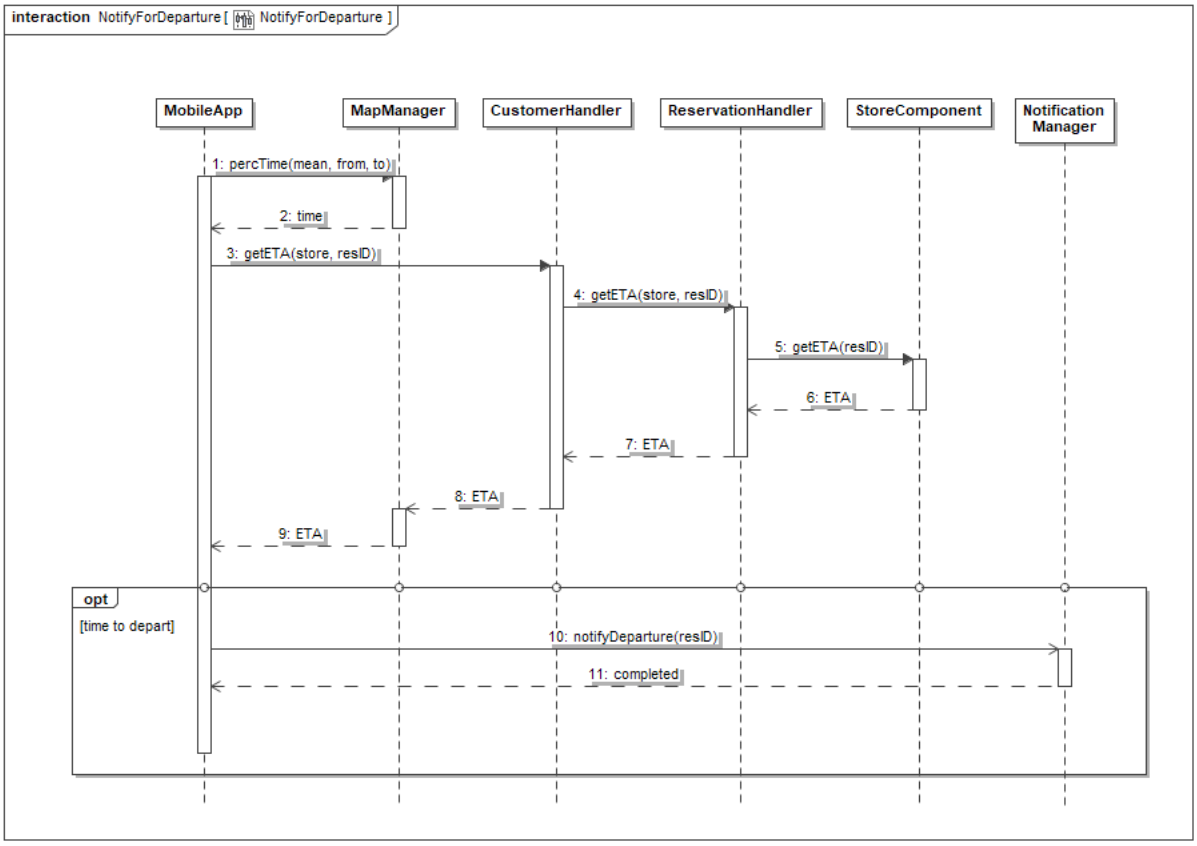


Figure 24: *Calling Number Sequence Diagram*

The process is quite simple: the MobileApp requests to the MapManager the travel time from the current place, to the considered store, and by the mean of transport selected by the customer. Than, the ETA for the call of the reservation is asked to the CLup Server trough the ReservationHandler, that will forward the request to the correct StoreComponent. Finally, when all the data are available, the MobileApp processes them to infer if it's necessary to send an "internal" notification, in order to alert customer it's time to depart. If so, the NotificationManager is invoked, with a request for a *Departure Notification*.

2.5 Component Interfaces

The previous diagram represents the interfaces of the most important and critic components of witch *CLup* is composed. All of these interfaces contains the methods used in the previously described *Sequence Diagrams*, and in such a way shows how the methods used by a component uses other parts of the system to complete its tasks. Not all the interfaces are described here, such as the one

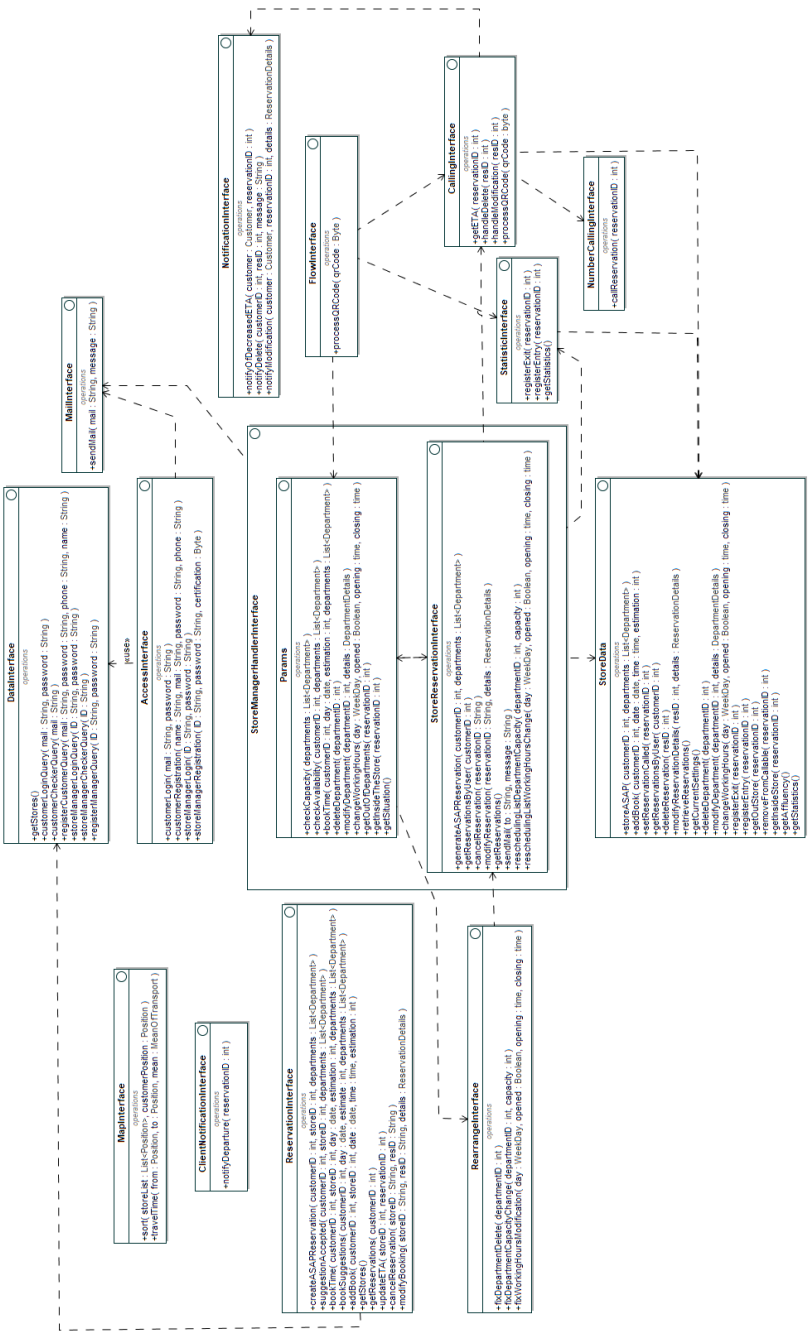


Figure 25: *Component Interfaces Diagram*

related to *CustomerHandler*, since it is an adapter between *presentation Tier* and *Business Tier*, only going to repeat methods already reported.
As *Sequence Diagrams* are simplified, here the methods type and parameters are totally fictitious, and used only to gave an idea of the content transmitted from a component to another.

- **StoreData**: it's exposed by the **DataManager** of each **StoreComponent**, and it's used by all the other component that needs to access to *DBMS* to retrieve data, or to permanently modified informations related to the associated Store.
- **StoreManagerHandlerInterface**: represents all the functionalities that a *Store Manager* can access to using the *MobileApp*. Since the main things that a manager can do is to manage reservations and settings of the store, this interface implements the two interfaces relative to the **ParameterComponent** and to the **ReservationComponent**. The first expose all the methods necessary to manage the settings of the store, and can also be used by other components, that need to access store preferences; the second is related to the management of store's reservation, and expose all the methods necessary to access and manage store's reservation.
- **RearrangeInterface**: all the methods here described are related to the processes needed to put the store in a consistent state after some parameters (such as departments and working hours) are modified in a way that not all the requests can be satisfied.
- **ReservationInterface**: implements all the methods necessary to allow a customer to do and to manage his own reservations. It acts like a forwarder of requests from the general server, where these methods are implemented, to the correct component of the store involved in the request. Moreover, there are functionalities designated to query all the stores in order to give customers suggestions on reservations, both ASAP and Booked.
- **AccessInterface**: all the operations needed to process login and registration features are contained in this interface, that can be directly accessed from the *Mobile App*. It makes use of the **DataManager** to check and store credentials on the *DBMS*.
- **Data Interface**: exposes the methods needed by the components of the general server to access data necessary to do the business tasks related to the system, such as checking and storing credentials in respectively login and registration process, and to store all the stores that are managed by the system.
- **FlowInterface**: defines the methods used by QR Code Scanner App to make requests at the system, precisely to invoke the analysis and processing of a scanned QR Code, so that precise actions are made depending on the given code, and a feedback is sent to the local store.

- **CallingInterface**: contains the methods necessary to communicate and make requests to the component devoted for gradually admit people inside the store. For example these methods allow to rearrange the queue if some reservation is modified, or to get the *ETA* associated to a specific reservation.
- **StatisticInterface**: this interface allow to communicate with the **StatisticComponent** in order to get visits' statistics, to create and update visits' data.
- NumberCallingInterface: specifies methods necessary to notify something using the **NumberCallingApp** that some reservation have been called inside the store.
- **NotificationInterface**, **MailInterface**, **MapInterface** and **ClientNotificationInterface**: the interfaces implemented by the adapters for using the respectively associated service.

2.6 Selected Architectural Styles and Patterns

2.6.1 System Architecture

As briefly introduced before, since *CLup* is a service usable by a great amount of people, and the operations made trough the systems must be considered in each function everyone can access, the architectural style best fitting for the described service is the **client-server** one. In fact, there are clients that make requests to a server, that processes them and reply to clients with the response of each computation.
In implementing the **client-server** architecture many alternatives have been analysed, but at the end the more suitable for *CLup* scope is a **three tier architecture**, that, as previously said and described, logically separates the **presentation**, **business** and **data** tier.
A particular description must be done about the particular **business** and **data** tier here implemented. In fact, in the system there isn't a unique principal business component, and/or a unique data tier. Since each store is independent from the other, and should access only to data of their competence, the business tier is divided so that each store has his own business logic. This helps in various situations, such as greater flexibility in accepting managers requests to implement a certain business logic, maybe different from the one of some other store; in case of dysfunctionality, only the broken store won't work, while the components of the others will work, and it's easier to isolate the broken one and repairing it without touching anything else; each store's **business tier** has an associated and unique **data tier** to access: this allows to separate each store's data, in order to make data consultation faster and more secure, since each store can access only to the associated *DBMS*.
An advantage of this is that each client that makes operation on a single store, can connect directly to the correct **business tier**. Regarding customers,

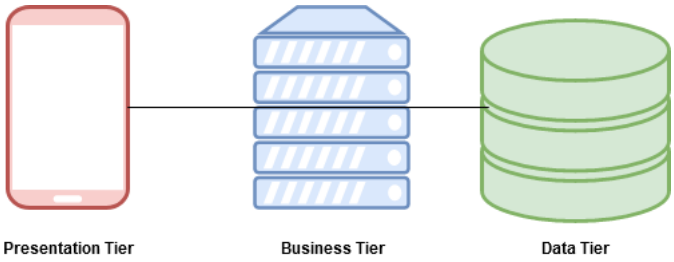


Figure 26: *The Three Tier Architecture*

instead, they connect to a dedicated **business tier** with its own **data tier**; it's scope it's to redirect requests from customers to a specific store, and to generate suggestions querying all the possible stores, which informations are stored in the general *DBMS*.

2.6.2 Used Patterns

In this section will be described some of the *design patterns* used in the development of *CLup*.

- **Pattern Adapter**: the adapter pattern have been used in multiple part *CLup*. For example, each external service is accessed though an adapter that takes requests, and send them to the requires service. This help to easily implement multiple external service or to change them without a great effort. Moreover, also the **CLup server** is an adapter, since takes in requests from the *mobile app* and redirects them to the correct **StoreComponent**.
- **Model View Controller**: this design pattern is used in each client developed for the system.
- **Facade Pattern**: used to allow the **StoreManagerHandler** to access more easily to multiple components trough the different interfaces they implement.

3 User Interface Design

This section shows and describes accurately the mockups of the application, in order to explain how the main functionalities will be offered. The first subsection is dedicated to the illustration of the application flow, shown using two different UX diagram, one for the customer and one for the store manager.

3.1 UX Diagrams

The following figures shows the application flow from the customer and the store manager point of view.

3.1.1 Customer Flow

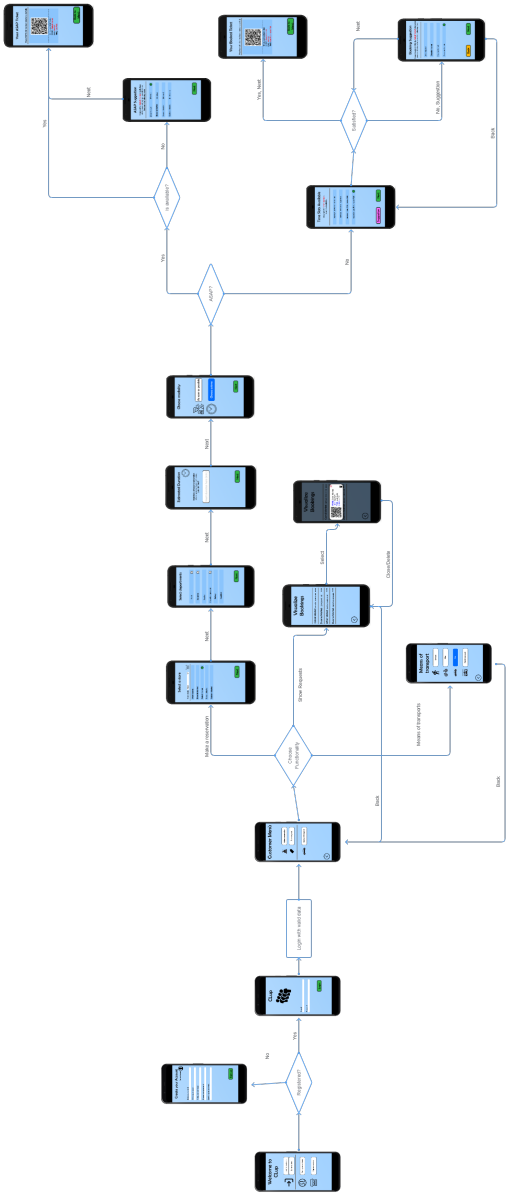
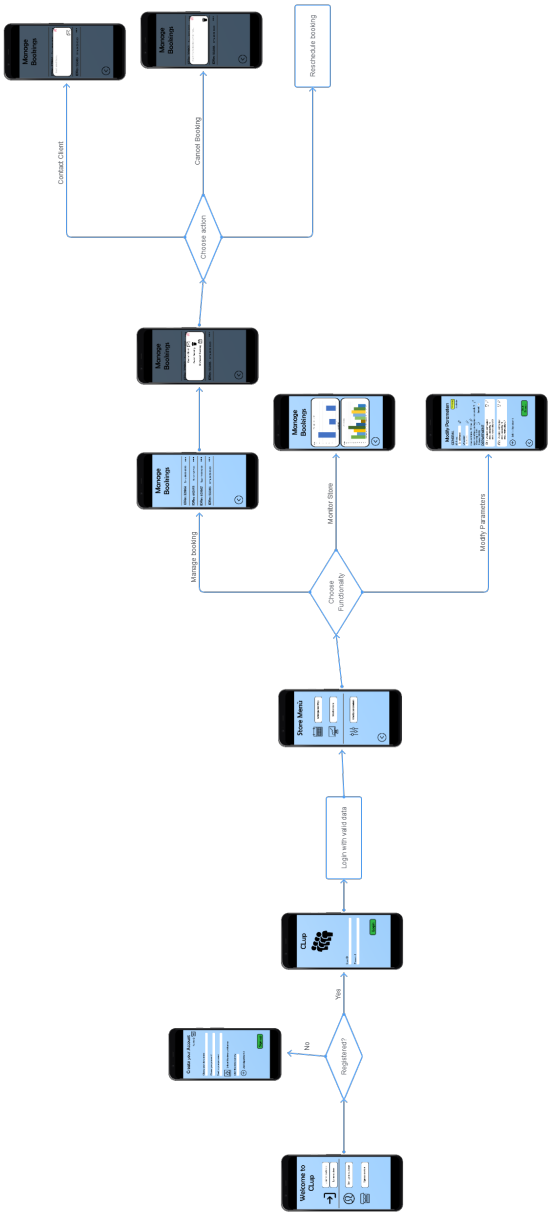


Figure 27: *Customer Flow*

3.1.2 Store Manager Flow

3.2 User Mobile App

In this paragraph are shown mockups that are used to define which interfaces are present within the application. All interfaces are shown and accompanied by a description explaining their functionality. The **Mobile App** version is truly intuitive and can be used without problems by everyone, customers or store managers. The application is flanked by totems placed at the entrance of each store, in case someone does not have the application or does not have a telephone and they only have one type of reservation, that is the *ASAP* one, in the shop they belong to.


Manage Bookings
Reschedule booking
Contact Client
Cancel Booking
Choose action
Manage Bookings
Manage Bookings
Manage Bookings
Monitor Store
Modify Parameters
Manage booking
Choose Functionality
Store Menu
Login with valid data
Club
Yes
No
Registered?
Welcome to Club

3.2.1 Home Page

In this section is shown the initial page that a user finds when opening the app. From this section the user can make a new registration, if is not registered yet, or can sign-in in order to use the functionalities that the app offers.

Figure 29: *Home page*

3.2.2 Registration

In this section are shown the mockups about the registration of an user. In fact an user can make a registration as a customer or as a store, based on the role it must have inside the app.

- If an user decides to create a customer account have to enter some parameter (email, name, password) in order to complete the registration.
- If an user decided to create a store manager account have to enter a valid store certification and a password.

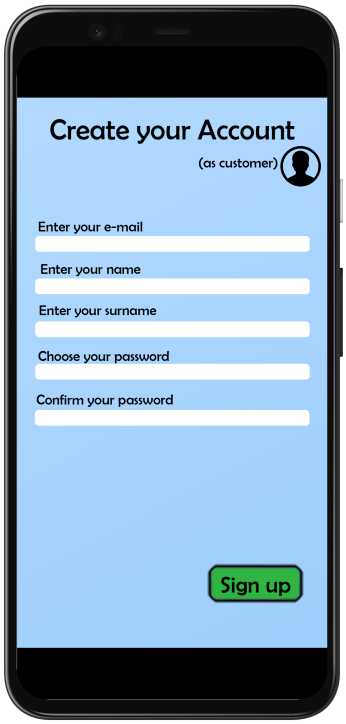


Figure 30: *Sign up as customer*

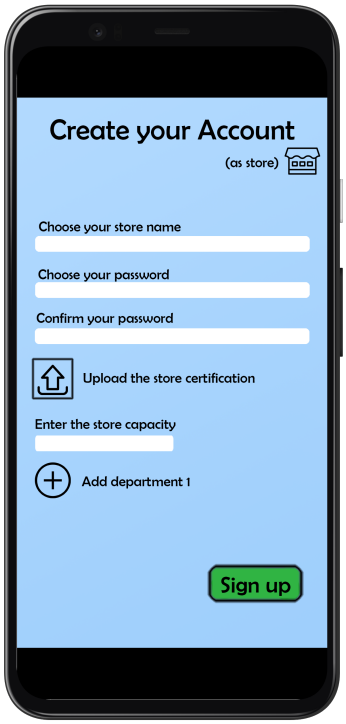


Figure 31: *Sign up as store manager*

3.2.3 Login

In this section are shown the mockups about the login of an user. In fact an user can perform two types of login based on the type of registration made at a previous time.

- If an user has registered as customer have to put email and password in the respective fields.
- If an user has registered as store manager have to put Store *ID* and password in the respective fields.

Figure 32: *Login as customer*

Figure 33: *Login as a store*

3.2.4 Menù

In this section are shown the mockups about main menù. After the login the users have a main menù that can be of two type, based on the type of the account: customer or store manager.

- In case of the customer, the menù allows to start a new reservation, show current requests and select the mean of transport.
- In case of the store manager, the menù allows to manage all reservations, see the store statistic and modify all store parameters.

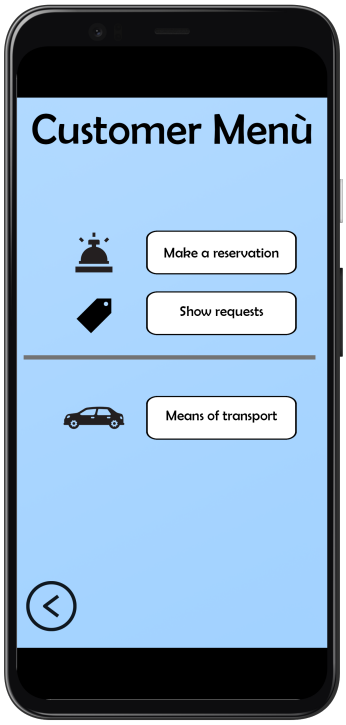


Figure 34: *Customer menù*

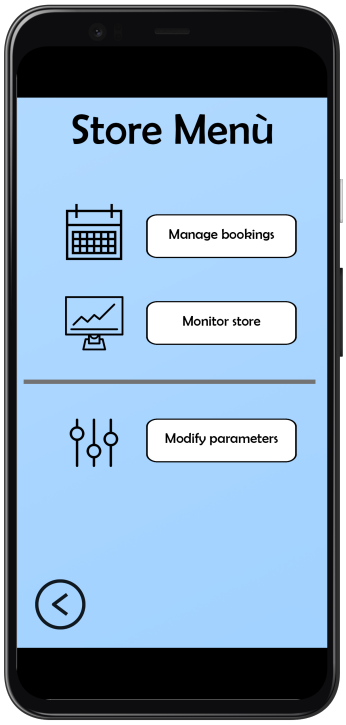


Figure 35: *Store manager menù*

3.2.5 Make a reservation

In this section are shown the mockups about the booking of a reservation. When a customer wants to make a reservation have to select the store where he wants to make his purchases. The store, thanks to *Maps API*, are ordered in ascending order by distance. After the selection, the customer can selected the products that he wants to purchase or, at least, the departments that he intends to visit during his purchases. After this first phase, the customer can insert

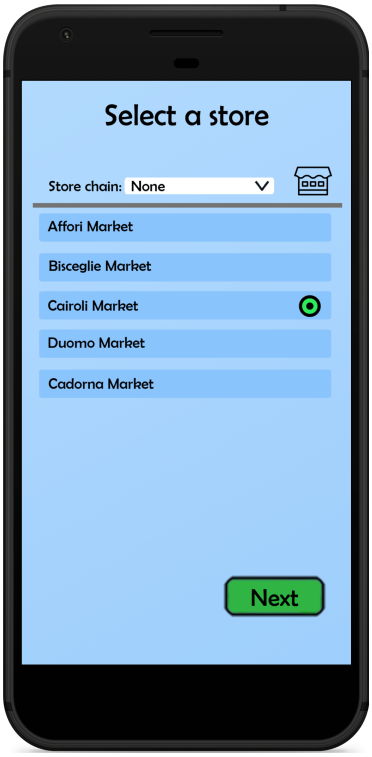


Figure 36: *Store selection*

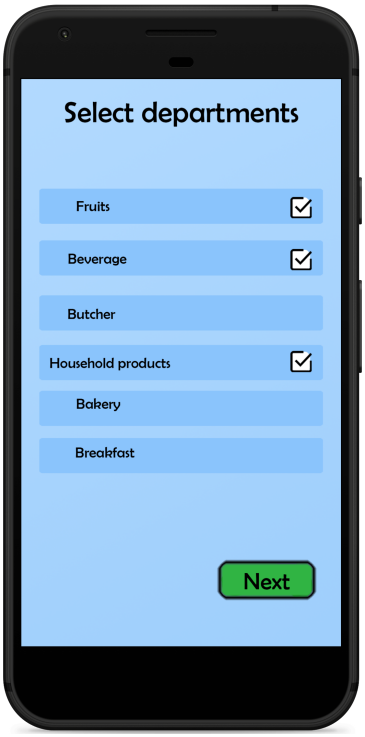


Figure 37: *Departments selection*

the estimation time of the duration of his expense. In case this parameter is not inserted by the customer, the system will calculate a reasonable estimation time with two criteria, based on the situation:

- In case of the system have sufficient data about customer, it provided to calculate the estimation time based on customer's previous expenses.
- In case of the system does not have sufficient data about customer, it provides to calculate the estimation time based on the data of other cus-

tomers who have made a similar expense, referring to the departments that the customer has previously selected.

Then, the customer must select the reservation type which can be of two types:

- **ASAP**: In this case the customer will be placed inside the queue occupying the first available slot.
- **Choose Slot**: In this case the customer is going to select an available slot with a length equal to the duration entered or calculated previously.

Figure 38: *Estimated duration*

Figure 39: *Choice of reservation type*

3.2.6 Choose Slot

In this section is shown the mockup about the choice of time slot. If a customer decide to plan a visit to the store, he must select a time slot which indicates the time spent by the customer inside the store. The customer can complete his reservation, but, in case of he is not fully satisfied with the time slots, he can asks for suggestions.

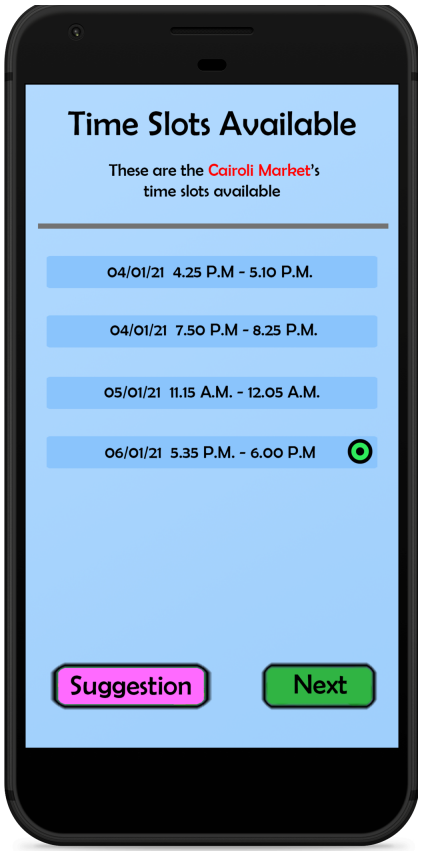


Figure 40: *Time slots*

3.2.7 Suggestions

In this section is shown the mockups about the suggestions of a reservation. If a customer is not satisfied with the available time slots the app allows to receive suggestions. These one are different based on the type of reservation selected previously:

- In case of customer has selected an *ASAP* reservation, if the wait is very long, suggestions are shown automatically. In this case are show, in order of waiting, all the stores that have a better time slot available (including the chosen one) and the customer can accept one of the suggestion or select again the initial store.
- In case of customer has planned a visit but the time slots do not fully satisfy him, he can select a new store, ordered from best to worst, and see the time slots of these new stores. When the customer identify the better choice for him, he can complete the reservation.

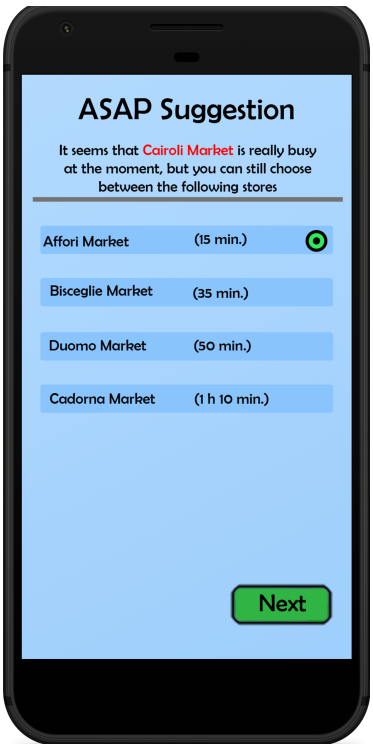


Figure 41: *ASAP suggestion*

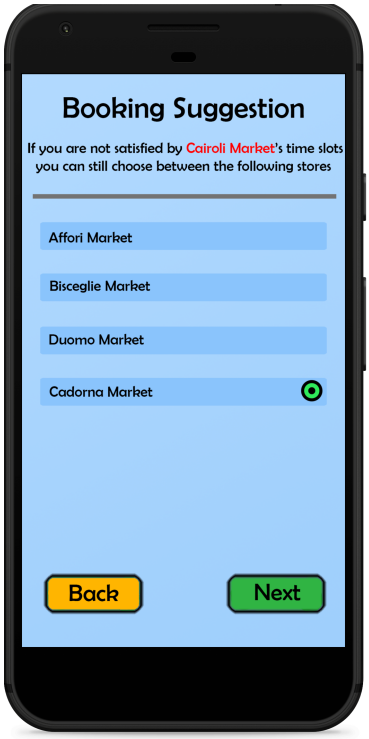


Figure 42: *Reservation suggestion*

3.2.8 Ticket

In this section are shown the mockups about the ticket reservation. When the customer terminates his reservation, a ticket with a *QR Code* is generated and saved. The ticket is used for enter and exit from the store and for both reservations it is the same. the only difference is that in the *ASAP* one the entrance time is updated in real time.

Figure 43: *ASAP ticket*

Figure 44: *Reservation ticket*

3.2.9 Visualize Booking

In this section are shown the mockups about the visualization of previous reservation. When the customer wants can see the previous reservation that he made. This functionality helps customer to build a detailed history.

Figure 45: *Visualize reservations*

Figure 46: *Visualize reservation detail*

3.2.10 Means of Transport

In this section is shown the mockup about means of transport. A customer can select his means of transport and that is very important beacuse, based on the choice made, all travel times are calculated accordingly. Moreover, means of transport used influence also the moment in which the notification, that warns the customer to leave his current position to reach the store in time, is sent.

Figure 47: *Means of transport*

3.2.11 Manage Bookings

In this section are shown the mockups about the management of the customers' reservations. A store manager can:

- View the current entry queue
- Contact a customer via email
- Delete a customer reservation
- Reschedule a customer reservation

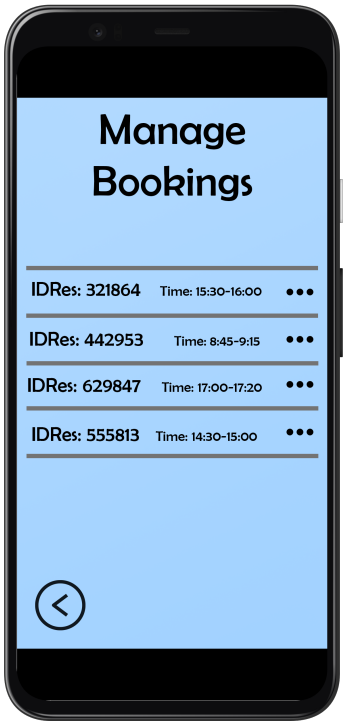


Figure 48: *Manage bookings*

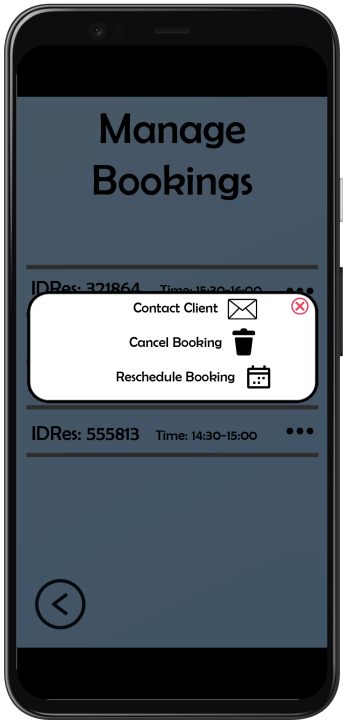


Figure 49: *Manage bookings functionalities*

Figure 50: *Contact customer via email*

Figure 51: *Delete reservation*

3.2.12 Monitor store

In this section is shown the mockup about the store statistics. A store manager can view the statistics about flow of people inside the store, when he wants. Indeed statistics analyze the number of people that visit the store during an entire day and the time spent in average inside the store and each departments.

Figure 52: *Monitor statistics*

3.2.13 Modify parameters

In this section is shown the mockup about the management of the store parameters. A store manager can decide to modify many personal parameters like Store *ID* and password and many parameters that influence the customers' reservations. Indeed is possible to modify the opening and closure time of the store, the max simoultaneous booking per week by a customer and the capacity allowed for each deaprtments and for the entire store

Figure 53: *Modify parameters*

4 Requirements Traceability

In this section it is described how the requirements of the application are mapped to the components defined previously.

- **R1**: The system must allow the customers to register
  AccessManager
- **R2**: The system must allow store managers to register their store
  AccessManager
- **R3**: The system must allow customers to log in
  AccessManager
- **R4**: The system must allow store managers to log in
  AccessManager
- **R5**: The system allows the customers to view their visits
  CustomerHandler
- **R6**: The system allows the customers to cancel their visits
  CustomerHandler
- **R7**: The system allows the customers to modify their visits
  CustomerHandler
- **R8**: The system allows the customers to select their favourite means of transportation
  MobileApp
- **R9**: The system allows customers to select the departments in which they are interested in doing shopping
  ReservationHandler
- **R10**: The system must let in customers only if it's their turn
  StoreComponent
- **R11**: The system must consider the estimate shopping time inserted by customers
  ReservationHandler
- **R12**: The system must show the customers of the time periods in which they can enter the store, according to store's booking policies and the estimate of customers' shopping time
  DataManager
- **R13**: The system have to make a reasonable estimate of when a user with a spot on the queue is able to enter the store
  StatisticComponent

- **R14**: The system can send notification to the clients
  NotificationManager
- **R15**: The system is able to ask for the position of the customers
  MapManager
- **R16**: The system permits to store manager to modify some critical parameters of the store related to customers' affluence management
  StoreManagerHandler
- **R17**: The system allows the manager to establish the maximum simultaneously allowed booked clients in a specific department
  StoreManagerHandler
- **R18**: The store manager can cancel the reservation of each client
  StoreManagerHandler
- **R19**: The store manager can modify the reservation of each client
  StoreManagerHandler
- **R20**: The store manager can view the reservation of each client
  StoreManagerHandler
- **R21**: The store manager can handle the working days and hours of the store, for each day of the week
  StoreManagerHandler
- **R22**: The system knows the situation in real time of each store
  DataManager
- **R23**: The system takes trace of each customer entry and exit from the store
  DataManager
- **R24**: The system contains a list of bookable stores
  DataManager
- **R25**: The system is able to print a paper ticket
  TotemApp
- **R26**: The system can reasonably estimate the time needed from a customer to complete his shopping
  StatisticComponent
- **R27**: The system saves clients' tickets
  DataManager
- **R28**: The system is able to smartly call clients with a ticket to enter the building depending on reservations and people inside the building
  NumberCallingApp

- **R29**: The system is able to scan and analyse a *QR Code*
  ScannerApp
- **R30**: The system is able to send emails
  MailComponent
- **R31**: The system knows how the maximum number of bookings allowed weekly per customer
  DataManager
- **R32**: The system can automatically rearrange reservations, if necessary
  DataManager

5 Implementation, Integration and Test Plan

5.1 Overview

In this section we are going to show a clear plan on how implementation, integration and test plan should be done in order to optimize the work flow.

After considered the various options available to define a *Implementation Plan*, it's been decided to use the **bottom-up approach**. So, each subcomponent will be developed, following an order different from to component to another, depending on the dependencies needed. Moreover, our system has been created with the intention of preserving modularity, in order to have loose coupling between components, thus simplifying future changes and maintenance. In the described plan, the component that can be simultaneously developed are grouped, to give an idea how the work can be managed in order to save time in the integration process. The first two components that must be implemented and integrated are **StoreComponent** and **CLupServer**.

5.2 StoreComponent

Before defining the Implementation Plan of the **Store Component**, a *Use Relation Hierarchy Diagram* have been traced, to understand the dependencies between the various components, so that an order of implementation can be defined.

- the components that have no other dependencies to others are **DataManager**, **MailComponent**, **NotificationManager**. So, they are the first components that will be implemented. In particular, **DataManager** will be implemented after the installation of the *DBMS*, to avoid the use of Stubs since the DBMS is already implemented and deployed, without need to test it. Instead, for **NotificationManager** and **MailComponent** there is the necessity to use some *Drivers* waiting for the implementation of the components that uses them.

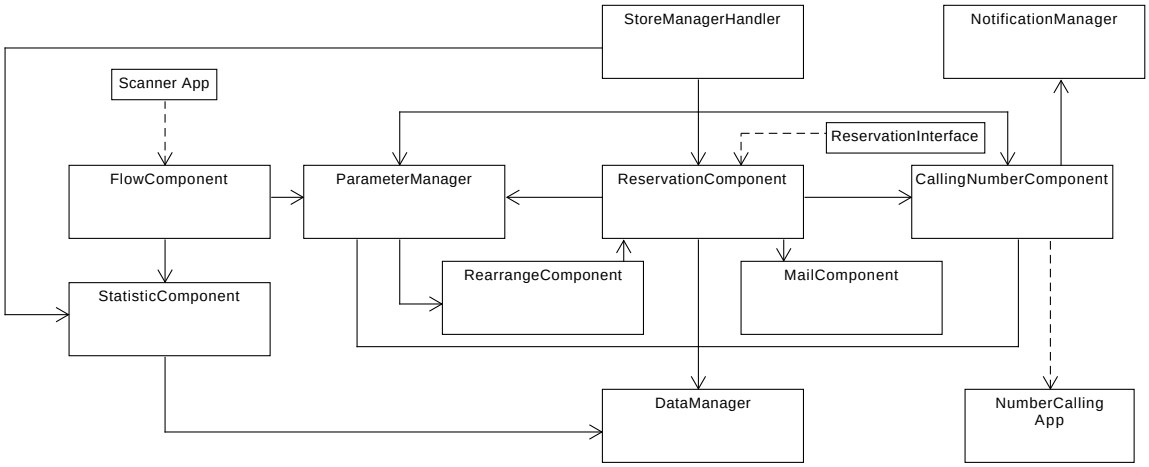


Figure 54: *Use Relation Hierarchy Diagram of StoreComponent*

- then, it's easy to implement the **StatisticComponent**, integrating it with the already developed **DataManager**, also using here some Drivers.
- at this point, the most critic part of the integration plan is about **ParameterManager**, **ReservationComponent** and **CallingNumberComponent**, since are related in couple. For this reason, we can use some *Drivers* and *Stubs* for the other components not already implemented, but in the simultaneous development of the three, it's suggested to implement and integrate them using the "Thread" technique, so that the dependencies between them can be integrated going on with the work. Done the three component, it's possible to integrate them with the one they use, as reported in the graph (**NotificationManager**, **MailManager**, **DataManager**). For **CallingNumberComponent**, it's necessary to use some *Stubs* to simulate the **NumberCallingApp**, and some *Drivers* for the **ReservationComponent**.
- after that these have been completed, it's possible to implement **FlowComponent**, **RearrangeComponent** and **StoreManagerHanlder**, without the need of using *Stubs* or *Driver* since their dependencies are implemented, but for **ScannerApp**, for which a *Driver* is needed in order to complete the testing of **FlowComponent**.

5.3 CLupServer

Also for **CLupServer** a *Use Relation Hierarchy Diagram* have been done, in order to identify the order with the single components can be implemented.

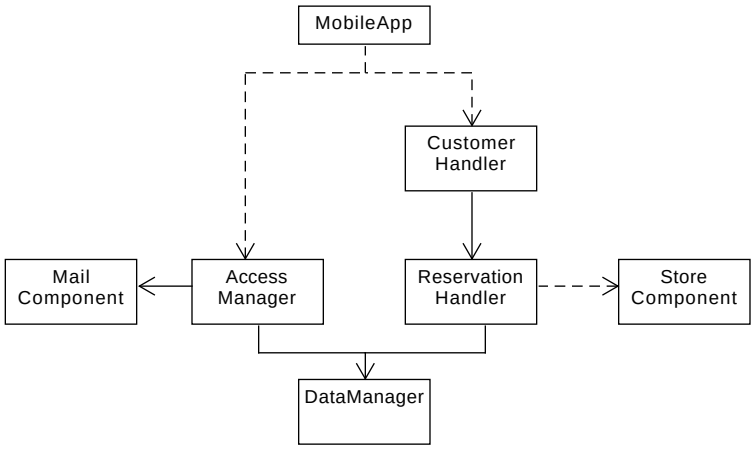


Figure 55: *Use Relation Hierarchy Diagram of CLupServer*

- the first components that can be implemented are **MailComponent** and **DataManager**; as for the **StoreComponent**, the **DataManager** should be implemented and integrated after the associated *DBMS* has been installed.
- then, it's possible to implement the **AccessManager** and the **ReservationHandler**, which can be integrated with the **DataManager**. Some *Drivers* simulating **CustomerHandler** and **MobileApp** are required to complete the test phase, and a *Driver* is needed to simulate the **StoreComponent** in testing the **ReservationHandler**
- finally, the **CustomerHandler** can be implemented, integrated with the **ReservationHandler** and tested. In this last phase, is needed a *Driver* to simulate the **MobileApp**

After the previously described components have been developed, integrated and unit tested, it's possible to develop all the clients using them, and integrate the system in his completeness.

5.4 System Testing

Once the system is completely integrated, we can proceed testing it, to verify that functional and non-functional requirements are satisfied. In order to consider all the possible issues that our system could have, we have to test it in different ways:

- **Functional Testing**: ascertains that the system meet the functional requirements described in the RASD. Some tests that can be conducted are

creating a reservation and check if it's created correctly, checking if correct suggestions are generated, trying the working of the management of QR Codes and of number calling. Moreover, also the correctness of calculation of statistics (and so of shopping time estimation) must be tested.

- **Performance Testing**: highlights problems related to the performance and bottlenecks; this test aims to check that the time constraint introduced in the RASD are respected, such as *QR Codes* processing time, the time needed to call a new reservations and the one necessary to process a new one, general system responsiveness and departure notification generated in time
- **Load Testing**: highlights problems memory-related such as memory leaks and buffer overflows (e.g. verify if the system works even with a large number of customer and store manager operating at the same time, respecting the constraints of 200 simultaneous users per store registered in the system, as described in the RASD)
- **Stress Testing**: ascertains that the application recovers in the right way after failure (e.g. see what happens after an unexpected internet disconnection during an operation, and how the system responds to recover the data and operativity)

6 Effort Spent

Task	Daniele Mammone's Hours
Introduction	0
Architectural Design	0
User Interface Design	0
Requirements Traceability	0
Implementation, Integration and Test Plan	0
Total Hours	0

Task	Gianmarco Naro's Hours
Introduction	0
Architectural Design	0
User Interface Design	0
Requirements Traceability	0
Implementation, Integration and Test Plan	0
Total Hours	0

Task	Massimo Parisi's Hours
Introduction	0
Architectural Design	0
User Interface Design	0
Requirements Traceability	0
Implementation, Integration and Test Plan	0
Total Hours	0

7 References