DD – Design Document

Immordino Alessandro – 969549
Pala Riccardo – 969598
Polvanesi Giacomo – 971083

Politecnico di Milano

A.A. 2020/21

Table of Contents

1 Introduction

1.1 Purpose

The purpose of this Design Document is to give more technical details regarding the CLup application by following the guidelines that are stated in the RASD document. This document describes architectural choices made for implementing the software in a proper way. It starts with a general view about the system design, then it goes deeper describing the functionalities of al the components and how they work together. Each choice is described and justified by listing all the advantages and it is represented by diagrams where the main features are depicted. The main topics, developed in this document, are listed below:

- The high-level architecture
- The main components, their interaction and the deployment view
- The runtime behaviour by means of sequence diagrams
- The design patterns
- The user interfaces by means of mock-ups
- A mapping of the requirements described in the RASD on the various components
- The implementation and testing plans

1.2 Scope

The scope of the application, as also explained in the RASD, is to provide a service that monitors people turnout in grocery stores, especially during the situation of pandemic we are going through. The aim is to avoid crowdings both outside and inside the supermarkets by adopting a mechanism of access regulation: customers that need to do grocery shopping must first reserve a place via app or by means of designated devices placed outside the stores. Devices addressed to the reservations allow to not leave out clients that, for some reasons, cannot access the necessary technologies for making a reservation via mobile app.

As mentioned before, the system avoids people from crowding near the store. In order to prevent this situation, reservations are identified with a unique token that is placed in a “virtual line-up”. The application provides clients with the number of their own position in the virtual line-up and with an expected waiting time, in order to ensure that users approach the building if their turn is coming. The customers must also indicate how much time they expect of spending within the supermarket, indeed this parameter is essential to compute the waiting time.

There is also an advanced function that consists of planning the visit by indicating the starting time of the reservation. This option regards only users and it provides users with the priority over the real-time reservations.

CLup guarantees a service that involves a very huge number of clients, since doing grocery shopping is a daily activity which has to be accomplished with respect to governement restrictions and safety rules.

1.3 Definitions, acronyms and abbreviation

1.3.1 Definitions

• Real-time reservation: reservation allowing Client to join immediately the virtual line-up in order to enter the supermarket as soon as possible.

• Planned reservation: reservation allowing User to schedule a visit for doing grocery shopping afterwards.

• Queue (Line): physical line-up of people waiting for their turn outside a supermarket.

• Virtual line-up: ordered set of all real-time reservations, concerning a certain supermarket, including those neither scanned nor expired.

• Expected time of entrance: accurate estimate of the visit starting instant for a real-time reservation.

• Maximum tolerated delay: time interval, from the visit starting instant, after which a reservation, not scanned yet, expires

• Schedule: time slot of a planned reservation from the starting, to the ending instant.

• Ticket number: number of a real-time reservation describing its relative position in the virtual line-up.

• Visit: activity of doing grocery shopping

1.3.2 Acronyms

• RASD = Requirement Analysis and Specification Document.

• API = Application Programming interface.

• QR = Quick Response.

1.3.3 Abbreviations

• ETE = expected time of entrance.

• MTD = maximum tolerated delay.

1.4 Reference documents

- Specification Documents: “R&DD Assignment AY 2020-2021”
- Rich Mobile Application: “https://en.wikipedia.org/wiki/Rich_mobile_application”
- Multitier architecture: “https://en.wikipedia.org/wiki/Multitier_architecture”
- For diagrams: “https://app.diagrams.net/”
- For mockups: “https://app.moqups.com/”

1.5 Document structure

Chapter 1: it describes the purposes and the scope of the DD by introducing the main functions of the system and the main aspects relevant for the design analisys.

Chapter 2: it proposes and describes the architectural design principles that are more suitable for the CLup software-to-be by analyzing all the aspects: from the component view to some detailed features regarding the guidelines for the algorithms.

Chapter 3: it provides a description of the interfaces with which the user interfacts once the software is deployed by means of mockups and a diagram that represents the user experience.

Chapter 4: it assigns the role of each component for the satisfaction of the requirements exposed in the RASD.

Chapter 5: this last chapter provides some details regarding the implementation strategy and the integration plan to follow in order to obtain a reliable and ready-to-use product.

2 Architectural design

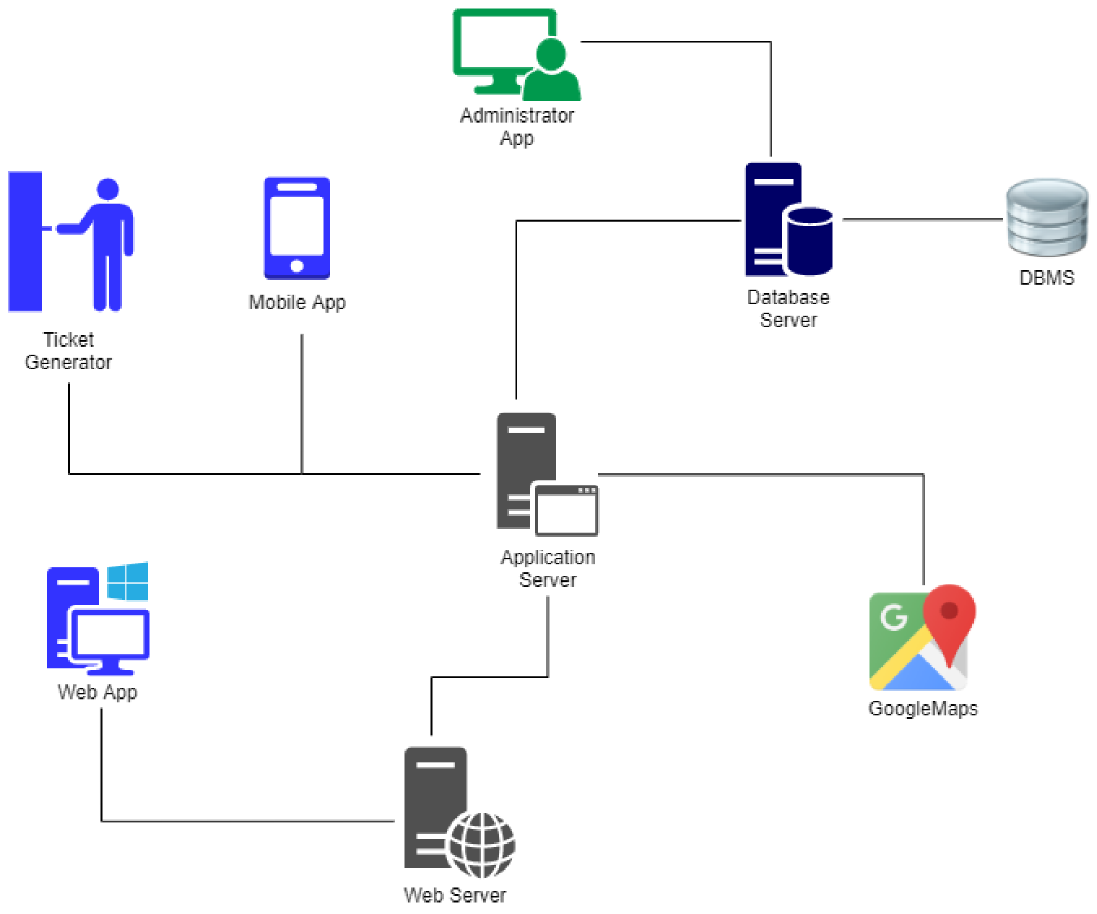
2.1 Overview

From a logical point of view the system structure can be represented by a *three-layered architecture*:

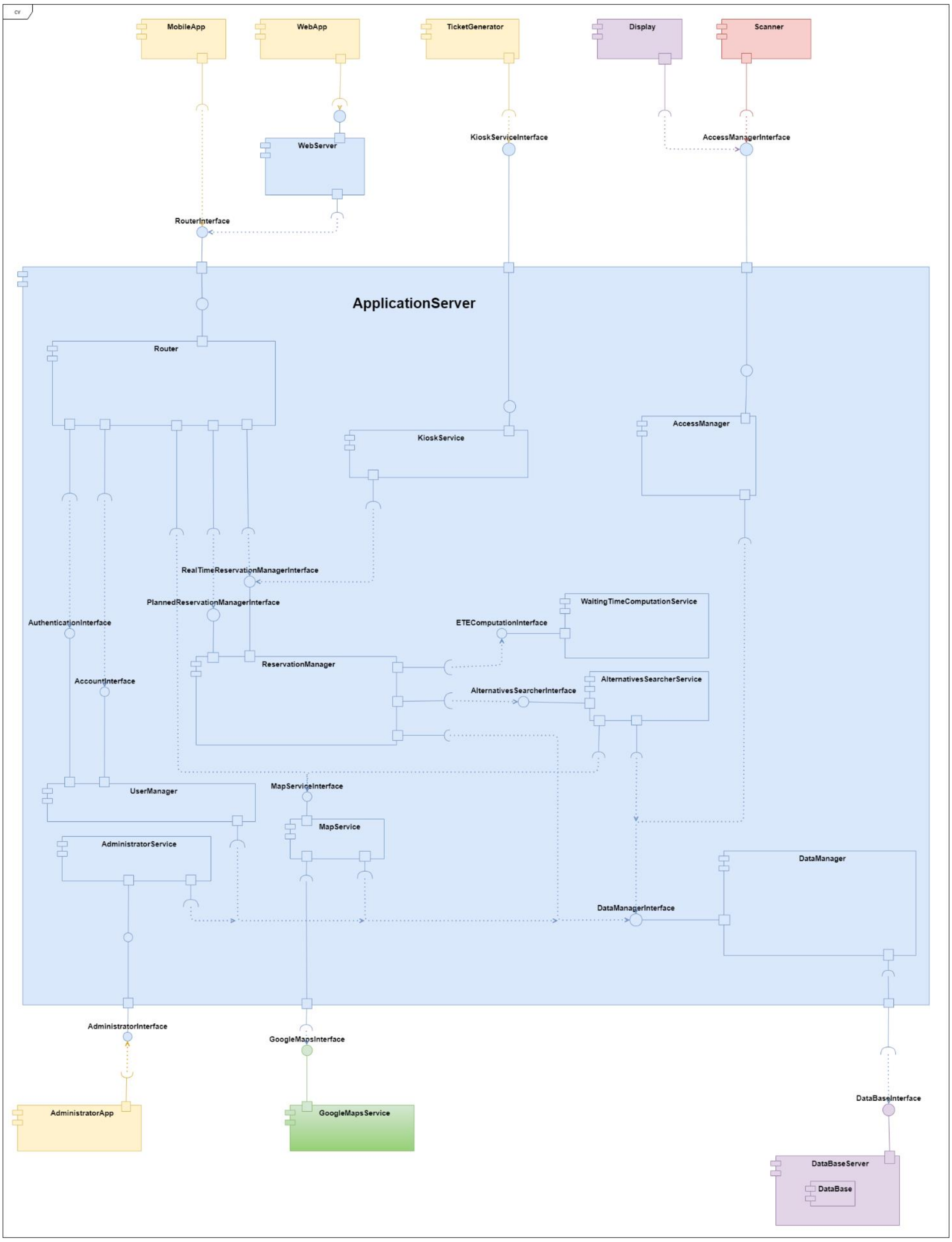
- **Presentation layer**: handles the user interface;
- **Business or Application layer**: mediator between PL and DL, it takes care of doing logical computation to perform operations;
- **Data layer**: deals with database access and data storage.

These *abstract layers* are concretely translated into a *physical* structure that is composed by different levels, the *tiers*. Afterwards, the *multi-tier architecture* adopted by the system is described.

The following figure depicts an abstraction of our software application in order to highlight the main high-level components. In the next sections the system is described by using a more specific approach both from logical (“*Component view*” and “*Component interfaces*”) and physical (“*Deployment view*”) way.



2.2 Component view description



The “component” is an abstraction of the concept of class, and it represents a logic unit of the system. Interaction among components is described through the definition of interfaces, which defines a set of operations used, required, created and provided by the components.

The following diagram describes the communication network among the different parts of the system.

- *Router*: this component behaves as a “façade” (view “Selected architectural style and patterns” section) through which Client can easily interact with other components. It takes care of Client-side requests despatching them to the related components of the Application Server. It also sends back properly the received responses.

- *ReservationManager*: this component is designated to manage all the operations requests concerning the reservations. In order to fulfil its requirements it needs to get several information through the *DataManager* interfaces.

- *AccessManager*: this component behaves as the driver of supermarkets access process at code level. It focuses on providing the access devices (*Scanner* and *Display*) with interfaces, and so methods, for the purpose of driving them to successfully handle the entrance procedure.

- *DataManager*: this component gathers all the requests directed to the Database Server. It is linked to every component of the application because all of them have to interact with the database. It is an essential element since it provides others with methods allowing to access and make operations on data.

- *WaitingTimeComputationService*: this component provides an interface with methods for the waiting time computation of a *real-time reservation*. Choice of separating this component from *ReservationManager* sub-system derives from the need of highlighting the importance of this service. The provided methods implement algorithms that are the key of reservation and access processes. Therefore, this service is essential for goals achievement and for functionality of the software-to-be.

- *MapService*: this component acts as a mediator between *GoogleMapsService* and logic components which exploit its functionalities. In this way, if we decide to use another mapping service, rather than GoogleMaps, we just need to change this intermediate component without intervening on the logic part of the application.

- *KioskService:* such as *MapService*, this component provides *KioskApp* with means for facing Application Server, in particular *RealTimeReservationManager*, without referring to the *Router*. Moreover, some changes on *KioskApp* software do not imply total substitution of component and related interface. Changes might be figured out just modifying the mediator.

- *AlternativesSearcherService*: this component is designated to carry out the specific operation of finding out alternative solutions in order to satisfy user requests. Invoked by *PlannedReservationManager*, it communicates with *MapService* to look for supermakets position and *DataManager* to check for availability.

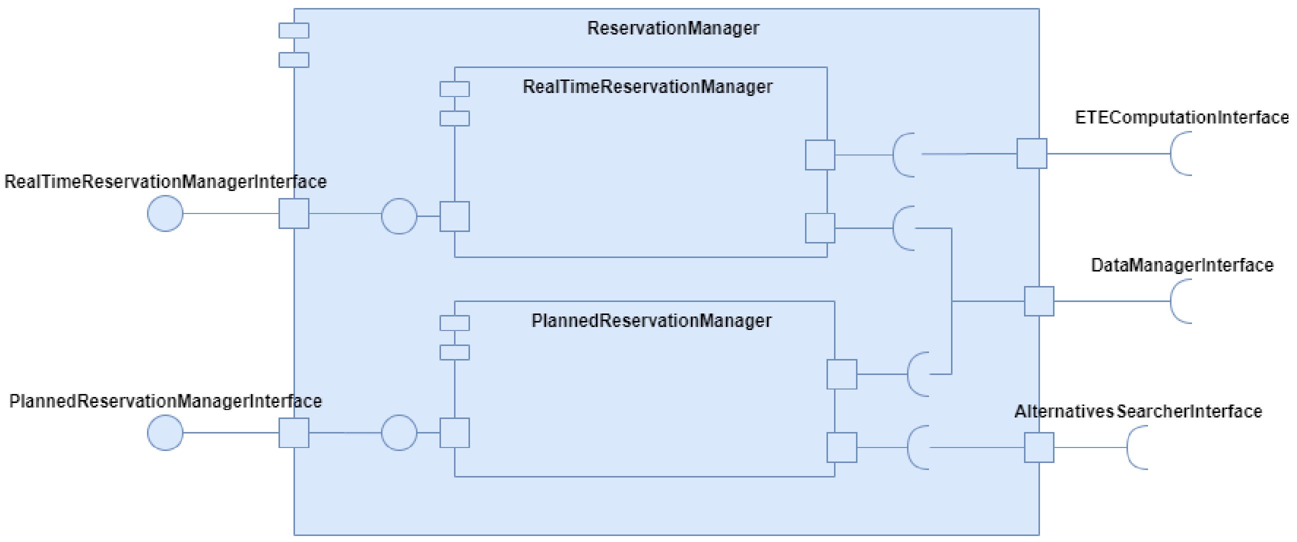
- *UserManager*: this component handles the access procedure of Client to the system. This process must be provided with a security mechanism in order to prevent undesirable entries from visitors not allowed to access the application services.

- *AdministratorService*: this component represents an intermediate step between Administrator interface and Data Server. *AdministratorApp* does not belong to *CLup* system, it is just a comfortable way for admins to interact with DBMS. It has not got many functionalities and it can follow only one path towards the database, therefore *Router* does not need to catch and redirect his request. Anyway, Client cannot directly access to the database service, indeed, *AdministratorService* operates as a mediator between *DataManager* component and *AdministratorApp*.

2.2.1 ReservationManager view

- *RealTimeReservationManager*: this sub-component provides user with an interface that allows him to create and manage his *real-time reservation* and update the *virtual line-up* of supermarkets. It has to interact with *DataManager* and *WaitingTimeComputationService* for the purpose of getting all useful information and performing operations to fulfil its tasks.

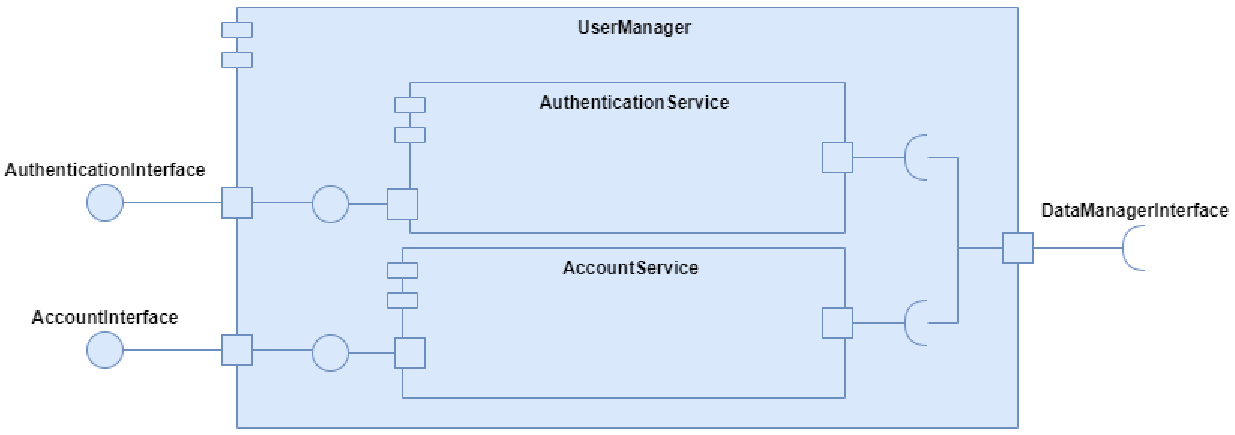
- *PlannedReservationManager*: this sub-component handles user operations concerning a *planned reservation.* It uses the *DataManager* interface that is essential to accomplish those procedures, while *AlternativeSearcherService* is useful for satisfying specific user requests. Furthermore, it manages the timetable of supermarkets.


ReservationManager
RealTimeReservationManager
PlannedReservationManager
ETEComputationInterface
RealTimeReservationManagerInterface
DataManagerInterface
PlannedReservationManagerInterface
AlternativesSearcherInterface

2.2.2 UserManager view

- *AuthenticationService*: this sub-component deals with a login attempt validation. It is connected to *DataManager* because it needs to check inserted data.

- *AccountService*: this sub-component monitors registration and information update processes of a user. It communicates with *DataManager* in order to store obtained data on database proper table.

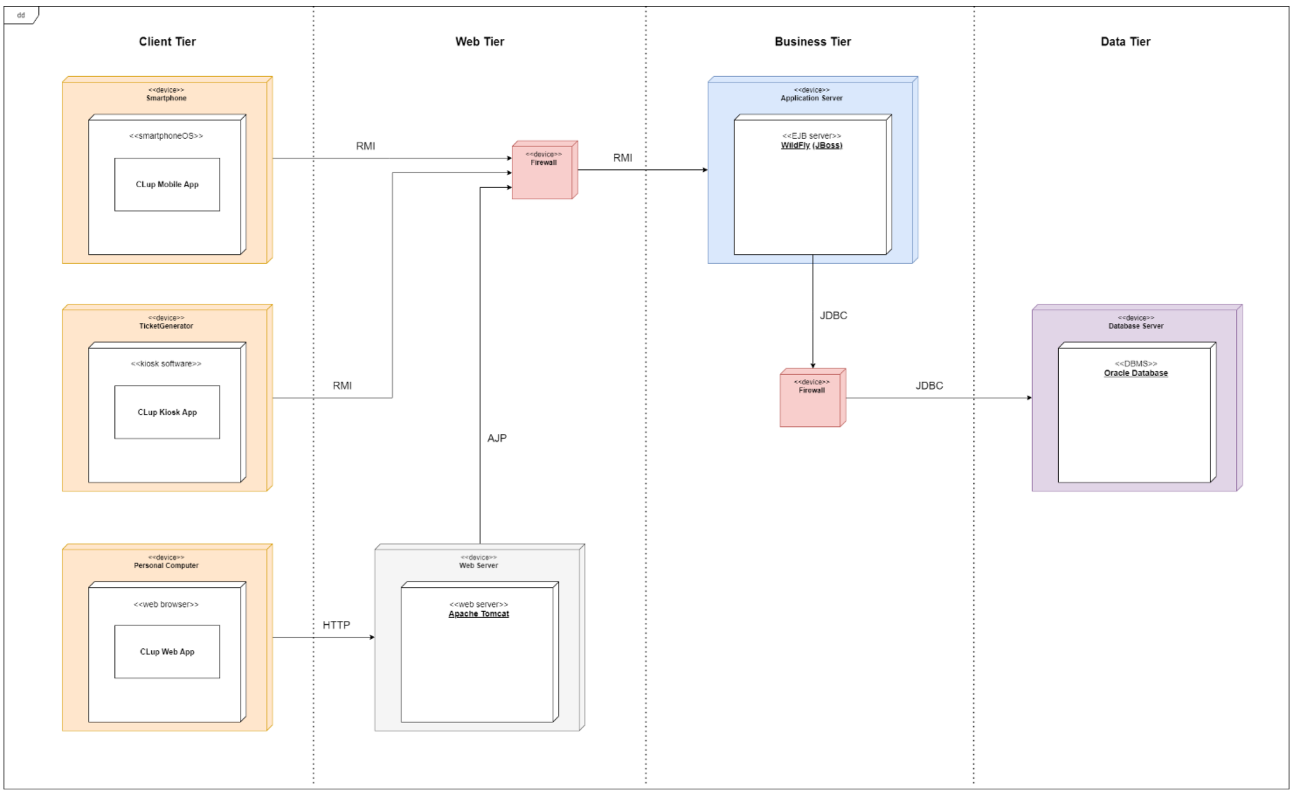


2.3 Deployment view

2.3.1 Deployment diagram

The following diagram describes the physical architecture of the application software. A multitier architecture is adopted. Logical layers (Presentation, Application and Data) are distributed among different platforms following an architectural pattern based on 4 tiers.

- **Client tier:** composed by devices client uses to interface with the system.
- **Web tier:** includes the Web Server that is able to manage client requests come from a web browser.
- **Business tier:** contains Application Server which performs business tasks and logic operations.
- **Data tier:** refers to DBMS charged of database accessing.



2.3.2 Recommended implementation

The above-showed schema also represents an idea of implementation. Indeed, specific platforms are suggested to be used.

For instance, Apache Tomcat (or simply Tomcat) server is proposed as Web Server. It deals with generating dynamic contents by using Java Servlet or JavaServer Pages (JSP) technologies. Requests of dynamic contents comes from the world wide web through the protocol of transmission HTTP.

Tomcat supports Apache JServ Protocol (AJP) that might be used as transimission protocol to send information from the Web Server to the Application Server. For this purpose, WildFly (formerly known as JBoss) may be the choice for the Business Tier, since it includes Enterprise JavaBean (EJB) among its components. EJB are software components which implement business logic on Java Enterprise Edition (JEE) architecture.

Other client devices (Mobile App and TicketGenerator App) send requests directly to the Application Server via Remote Method Invocation (RMI) that is a technology allowing distributed processes to communicate through a network.

Eventually, Java Database Connectivity (JDBC) is an API which establishes the connection between Application Server and Database Server providing methods for database accessing. Oracle Database might be the chosen DBMS for the implementation of the application.

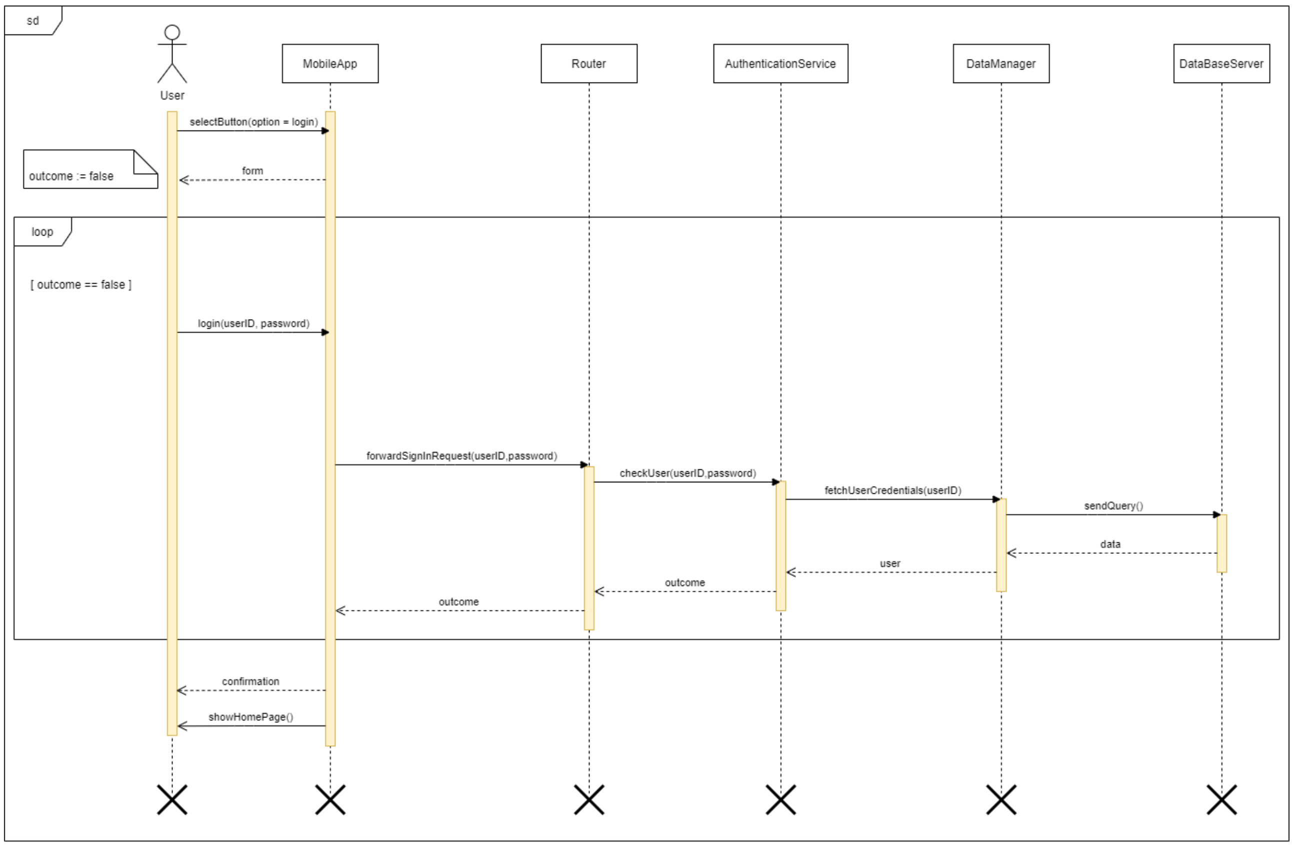
2.3.3 Other decisions

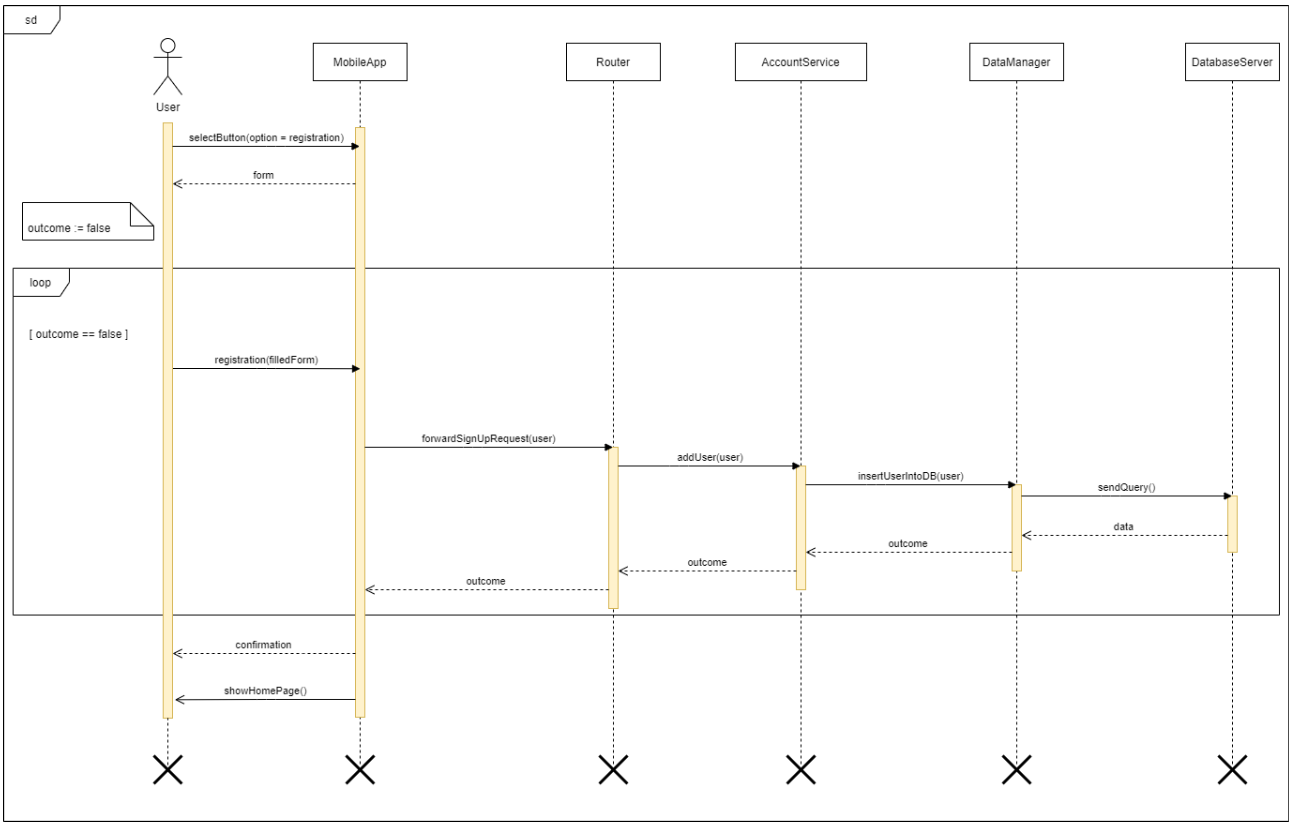
- As we can see in the figure of "Deployment diagram", two firewalls are represented. These components provide a security system to prevent malicious agents from accessing the network.
- Application and Web Server are doubled in order to prevent overload and guarantee an higher requests capacity. Load balancers are installed for the purpose of distributing workload among the servers. This technique improves scalability and reliabality of the system.

2.4 Runtime view

In this section, diagrams concerning the runtime views of the most relevant occuring situations are represented. They decribe the interaction among all the components in a more detailed way than sequence diagrams did in the RASD document. For some of these a further description is added to facilitate the comprehension of those step that could be confusing at first sight.

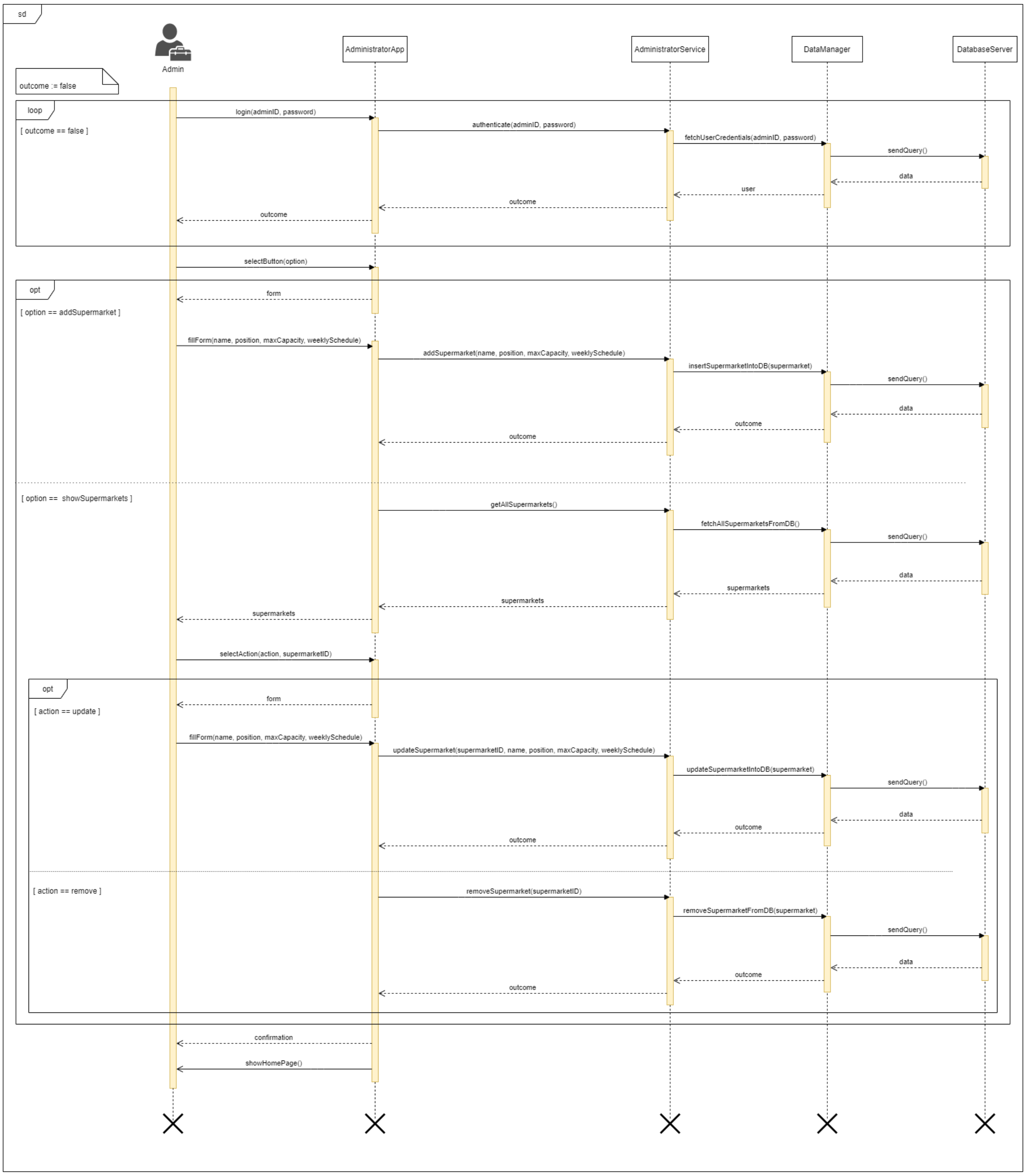
2.4.1 Login and Registration sequence diagrams



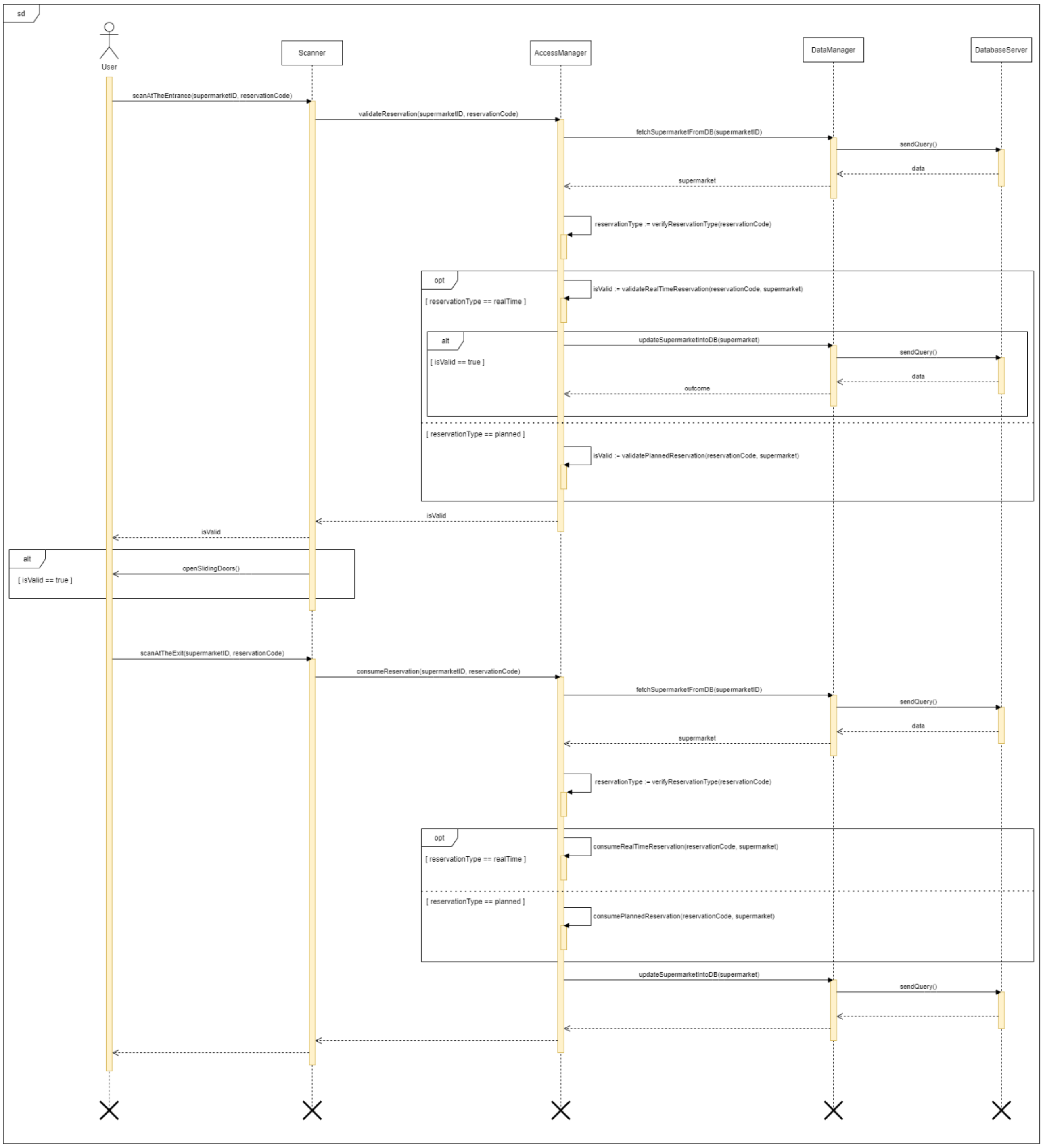


A button to come back to the starting page is assumed to be in the login and registration forms.

2.4.2 Administration sequence diagram


sd
Admin
AdministratorApp
AdministratorService
DataManager
DatabaseServer
outcome := false
loop
[outcome == false]
login(adminID, password)
authenticate(adminID, password)
fetchUserCredentials(adminID, password)
sendQuery()
data
user
outcome
outcome
selectButton(option)
opt
[option == addSupermarket]
form
fillForm(name, position, maxCapacity, weeklySchedule)
addSupermarket(name, position, maxCapacity, weeklySchedule)
insertSupermarketIntoDB(supermarket)
sendQuery()
data
outcome
outcome
[option == showSupermarkets]
getAllSupermarkets()
fetchAllSupermarketsFromDB()
sendQuery()
data
supermarkets
supermarkets
supermarkets
selectAction(action, supermarketID)
opt
[action == update]
form
fillForm(name, position, maxCapacity, weeklySchedule)
updateSupermarket(supermarketID, name, position, maxCapacity, weeklySchedule)
updateSupermarketIntoDB(supermarket)
sendQuery()
data
outcome
outcome
[action == remove]
removeSupermarket(supermarketID)
removeSupermarketFromDB(supermarket)
sendQuery()
data
outcome
outcome
confirmation
showHomePage()

2.4.3 Scanning sequence diagram


sd
User
Scanner
AccessManager
DataManager
DatabaseServer
scanAtTheEntrance(supermarketID, reservationCode)
validateReservation(supermarketID, reservationCode)
fetchSupermarketFromDB(supermarketID)
sendQuery()
data
supermarket
reservationType := verifyReservationType(reservationCode)
opt
[reservationType == realTime]
isValid := validateRealTimeReservation(reservationCode, supermarket)
alt
[isValid == true]
updateSupermarketIntoDB(supermarket)
sendQuery()
data
outcome
[reservationType == planned]
isValid := validatePlannedReservation(reservationCode, supermarket)
isValid
isValid
alt
[isValid == true]
openSlidingDoors()
scanAtTheExit(supermarketID, reservationCode)
consumeReservation(supermarketID, reservationCode)
fetchSupermarketFromDB(supermarketID)
sendQuery()
data
supermarket
reservationType := verifyReservationType(reservationCode)
opt
[reservationType == realTime]
consumeRealTimeReservation(reservationCode, supermarket)
[reservationType == planned]
consumePlannedReservation(reservationCode, supermarket)
updateSupermarketIntoDB(supermarket)
sendQuery()

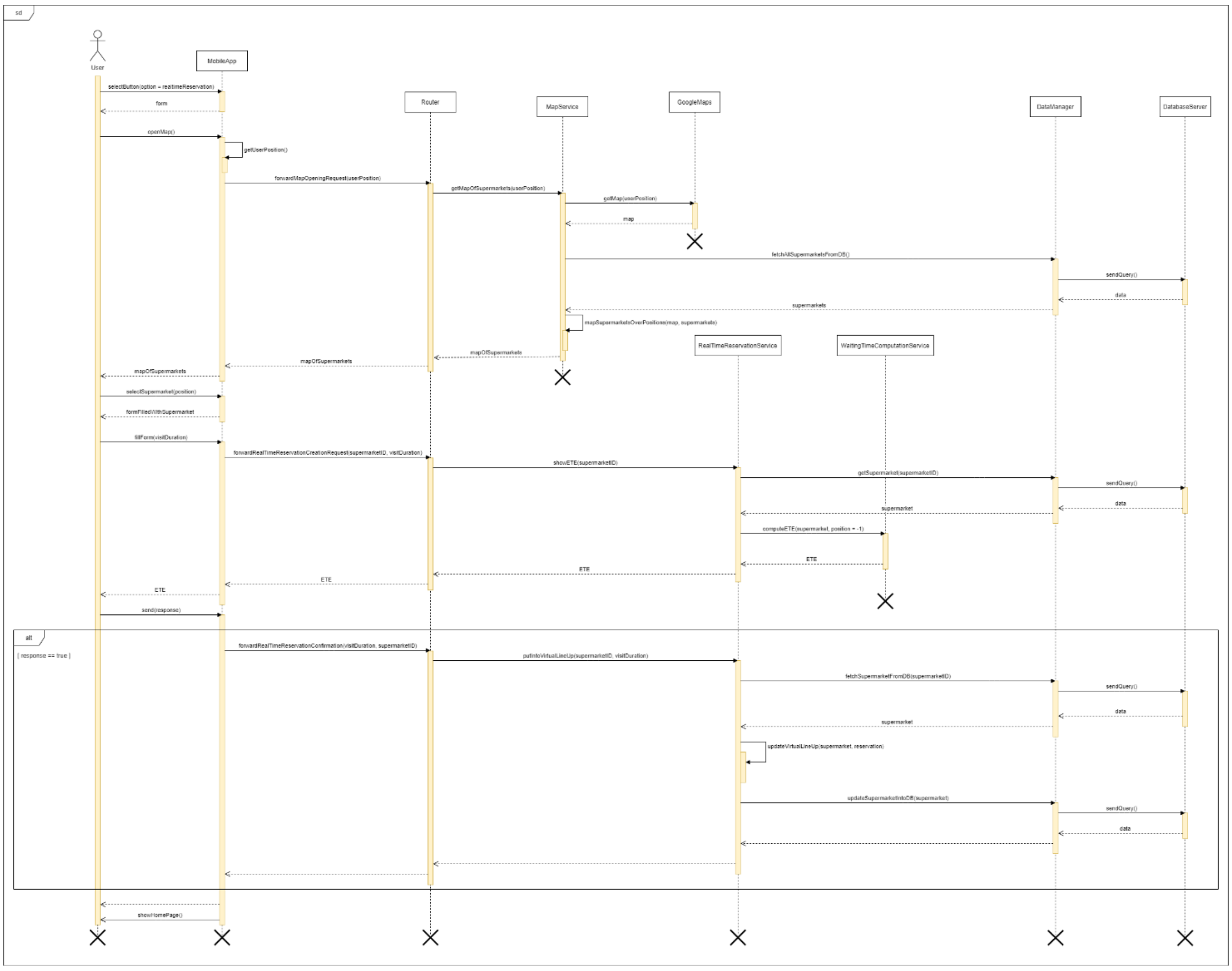
This sequence diagram depicts the procedure of validation of a reservation. Once the QR code of the reservation is read by *Scanner*, it is translated in a string that represents the reservation code of a certain supermarket to be stored in the database. These information are sent to *AccessManager* that can recognize either if the reservation is *real-time* or *planned*, basing on the reservation code and then, depending on the type, it invokes differents methods.

The function *validateRealTimeReservation* updates the attribute concerning the last scanned *ticket number*, the attribute *EntranceTime* of the reservation, assigning the time that represents the instant of the physical scan performed by *Scanner* and, finally, it updates also attributes of the involved Supermarket object, which collect waiting reservations and already scanned reservations doing grocery shopping.

Instead, the function *validPlannedReservation* updates only the *EntranceTime* attribute.

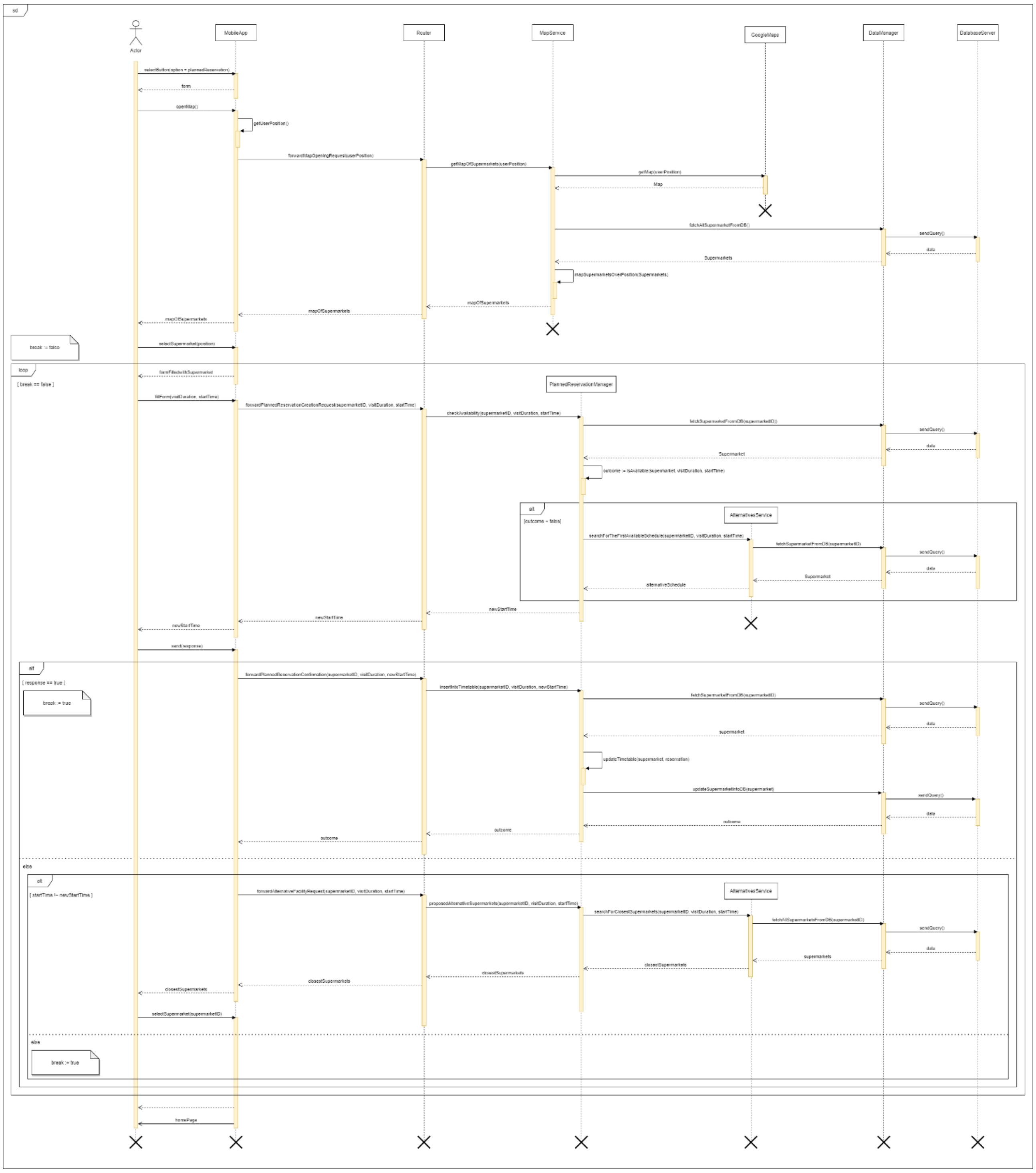
All the changes become persistent in the database once the function *updateSupermarketIntoDB* is called.

2.4.4 Real-time reservation sequence diagram


sd
User
MobileApp
Router
MapService
GoogleMaps
DataManager
DatabaseServer
selectButton(option = realtimeReservation)
form
openMap()
getUserPosition()
forwardMapOpeningRequest(userPosition)
getMapOfSupermarkets(userPosition)
getMap(userPosition)
map
fetchAllSupermarketsFromDB()
sendQuery()
data
supermarkets
mapSupermarketsOverPositions(map, supermarkets)
RealTimeReservationService
WaitingTimeComputationService
mapOfSupermarkets
mapOfSupermarkets
mapOfSupermarkets
selectSupermarket(position)
formFilledWithSupermarket
fillForm(visitDuration)
forwardRealTimeReservationCreationRequest(supermarketID, visitDuration)
showETE(supermarketID)
getSupermarket(supermarketID)
sendQuery()
data
supermarket
computeETE(supermarket, position = -1)
ETE
ETE
ETE
ETE
send(response)
alt
[response == true]
forwardRealTimeReservationConfirmation(visitDuration, supermarketID)
putIntoVirtualLineUp(supermarketID, visitDuration)
fetchSupermarketFromDB(supermarketID)
sendQuery()
data
supermarket
updateVirtualLineUp(supermarket, reservation)
updateSupermarketIntoDB(supermarket)
sendQuery()
data
showHomePage()

This runtime view shows all the steps needed to create a *real-time reservation* via mobile application.

2.4.5 Planned reservation sequence diagram


sd
Actor
MobileApp
Router
MapService
GoogleMaps
DataManager
DatabaseServer
selectButton(option = plannedReservation)
form
openMap()
getUserPosition()
forwardMapOpeningRequest(userPosition)
getMapOfSupermarkets(userPosition)
getMap(userPosition)
Map
fetchAllSupermarketFromDB()
sendQuery()
data
Supermarkets
mapSupermarketsOverPosition(Supermarkets)
mapOfSupermarkets
mapOfSupermarkets
mapOfSupermarkets
break := false
selectSupermarket(position)
loop
[break == false]
formFilledwithSupermarket
PlannedReservationManager
fillForm(visitDuration, startTime)
forwardPlannedReservationCreationRequest(supermarketID, visitDuration, startTime)
checkAvailability(supermarketID, visitDuration, startTime)
fetchSupermarketFromDB(supermarketID))
sendQuery()
data
Supermarket
outcome := isAvailable(supermarket, visitDuration, startTime)
alt
[outcome = false]
AlternativesService
searchForTheFirstAvailableSchedule(supermarketID, visitDuration, startTime)
fetchSupermarketFromDB(supermarketID)
sendQuery()
data
Supermarket
alternativeSchedule
newStartTime
newStartTime
newStartTime
send(response)
alt
[response == true]
break := true
forwardPlannedReservationConfirmation(supermarketID, visitDuration, newStartTime)
insertIntoTimetable(supermarketID, visitDuration, newStartTime)
fetchSupermarketFromDB(supermarketID)
sendQuery()
data
supermarket
updateTimetable(supermarket, reservation)
updateSupermarketIntoDB(supermarket)
sendQuery()
data
outcome
outcome
outcome
else
alt
[startTime != newStartTime]
AlternativesService
forwardAlternativeFacilityRequest(supermarketID, visitDuration, startTime)
proposedAlternativeSupermarkets(supermarketID, visitDuration, startTime)
searchForClosestSupermarkets(supermarketID, visitDuration, startTime)
fetchAllSupermarketsFromDB(supermarketID)
sendQuery()
data
supermarkets
closestSupermarkets
closestSupermarkets
closestSupermarkets
closestSupermarkets
selectSupermarket(supermarketID)
else
break := true
homePage

This runtime view shows all the steps needed to create a *planned reservation* via mobile application.

Insertion of the guard variable (break) is useful to distinguish all the situations the system could face during the loop of the above-showed sequence diagram.

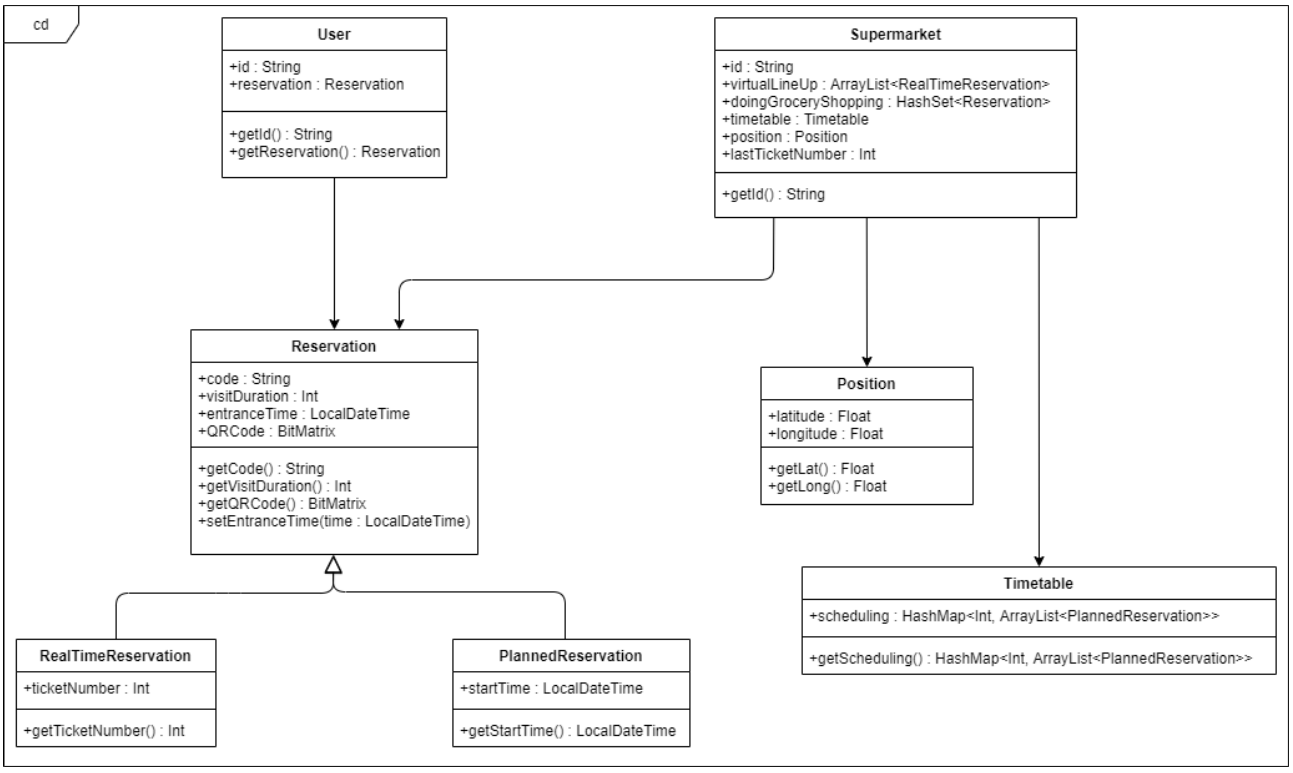
There are three possible situation:

1. The system suggests an alternative *schedule* (*startTime* != *newStartTime*) in case the selected one in not available and the user refuses it. In this situation, the system proposes alternatives basing on the closest supermarkets with the selected *schedule* available in the resepective timetable. Then, either the user can select one of the alternatives and refill the form or he can refuse the options returning at the home page.
2. The system suggests an alternative *schedule* (*startTime* != *newStartTime*) in case the selected one in not available and the user accepts it. In this situation a new reservation is generated.
3. The system confirms that the selected *schedule* is available (*startTime* == *newStartTime*) so it creates the reservation.

2.5 Component interfaces

This section focuses on interfaces description: by means of "Interface diagram" the main functions are represented and interactions among different interfaces are outlined. In the methods, the types of some arguments are objects derived from classes that are depicted in the following "Class diagram".

2.5.1 Class diagram

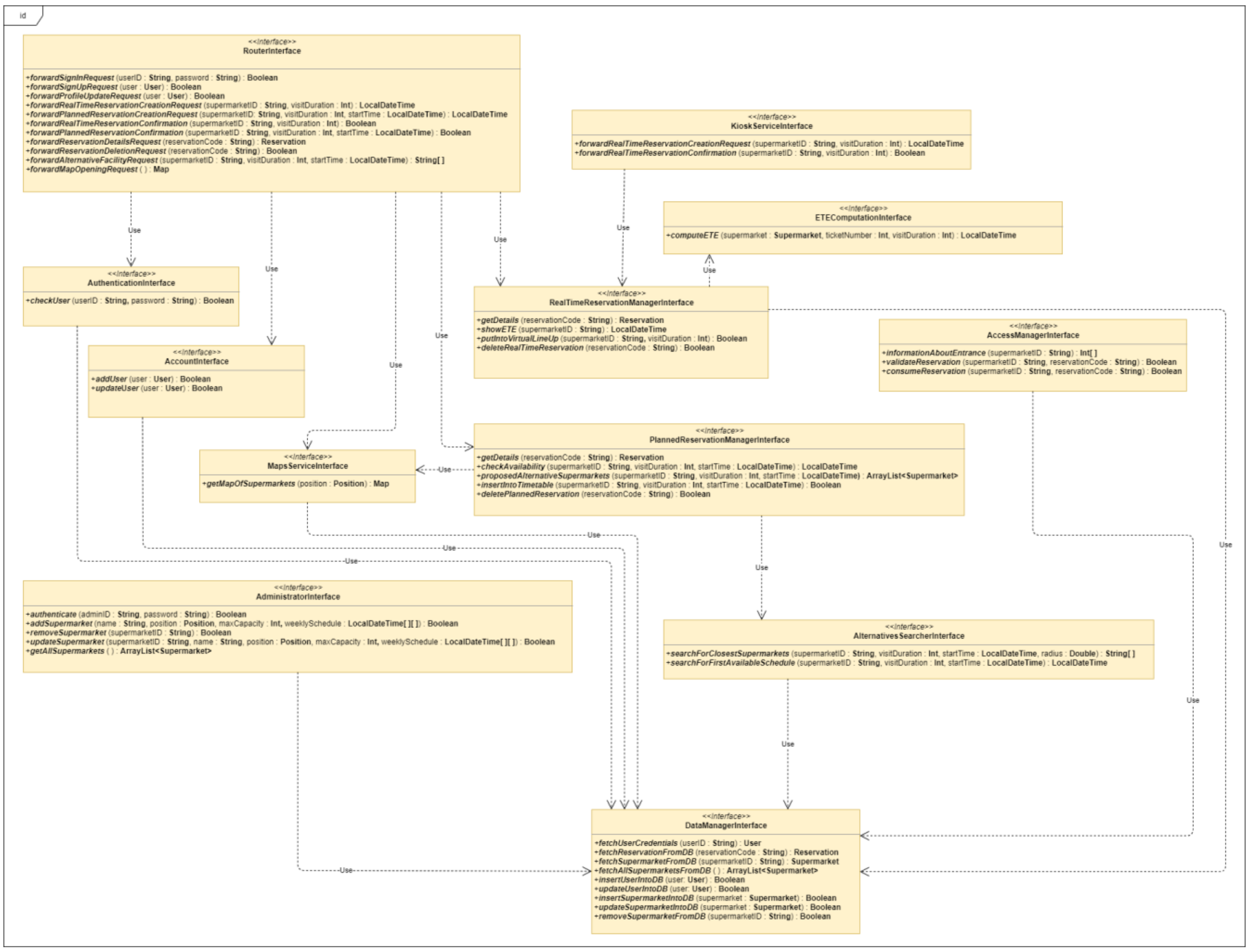

cd
User
+id : String
+reservation : Reservation
+getId() : String
+getReservation() : Reservation
Supermarket
+id : String
+virtualLineUp : ArrayList<RealTimeReservation>
+doingGroceryShopping : HashSet<Reservation>
+timetable : Timetable
+position : Position
+lastTicketNumber : Int
+getId() : String
Reservation
+code : String
+visitDuration : Int
+entranceTime : LocalDateTime
+QRCode : BitMatrix
+getCode() : String
+getVisitDuration() : Int
+getQRCode() : BitMatrix
+setEntranceTime(time : LocalDateTime)
Position
+latitude : Float
+longitude : Float
+getLat() : Float
+getLong() : Float
Timetable
+scheduling : HashMap<Int, ArrayList<PlannedReservation>>
+getScheduling() : HashMap<Int, ArrayList<PlannedReservation>>
RealTimeReservation
+ticketNumber : Int
+getTicketNumber() : Int
PlannedReservation
+startTime : LocalDateTime
+getStartTime() : LocalDateTime

The class Supermarket has got all the information about the reservations in the *virtual line-up* saved into an ArrayList of RealTimeReservation objects and about the reservations of users inside the facility saved ArrayList of reservations.

The *schedule* of *planned reservations* in a supermarket is stored in a data structure defined as an HashMap with an integer as key, and an Arraylist of PlannedReservation objects as value.

The choice of setting Integer as key value could be ambiguous since the HashMap can be thought as two nested ArrayLists; actually each Integer key identifies a time slot included between the supermarket opening and closing time. Every *planned reservation* will be inserted in the bucket related to its starting time by means of an hashing function; doing that it is possible to retrieve all the reservations which have a particular starting time.

2.5.2 Interface diagram



```
<<Interface>>
RouterInterface
+forwardSignInRequest (userID : String, password : String) : Boolean
+forwardSignUpRequest (user : User) : Boolean
+forwardProfileUpdateRequest (user : User) : Boolean
+forwardRealTimeReservationCreationRequest (supermarketID : String, visitDuration : Int) : LocalDateTime
+forwardPlannedReservationCreationRequest (supermarketID: String, visitDuration : Int, startTime : LocalDateTime) : LocalDateTime
+forwardRealTimeReservationConfirmation (supermarketID : String, visitDuration : Int) : Boolean
+forwardPlannedReservationConfirmation (supermarketID : String, visitDuration : Int, startTime : LocalDateTime) : Boolean
+forwardReservationDetailsRequest (reservationCode : String) : Reservation
+forwardReservationDeletionRequest (reservationCode : String) : Boolean
+forwardAlternativeFacilityRequest (supermarketID : String, visitDuration : Int, startTime : LocalDateTime) : String[ ]
+forwardMapOpeningRequest ( ) : Map

<<Interface>>
KioskServiceInterface
+forwardRealTimeReservationCreationRequest (supermarketID : String, visitDuration : Int) : LocalDateTime
+forwardRealTimeReservationConfirmation (supermarketID : String, visitDuration : Int) : Boolean

<<Interface>>
ETEComputationInterface
+computeETE (supermarket : Supermarket, ticketNumber : Int, visitDuration : Int) : LocalDateTime

<<Interface>>
AuthenticationInterface
+checkUser (userID : String, password : String) : Boolean

<<Interface>>
RealTimeReservationManagerInterface
+getDetails (reservationCode : String) : Reservation
+showETE (supermarketID : String) : LocalDateTime
+putIntoVirtualLineUp (supermarketID : String, visitDuration : Int) : Boolean
+deleteRealTimeReservation (reservationCode : String) : Boolean

<<Interface>>
AccessManagerInterface
+informationAboutEntrance (supermarketID : String) : Int[ ]
+validateReservation (supermarketID : String, reservationCode : String) : Boolean
+consumeReservation (supermarketID : String, reservationCode : String) : Boolean

<<Interface>>
AccountInterface
+addUser (user : User) : Boolean
+updateUser (user : User) : Boolean

<<Interface>>
PlannedReservationManagerInterface
+getDetails (reservationCode : String) : Reservation
+checkAvailability (supermarketID : String, visitDuration : Int, startTime : LocalDateTime) : LocalDateTime
+proposedAlternativeSupermarkets (supermarketID : String, visitDuration : Int, startTime : LocalDateTime) : ArrayList<Supermarket>
+insertIntoTimetable (supermarketID : String, visitDuration : Int, startTime : LocalDateTime) : Boolean
+deletePlannedReservation (reservationCode : String) : Boolean

<<Interface>>
MapsServiceInterface
+getMapOfSupermarkets (position : Position) : Map

<<Interface>>
AdministratorInterface
+authenticate (adminID : String, password : String) : Boolean
+addSupermarket (name : String, position : Position, maxCapacity : Int, weeklySchedule : LocalDateTime[ ][ ]) : Boolean
+removeSupermarket (supermarketID : String) : Boolean
+updateSupermarket (supermarketID : String, name : String, position : Position, maxCapacity : Int, weeklySchedule : LocalDateTime[ ][ ]) : Boolean
+getAllSupermarkets ( ) : ArrayList<Supermarket>

<<Interface>>
AlternativesSearcherInterface
+searchForClosestSupermarkets (supermarketID : String, visitDuration : Int, startTime : LocalDateTime, radius : Double) : String[ ]
+searchForFirstAvailableSchedule (supermarketID : String, visitDuration : Int, startTime : LocalDateTime) : LocalDateTime

<<Interface>>
DataManagerInterface
+fetchUserCredentials (userID : String) : User
+fetchReservationFromDB (reservationCode : String) : Reservation
+fetchSupermarketFromDB (supermarketID : String) : Supermarket
+fetchAllSupermarketsFromDB ( ) : ArrayList<Supermarket>
+insertUserIntoDB (user: User) : Boolean
+updateUserIntoDB (user: User) : Boolean
+insertSupermarketIntoDB (supermarket : Supermarket) : Boolean
+updateSupermarketIntoDB (supermarket : Supermarket) : Boolean
+removeSupermarketFromDB (supermarketID : String) : Boolean
```

2.5.3 Interface description

- *RouterInterface*: this interface includes methods that carry out the forwarding and redirection of Client-side requests through the Application Server environment. Its methods are:

    - *ForwardSignUpRequest* and *ForwardSignInRequest*: these methods make respectively use of *AccountInterface* and *AuthenticationInterface* in order to send out a request of subscription realization and validation of an user authentication.
    - *ForwardUpdateProfileRequest*: this method make use of *AccountInterface* in order to send out a request of profile information update.
    - *ForwardMapOpeningRequest*: this method makes use of *MapServiceInterface* to call the method that provides user with the map of all supermarkets.
    - *ForwardRealTimeReservationCreationRequest* and *ForwardPlannedReservationCreationRequest*: these methods are designated to call the specific methods for fulfilling a *real-time reservation* or a *planned reservation* request of creation. User inputs the duration of the visit, the start time (if *planned*) and the chosen supermaket through a given form. After this, Client is provided with a LocalDateTime object indicating the ETE (if *real-time*) or the start time (if *planned*). The latter would represent an alternative *schedule* proposal, if the supermarket reached maximum capacity.
    - *ForwardAlternativeFacilityRequest:* this method calls the specific functions providing Client with the possibility of choosing an alternative facility in which he can book a *planned reservation* at the same selected *schedule*, in case the alternative *schedule* does not fulfil the request.
    - *ForwardRealTimeReservationConfirmation* and *ForwardPlannedReservationConfirmation*: these methods are invoked when an user decides whether to confirm or cancel the operation of creating a reservation. In the former case all the arguments are set to the corresponding values, in the latter all are set to null values.
    - *ForwardRealTimeReservationDetailsRequest* and *ForwardPlannedReservationDetailsRequest*: these methods make use of *RealTimeReservationInterface* and *PlannedReservationInterface* to fetch details of a given reservation and provides Client with them.
    - *ForwardRealTimeReservationDeletionRequest* and *ForwardPlannedReservationDeletionRequest:* these methods are invoked when a Client-side reservation deletion request is received. They call all the proper methods to remove from the database the reservation which the given code belongs to.

- *RealTimeReservationInterface*: this interface includes methods that are used to create, delete or retrieve details of a *real-time reservation*. Its methods, that make use of *DataManagerInterface* and *ETEComputationInterface,* are:

    - *GetDetails:* this method provides details, such as visit duration, ticket number and ETE, concerning the *real-time reservation* with the provided code.

- *PutIntoVirtualLineUp*: this method calls *fetchSupermarketFromDB* for retrieving information of the selected supermarket, mainly the *virtual line-up*. After that it creates a new *RealTimeReservation* object, puts it into the *virtual line-up* of this supermarket. Then it calls *updateSupermarketIntoDB* to insert the supermarket with the updated *virtual line-up* on the database.
- *ShowETE:* this method retrieves the ETE of a *real-time reservation* during the process of booking, calling *computeETE* method, and then it provides that with the user.
- *DeleteRealTimeReservation*: this method is designated to remove a *real-time reservation* from the system, as well as from the corresponding *virtual line-up*.

- *PlannedReservationInterface*: this interface includes methods used to create, delete or retrieve details or alternative proposals for a *planned reservation*. Its methods, that make use of *DataManagerInterface* and *AlternativesSearcherInterface,* are:

  - *GetDetails*: this method provides details, such as visit duration and start time, about the *planned reservation* with the given code.
  - *CheckAvailability*: this method is designated to verify the possibility of booking a *visit* in the selected *schedule* and it returns an alternative *schedule* option if the booking cannot be fulfilled in that time.
  - *ProposedAlternativeSupermarkets*: this method returns a list of alternative facilities in which an user can book the *visit*, in case the proposed alternative *schedule* does not fit his request.
  - *InsertIntoTimeTable*: this method is called once a *planned reservation* creation request has been confirmed. It retrieves the timetable of the given supermarket, in which it inserts a new *PlannedReservation* object created basing on the specified visit duration and start time.
  - *DeletePlannedReservation*: this method is designated to remove a *planned reservation* from the system, as well as from the corresponding timetable.

- *AlternativeSearcherInterface*: this interface includes a method to provide user with alternatives in case his request cannot be satisfied. This method, using *MapServiceInterface* and *DataManagerInterface*, is:

  - *SearchForClosestSupermarkets*: this method retrieves from the database the selected supermarket, then it searches for and returns the closest facilities on the map, basing on its position, within a certain radius (tipically a default value).

- *AccessManagerInterface*: this interface includes methods to manage accesses to each facility, provided to *Scanner* and *Display*. Its methods, that makes use of *DataManager*, are:

  - *ValidateReservation*: this method is designated to check whether the reservation with the specified code is allowed to enter in the given supermarket, or not.
  - *InformationAboutEntrance*: this method returns a data structure with some details, useful for the Display, about the entrance of a given supermarket. For instance, it

provides maximum capacity, number of people inside the facility and last ticket number allowed to enter.
    - *ConsumeReservation*: this method is called once a reservation has been scanned at the exit. It deals with removing such reservation from the database and, eventually, from *doingGroceryShopping* list.

- *DataManager*: this interface includes methods that act as intermediates between all Application Server interfaces and database. All methods are clearly assumed to communicate with the database through queries. Its methods are:

    - *FetchUserCredentials*: this method returns an user object containing his credentials. It returns null value if no such record is found.
    - *FetchReservationFromDB*: this method returns a *Reservation* object corresponding to the one with the specified code. It returns null value if no such record is found.
    - *FetchSupermarketFromDB*: this method returns a Supermarket object concerning the supermarket with the specified ID. It returns null value if no such record is found.
    - *FetchAllSupermarketsFromDB*: this method returns a list containing all supermarket registered to the system.
    - *InsertUserIntoDB*: this method sends an INSERT query to add the given user into the database.
    - *UpdateUserIntoDB*: this method sends an UPDATE query to edit the given user information into the database.
    - *InsertSupermarketIntoDB*: this method sends an INSERT query to add the given supermarket into the database.
    - *RemoveSupermarketFromDB*: this method sends a DELETE query to remove the supermarket with the specified ID from the database.

- *ETEComputationInterface*: this interface includes a method to provide user with the ETE of its reservation, either during its creation phase or already existing. Its method is:

    - *ComputeETE*: this method is designated to obtain details concerning the status of the given supermarket (such as reservations in the *virtual line-up*, people doing grocery shopping and *scheduled* visits) in order to compute the ETE of the reservation referring to the provided ticket number. In case this method is called during a *real-time reservation* creation (i.e. ticket number unknown), *ticketNumber* parameter is set to “-1” value, and the ETE would be computed as the reservation was the last in the *virtual line-up*.

- *MapsServiceInterface*: this interface includes a method to provide user with the map of the supermarkets belonging to the system, useful to search either for a facility nearby the user position or for an alternative in case of unsatisfied requests. Its method, that makes use of *DataManagerInterface,* is:

    - *GetMapOfSupermarkets*: this method makes use of GoogleMapsAPI services to retrieve a map section focused on the given position. Then, it obtains a list of all supermarkets in order to put some markers (containing all information about stores) over their positions on the map, and return that to the user.

- *KioskServiceInterface*: this interface acts as a “personal *Router*” for the *TicketGenerator*. Its methods manage requests, that come from this device, calling the proper methods of Application Server interfaces. In particular:

    - *ForwardRealTimeReservationCreationRequest:* this method is designated to call the specific methods for fulfilling a *real-time reservation* via *TicketGenerator*.
    - *ForwardRealTimeReservationConfirmation:* these methods are invoked when an user decides whether to confirm or cancel the operation of creating a reservation.

- *AccountInterface*: this interface includes a method to carry out all subscription procedures. Its method, that make use of *DataManagerInterface*, is:

    - *AddUser*: this method takes care of the execution of all input sanitization procedures (in addition to the ones already done by the Client-side functions), and the call of *insertUserIntoDB* in order to register a user to the system.
    - *UpdateUser*: this method takes care of managing user data, and, in particular, it allows to edit his information and preference, calling *updateUserIntoDB* from *DataManagerInterface*.

- *AuthenticationInterface*: this interface includes a method to validate all authentication procedures. Its method, that make use of *DataManagerInterface*, is:

    - *CheckUser*: this method takes care of the execution of all input sanitization procedures (in addition to the ones already done by the Client-side functions), and the call of *fetchUserCredentials* in order to retrieve and check the truthfulness of the login.

- *AdministratorInterface*: this interface includes methods that allow the system administrator to execute maintenance operations without having to access directly to the database. Its methods, that make use of *DataManagerInterface*, are:

    - *Authenticate*: this method allows an administrator to login and access to the system maintenance functionalities. Like *checkUser* method, it calls *fetchUserCredentials* method, but it will return a positive response only if such user is also an administrator.
    - *AddSupermarket:* this method creates a new Supermarket object with the given attributes and calls *insertSupermarketIntoDB* passing that object as parameter.
    - *RemoveSupermarket*: this method allows an administrator to delete an existing supermarket from the database, calling *removeSupermarketFromDB* method, passing the given ID as parameter.
    - *UpdateSupermarket:* this method updates the specified Supermarket object with the given attributes and calls *updateSupermarketIntoDB* passing that object as parameter.

- *GetAllSupermarkets*: this method retrieves the list of all supermarkets belonging to the system, calling *fetchAllSupermarketsFromDB* method, and returns that list.

2.6 Selected architectural styles and patterns

The system is developed by using a multitier architecture which derives from the more traditional client-server architectural style. In particular, a **four-tier architecture** seems to be the best choice among all the available ones. Thanks to the subdivision of the system in different parts, the application acquires scalability and maintainability, due to the possibility of substituting or modifying only one tier without changing the whole system.

Physical partition of a software application is usually associated to a logical division of the system at code level. For this reason, the system is based on a **multi-layered architecture** also from the logical point of view.

This kind of patterns suggest a physical separation of logical layers on different platforms. The software is divided into three independent modules (Presentation, Application and Data) running on several tiers:

- user interface is represented by the Web Server and Client devices (Client and Web tiers);
- business logic is managed by the Application server (Business tier);
- data are persistently warehoused on the DBMS and controlled by Database Server (Data tier).

Multitier architecture is strictly related to an architectural pattern known as Model-View-Controller. MVC pattern main purpose is the detachment of UI (represented by the view) from data (model). In order to achieve this goal, every change on the model should not involve the view, from the other side, changes at the user interface level should not affect persistent data.

This software application adopts a variant of the Model-View-Controller called **Model-View-Adapter pattern**. MVA intends to solve the problem MVC also wants to fulfil, but it suggests a different approach referring to the interaction among the three layers of the architecture. Indeed, the main difference between MVC and MVA concerns the architectural geometry:

- MVC has a triangular structure, because view and model can exchange information indirectly through the controller, but some operation put them directly in communication without passing from the middle, for instance view can get updates directly from the model;
- **MVA** proposes a linear structure, so information is prevented from passing outside the controller (in this case called *adapter*) which is always involved as an essential mediator, since model is oblivious of view and view is not aware of model presence.

Choice of adopting MVA could appear strict, but this tied communication makes the view completely decoupled from the model. This represents a big advantage, because independency among different layers eases debugging, testing and evaluation of the several units of the software; moreover, as well as MVC, it ensures maintainability and reusability of the code.

Focusing on the interaction between Client and Application Server, as we have already seen in the "Component view" section, the usage of a component serving as a dispatcher between the first two layers is fundamental in this software. Router is the component designated to play this role: it is able to handle request coming from the Client side and then to redirect them towards the proper Application Server component.

In order to highlight Router main task, **Façade design pattern** is applied. This structural pattern suggests the usage of one component acting as a front-facing interface ("façade") charged of concealing other components. By using the Router as a Façade, Client is provided with a very simple interface and he is allowed to call general methods without taking care of directly forwarding requests to the right components. From the opposing point of view Router deals with User requests sending them out to the designated components of the Application Server. Façade pattern is very useful when a simple interface is required to access a complex system, it makes the software more readable, reusable and loosely coupled because of the less number of associations among components, moreover it eases the system changes which not involve the Client layer.

For the implementation, some design patterns are recommended to use (see "Class Diagram"):

- **Factory Method:** it is a creational design pattern that might be useful in the instantiation process of an object *Reservation*. It defines an interface to create objects but let the subclasses (*RealTimeReservation* and *PlannedReservation*) decide which type to instantiate basing on a certain input.
- **Bridge Pattern:** from the structural point of view this pattern decouples an abstraction (class *Reservation*) from its methods, giving the responsibility of their implementation to other classes (*RealTimeReservation* and *PlannedReservation*) which extends it using an inheritance mechanism.

2.7 Other decisions

For what concerns the mobile application environment, the choice of a Rich Mobile Application (RMA) seems more suitable for the system. RMA derives from the Rich Internet Application, but, instead of charging the Application layer with all the logic part, it transfers some of it in the Presentation layer. This choice is made to allow mobile application to work offline in some situations, for example, retrieving all the reservation details from the proper section, downloading the QR code, or notifying user if his turn is coming. For the latter case, the system performs computations basing on the last computed ETE when the mobile was online. Such operations could be performed offline, in order to prevent people from entering the supermarkets if they have not previously downloaded the QR code.

2.8 Algorithms

This algorithm can be considered the core of the application. It is charged with the responsibility of checking the status of a given supermarket and, basing on this, to provide user with an *expected time of entrance* corresponding to his *real-time reservation*, either of a new or already existing one.

```
computeETE (S : Supermarket, ticketNumber : Numeric, visitDuration : Numeric) : LocalDateTime {

      now <- LocalDateTime.now()

      queuePosition <- 0

      if ticketNumber < 0 then
             queuePosition <- S.getVirtualLineUp().size()

      else for i <- 0 to S.getVirtualLineUp().size() - 1 do
             if (S.getVirtualLineUp().get(i).getTicketNumber() = ticketNumber)
                    queuePosition <- i

      precSchedules <-
       S.getTimetable().getAllScheduledNotScanned(now, now.plusMinutes(visitDuration))

      allowedEntrances <-
             S.maximumCapacity() – S.getDoingGroceryShopping().size() - precSchedules.size()

      if queuePosition < allowedEntrances
             return LocalDateTime.now()

      dim <- new Numeric
      precReservation <- new Reservation
      ETE <- new LocalDateTime

      do {
             precList <- new List<Reservation>

             for each res in S.getDoingGroceryShopping() do
                    precList.add(res)

             for each res in precSchedules do
                    precList.add(res)

             for i <- 0 to queuePosition - 1 do
                    precList.add(S.getVirtualLineUp().get(i))

             dim <- precSchedules.size()

             sortByExitTime(precList)

             precReservation <- precList.get(queuePosition – allowedEntrances)

             ETE <- precReservation.getExitTime()

             precSchedules <- S.getTimetable().getAllScheduledNotScanned(
                                  now,ETE.plusMinutes(visitDuration))

      } while dim < precScheds.size()

      return ETE

 }
```

Before the explanation, it is necessary to keep in mind that the *expected time of entrance* is NOT a fixed attribute of a *real-time reservation*, instead its value needs to be re-computed each time we want to know it, due to the fact that the status of a supermarket continuously changes.

Another necessary clarification is that this algorithm is recursively called in an implicit way. In fact, *sortByExitTime* method needs to know the exit time of all reservations on which it is called, which is given by:

- Scanned (*planned* or *real-time*) reservation: entranceTime + visitDuration
- Not-scanned *planned reservation*: startTime + visitDuration
- Not-scanned *real-time reservation*: ETE + visitDuration

So, for each not-scanned *real-time reservation* (previous to the given one), *computeETE* must be recursively called.

A control on *ticket number* is initially done. If the *ticket number* passed is "-1" it means that the reservation is on its creation phase, so it will be considered as the last in the *virtual line-up*. After this, the number of *real-time reservations* allowed to enter is calculated. The reservations on *precSchedules* array are those *scheduled* for the current time slot and those having a *schedule* that overlaps the *visit* duration of considered reservation.

Then, if the reservation is among those allowed to enter, the returned ETE will be the current time. Else, a *precList* array is filled with all reservations that have a precedence with respect to the considered one, and then ascendently ordered by exit time.

The expected reservation that will leave its place to the considered one is that which is in *precList* in a position corresponding to *queuePosition* (not considering those reservations that are already allowed to enter, thus *queuePosition - allowedEntrances*), so the ETE will be its exit time.

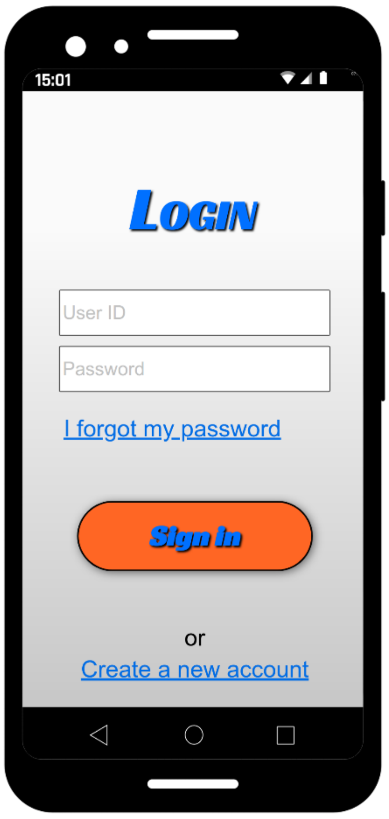
However, the just found ETE might not be the definitive one. In fact, it is necessary to verify that in such time interval there are no *scheduled* reservations that have not been considered yet. In order to do so, *precSchedules* dimension is saved into variable *dim* to compare it with the size of the new computed one. If the new one has a greater size, then there are some new *planned reservations* to consider in our computation, and the cycle must be re-done.

After exiting the loop, the ETE is returned.

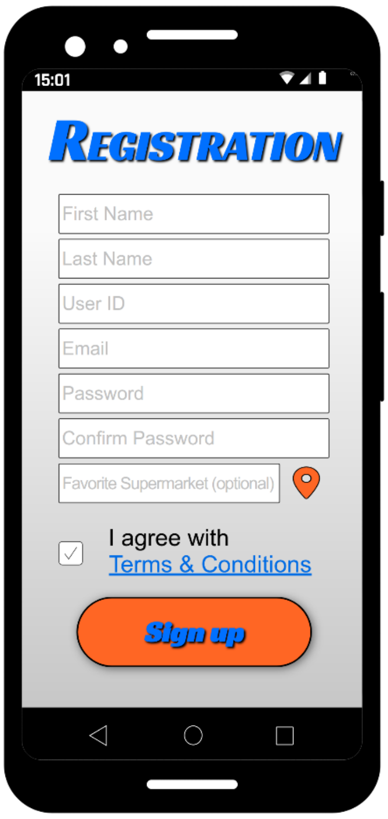
3 User Interface Design

In this section the interfaces which the user interacts with are represented by means of mockups and a User Experience diagram that represents all the screen templates in which the navigation of the user can go.

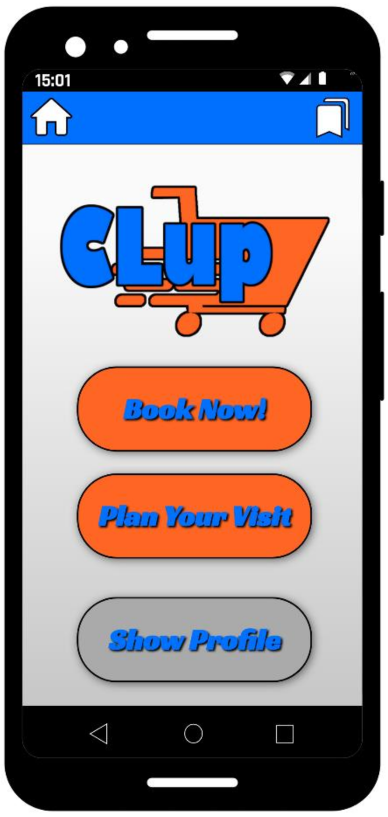
3.1 Mockups



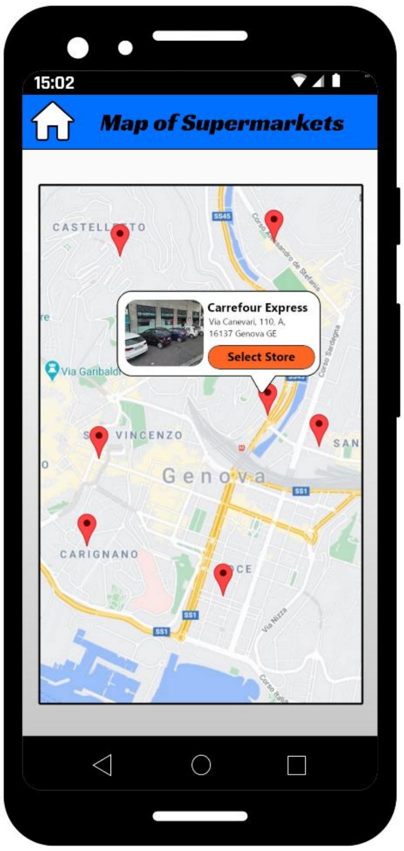
Login



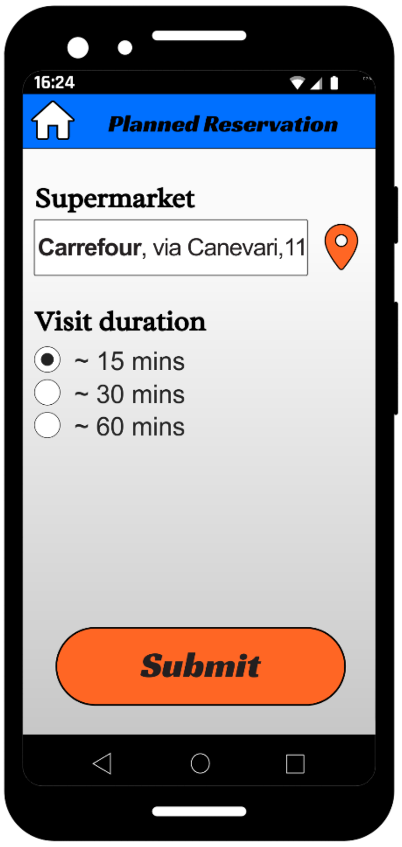
Registration



HomePage



Map of Supermarkets



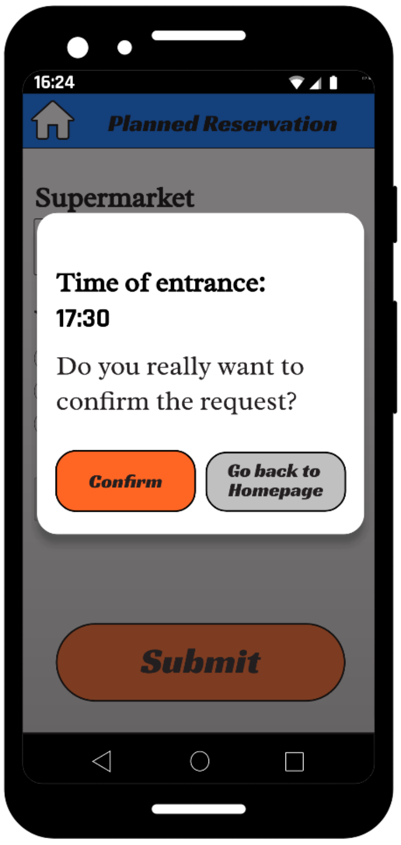
Supermarket

Carrefour, via Canevari,11

Visit duration

(●) ~ 15 mins
( ) ~ 30 mins
( ) ~ 60 mins

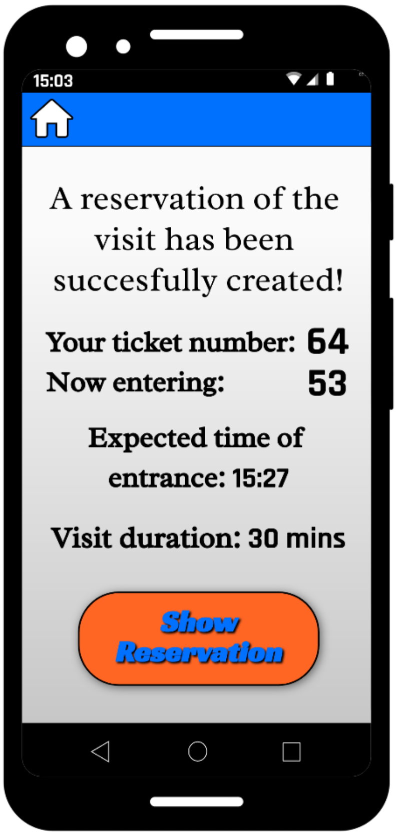
Real-Time Reservation Form



Time of entrance:
17:30

Do you really want to confirm the request?

Real-Time Reservation Confirmation



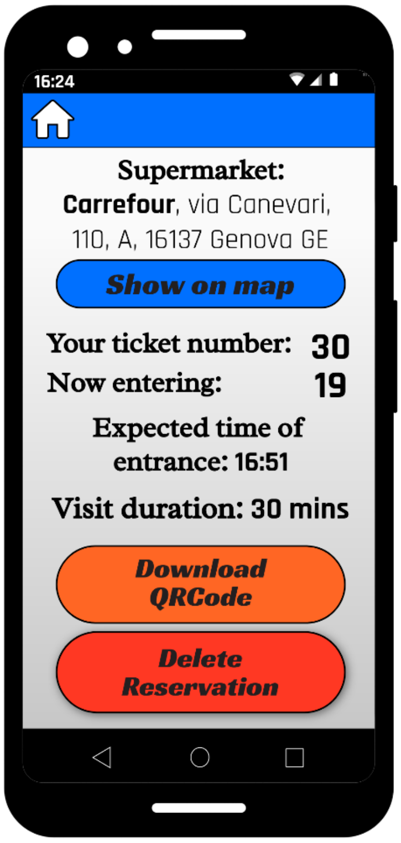
A reservation of the visit has been succesfully created!

Your ticket number: 64
Now entering: 53

Expected time of entrance: 15:27

Visit duration: 30 mins

Real-Time Reservation Created



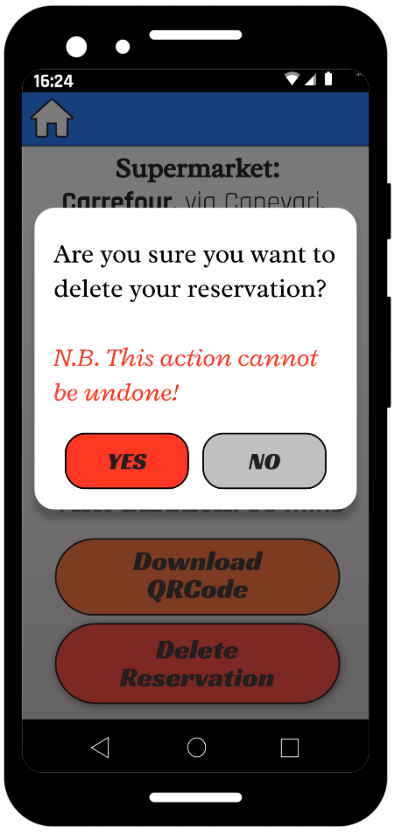
Supermarket:
Carrefour, via Canevari, 110, A, 16137 Genova GE

Your ticket number: 30
Now entering: 19

Expected time of entrance: 16:51

Visit duration: 30 mins

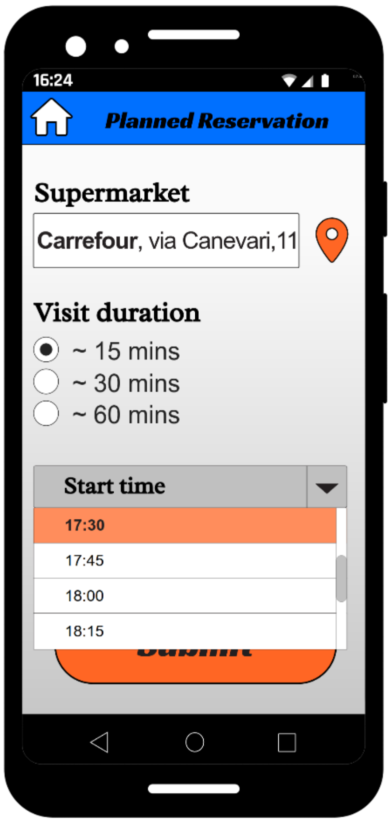
Real-Time Reservation Details



Are you sure you want to delete your reservation?

N.B. This action cannot be undone!

Real-Time Reservation Deletion



Supermarket

Carrefour, via Canevari,11

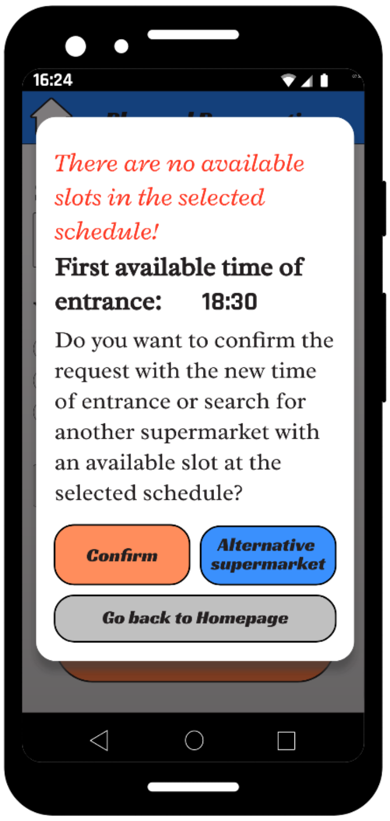
Visit duration

● ~ 15 mins
○ ~ 30 mins
○ ~ 60 mins

Start time

17:30
17:45
18:00
18:15

Planned Reservation Form

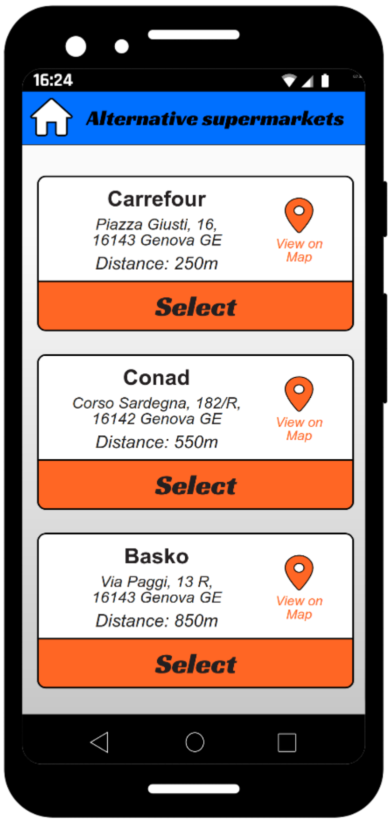


There are no available slots in the selected schedule!

First available time of entrance: 18:30

Do you want to confirm the request with the new time of entrance or search for another supermarket with an available slot at the selected schedule?

Alternative Schedule Proposal

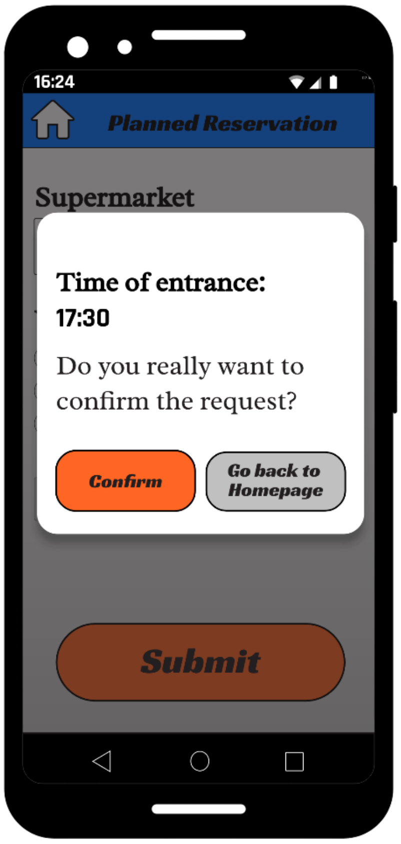


Carrefour
Piazza Giusti, 16, 16143 Genova GE
Distance: 250m

Conad
Corso Sardegna, 182/R, 16142 Genova GE
Distance: 550m

Basko
Via Paggi, 13 R, 16143 Genova GE
Distance: 850m

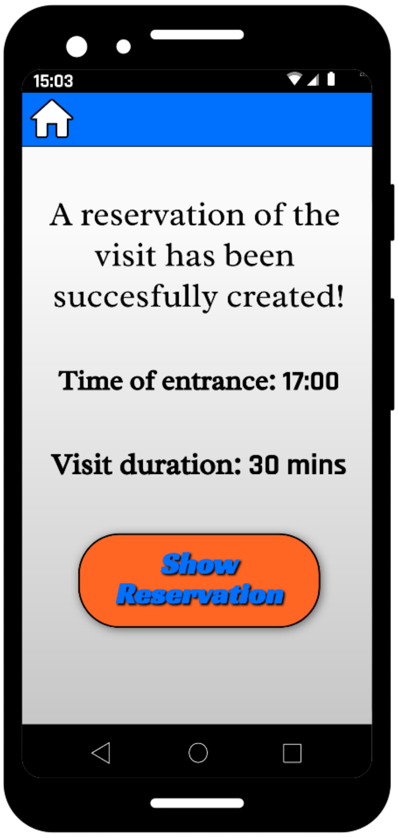
Alternative Supermarkets Proposal



Time of entrance: 17:30

Do you really want to confirm the request?

Planned Reservation Confirmation

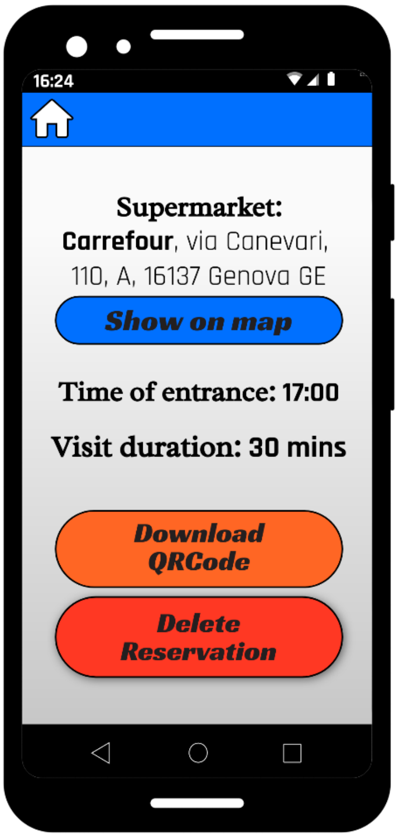


A reservation of the visit has been succesfully created!

Time of entrance: 17:00

Visit duration: 30 mins

Planned Reservation Created

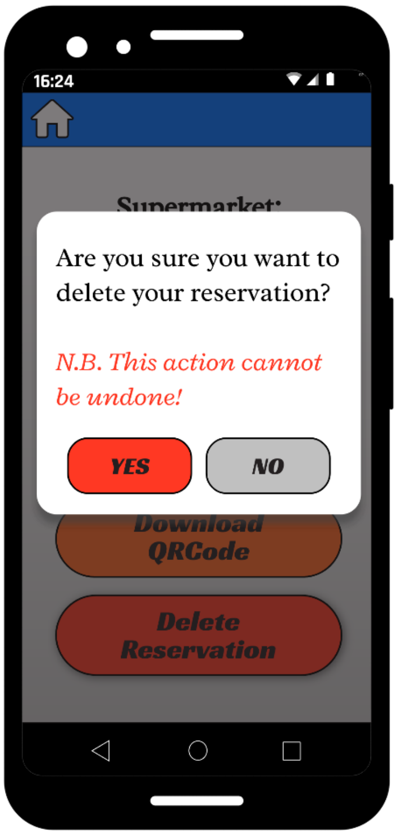


Supermarket:
Carrefour, via Canevari, 110, A, 16137 Genova GE

Time of entrance: 17:00

Visit duration: 30 mins

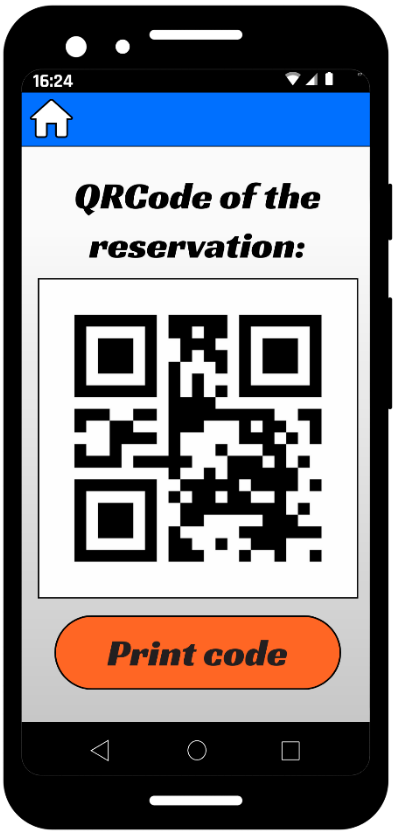
Planned Reservation Details



Are you sure you want to delete your reservation?

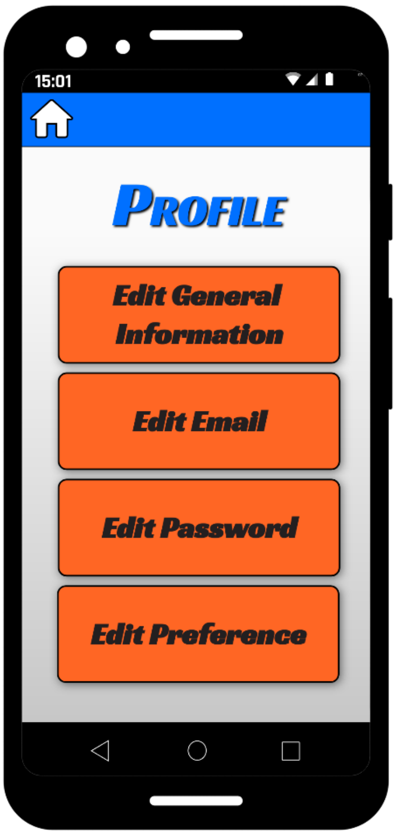
N.B. This action cannot be undone!

Planned Reservation Deletion

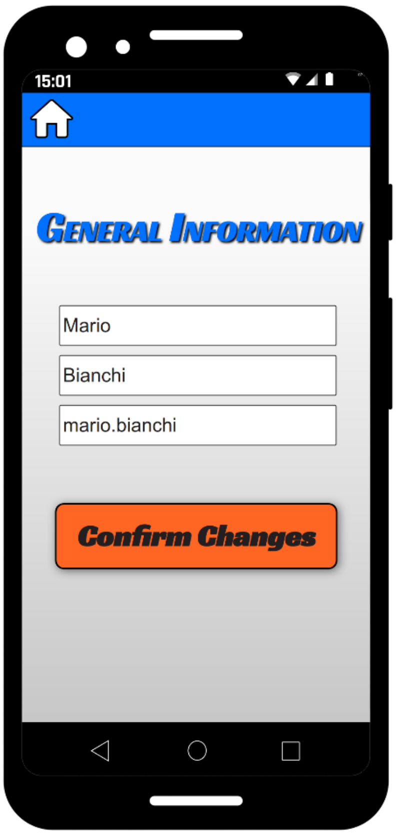


QRCode of the reservation:

QRcode



Show Profile

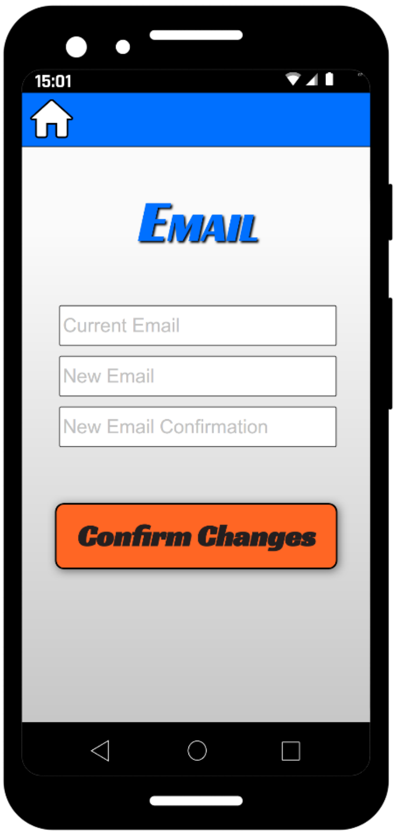


Mario

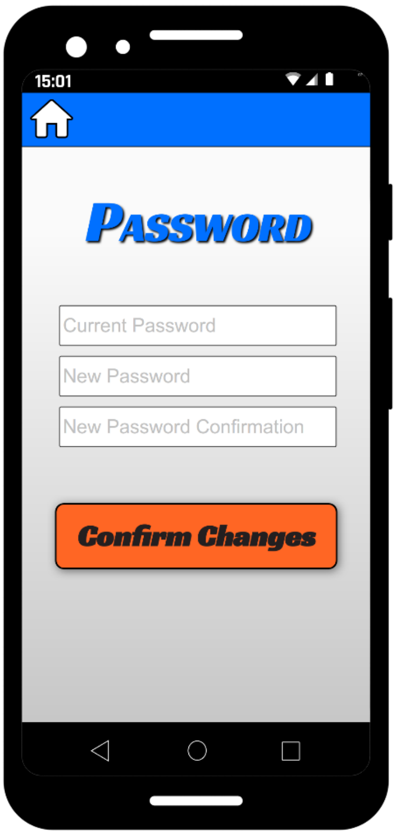
Bianchi

mario.bianchi

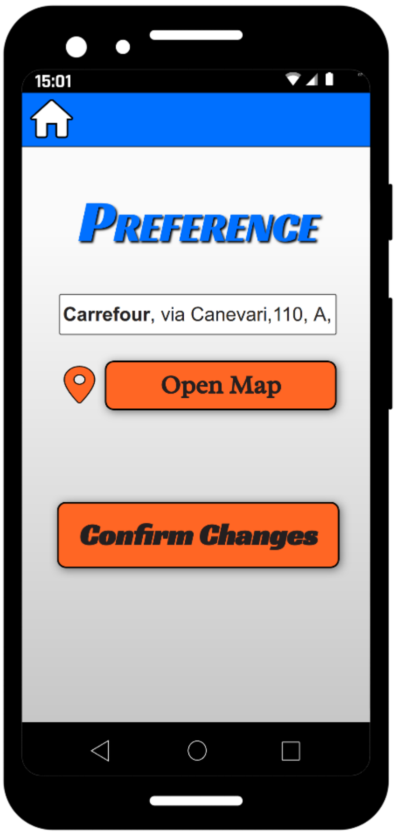
Edit General Information



Edit Email

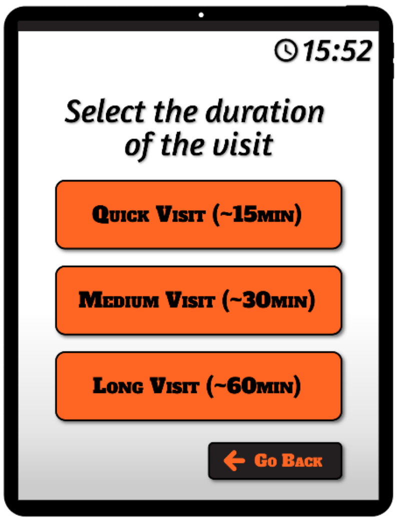


Edit Password

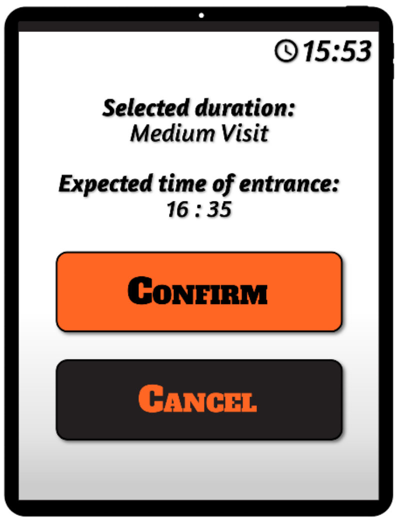


Carrefour, via Canevari,110, A,

Edit Preference



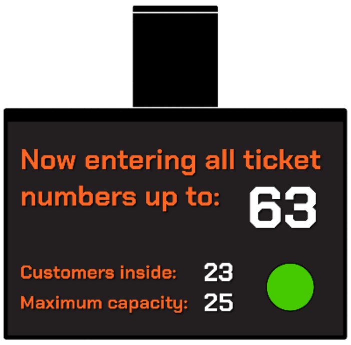
Ticket Generator – Visit Duration



Selected duration:
Medium Visit

Expected time of entrance:
16 : 35

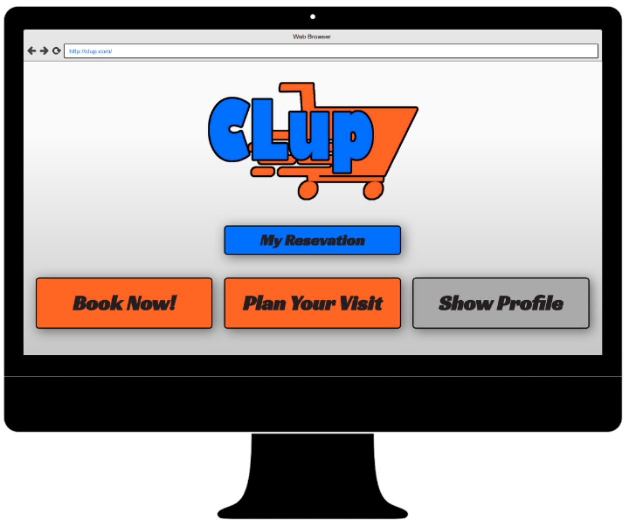
Ticket Generator – Confirmation



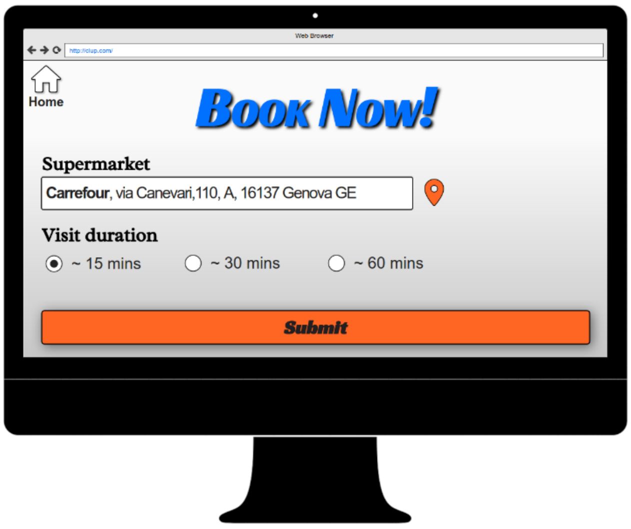
Now entering all ticket numbers up to: 63

Customers inside: 23
Maximum capacity: 25

Display



WebApp - Homepage

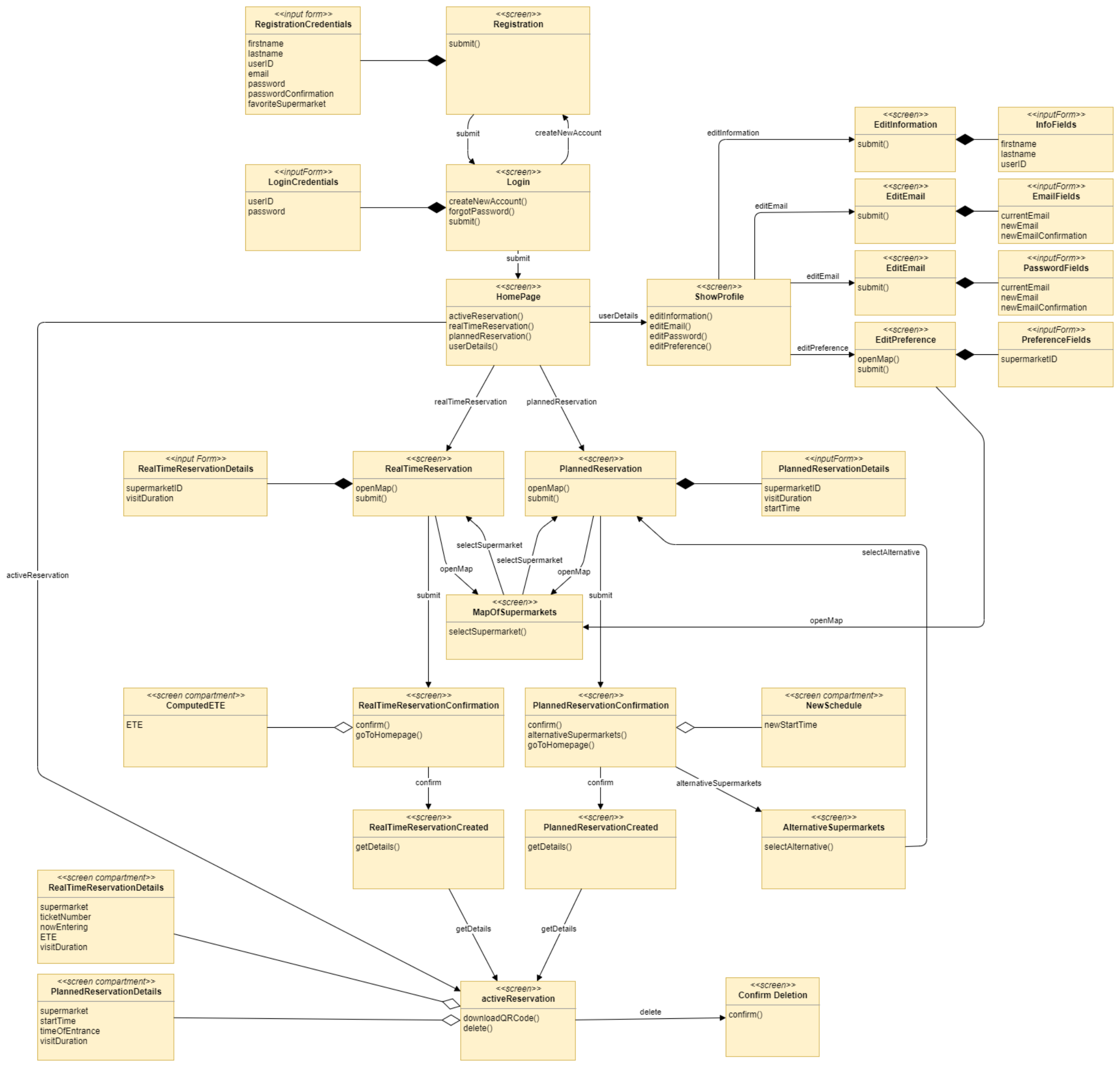


Supermarket
Carrefour, via Canevari,110, A, 16137 Genova GE

Visit duration
(● ~ 15 mins) ○ ~ 30 mins ○ ~ 60 mins

WebApp – Real-Time Reservation Form

3.2 UX diagram



4 Requirements Traceability

This chapter lists all the components that play a role for the satisfaction of requirements that are exposed in the RASD. The only not mentioned component is AdministratorService, since the administrator figure is assumed to exist, but it does not affect specifications and requirements of the *software-to-be*.

[**R1**]: System must assign a unique code to every reservation.

- ✓ PlannedReservationManager
- ✓ RealTimeReservationManager
- ✓ DataManager

[**R2**]: System should produce a QR code for each reservation based on its unique code.

- ✓ PlannedReservationManager
- ✓ RealTimeReservationManager
- ✓ DataManager

[**R3**]: System must assign a *ticket number* to every *real-time reservation*.

- ✓ RealTimeReservationManager
- ✓ DataManager

[**R4**]: System must unlock the sliding doors if and only if the reservation code scanning gives a positive outcome.

- ✓ AccessManager
- ✓ DataManager

[**R5**]: System must lock sliding doors right after Client entrance.

- ✓ AccessManager

[**R6**]: A reservation code scanning must give a positive outcome only if such reservation concerns the supermarket Client is attempting to enter.

- ✓ AccessManager
- ✓ DataManager

[**R7**]: A code scanning of a *real-time reservation* must give a positive outcome only if the store has not reached its maximum capacity.

- ✓ AccessManager
- ✓ DataManager

[**R8**]: A code scanning of a *real-time reservation* must give a positive outcome only if the corresponding reservation number is among those of the *virtual line-up* authorized to enter, since their entrance does not make the store exceed the maximum capacity.

- ✓ AccessManager
- ✓ DataManager

[**R9**]: A code scanning of a *planned reservation* must give a positive outcome only if the actual time is included in the interval between the starting time of its *schedule* and the *maximum tolerated delay*.

- ✓ AccessManager
- ✓ DataManager

[**R10**]: System must keep track of the number of Clients within the store.

- ✓ AccessManager
- ✓ DataManager

[**R11**]: System should provide Client, having a *real-time reservation*, with *expected time of entrance*.

- ✓ Router
- ✓ AccountService
- ✓ AuthenticationService
- ✓ RealTimeReservationManager
- ✓ WaitingTimeComputationService
- ✓ DataManager
- ✓ KioskService

[**R12**]: System could notify User, who made a reservation, that his turn is coming.

- ✓ Router
- ✓ AccountService
- ✓ AuthenticationService
- ✓ DataManager
- ✓ PlannedReservationManager
- ✓ RealTimeReservationManager

[**R13**]: System should allow the *line* to flow anyway, even if one of Clients next in the *queue* did not arrive within the *maximum tolerated delay*.

- ✓ AccessManager
- ✓ DataManager

[**R14**]: System must validate a *real-time reservation* request only if it occurs during the hours of operation of the chosen supermarket.

- ✓ Router
- ✓ WaitingTimeComputationService
- ✓ RealTimeReservationManager
- ✓ DataManager
- ✓ KioskService

[**R15**]: System must validate a *planned reservation* request if and only if at least one slot is available in the chosen store for the selected *schedule*.

- ✓ Router
- ✓ PlannedReservationManager
- ✓ DataManager

[**R16**]: System must forbid User from making a reservation if he has got another active one.

- ✓ Router
- ✓ PlannedReservationManager
- ✓ RealTimeReservationManager
- ✓ DataManager
- ✓ AccountService
- ✓ AuthenticationService

[**R17**]: System must propose making a *planned reservation* starting from some hours later the current time.

- ✓ Router
- ✓ PlannedReservationManager
- ✓ DataManager

[**R18**]: System should detect an alternative *schedule* for doing grocery shopping in the chosen store if the selected one is not available.

- ✓ Router
- ✓ AlternativeSearcherService
- ✓ PlannedReservationManager
- ✓ DataManager

[**R19**]: System could find the closest stores to the chosen one, in which the selected *schedule* is available.

- ✓ Router
- ✓ PlannedReservation
- ✓ MapService
- ✓ DataManager

[**R20**]: System must deallocate the slot occupied by a reservation when its *schedule* expires, or Client terminates his *visit,* or he does not scan his reservation within a certain *maximum tolerated delay*.

- ✓ AccessManager
- ✓ DataManager

	[R1]	[R2]	[R3]	[R4]	[R5]	[R6]	[R7]	[R8]	[R9]	[R10]
Router										
AccessManager				✓	✓	✓	✓	✓	✓	✓
PlannedReservationManager	✓	✓								
RealTimeReservationManager	✓	✓	✓							
AlternativeSearcherService										
WaitingTimeComputationService										
DataManager	✓	✓	✓	✓		✓	✓	✓	✓	✓
AccountService										
AuthenticationService										
MapService										
KioskService										

	[R11]	[R12]	[R13]	[R14]	[R15]	[R16]	[R17]	[R18]	[R19]	[R20]
Router	✓	✓		✓	✓	✓	✓	✓	✓	
AccessManager			✓							✓
PlannedReservationManager		✓			✓	✓	✓	✓	✓	
RealTimeReservationManager	✓	✓		✓		✓				
AlternativeSearcherService								✓		
WaitingTimeComputationService	✓			✓						
DataManager	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
AccountService	✓	✓				✓				
AuthenticationService	✓	✓				✓				
MapService									✓	
KioskService	✓			✓						

5 Implementation, integration and test plan

5.1 Implementation plan

Implementation and integration of the sub-components of the software-to-be, regarding the Application Server, should follow the bottom-up approach, starting from the modules that are at lower levels, thus, those components that are independent from the others.

For instance, *WaitingTimeService* might be the first, starting from the implementation of the algorithm for the *expected time of entrance* computation. At the first stage, also *DataManager* could be implemented because its interaction with the Database Server involves every component. The implementation of these two modules could be performed in parallel since they are not linked each other.

Having done that, we can move to the second level where there are all the components that require only *DataManagerInterface* or *ETEComputationInterface* or both.

Meantime *AccessManager* could be implemented by means of a message-based interface through which *Scanner* and *Display* can extract information, about the reservation for the first device and supermarket status information for what concerns the second one.

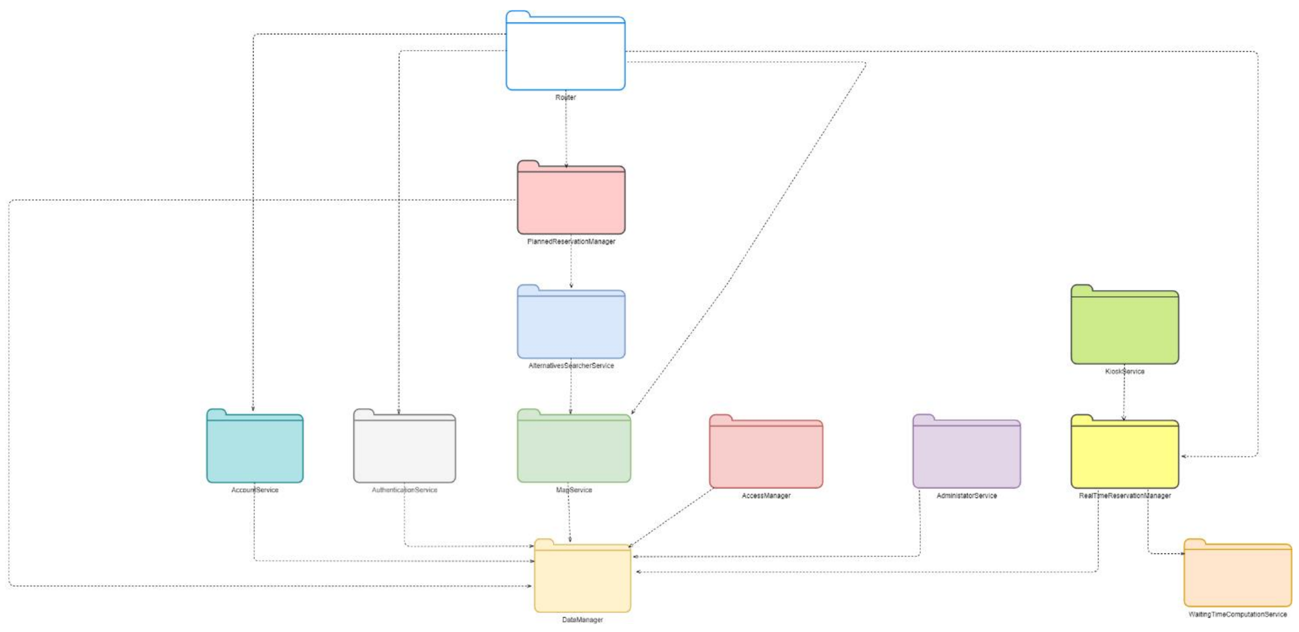
Another component that could be implemented in parallel in this phase is *RealTimeReservationManager*: it provides the services through which users can make a *real-time reservation* by mobile or web application.

Other components to implement at the second level are *AuthenticationService*, *AccountService*, *AdministratorService* and *MapService*, because, assuming that external services as Google Maps are reliable, the latter does not require other interfaces.

At the third level, there are *AlternativesSearcherService*, that relies on *MapService,* and also *KioskService,* that requires *RealTimeReservationManager* to handle the requests made by kiosks.

At the fourth level, *PlannedReservationManager* is ready to be implemented: the choice to implement the two components of *ReservationManager* at two different stages derives from the fact that functions of the second one relies on many more components than *RealTimeReservationManager* functions.

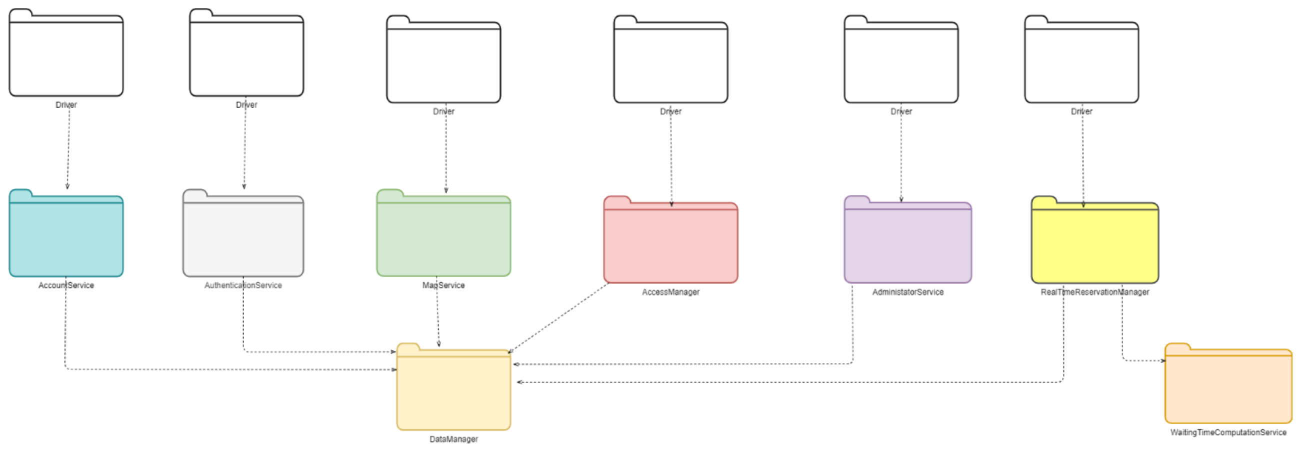
Router is implemented at the last stage, when all the other modules have already been integrated and implemented. Below there is a picture that represents the hierarchy process explained so far, including all the levels and all the components.


Router
PlannedReservationManager
AlternativesSearcherService
KioskService
AccountService
AuthenticationService
MapService
AccessManager
AdministatorService
RealTimeReservationManager
DataManager
WaitingTimeComputationService

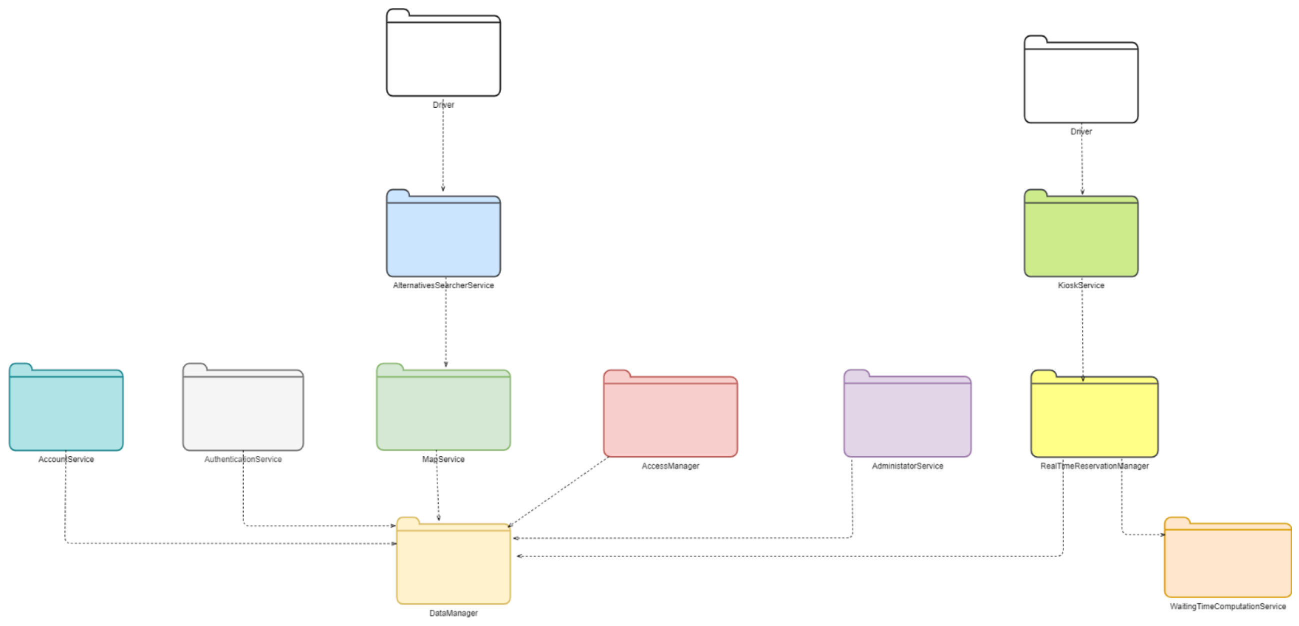
5.2 Integration strategy

The following section explains the strategy which the process of integration of all the components refers to. In this case, bottom-up approach may be helpful, in order to make the debugging more effective and to discover any type of errors as soon as possible, by proceeding hand in hand with the implementation.

Once the first two components, *DataManager* and *WaitingTimeComputationService*, are implemented, all the components that are at the second level of the implementation plan might be integrate in this subsystem, after having unit tested them with drivers that replace higher levels components or external ones, as shown in the picture.



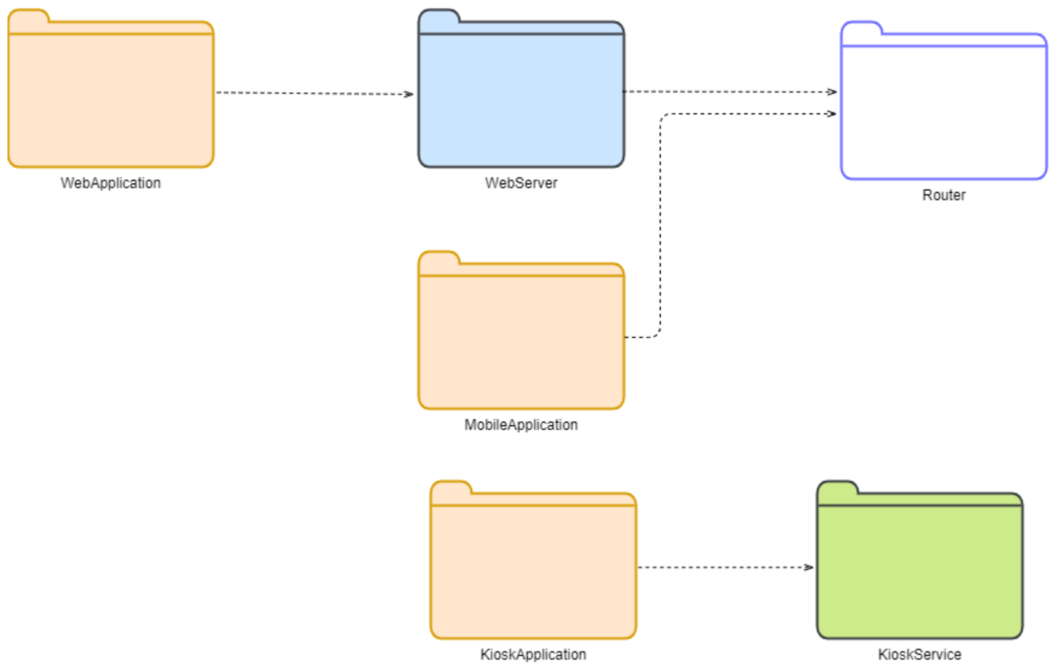
Therefore, integration strategy goes through *AlternativesSearcherService* and *KioskService* components, which are unit tested and added to the subsystem:



Finally, the integration of *PlannedReservationManager* and therefore *Router* terminate the procedure for what concerns the Application Server.

A further step is the integration of external components, for instance GoogleMaps service, which *MapService* deals with, and the integration of all the client-side applications, such as:

- The web application with the WebServer that deals with *Router*.
- The kiosk application that interacts with *KioskService*.
- The mobile application that, as the WebServer, requires the router interfaces in order to access the software functionalities.



5.3 System testing

Once the system is completely integrated, it needs to be entirely tested to verify whether functional and non-functional requirements are satisfied. The process of testing the whole system should go through different types of tests that are listed below:

- **Functional testing**: it verifies whether the non-functional and functional requirements described in the RASD document are fulfilled or not.
- **Stress testing**: it may happen that the system faces a failure of some components during its lifecycle, this type of test checks if the recovery time of the system can be tolerated.
- **Performance testing**: this test concerns the software performances, so the response time that affects some components; this type of test can suggest which version of the product is more reliable.
- **Load testing**: it exposes bugs such as memory leaks, mismanagement of memory, buffer overflows and it identifies upper limits of components.

The enviroment in which tests take place must be similar to the real enviroment, because there are a lot of interactions with physical devices to be tested, since they play an essential role in the system.

6 Effort spent

Immordino Alessandro

Introduction	1h
Architectural design	29 h
Requirements traceability	3h
Implementation, Integration and Test Plan	6h
UI design and Others	5h
Total	44h

Polvanesi Giacomo

Introduction	1.5h
Architectural design	33h
Requirements traceability	3h
Implementation, Integration and Test Plan	3h
UI design and Others	2.5h
Total	43h

Pala Riccardo

Introduction	1h
Architectural design	27.5h
Requirements traceability	1h
Implementation, Integration and Test Plan	1h
UI design and Others	13h
Total	43.5h