DD
Design Document

POLITECNICO
MILANO 1863

ANDREA MANGLAVITI
DAVIDE MARINARO
LUCA MARINELLO

Version 1.0
Date 07/01/2021
Prof. Matteo Giovanni Rossi

Contents

1. Introduction

1.1 Purpose

The purpose of this document is to explain more technically and in a more detailed way the software discussed in the RASD document. This will represent a robust manual for the programmers that will develop this software application.

In this DD we present hardware and software architecture of the system in terms of components and interactions among them. Moreover, this document defines a collection of design characteristics needed for a correct implementation that guarantees standard quality attributes.

Finally, it also gives a detailed presentation of the implementation plan, integration plan and the testing plan. Overall, the main different features listed in this document are:

- The high-level architecture of the system
- Main components of the system
- Interfaces provided by the components
- Design patterns adopted
- User interfaces designed for clients
- Requirement traceability
- Implementation, integration, and test plan

Reading this document would be very useful for stakeholders who want to understand the main characteristics of the software to be and, consequently, all the choices made to offer its functionalities satisfying the quality of requirements.

1.2 Scope

The main goals of CLup application are on one hand helping users not to wait forming long lines outside the store if it is full and, on the other hand, allowing store manager to regulate accesses of people to their store, analysing data collected by ticket machines outside supermarkets. Furthermore, CLup offers interesting functionalities to registered customers, such as the possibility to book visits to go only to specific areas of the supermarket, or even to receive notification acting as suggestion to go shopping in the favourite period of the day. This is just an overview of the main functionalities offered by the software: more detailed information can be found on the RASD document.

This application will have a strong impact in avoiding crowds next to the supermarkets during this difficult period we are living, due to COVID-19 pandemic.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

Customer	A person that has to buy something at the supermarket
User	A person with a smartphone that downloaded the application and uses it
Non-User	A person without a smartphone or that do not have the application nor uses it
Valid Entrance	A ticket or visit that allows the customer (if visit, only user) to enter the supermarket
[General] Entrance	Term used to refer to both visits and tickets, pointing out their common attributes instead of their differences

1.3.2 Acronyms

RASD	Requirement Analysis and Specification Document
DD	Design Document
GPS	Global Positioning System
QR Code	Quick Response Code
REST	REpresentational State Transfer
API	Application Programming Interface
RAC	Real Application Clusters
RACS	Reliable Array of Cloned Servers
OLTP	On-Line Transaction Processing
OLAP	On-Line Analytical Processing

1.3.3 Abbreviations

CLup	Customer Line-Up (name of the application)
Gn	Goal number n
Rn	Requirement number n

1.4 Revision History

11/12/2020	DD of CLup is created
24/12/2020	First Version of DD (draft)
07/01/2021	Last Version of DD

1.5 Reference Documents

- Specification Document: "CLup - Customer Line-up Optional Project Assignement.pdf"
- Slides of the lectures

1.6 Documentation Structure

Chapter 1 describes the scope and purpose of the DD, including the structure of the document and the set of definitions, acronyms and abbreviations used.

Chapter 2 shows the different views of the program, especially showing the deployment view, the component view, the patterns implemented and the architectural styles, the Runtime view and the how the different components are connected between themselves through their interfaces.

Chapter 3 shows how the user interface should be on the mobile application for customers and the user interface of the WebApp for supermarket managers.

Chapter 4 focuses on the requirements traceability. It shows the requirements needed to fulfil a goal and the design components needed by the requirements to achieve the goal. Finally, it illustrates how the requirements defined in the RASD are mapped onto the design components.

Chapter 5 identifies the schedule to follow for the chosen strategy to implement, integrate and test all the components, highlighting when and how all the components are implemented, integrated with each other and tested to verify the correctness of the desired behaviour.

Chapter 6 shows the effort spent for each member of the group.

Chapter 7 includes the reference documents.

2. Architectural Design

2.1 Overview

We decided to develop the application architecture using the 3 layers logic, structured in the following way:

- Presentation layer (P): This layer function is to handle the interactions with the user. It presents all the interfaces needed to communicate with the user, in such a way that will result simple for him/her to use all the functions of the application
- Business logic or Application layer (A): It manages the data flows between Presentation Layer and Data access layer. Moreover, it hosts the logic of the application and its functions.
- Data access layer (D): This layer function is to store data generated by the Application Layer and Presentation Layer. In Addition, this layer manages data stored to satisfy Application Layer's queries.

The Client-Server style is the most appropriate architecture for this software. Communication between these two components performs through the middle tier component.

The first step is to allow the client to invoke a method to access the desired functionality, like creating a ticket or booking a visit. Then, the server receives the request, analyses it and finally dispatches the correct service. Finally, we report a brief list of the functionalities the server can provide:

- Create a ticket: the client (user) selects the supermarket; the server analyses the line-up in the selected supermarket and if the waiting time is too long, it suggests alternatives. The client decides whether to keep the selected one or change it with an alternative (if the above case is detected). Then finishes inserting required information and sends the request; The server collects data received and sends back the QR code with its information.
- Book a visit: the client (user) inserts some data; the server collects data from database about the client and estimates an expected time for the visit, if client's data are enough; Client accepts or refuses the proposed time (if the above case is detected) and in case of refusal inserts time preferred; the server receives this information and checks availability of the slot required. If the slot is not free the server sends back alternative slots for the same supermarket or same slot for different supermarkets; the client, in this case, selects an alternative; the process ends with receiving QR code and its information.
- Monitor the access: the ticket machine collects data of QR codes scanned in entrance and exit and sends them to the server; Supermarket manager can request to server to send those data and monitor the accesses.

- **Send notification:** the client application requests server to infer data in order to find a slot among the ones the user usually uses to visit the supermarket; server takes the proper information from the database and sends results back to client application; finally, the client application verifies whether those slots fit the current day and hour, and, in affirmative case, it sends a notification to the user.

This is a distributed application and in the below section we describe the 3-tier architecture.

2.1.1 High Level Components

For the distributed application we decide to use the Three-Tier architecture because adding the middle tier enables to achieve a better separation between the client and the server, since the middle tier:

- Centralizes connections to the data server, reducing costs and guaranteeing a good level of security and control of accesses
- Masks data model to the clients, guaranteeing a good level of data integrity
- Can be replicated to scale up, simplifying this process

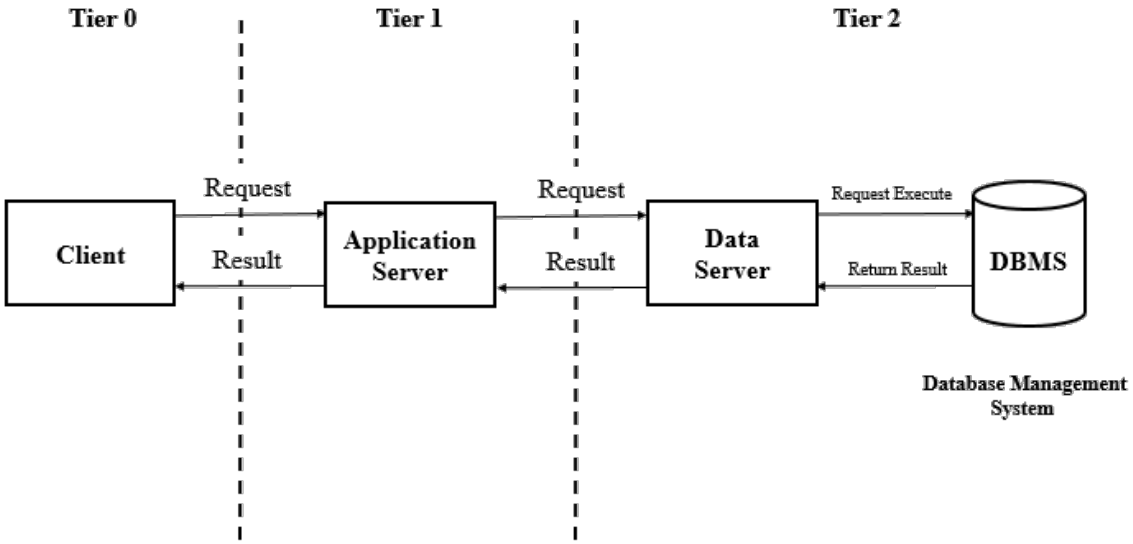


Figure 1 - Three-Tier architecture

Going more in-depth in this analysis of the high-level components, the following picture will help understanding how the internal components behave and communicate between each other.

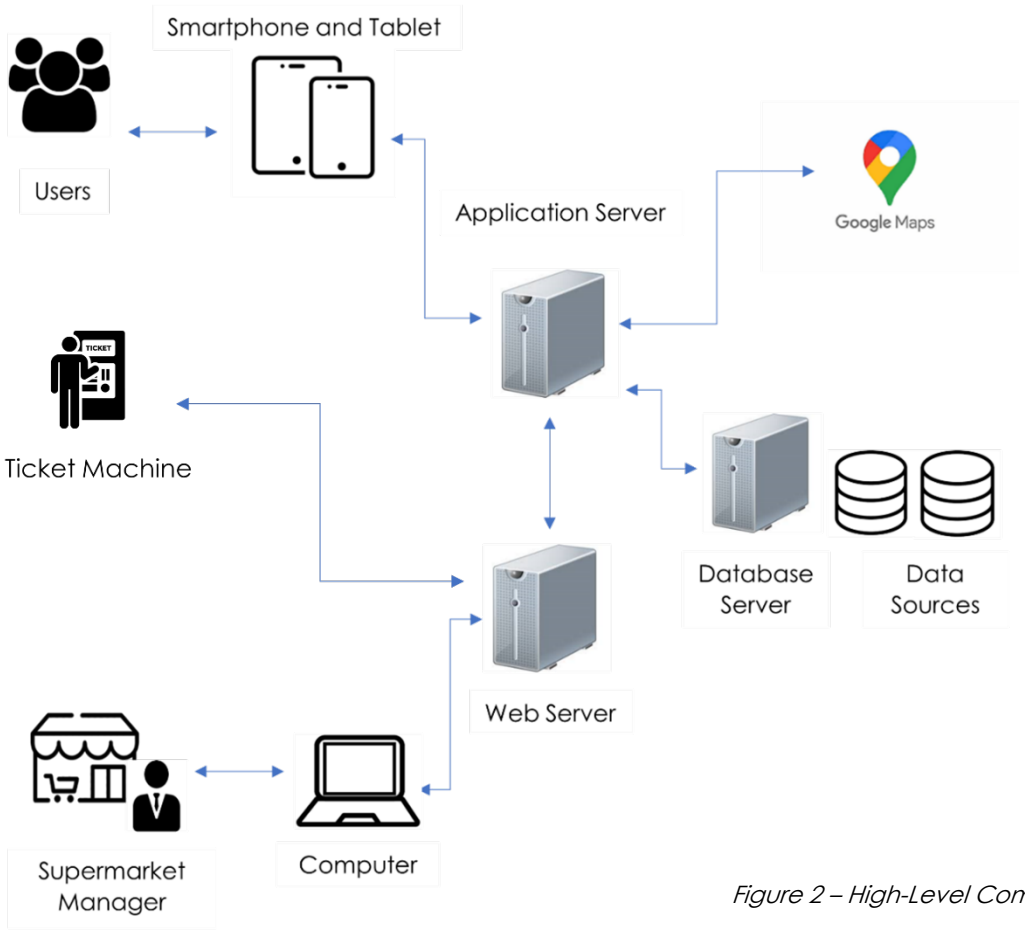


Figure 2 – High-Level Components

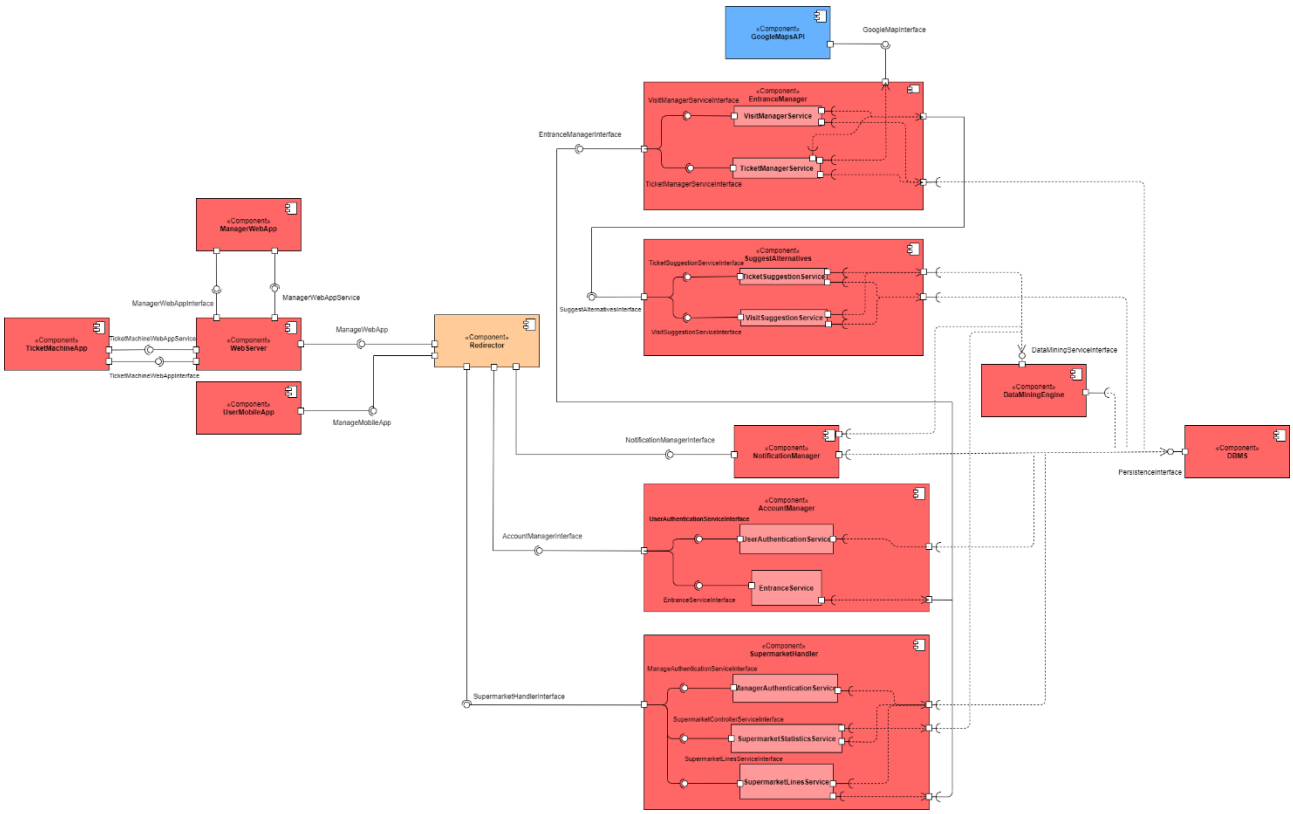
As the picture shows, client devices can be smartphones or tablets (for customers) and a computer for supermarket managers: this distinction is due to the fact that customers need a portable device provided with internet connection and GPS sensors, instead supermarket managers only need services that can be requested directly from a WebApp and, therefore, from a computer. Furthermore, the ticket machine, that validates entrances and distributes them to unregistered customers, is the last device of the first tier; for the sake of simplicity, it uses a web app interface.

The second tier is made of two servers: the application server that manages the connection of clients belonging to customers (smartphones and tablets), and the web server that manages the connection of supermarket managers and ticket machines (webapp). These two servers communicate between each other and, moreover, the first one is responsible for using the API that enables the usage of Google Maps inside the application.

Finally, the third tier is made of the database server (DBMS) that manages data of the entire system. The DBMS communicate with the application server to offer all the desired functionalities.

2.2 Component View

The following picture represents the component diagram of the system, focusing on the internal structure of the application server, that holds the business logic of the software. As the picture well highlights, every message form and to clients (both mobile and web application) is caught by the redirector, that acts as the postman of the network. The picture also shows other components besides the application server, with the only purpose of highlighting how the interaction happens.



Hereafter it is reported a detailed overview on each component:

- **Redirector:** it is the core of the server because every request made by clients passes through this component. Indeed, it contains a mapping of every service offered to the corresponding component able to handle it. It is the communication keeper between clients and data server and, consequently, data sent back to clients is also dispatched by this useful component
- **AccountManager:** this component manages the accesses and the functionalities dedicated to customers of the supermarket chain: indeed, it includes two sub-components, dealing with the following services:
    1. **UserAuthenticationService:** this service manages the functionalities of login and signup to authenticate the user and make him able to exploit all other dedicated functionalities.
    2. **EntranceService:** this service manages the functionalities of creating, checking, and deleting an entrance. It is important to notice that this

component is not the one completing the requests, but it redirects the requests to the EntranceManager component.

- EntranceManager: this component manages all the functionalities related to entrances and it is directly connected with the previous component, which calls the services required by the user. Since there are two types of entrances, this component is divided in two sub-components:
  1. VisitManagerService: this first sub-component is the responsible of managing visits and contains a collection of algorithms appointed to book, check, and delete a visit, considering the specificity of Visits.
  2. TicketManagerService: this second sub-component is the responsible of managing tickets and contains a collection of algorithms appointed to book, check, and delete a visit, considering the specificity of Tickets. Moreover, this sub-component exploits the functionalities offered by GoogleMapsAPI (to calculate the time spent in reaching the supermarket).
- SuggestAlternatives: this component extends the functionalities of the previous one, implementing one of the advanced functionalities of the application, that is suggesting alternatives when the selected supermarket is not available. For the sake of homogeneity, also this component is divided into two sub-components, one dedicated to tickets and one dedicated to visits, TicketSuggestionSerivce and VisitSuggestionService, respectively.
- NotificationManager: this component implements the advanced functionality of notifying users when one of his usual slots is available and recommend a visit to the supermarket. Therefore, it includes methods to inquire whether there is an active entrance or not and, also, a set of methods directly connected to the DataMiningEngine, which helps inferring the usual slots.
- SupermarketHandler: this component is the counterpart of AccountManager one, but dedicated to supermarket managers; indeed, it contains methods that allows managers to authenticate (through the sub-component ManagerAuthenticationService) and methods appropriate to build statistics and get useful information about accesses into his/her supermarket (through the sub-component SupermarketStatisticsService). It is important to notice that this last sub-component uses data mining services offered by the following component. Finally, there is another sub-component (called SupermarketLinesService) whose purpose is to update the lines of the corresponding supermarket using data form the ticket machine that scans every QR code before entrance and after exit; for this reason, it has methods that directly use the DBMS interface.
- DataMiningEngine: this component offers many services of data mining to other components and acts as an assistant to deliver the functionalities of the application. It has different purposes:

1. Infer data to find more advantageous alternatives during the creation of an entrance, sending back the results to EntranceManager
2. Infer data to find which are the usual slots of a user and send back this information to NotificationManager
3. Analyse data to present statistics to managers, sending the information required back to the sub-component SupermarketStatisticsService, that will organize them for the final target

- DBMS: this component allows every other component of the architecture to access data stored in databases; indeed, the interface provided contains a set of methods dedicated to store, retrieve, update, delete and insert data into databases from different components. Every other component that contains the business logic of the software has interfaces to interact with the DBMS.
- GoogleMapsAPI: this last component is an external one, that provides useful services to complete the process of creation of a ticket and the continuous update of active tickets, providing users information on when leave to go to the supermarket arriving in time with the selected means of transportation.

2.3 Deployment View

Combining the information of sections number 2.1 and 2.2, the chosen configuration of the architecture will include the Fat Client, in which part of the business logic of the system is embedded into the client: this happens, above all else, on the customers' side of the software, where other than requesting services and sending information, there are also included functionalities to verify the correctness of data sent or to visualize active entrances stored in the local device. For the sake of homogeneity, also on the managers' side the selected implementation includes the Fat Client, implementing functionalities to control accesses with the ticket machine.

One on hand, one of the advantages offered by this configuration is the increase in performances in communication speed due to the less frequent request of data, that are directly stored in the cache of local devices; on the other hand, obviously, this implementation requires validation algorithms to check the veracity data stored in the cache, invalidating those that are older than a certain time threshold.

Finally, as stated in the RASD document, the middle tier (web server and application server) should be duplicated following the pattern of a reliable array of cloned servers. There are two main advantages:

- The workload can be balanced between multiple servers through a load balancer, increasing performances and allowing more users to use the app.
- If one of the servers goes down, there is the other that can replace it, completing extra tasks.

Since the redirector is also a core part of the architecture, it should be replicated too, to continue guaranteeing services and the correct functioning of the whole system.

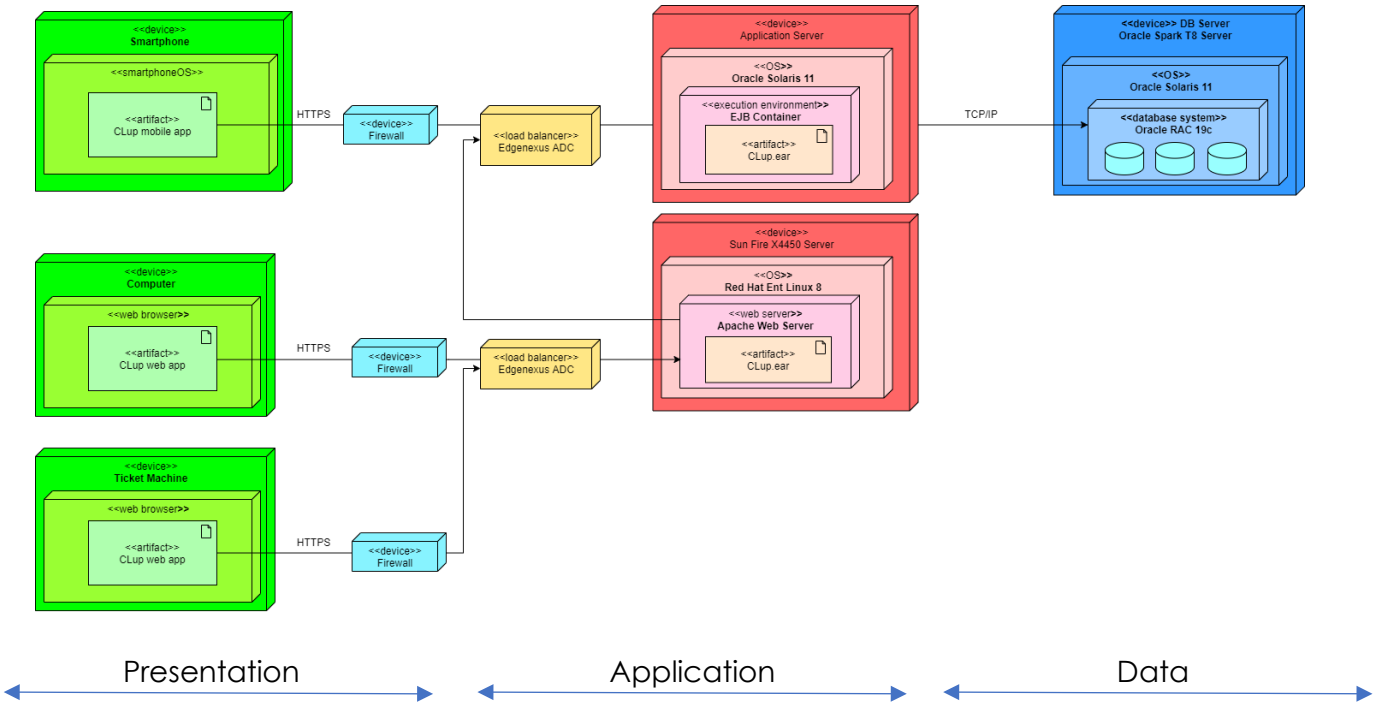


Figure 3 – Deployment Diagram

The Deployment View highlights in a more detailed way the three-tier architecture chosen for the system. A better explanation of each component and the choices made will follow:

- **Smartphone:** the appointed device that customers can use to reserve, check or delete tickets and book, check or delete visits. It also receives notification of available time slots for visits. It uses HTTPS protocol to communicate with the Application Server.
- **Computer:** the appointed device that managers can use to check information about their supermarket and analyse statistics about it. It uses HTTPS protocol to communicate with the Web Server.
- **Ticket Machine:** this devise acts as a controller between entrances and lines of supermarkets and has different goals; the first one is the scan of entrances before entry and after exit the supermarket; the second one is to create tickets and line-up unregistered customers. To fulfil these two goals, it uses an interface base on a WebApp and, therefore, it connects directly with the Web Server, using HTTPS protocol to communicate with it.
- **Firewall:** this device provides safety accesses to the internal network of the system blocking unwanted users, threats, and external attacks.
- **Load Balancer:** this device helps distributing the workload between multiple servers, providing benefits to the whole system such as increase capacity, reliability and avoiding overload issues.

- **Web Server:** this first type of server interacts with requests coming from the WebApp through the firewall and the load balancer and then dispatches those requests to the application server that contains the business logic to satisfy the request. The communication protocol establishes that those requests are implemented as java objects with a specific structure that includes headers to categorize them and the complete payload with specific data of the request. This device then redirects these messages to the application server with the correct format. The chosen operating system is, obviously a Linux-based one, due to its transparency and reliability. As regard the hardware, the Sun Fire X4450 has been chosen due to its flexibility and variety of compositions, each one guaranteeing high performances.
- **Application Server:** the second type of servers interacts with the mobile application through the firewall and the load balancer (analogous behaviour as web server with web app); furthermore, it has a direct connection with the web server from which it receives messages of web applications. It contains the majority of the business logic (since, as we stated before, a slice of it is contained into clients) that allows it to handle every received request, providing correct answer in an appropriate time. Web Server communication protocol also applies to this server. The chosen operating system is Oracle Solaris 11 because it is a very solid, simple to use and safe UNIX-based system and well integrates with other Oracle hardware and software solutions appointed for this architecture.
- **Database Server (DBMS):** the last part of this section is dedicated to one of the most important elements of the system, that represents the third tier of the architecture. The database server directly interacts with the Application Server, providing the business logic all needed data to satisfy requests from clients with a complete set of methods to retrieve, delete, insert and update it continuously. The chosen hardware is an Oracle SPARC T8 Server for its high performances and integrability with Java software and other Oracle components. As before, the operating system chosen is Oracle Solaris 11, both for the sake of homogeneity and thanks to its useful integration with hardware and, above all else, with the database management system. Indeed, the DBMS is Oracle RAC (Real Application Cluster) 19c, that harnesses the processing power of multiple interconnected servers, creating a robust computing environment. This is the preferred solution because, due to the introduction of load balancers that distribute the workload on multiple servers that have to work simultaneously on the same database, it can provide a parallel and synchronous access to datafiles. Moreover, it is very important to notice that RAC is always a Relational DBMS. This choice also guarantees:
    1. High performance due to the load balancing features well integrated with the parallelized access to data.
    2. High Availability because the system has well synchronized redundant components providing a consistent and uninterrupted service even during failures.

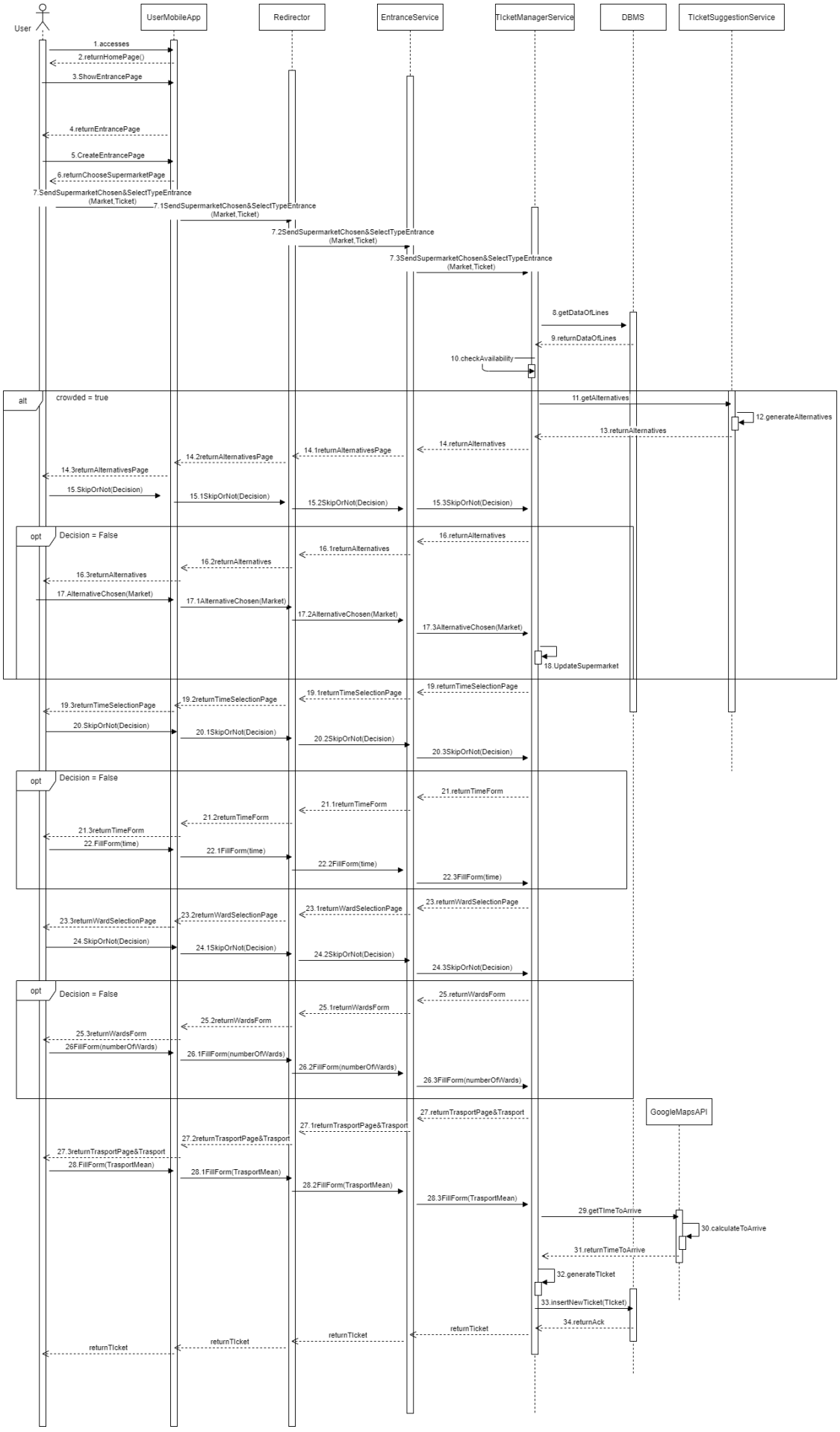
3. Scalability because it is very simple to add more machines since load balancers and Oracle software are designed with this purpose.
4. Resilience for the same reason of high availability: if one component fails, there are other redundant components that will continue guaranteeing the services, allowing to take down the broken machine and repair it.
5. Transparency since this relational environment is functionally equivalent to a single-instance one; this means that there are no differences between the code to manage this solution and the code to manage normal ones.

Finally, we can consider of extending Oracle RAC DBMS with Oracle Data Guard, an extension establishing and maintaining a secondary database (called standby), continuously updated alongside the primary one, that can be used in case of failure. This effective solution implements the concept of RACS (Reliable Array of Cloned Servers) for the third tier of our architecture.

2.4 Runtime View

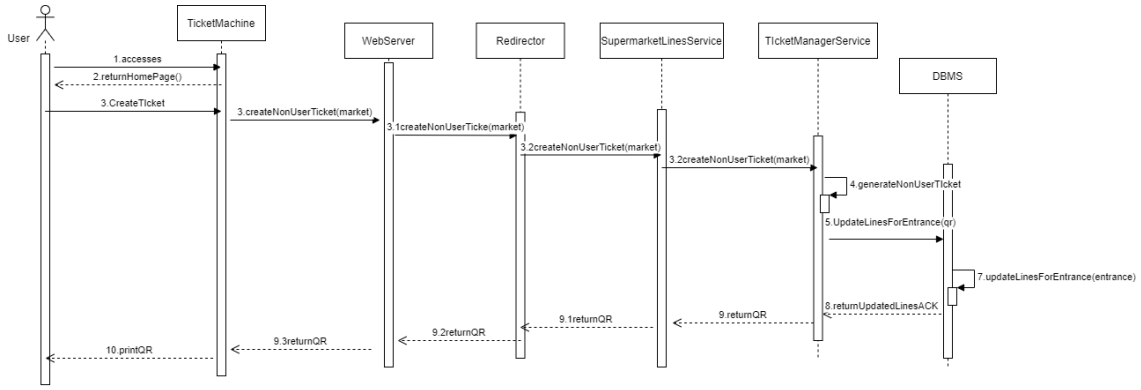
2.4.1 Create a Ticket

The Process described below, at runtime, is the creation of a ticket by a user. The first interaction is between the user and the UserMobileApp and is relative to the access to the home page of the application. After returning the Supermarket page, the Redirector starts to distribute all the requests from the UserMobileApp to the EntranceService, since it manages the phases of creating, checking, and deleting an entrance. Immediately after, EntranceService forwards all the information to the TicketManagerService, which will request and receive all the information it needs. However, after the selection of the supermarket, the TicketManagerService, through the DBMS, extracts the queue of users currently lined-up in the selected supermarket. After this extraction, the TicketManagerService checks if the line is too long. If so, the TicketManagerService asks the TicketSuggestionService to provide him some suggestion, that the user can choose to select or not. After the optional insertion of the time that the user expects to spend in the supermarket, and the number of wards he/she intends to visit, is required to insert the means of transportation to reach the supermarket. This information is fundamental to the proper management of the lines, since through the GoogleMapsAPI, the TicketManagerService can obtain the time that the user will spend to reach the supermarket, information needed to allow the user to understand when he should move. At last, when all the information to create has been collected by TicketManagerService, it will create the Ticket and send it to the DBMS to be stored in the database.


User
UserMobileApp
Redirector
EntranceService
TicketManagerService
DBMS
TicketSuggestionService
1.accesses
2.returnHomePage()
3.ShowEntrancePage
4.returnEntrancePage
5.CreateEntrancePage
6.returnChooseSupermarketPage
7.SendSupermarketChosen&SelectTypeEntrance (Market,Ticket)
7.1SendSupermarketChosen&SelectTypeEntrance (Market,Ticket)
7.2SendSupermarketChosen&SelectTypeEntrance (Market,Ticket)
7.3SendSupermarketChosen&SelectTypeEntrance (Market,Ticket)
8.getDataOfLines
9.returnDataOfLines
10.checkAvailability
alt
crowded = true
11.getAlternatives
12.generateAlternatives
13.returnAlternatives
14.returnAlternatives
14.1returnAlternativesPage
14.2returnAlternativesPage
14.3returnAlternativesPage
15.SkipOrNot(Decision)
15.1SkipOrNot(Decision)
15.2SkipOrNot(Decision)
15.3SkipOrNot(Decision)
opt
Decision = False
16.returnAlternatives
16.1returnAlternatives
16.2returnAlternatives
16.3returnAlternatives
17.AlternativeChosen(Market)
17.1AlternativeChosen(Market)
17.2AlternativeChosen(Market)
17.3AlternativeChosen(Market)
18.UpdateSupermarket
19.returnTimeSelectionPage
19.1returnTimeSelectionPage
19.2returnTimeSelectionPage
19.3returnTimeSelectionPage
20.SkipOrNot(Decision)
20.1SkipOrNot(Decision)
20.2SkipOrNot(Decision)
20.3SkipOrNot(Decision)
opt
Decision = False
21.returnTimeForm
21.1returnTimeForm
21.2returnTimeForm
21.3returnTimeForm
22.FillForm(time)
22.1FillForm(time)
22.2FillForm(time)
22.3FillForm(time)
23.returnWardSelectionPage
23.1returnWardSelectionPage
23.2returnWardSelectionPage
23.3returnWardSelectionPage
24.SkipOrNot(Decision)
24.1SkipOrNot(Decision)
24.2SkipOrNot(Decision)
24.3SkipOrNot(Decision)
opt
Decision = False
25.returnWardsForm
25.1returnWardsForm
25.2returnWardsForm
25.3returnWardsForm
26FillForm(numberOfWards)
26.1FillForm(numberOfWards)
26.2FillForm(numberOfWards)
26.3FillForm(numberOfWards)
GoogleMapsAPI
27.returnTrasportPage&Trasport
27.1returnTrasportPage&Trasport
27.2returnTrasportPage&Trasport
27.3returnTrasportPage&Trasport
28.FillForm(TrasportMean)
28.1FillForm(TrasportMean)
28.2FillForm(TrasportMean)
28.3FillForm(TrasportMean)
29.getTImeToArrive
30.calculateToArrive
31.returnTimeToArrive
32.generateTicket
33.insertNewTicket(TIcket)
34.returnAck
returnTicket
returnTIcket
returnTIcket
returnTIcket

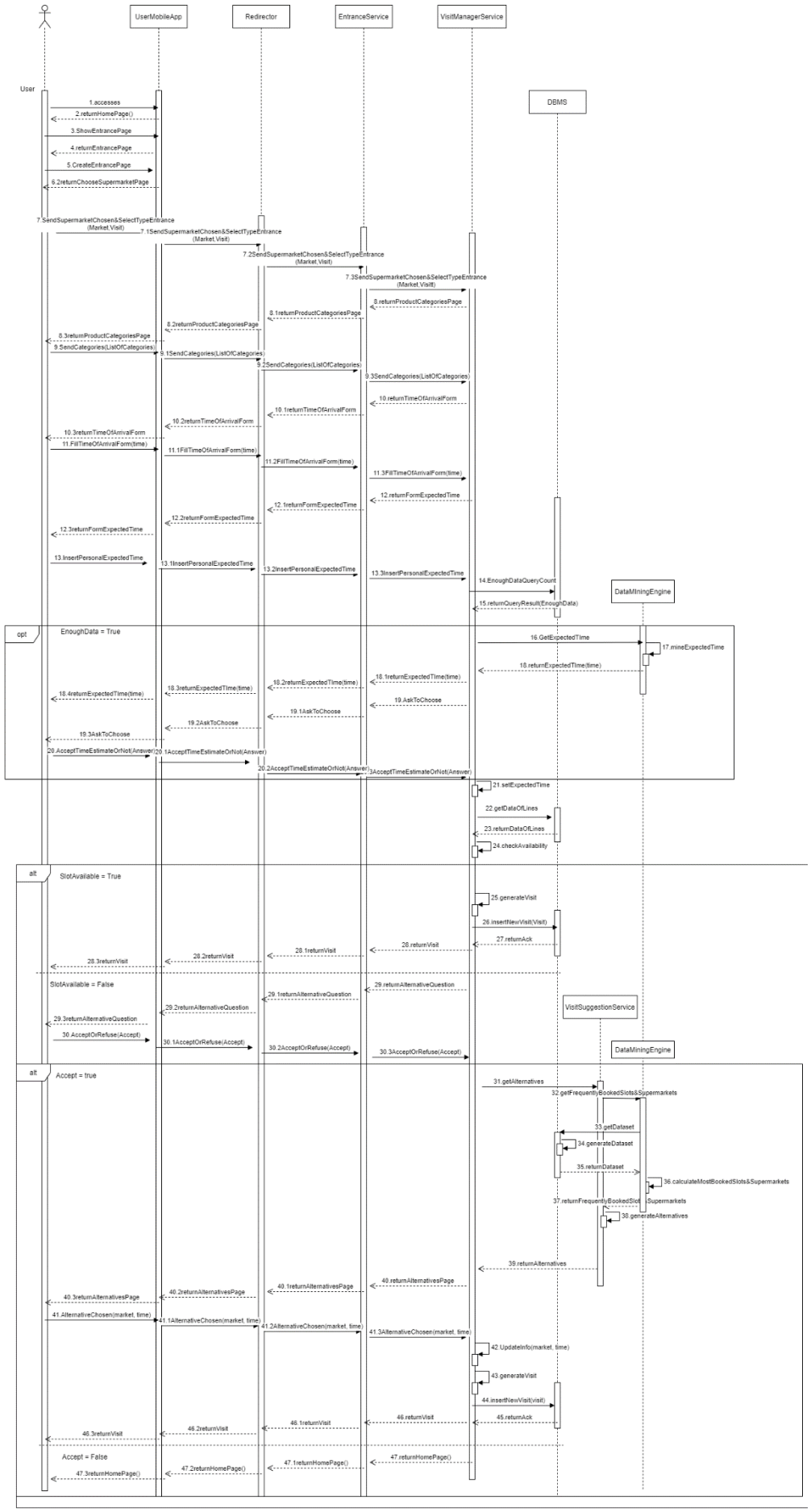
2.4.2 Create a Ticket for Non-User

This function is specific for non-users. Because the ticket for non-users is created directly at the supermarket through the ticket machine, all the steps of creation of a ticket for a user are jumped. However, to create a ticket SupermarketLinesService must have an interface with TicketManagerService. As the creation of a ticket for a user, once the creation of the ticket is created the DBMS adds the entrance in the non-user line.



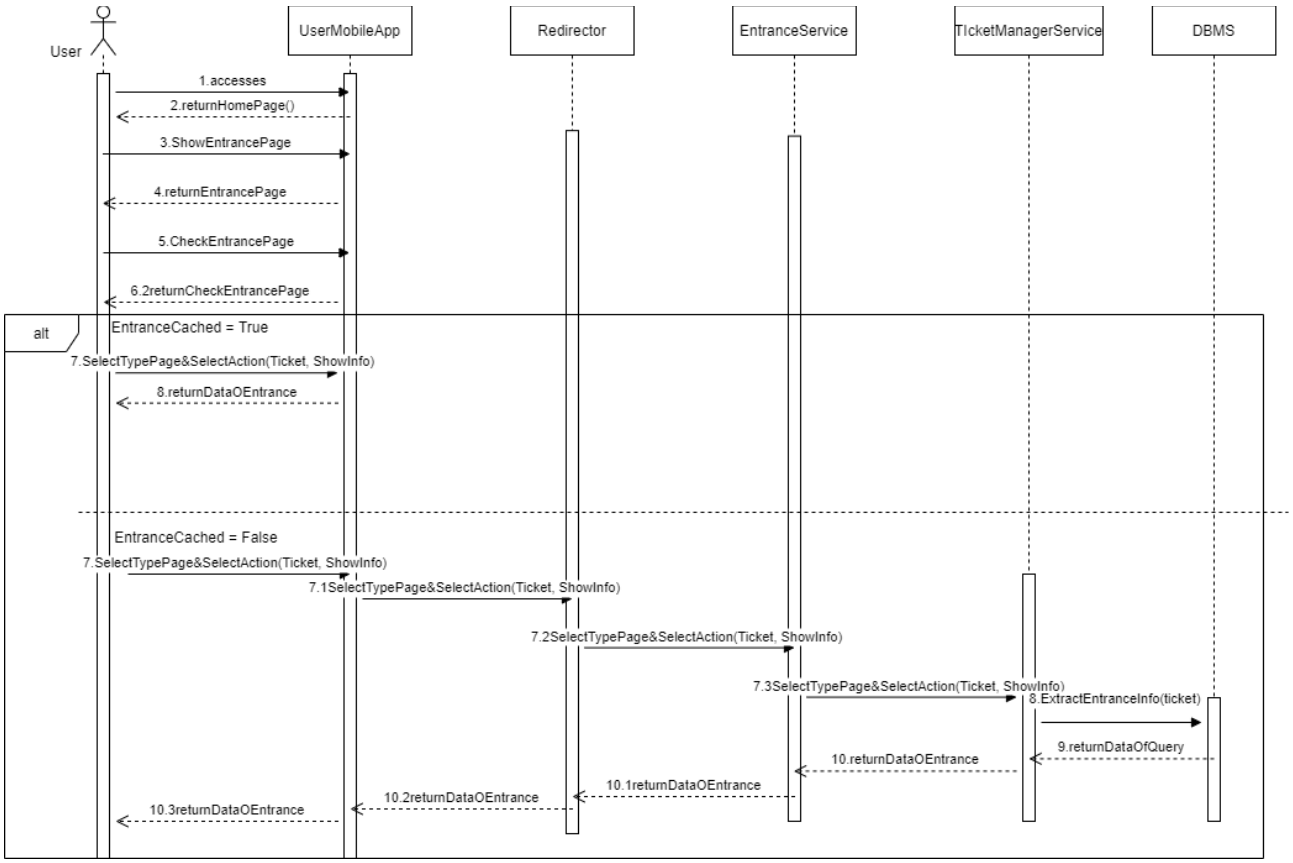
2.4.3 Book a Visit

Although the function of book a visit is structured in a similar way to create a ticket, there are some important differences, since it involves the use of methods of DataMiningEngine both for estimating the time that the user will spend for the visit and for generating the eventual alternatives. In fact, VisitSuggestionService is interfaced with the DataMiningEngine, and indirectly with the DBMS, to get the most frequently booked slots in the most frequented supermarkets by the user. However similarly to create a ticket, book a visit function requires EntranceService, VisitManagerService and indeed Redirector components.


User
UserMobileApp
Redirector
EntranceService
VisitManagerService
DBMS
DataMiningEngine
VisitSuggestionService
1.accesses
2.returnHomePage()
3.ShowEntrancePage
4.returnEntrancePage
5.CreateEntrancePage
6.2returnChooseSupermarketPage
7.SendSupermarketChosen&SelectTypeEntrance (Market,Visit)
7.1SendSupermarketChosen&SelectTypeEntrance (Market,Visit)
7.2SendSupermarketChosen&SelectTypeEntrance (Market,Visit)
7.3SendSupermarketChosen&SelectTypeEntrance (Market,Visit)
8.returnProductCategoriesPage
8.1returnProductCategoriesPage
8.2returnProductCategoriesPage
8.3returnProductCategoriesPage
9.SendCategories(ListOfCategories)
9.1SendCategories(ListOfCategories)
9.2SendCategories(ListOfCategories)
9.3SendCategories(ListOfCategories)
10.returnTimeOfArrivalForm
10.1returnTimeOfArrivalForm
10.2returnTimeOfArrivalForm
10.3returnTimeOfArrivalForm
11.FillTimeOfArrivalForm(time)
11.1FillTimeOfArrivalForm(time)
11.2FillTimeOfArrivalForm(time)
11.3FillTimeOfArrivalForm(time)
12.returnFormExpectedTime
12.1returnFormExpectedTime
12.2returnFormExpectedTime
12.3returnFormExpectedTime
13.InsertPersonalExpectedTime
13.1InsertPersonalExpectedTime
13.2InsertPersonalExpectedTime
13.3InsertPersonalExpectedTime
14.EnoughDataQueryCount
15.returnQueryResult(EnoughData)
opt
EnoughData = True
16.GetExpectedTime
17.mineExpectedTime
18.returnExpectedTime(time)
18.1returnExpectedTime(time)
18.2returnExpectedTime(time)
18.3returnExpectedTime(time)
18.4returnExpectedTime(time)
19.AskToChoose
19.1AskToChoose
19.2AskToChoose
19.3AskToChoose
20.AcceptTimeEstimateOrNot(Answer)
20.1AcceptTimeEstimateOrNot(Answer)
20.2AcceptTimeEstimateOrNot(Answer)
20.3AcceptTimeEstimateOrNot(Answer)
21.setExpectedTime
22.getDataOfLines
23.returnDataOfLines
24.checkAvailability
alt
SlotAvailable = True
25.generateVisit
26.insertNewVisit(Visit)
27.returnAck
28.returnVisit
28.1returnVisit
28.2returnVisit
28.3returnVisit
SlotAvailable = False
29.returnAlternativeQuestion
29.1returnAlternativeQuestion
29.2returnAlternativeQuestion
29.3returnAlternativeQuestion
30.AcceptOrRefuse(Accept)
30.1AcceptOrRefuse(Accept)
30.2AcceptOrRefuse(Accept)
30.3AcceptOrRefuse(Accept)
alt
Accept = true
31.getAlternatives
32.getFrequentlyBookedSlots&Supermarkets
33.getDataset
34.generateDataset
35.returnDataset
36.calculateMostBookedSlots&Supermarkets
37.returnFrequentlyBookedSlots&Supermarkets
38.generateAlternatives
39.returnAlternatives
40.returnAlternativesPage
40.1returnAlternativesPage
40.2returnAlternativesPage
40.3returnAlternativesPage
41.AlternativeChosen(market, time)
41.1AlternativeChosen(market, time)
41.2AlternativeChosen(market, time)
41.3AlternativeChosen(market, time)
42.UpdateInfo(market, time)
43.generateVisit
44.insertNewVisit(visit)
45.returnAck
46.returnVisit
46.1returnVisit
46.2returnVisit
46.3returnVisit
Accept = False
47.returnHomePage()
47.1returnHomePage()
47.2returnHomePage()
47.3returnHomePage()

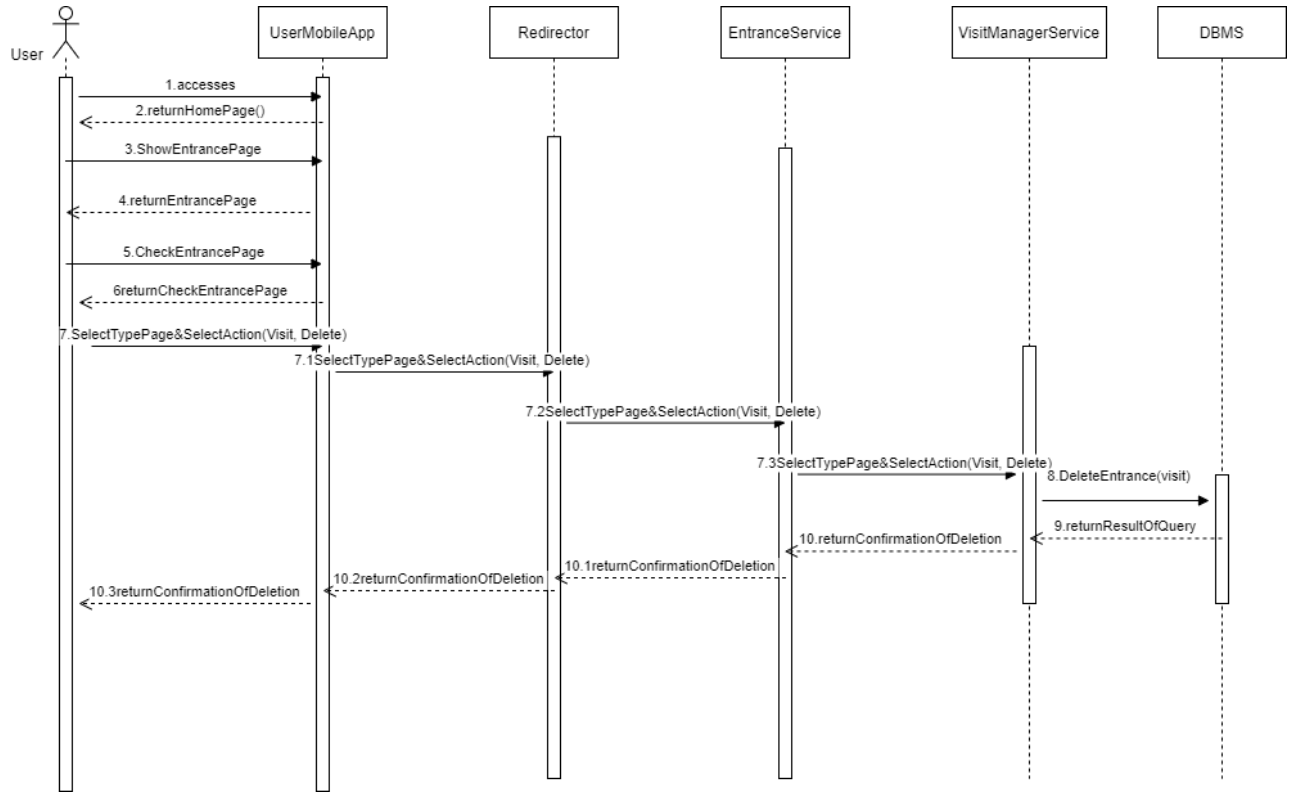
2.4.4 Check a Ticket

Check a Ticket is structured in the same way of Check a Visit, so for the sake of simplicity we represent only Check a Ticket. This process involves the use of the cache, in fact if the entrance is cached, no interaction with the DBMS is required. This function uses indeed UserMobileApp component, and other components such as Redirector, EntranceService, TicketManagerService and DBMS if the information of the entrance is not cached. The process starts when the user interacts with the UserMobileApp which will return the home page. After that the user accesses the CheckEntrancePage, it chooses which kind of entrance it wants to interact with and the type of interaction, since as can be seen in the mock-ups CheckEntrancePage can both be used both to check the info of an entrance and to delete one. If the ticket information is not cached the request from the User is forwarded to TicketManagerService that then will request the DBMS to extract through a query the information of the tickets and then it will be forwarded to the User. If the information is cached UserMobileApp will return it.



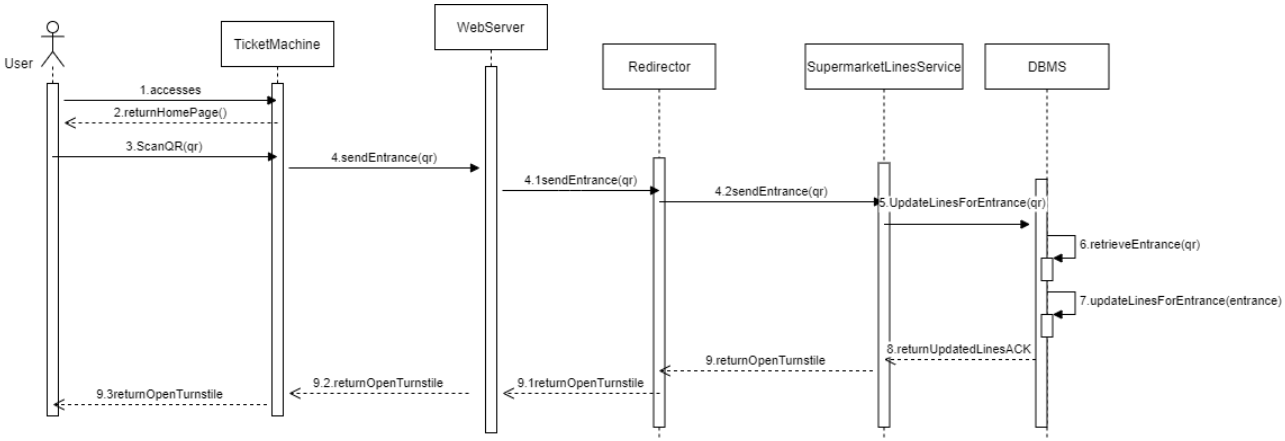
2.4.5 Delete a Visit

In this process, the user requests to delete a visit. Till the selection of the type of entrance and the type of the action, which are the main differences, the process is the same as above. The delete action is forwarded by the EntranceService to VisitManagerService and then to DBMS that will perform a remove query. As the process described above, for the sake of simplicity we describe only the process to delete a visit since it is analogous to the process of deletion of a ticket.



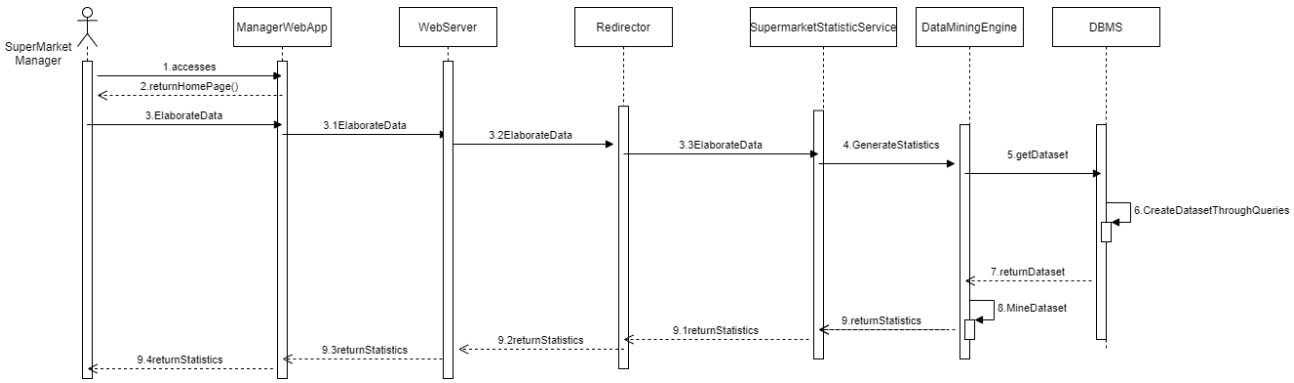
2.4.6 Enter the Supermarket

In this process, the TicketMachine, WebServer, Redirector, SupermarketLinesService and DBMS components are involved. The TicketMachine will scan the QR code of the entrance, that will send through the WebServer to the SupermarketLinesService that will request the DBMS to update the lines. This process is identical to the one for allowing the customers to exit the supermarket therefore we only reported this one.



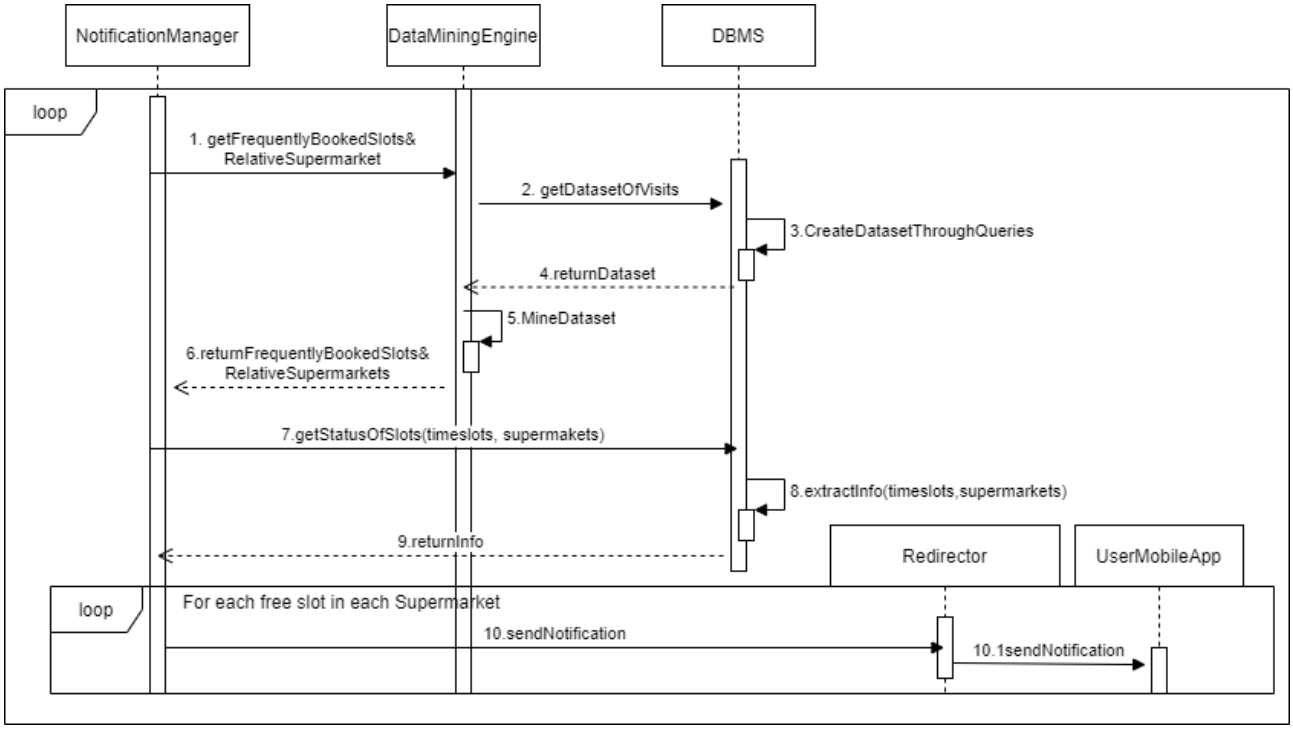
2.4.7 Statistics

The process involves the ManagerWebApp, WebApp, Redirector, SupermarketStatisticsService, DataMiningEngine and the DBMS. The ManagerWebApp will forward the request to elaborate data through the WebServer and the Redirector to the SupermarketStatisticsService, which will then request the DataMiningEngine to generate the statistics. The DataMiningEngine to mine the data, needs the DBMS to extract the information and group them in a dataset that will be returned to the previous component. After the DataMiningEngine creates the statistics of the data, it returns them to the user.

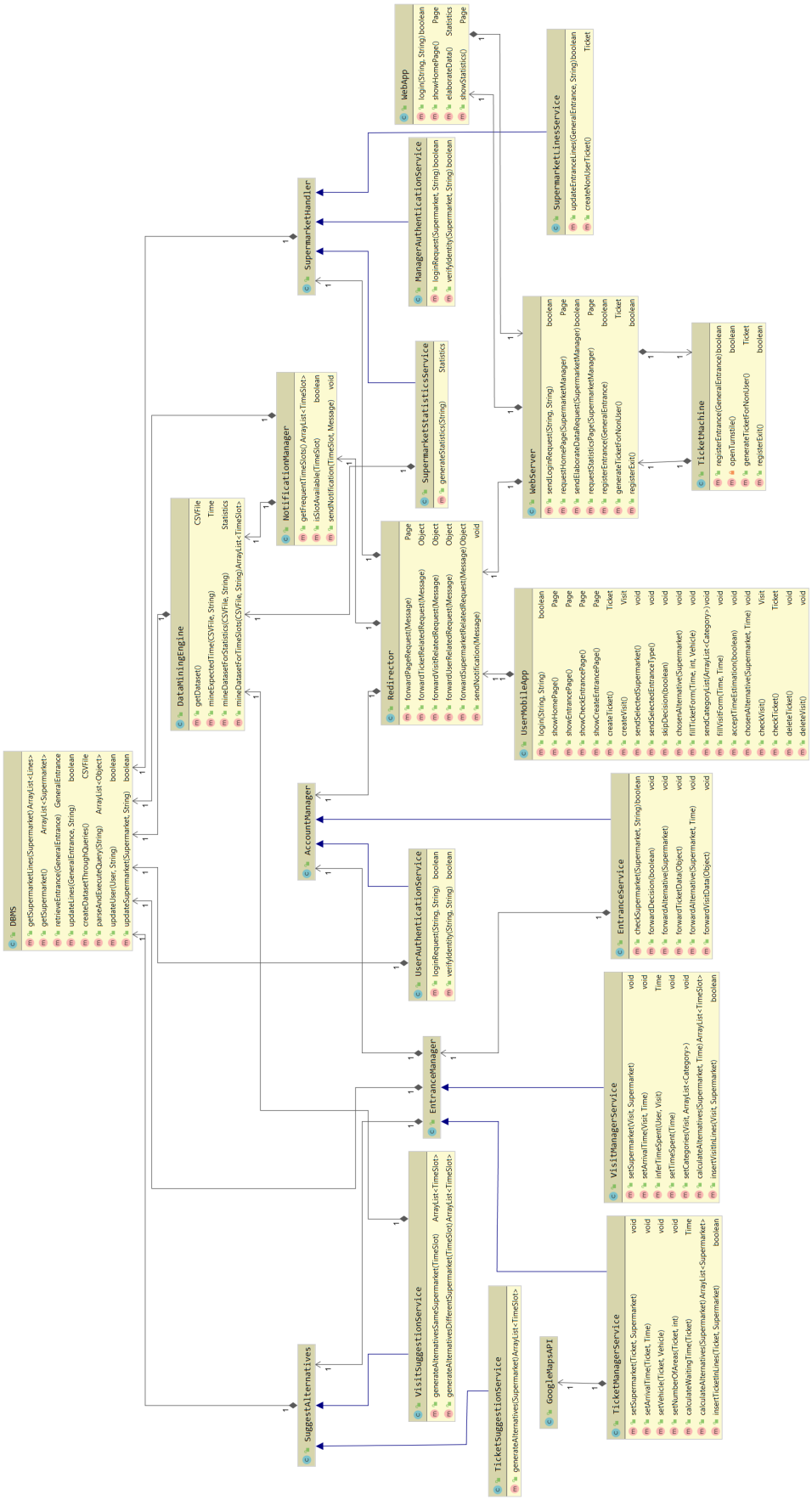


2.4.8 Send notification

This process is continuously active to be able to check when a slot becomes free. It involves the NotificationManager, the DataMiningEngine and the DBMS. The NotificationManager requests the DataMiningEngine to return to it the most frequently booked slots and supermarket of a user. To do that the DataMiningEngine requests the DBMS to return him a dataset from which he will mine the information requested. Once the NotificationManager receives the information required, it requires the DBMS to check if those slots are free, if they are free it sends a notification for each of them.



2.5 Component Interfaces



The diagram in the previous page describes the main methods that each component can invoke to complete the tasks it is requested. These methods are the most significant actors in the runtime view of the former section and well represents what the software needs to work properly.

For the sake of completeness, it is important to mention that all the methods in this picture have a logical value to better understand how each component should behave and interact with others, instead of a strict and dogmatic value. Indeed, the purpose of this graph is to give the guidelines of what is the idea behind the project, to then let implementers the possibility to adapt methods to the various aspects that will come out during that process.

2.6 Selected Architectural Style and Patterns

As described before, the architectural style chosen for the application is a three-tier client-server architecture. This choice will guarantee scalability, availability, and performance of the system. The client is structured as a fat client, to allow a check on the inputs of the user. Moreover, the fat client, as previously mentioned, includes a caching system to increase greatly performances reducing the number of queries that the DBMS would perform and at the same increasing the scalability of the system reducing the computational weight both on the other two layers.

The communication protocol chosen is HTTPS to guarantee a high level of security for the data exchanged between all the layers of the architecture. The format utilized will be JSON to maintain the messages easily readable and less complex implementation allowing however custom representation for objects.

Being Google Maps an external service, REST communication protocol is chosen to deal with Google Maps REST API since allows customization of the requested contents. Moreover, the use of REST API helps guaranteeing scalability being stateless, performance because it supports caching, while also providing simple implementation since it is a standardized approach to communication.

For the implementation of the software and the inner property of the system, the adopted patterns are the following ones.

MVC (Model – View – Controller)

MVC pattern guarantees a good level of maintainability, reusability and scalability of the code. Moreover, it divides very well the three layers in which the software is split, making it the most preferred pattern in web-based application. MVC can be implemented following several styles and the chosen one for this project is the variant that uses the virtual view server-side. This option allows to simplify the management of the network, since the behaviour of the view (implemented client-

side) is simulated into the server: controller and model, then, can behave as if the view lies in the same machine and, after all requested changes are made, these are communicated back to the client. Therefore, MVC with virtual view divides into:

- **Model:** it is the responsible of the data layer, managing and keeping them updated. Its changes are reflected onto the virtual view.
- **View:** it is the responsible of the presentation layer, showing all kind of information to let the user interact properly.
- **Virtual View:** it is the counterpart of the View server-side, filtering every request and acting as if it made them.
- **Controller:** it is the responsible of the business logic of the whole system; in our configuration (fat client) it is not fully on server-side, but some correctness checks lies onto client-side of the system.

The next section will better present how the MVC pattern with virtual view integrates with the network.

Observer Pattern

This pattern is very useful to implement the previous pattern (MVC) since whenever a change is made in the View, it sends a message to the Virtual View, the observer pattern notifies one its observer, the Controller, with a specific method. Consequently, the Controller changes the Model, completing the requests made by the View. Once a change is made on the Model, it notifies the Virtual View, that send the answer to the real View that shows the changes of the Model to the user.

Façade Pattern

This pattern is used to hide a complex system with a simple interface; with this brief description it is clear that the main component that carries out this task is the Redirector, that:

- Interacts externally with both mobile app and web server providing many services through a simple interface.
- Interacts internally with all other components of the business logic, dispatching, filtering and coordinating every operation.

This pattern is extremely useful in this situation since connects two different worlds with a simple interface.

Factory Pattern

This pattern is mainly used in the creation of entrances: indeed, in this design there are two different types of entrances (Tickets and Visits) with similar characteristics and the factory pattern can take advantage from this similarity to provide a simple, fast, reliable and not redundant way to generate those core parts of the software.

2.7 Other Design Decisions

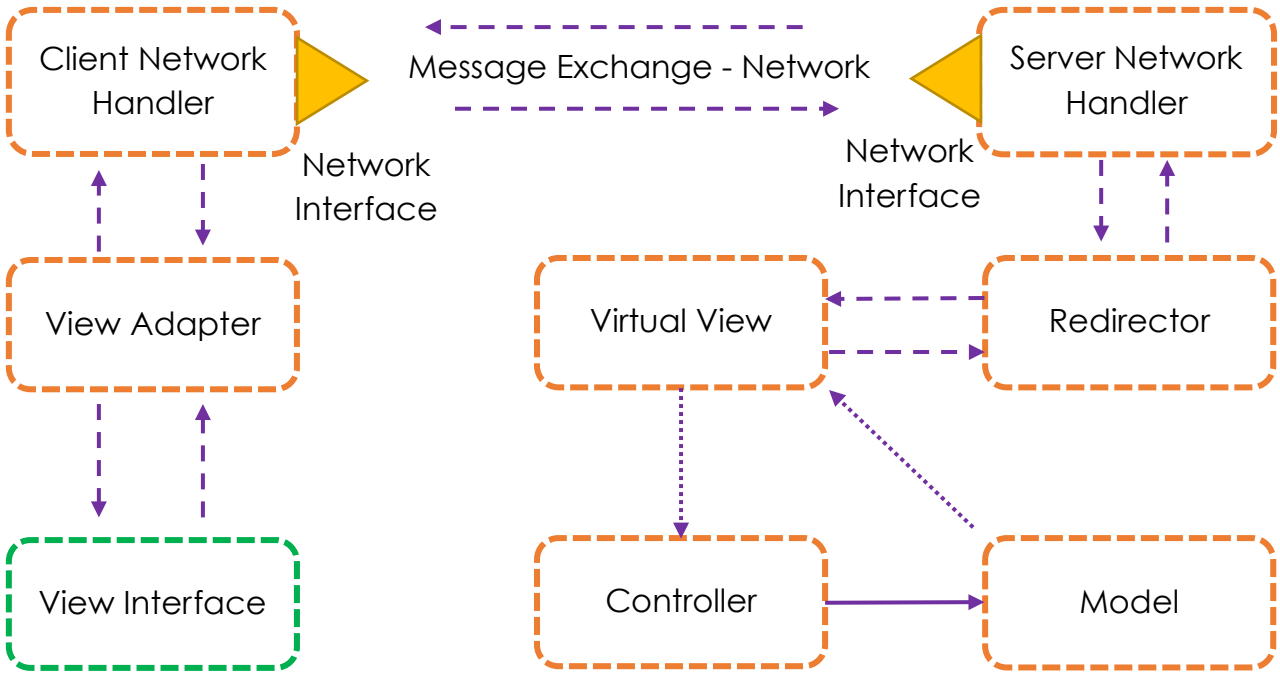
In this last paragraph of the second chapter, we are going to explain some other design choices concerning the software.

The first choice that should be made when considering a way to store data is the type of data management system to use between OLTP and OLAP. Since this system needs operations characterized by a huge number of synchronous and fast transactions (INSERT, DELETE, UPDATE), the relational OLTP solution is the preferred one, since they can guarantee:

- Data Integrity in a multi-access environment (Collaboration and Security)
- Efficiency and effectiveness, measuring for instance the number of transactions per second (Trust and Flexibility)
- Data stored always detailed, precise and updated (Accuracy)

Nevertheless, CLup system needs to perform queries at a higher level of complexity (let us think about the SupermarketStatisticsService) that would imply the usage of an OLAP system to store data, such as data warehouses. However, this conflict can be bypassed introducing a data mining engine able to perform those complex queries; besides, this addition simplifies the operations of other components such as NotificationManager, that needs mine data that can be easily filtered through a fast query performed by the DBMS. For the above reasons, the preferred choice is a relational OLTP integrated with a data mining engine, to separate two distinct functionalities.

Finally, it is necessary to spend a few words on the communication protocol and how it integrates with the MVC pattern, using as a support the images below.



On the left side of the picture is represented the client and, consequently, on the right side the server. The client is very simple and implements the View of the MVC pattern. However, it also checks the consistency of inputs using data stored in the cache of the application; this expedient lightens the communications introducing the concept of Fat Client we have already talked about. The client is divided into:

- Client Network Handler, that manages the exchange of message through the network adopting a Network Interface identical to the one of the server to standardize the methods.
- View Adapter, that analyses all the received messages and chooses what View the user has to interact with.
- View Interface, that standardizes the View and is the responsible of showing the user the correct page.

As regard the server, instead, it has a specular structure than the client's one:

- Server Network Handler, that is the server's counterpart of Client Network Handler
- Redirector, that is the core component we have already talked about
- Virtual View, Controller and Model, that are the remaining parts of the MVC pattern chosen. The picture better highlights the details of the pattern explained in the previous section.

Now that the network architecture is clear, the last part of this section is dedicated to the vehicles that use the network and allows the system to work properly: messages. These core elements are thought to be highly extendable: whenever a new functionality would be added to the system and this one requires an exchange of information between client and server, new types of messages can be added without touching any other piece of software than the main class Message.

The structure of a message is trivial and consists in:

1. First Level Header, that makes a first strong filter between the different types of message.
2. Second Level Header, that is a sub-header strongly dependent to the previous one that makes further differentiation.
3. Payload, that is the real information content of the whole message and contains data, requests and whatsoever.

This simplifies a lot the functioning of the network since only one class needs to be serialized (the Message class), standardizing, categorizing and organizing the communication protocol, and not the long list of objects representing the payloads.

Since these instructions regarding network management are only guidelines, and for the sake of clarity and immediacy, network is not treated while explaining the behaviour of sequence diagrams (section 2.4), otherwise its dissertation would have been uselessly complex and fallen outside the purposes of that section.

3. User Interface Design

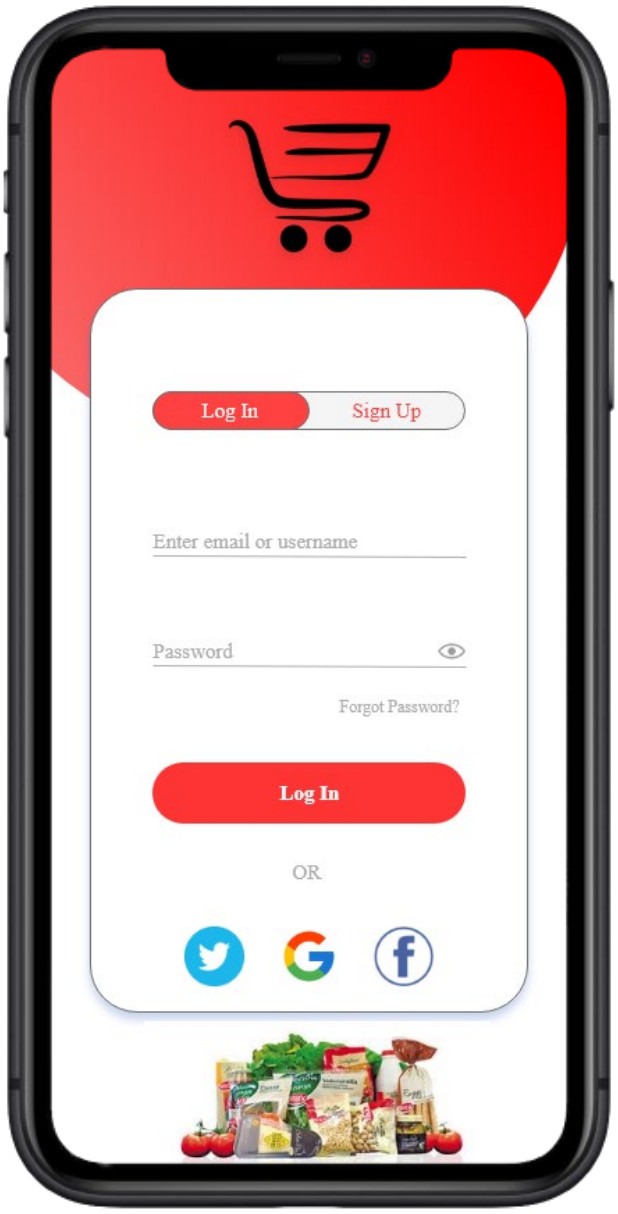
3.1 Mock-ups

The following mock-ups represents how the mobile application should look like used for a user (sections from 3.1.1 to 3.1.8) and the aspect of the web application used by the supermarket manager (sections 3.1.9 and 3.1.10)

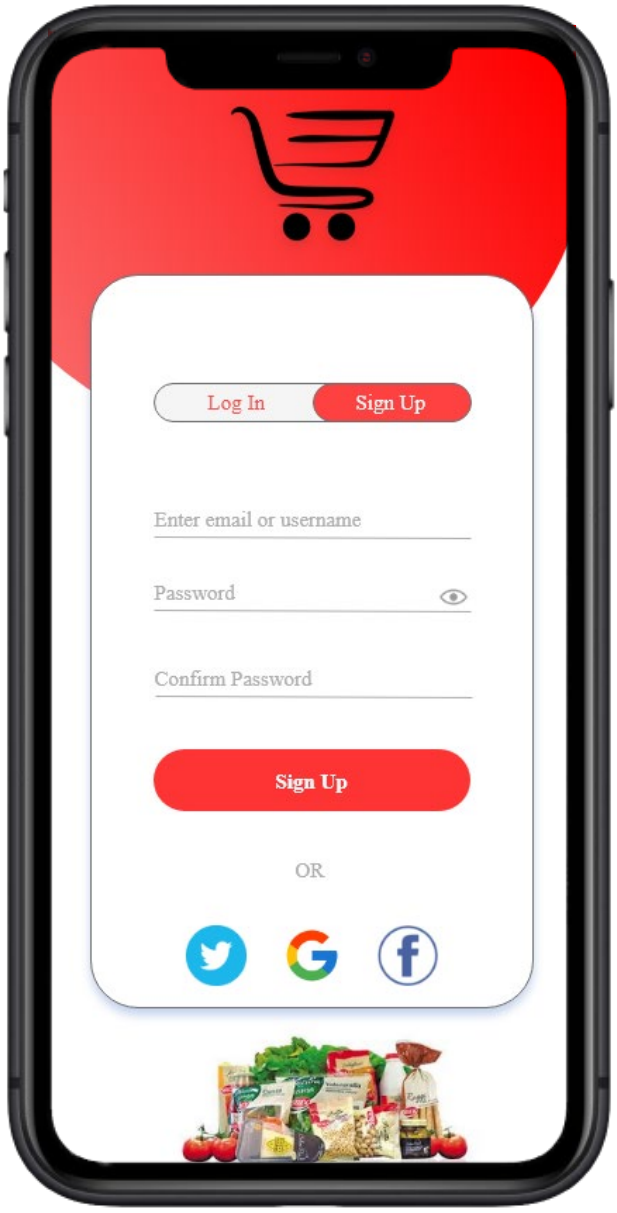
3.1.1 App Icon and Notification

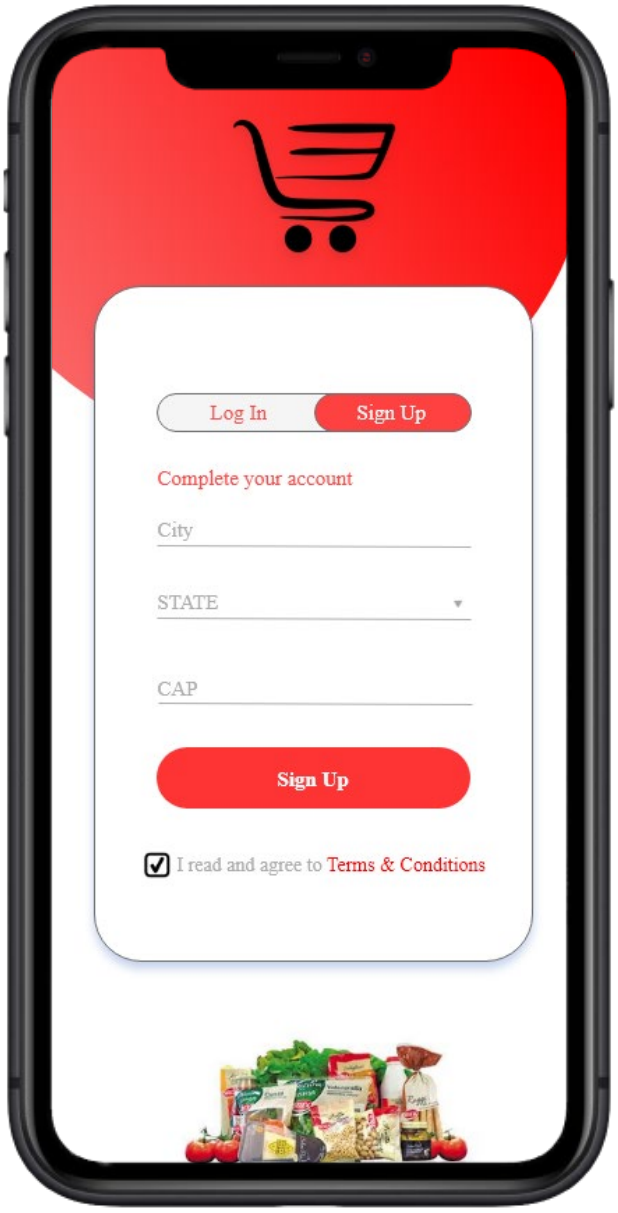
3.1.2 Splash Screen

3.1.3 Log-in

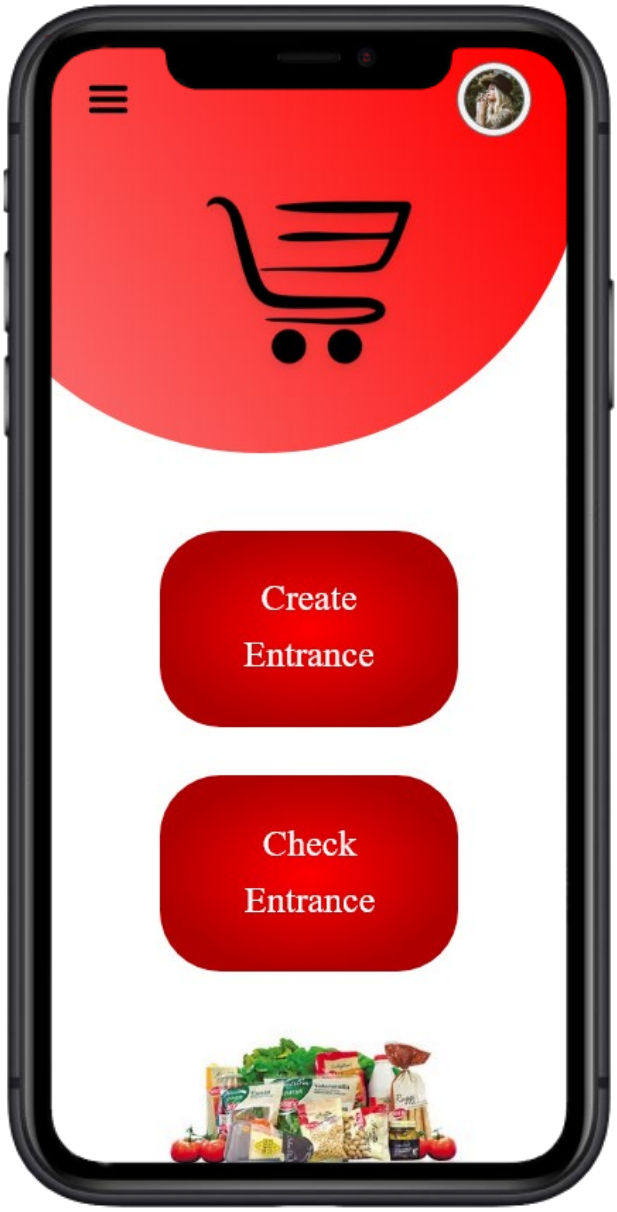


3.1.4 Sign Up

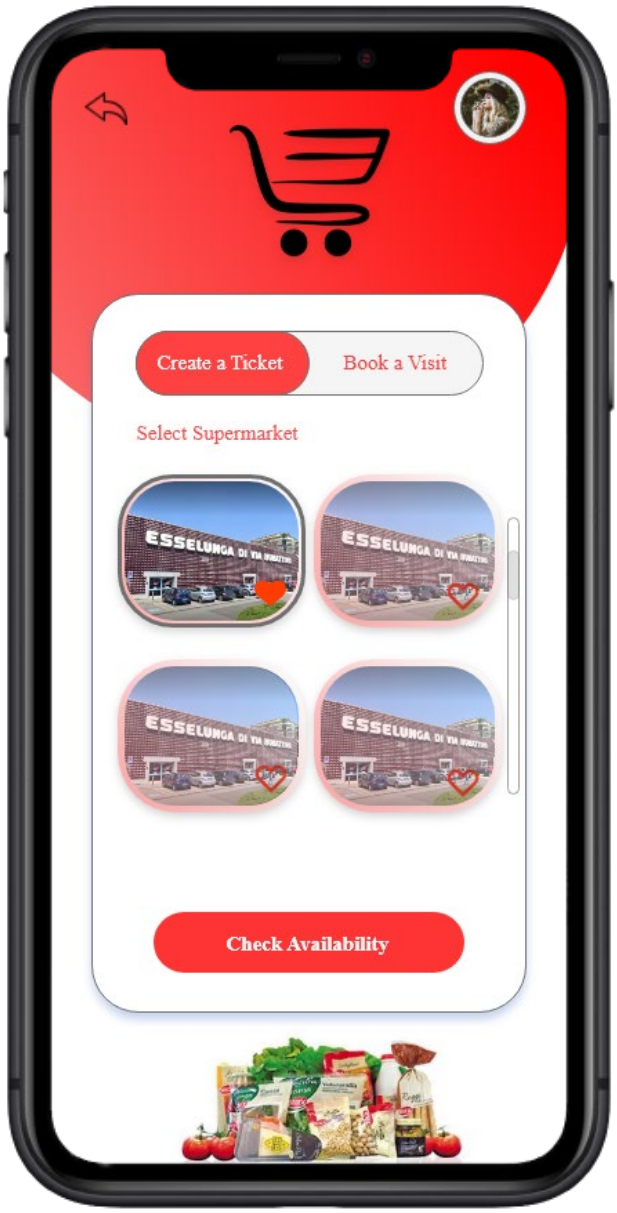

Log In
Sign Up
Enter email or username
Password
Confirm Password
Sign Up
OR

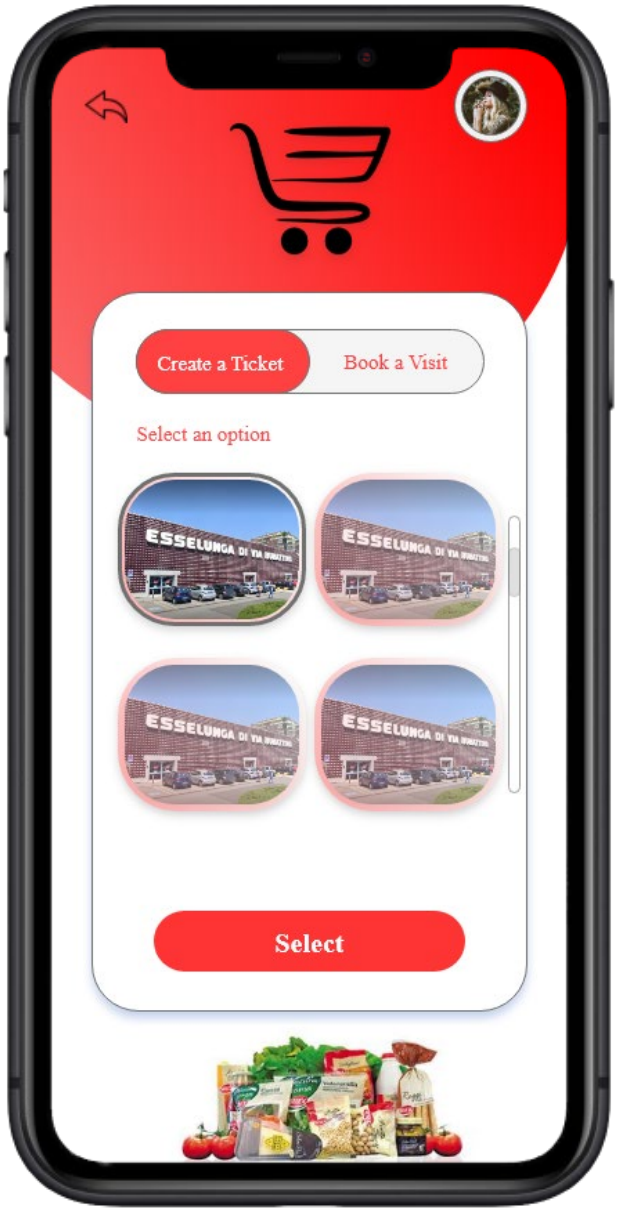

Log In
Sign Up
Complete your account
City
STATE
CAP
Sign Up
I read and agree to Terms & Conditions

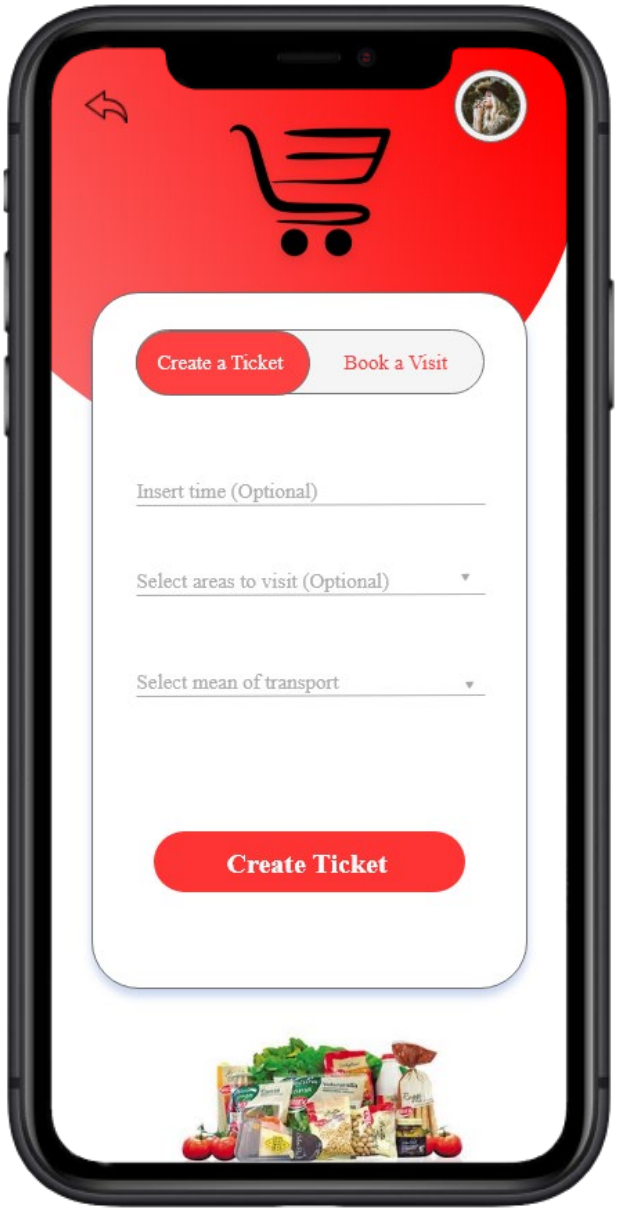
3.1.5 Customer Home Page



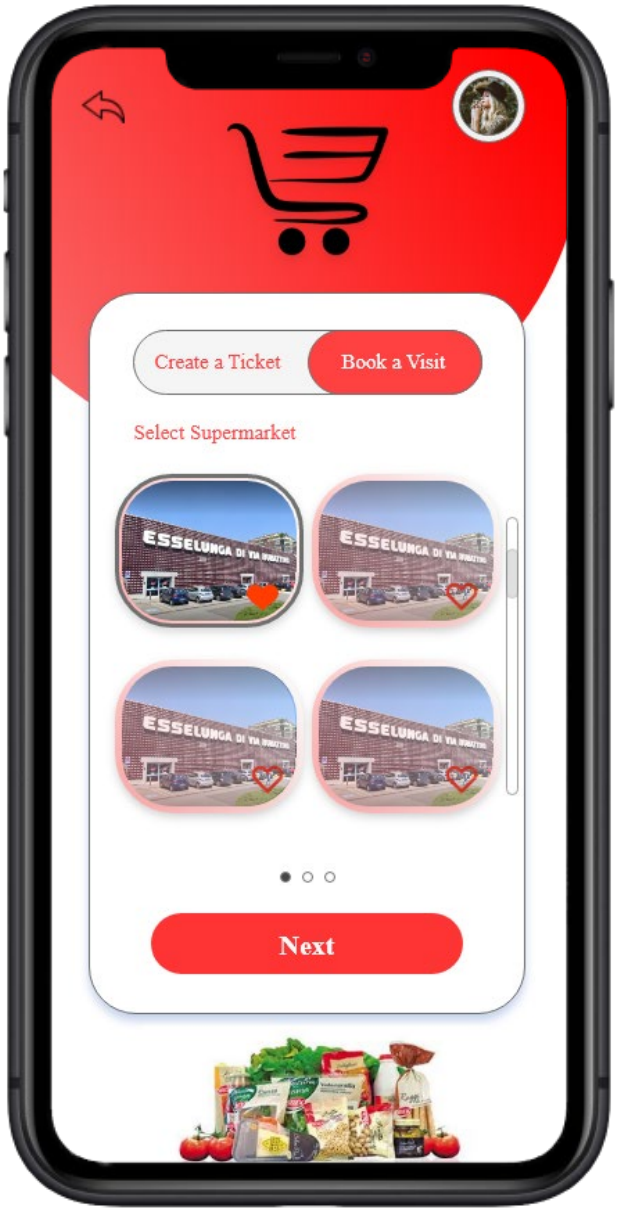
3.1.6 Create a Ticket

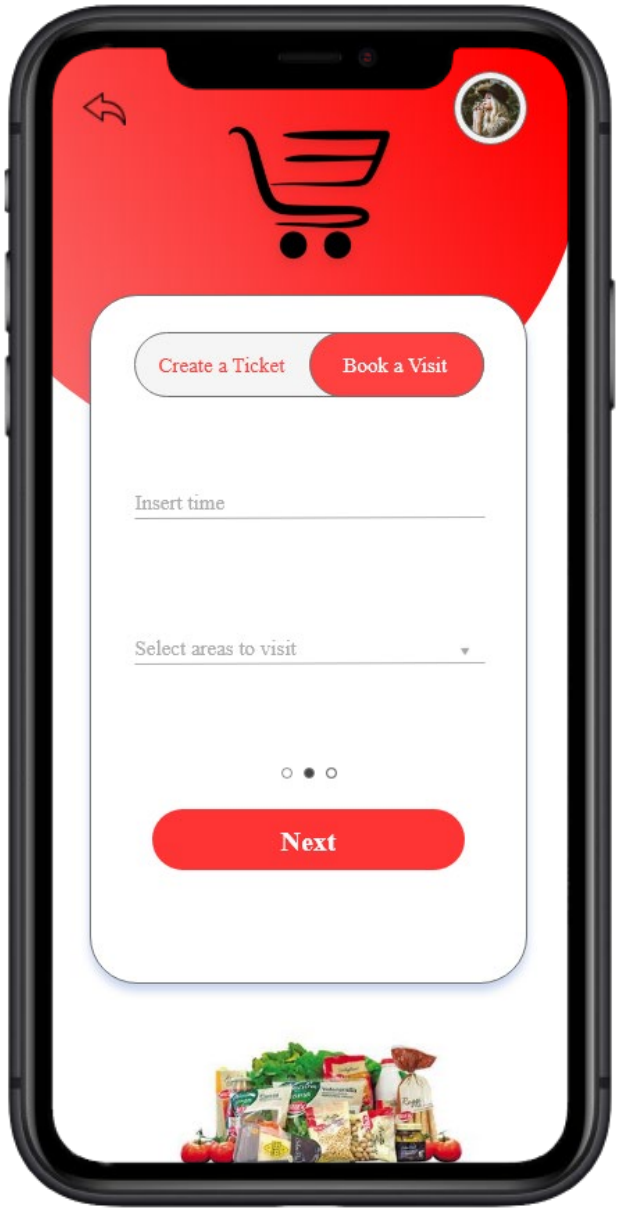



Create a Ticket
Book a Visit
Select an option
Select

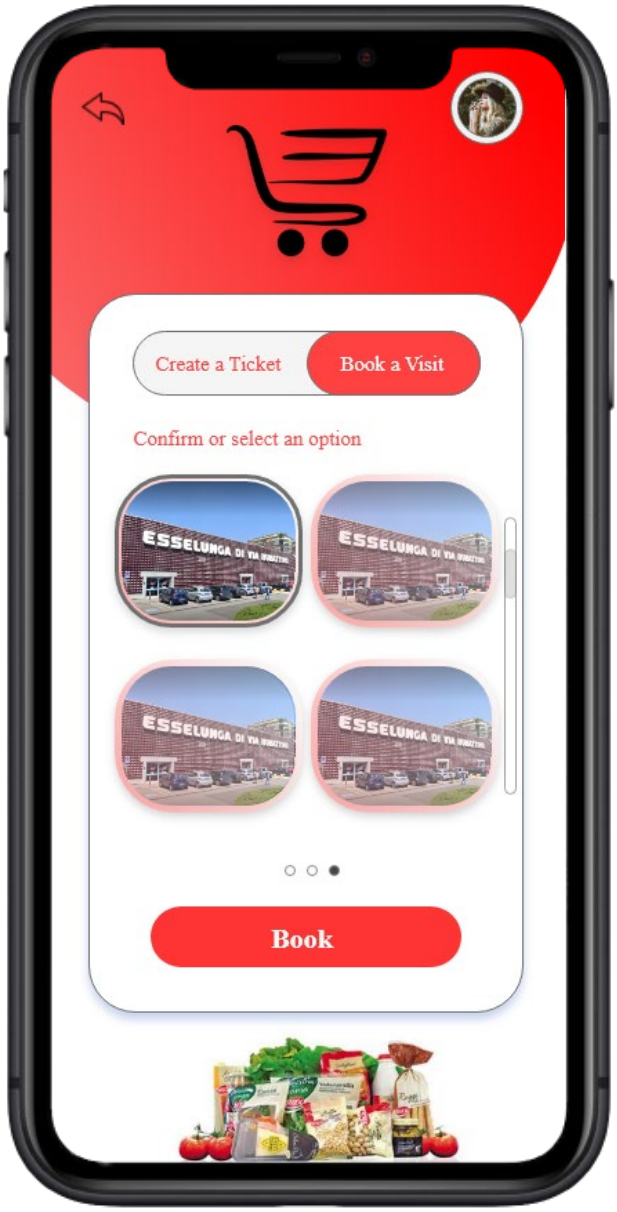

Create a Ticket
Book a Visit
Insert time (Optional)
Select areas to visit (Optional)
Select mean of transport
Create Ticket

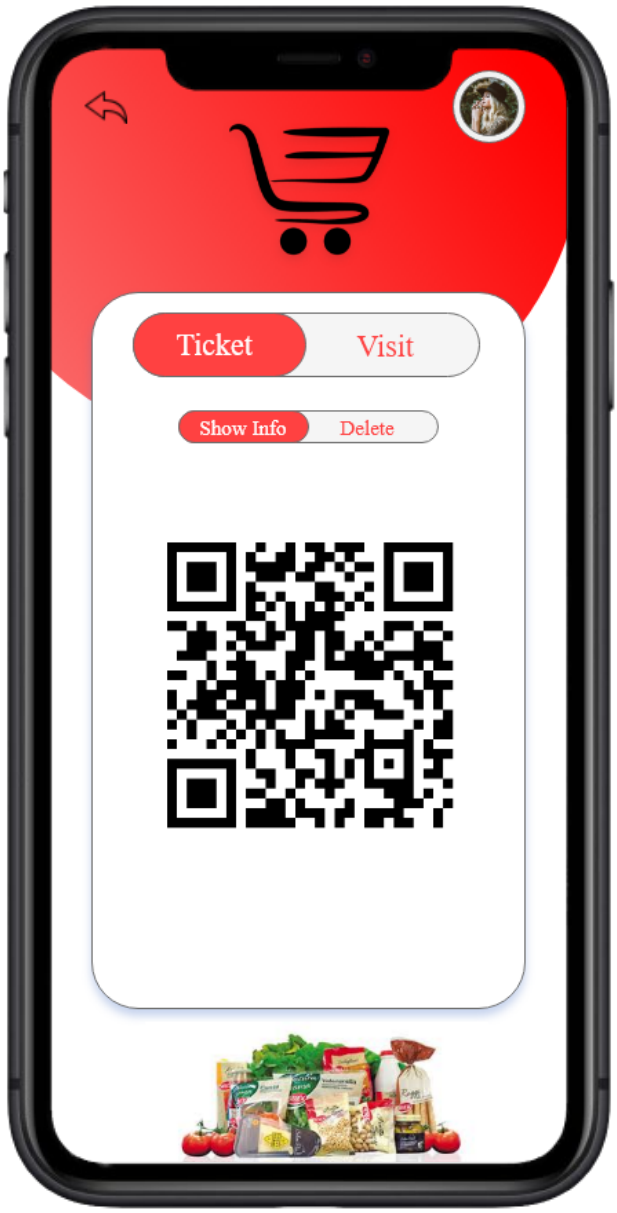
3.1.7 Book a Visit



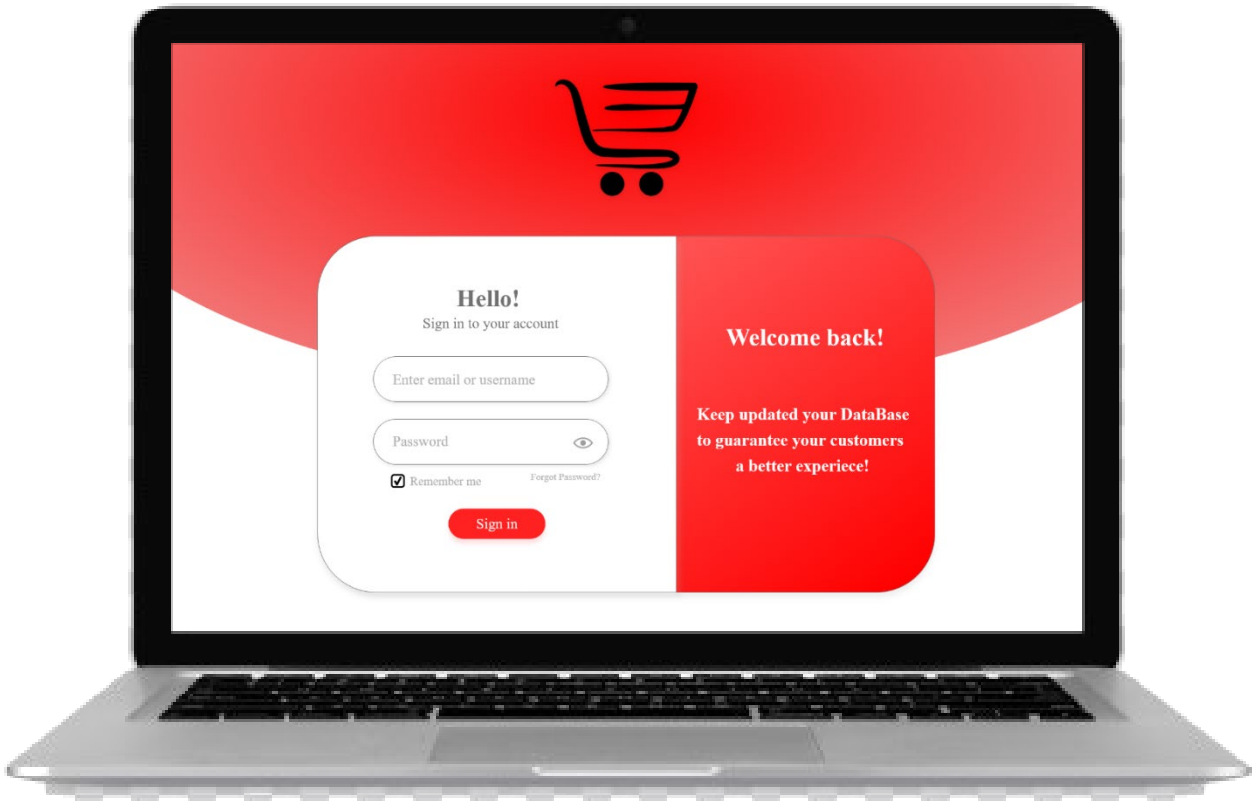


3.1.8 Ticket or Visit Info

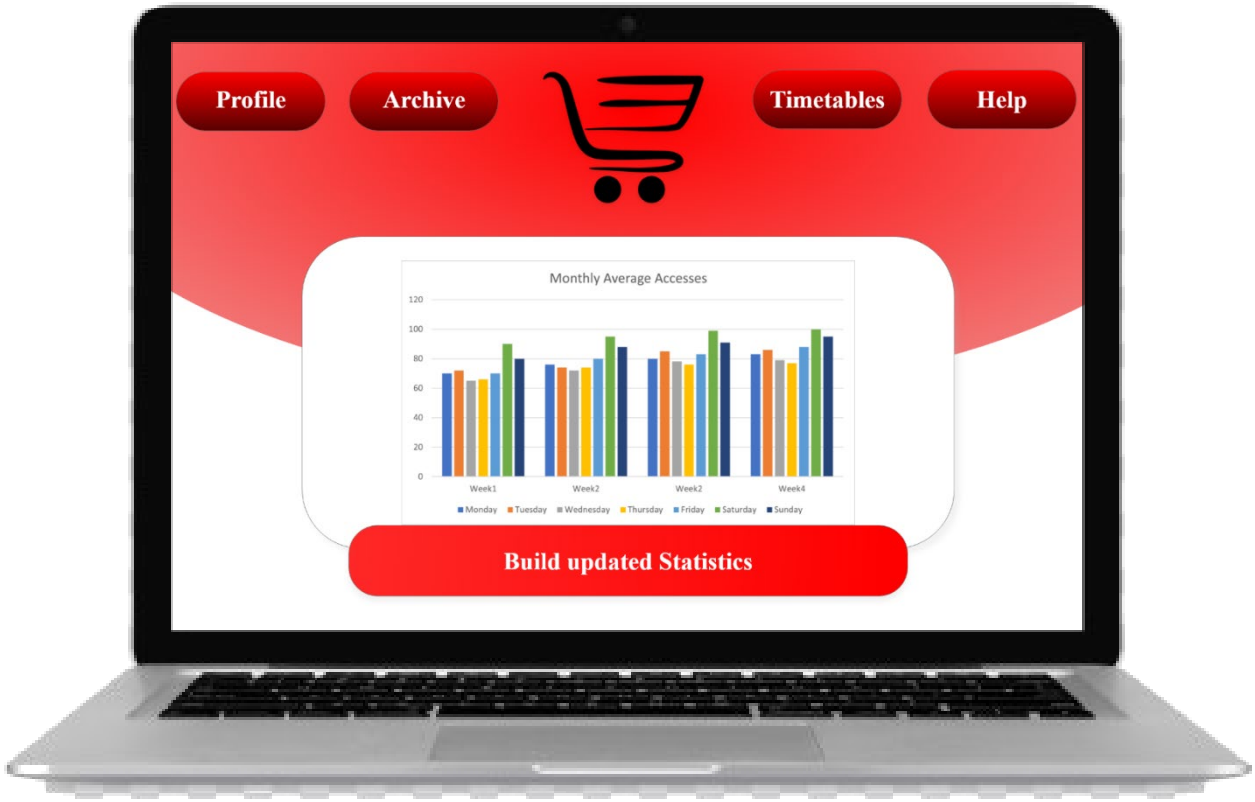




3.1.9 Supermarket Manager Sign in



3.1.10 Supermarket Manager Home Page



4. Requirement Traceability

In this section the goals specified in the RASD is bonded with design components

GOALS	DESCRIPTION	DESIGN COMPONENTS	REQUIREMENTS
G1	Allow user to create a ticket	UserMobileApp, EntranceService, Entrance Manager: TicketManagerService, SuggestAlternatives: TicketSuggestionService, DBMS, GoogleMapsAPI	R1, R2, R3, R4, R5, R20, R26, R27, R28, R44, R46
G2	Allow user to enter the supermarket	TicketMachine, WebServer, SupermarketHandler: SupermarketLinesService, DBMS	R1, R6, R7, R8, R9, R28, R29, R44, R45
G3	Allow non-user to enter the supermarket	TicketMachine, WebServer, SupermarketHandler: SupermarketLinesService, DBMS	R6, R7, R10, R11, R12, R28, R29
G4	Allow user to book a visit	UserMobileApp, EntranceService, EntranceManager: VisitManagerService, SuggestAlternatives: VisitSuggestionService, DataMiningEngine, DBMS	R1, R2, R3, R4, R13, R14, R15, R20, R26, R27, R28, R45, R46
G5	Allow user to check a ticket	UserMobileApp, EntranceService, EntranceManager: TicketManagerService, DBMS	R1, R7, R16

G6	Allow user to delete a ticket	UserMobileApp, EntranceService, EntranceManager: TicketManagerService, DBMS	R1, R16, R17, R18, R19, R21, R27, R28, R44
G7	Allow user to check a visit	UserMobileApp, EntranceService, EntranceManager: VisitManagerService, DBMS	R1, R7, R22
G8	Allow user to delete a visit	UserMobileApp, EntranceService, EntranceManager: VisitManagerService, DBMS	R1, R22, R23, R24, R27, R28, R45
G9	Suggest alternatives for tickets to user	SuggestAlternatives: TicketSuggestionService, DBMS	R1, R25, R28, R30, R31, R32, R33, R39, R44
G10	Suggest alternatives for visits to user	SuggestAlternatives: VisitSuggestionService, DBMS	R1, R25, R28, R30, R31, R32, R34, R39, R45
G11	Send notification to user about free available slots	NotificationManager, DataMiningEngine, DBMS, UserMobileApp	R1, R25, R27, R28, R35, R36, R37, R38, R44, R45
G12	Build statistics using collected data	ManagerWebApp, WebServer, SupermarketHandler: SupermarketStatisticsService, DataMiningEngine, DBMS	R28, R29, R40, R41, R42, R43, R44, R45
G13	Allow customer to exit the supermarket	TicketMachine, WebServer, SupermarketHandler: SupermarketLinesService, DBMS	R1, R10, R44, R45, R47

R1	User is registered
R2	User has to select supermarket from a given list
R3	Software should allow user to insert the time he would spend in the supermarket
R4	Software should allow user to insert the categories of products he would purchase in the supermarket
R5	User has to insert the means of transport he would use to reach the supermarket from a given list
R6	Customer has to scan QR code to enter
R7	QR code has to be valid
R8	If p < cu, the first user in the user queue can enter
R9	If a slot reserved to all becomes free, the first user in the user queue can enter
R10	Supermarket has to give QR to non-user
R11	If p < c AND u = 0, the first non-user in non-user queue can enter
R12	If a slot reserved for non-users becomes free, the first non-user in non-user queue can enter
R13	User has to select arrival time
R14	If possible, Software should estimate the time user would spend in the supermarket
R15	User has to select the time he would spend in the supermarket
R16	At least one valid ticket has to have been created
R17	User has to select the ticket he wants to delete

R18	Software must remove the ticket from the queue where it was placed
R19	Software must modify the queue where the ticket was placed
R20	Software must generate a valid ticket
R21	Software must update the waiting times of all other ticket of the queue of the supermarket
R22	One VALID visit has to be booked
R23	User has to select the visit he wants to delete
R24	Software must remove the visit from the scheduled ones of the supermarket
R25	Software must receive information from supermarket
R26	User cannot reserve in the past
R27	Connection must stay up for the duration of the operation
R28	Data of users are stored in databases
R29	Software tracks incoming and outcoming fluxes of customers of supermarket
R30	Software should check if more advantageous options than the current one are available
R31	Software should suggest alternatives found
R32	Software should allow user to choose an alternative
R33	If user chooses an alternative, software must modify the ticket with updated information
R34	If user chooses an alternative, software must modify the visit with updated information

R35	Software should mine information of the user about his usual slots
R36	If the current day is one of the days user usually goes to supermarket AND there is at least one slot available, software should notify user with slots available of that day
R37	If during the current day there is available at least one of the usual time slots of the user, software should notify user with that/those slot/s
R38	If user accepts one of the suggestions, software should create a valid ticket with all the information
R39	Data received are not corrupted
R40	Software stores data about fluxes in databases
R41	Supermarket managers have access to data of all databases of the supermarket and of users
R42	Software has to analyze data
R43	Software has to build statistics
R44	Data of VALID and USED tickets are stored
R45	Data of VALID and USED visits are stored
R46	Software should not allow user to generate a general entrance while there is another one still valid
R47	Customer has to scan QR code to exit

5. Implementation, Integration and Test Plan

"Program testing can be used to show the presence of bugs, but never to show their absence."

(Dijkstra 1972)

5.1 Overview

This last chapter is one of the most important because thanks to this we reduce the risk of having errors in the system. In particular, the phases of verification and validation play a central role in this section. Indeed, the main scope is to find as many bugs as possible.

5.2 Implementation Plan

The bottom-up approach is the most appropriate to test and integrate the system that has to be implemented. This approach will compensate the use of the top-down approach for the architectural style, allowing to create reusable components. Therefore, the implementation must be in a gradual way, from smaller components to the bigger ones. This would also allow to test each component along the implementation to reduce the propagation of errors.

The first component to be implemented would be the DBMS since allow the other components will be connected to it, both directly and indirectly therefore it has a high priority in the implementation plan. The DBMS would provide all the methods to execute queries on the database.

The second component to be developed would be the DataMiningEngine. It needs indeed a functional DBMS to correctly execute its methods. Its methods in fact generates:

- the statistics needed by the SupermarketStatisticsService.
- the most frequent visited supermarkets needed by VisitSuggestionService to suggest the alternatives in the creation of a visit.
- The most "booked" slots in a supermarket by a user for the NotificationManager since it has to notify to him/her when they become free.

SuggestAlternatives and NotificationManager components can be implemented and tested in parallel since they are not interfaced among themselves, because SuggestAlternatives component's methods are used during the creation of an entrance while the NotificationManager function is to communicate when certain slots can be booked by a user.

Since SuggestAlternatives is a component utilized during the creation of an entrance, the forthcoming component to be implemented would be the EntranceManager. Its sub-components TicketManagerService and VisitManagerService are fundamental for the application since they provide the methods for the creation, deletion of the entrances and the method for showing to the user the information of the entrances. Both the components are also interfaced to the DBMS since they will need to insert, remove, and get the entrances from the database. Moreover, the VisitManagerService is interfaced to DataMiningEngine to infer the expected time spent by a user during a visit.

AccountManager and SupermarketHandler can be implemented and tested in parallel because they do not need each other methods nor functions. The former is divided in two sub-components:

- UserAuthenticationService: its main function is to register a new user and to authenticate an already registered one. Therefore, its methods require only to have this component to be interfaced to the DBMS.
- EntranceService: it is strongly linked to EntranceManager component. It manages and forwards all the request of creation, deletion and extraction of an entrance to the EntranceManager.

The two sub-components can be implemented in parallel since they don't share methods between themselves. Thus, they can be tested in parallel. The latter component is composed by three sub-components:

- ManagerAuthenticationService: this sub-component is the analogous of UserAuthenticationService, it allows to register a new supermarket and its linked manager and to authenticate a registered one.
- SupermarketStatisticService: this sub-component produces and presents to the manager, the statistics of his/her supermarket. The statistics are generated by the DataMiningEngine.
- SupermarketLinesService: this last sub-component function is to allow the customers to enter and exit the supermarket while registering the flows of customers entering and exiting the store. Moreover, this sub-component is connected to TicketManagerService sub-component to create the tickets for the non-users.

The last component to be implemented would be the Redirector since its role is to dispatch the messages and requests between server and client.

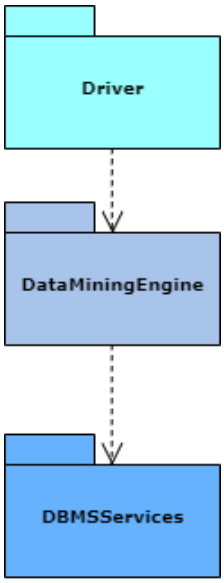
UserMobileApp and WebServer can be implemented at the same time since they are not interfaced between themselves. However, ManagerWebApp, TicketMachineApp can be developed in parallel only after the implementation of the WebServer for the opposite reason. The client and the server can be implemented in parallel at the same time thanks to the use of the redirector.

The implementation of each component and sub-component should be executed at the same time with the single-unit testing to, as stated before, to detect and solve errors and bugs as soon as possible to avoid propagation of error during the integration of the components.

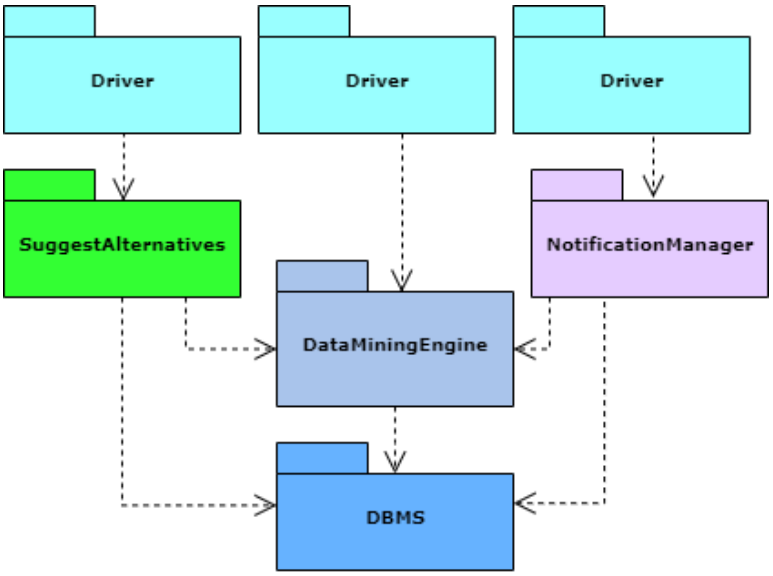
5.3 Integration Strategy

As explained above we decided to use a bottom-up approach to test the different functionalities of the system. Following diagrams explain the process in which these functionalities are integrated between each other and tested progressively, according to the approach chosen.

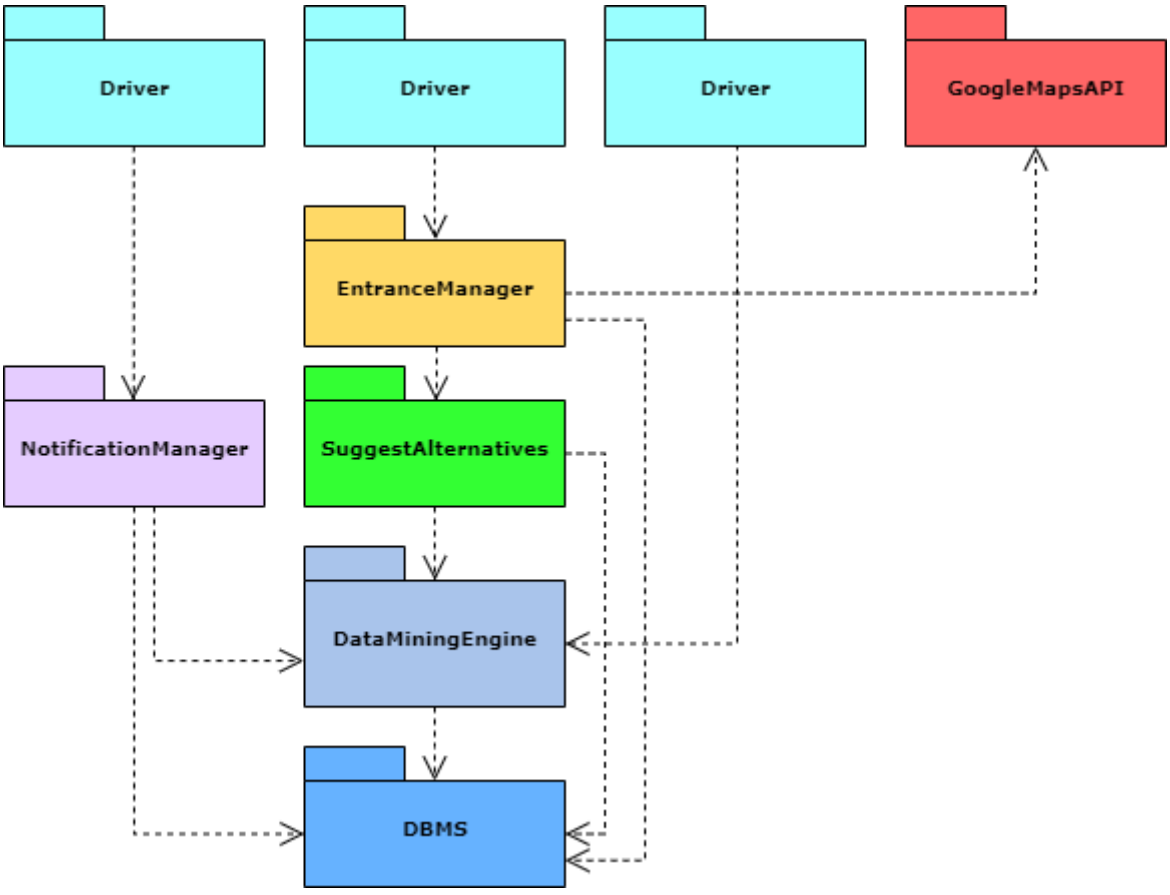
(1) First thing to be implemented and tested is the DataMiningEngine that is a core component for the systems and at the same time not a difficult one. To this is attached a driver to simulate other components that are under implementation yet.



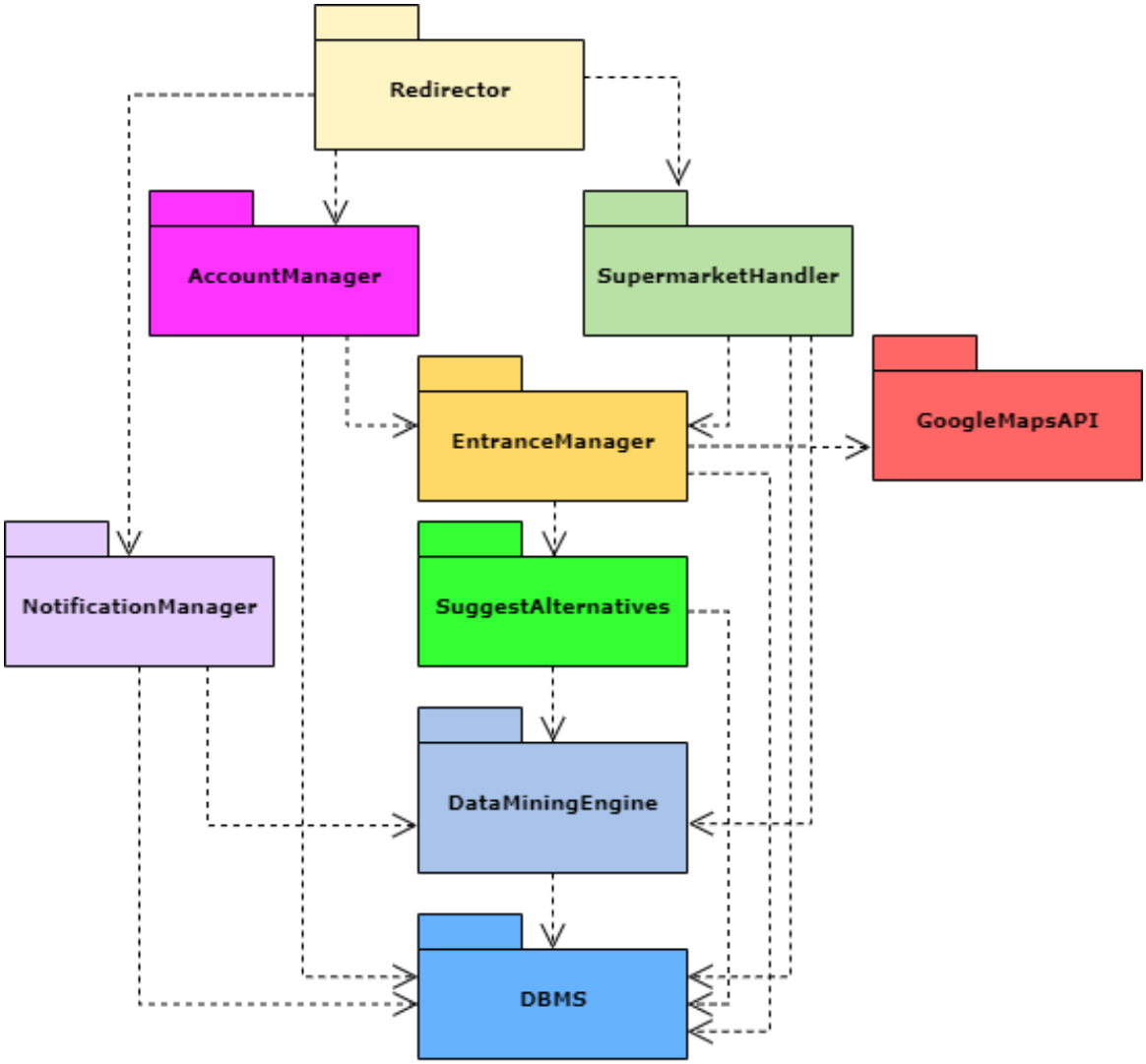
(2) Then, we integrate SuggestAlternatives and NotificationManager component that enable corresponding functionalities of the system, to which are connected correspondent driver for each component. These two components can be implemented and tested in parallel because they are not directly linked each other.



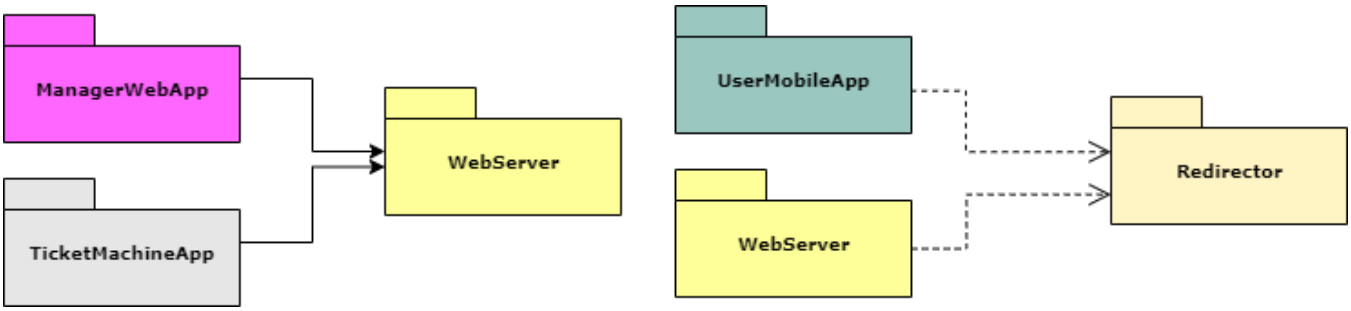
(3) At this point EntranceManager component is implemented and tested. Notice the importance to implement before the SuggestAlternatives component due to the usage-relation between these components. Moreover, it is important to pay attention to GoogleMapsAPI which is not going to be implemented because released by a trusted provider. However, it is going to be integrated in the system through EntranceManager component.



(4) Furthermore, all drivers are substituted by the redirector component which has to be implemented and unit-tested with other components of the application server.



(5) Finally, all the parts regarding the client side must be implement and unit-tested with the whole system. Just after each component is tested and integrated with the system, system testing can be performed.



5.4 System Testing

After a white box testing is completed, whose plan has been fully explained in the previous section alongside implementation and integration, a black box testing (in this case the system testing) is the final stage of the strategy to check the behaviour of the whole system, validating its end-to-end specifications. To reach this goal, the testing environment should be, obviously, as close as possible to the real environment in which the software will act.

The system testing process will follow these steps to in-depth inspect the software to be and try to detect as many bugs as possible to solve them:

1. Environment Setup: this is the first step and, as we stated before, it should be created with the maximum attention to fit as much as possible a real scenario, taking care of every detail.
2. Test Case Creation: during the second step, test cases should be created analysing what are the limit situations that could create problems to the software; this step is fundamental to perform a high-quality testing because a poor test case would not inspect properly possible defects of the system.
3. Test Data Creation: the third step is very correlated to the previous one and it take care of the data that should be use alongside the case formerly created.
4. Test Case Execution: after creating a solid test case with effective data to inspect the veracity of the software, that case should be executed, and its output recorded and analyzed later on.
5. Defect Reporting: the fifth step is to analyze all the generated output during the test case execution to find all the defects the software showed during that simulation.
6. Bug Fixing: in this sixth step, all the bugs, detected in the previous one, are fixed modifying the code.
7. Regression Testing: during this seventh step immediately following the previous one, the test is executed another time after all the modifications to check the presence of bugs that could be either unsolved one or new one generated while fixing the existing ones. If new bugs are spotted, then the process repats from the sixth step.
8. Retest: this last step is meant to repeat the system test cycle with new test cases and test data, before changing environment and scenarios in which the system should be tested.

After pointing out how each kind of system testing should be taken on, it is necessary to explain what are those type of testing planned for our software to be:

- Functional testing: first of all, this type of testing is mandatory to validate and verify the correct implementation of all the requirements and specifications owned by the software and stated into the previous document, the RASD.

- **Performance testing:** this second type aims to test the speed of the software, identifying possible bottlenecks due to bad algorithm design or poor balance of the entire workload among all components. Moreover, the scope of performance testing covers scalability, stability, availability and reliability of the whole system. Therefore, specific tests to underline those aspects should be designed.
- **Load testing:** this third type aims to test the system in an extreme environment with heavy workloads, to record its behaviour and find possible bugs related to a bad memory management, such as memory leaks, overflows; finally, with this type of test, it is possible to determine what the upper limits are and define possible strategies to enlarge those limits.
- **Stress testing:** this fourth type aims to stress the system under difficult situations with loads that consistently and frequently vary to check its robustness. Moreover, since our architecture includes two clones of both web server and application server, the stress test scope covers failure management, highlighting how one clone of both servers can handle the load and how the failure can be recovered.
- **Scalability testing:** the goal of this last type of test is to check the behaviour of the system when it receives too many or too little requests from clients, proving its ability of scaling up and scaling down the resources allocated in this type of situations.

5.5 Useful Guidelines

This last section is dedicated to some useful guidelines that should be followed to simplify and conduct a clear and linear work schedule.

First of all, after reading carefully the documents provided (RASD & DD), the team should be divided into two sub-teams: the development team and the testing team, since these two tasks have different purposes and scopes. While the development team focuses on implementation and integration of components, the testing team focuses on testing the functionalities of the components, both separated and integrated; these two teams should be coordinated since their tasks are dependent between each other:

1. **Implementation** should be done by the development team, which then submits its work to the testing team.
2. **Unit testing,** instead, should be conducted by the testing team, which receives implemented component and tests it.
3. **Integration** will follow and the turn goes back to the development teams, which integrates debugged components between each other.
4. **System testing,** finally, would be done by testing team, that stresses the whole system looking for bugs that need to be fixed.

However, testing process should always run in parallel with all other processes, beginning as soon as possible with the requirement analysis, the revision of the document provided and, obviously, implementation and integration. This is crucial because a software can never be 100% free from bugs and errors and, the sooner they are spotted, the simpler they are fixed, avoiding annoying, time-consuming and difficult to solve bugs in cascade. Another crucial point is to determine a priority queue of core components and sections, mandatory for the system: those have to be tested with the highest priority and as soon as possible.

While building test cases and test data, one of the most important things to remember is that users will surely make mistakes and inserting unexpected data as input and if this fortuity is not considered, it can cause undesired problems. For this reason, it is very important build tests with correct inputs to verify the correct behaviour of the software, but it is also important build tests with flawed, undesired inputs to check if the system is able to recognize the error and request other data as inputs or, in a simpler way, manage the error following the preferred policy.

Last but not least guideline on testing is to never create a test case on the bases of poor assumptions and hypothesis, because those choices could lead to a poor testing activity that will not detect the relevant defects of the system.

Finally, as regard the development part of the project, it is quite important to follow the usual guidelines, briefly reported here:

- Widely commented code to simplify communication between developers and testers and to make the code simpler to change and correct.
- Avoiding hardcodes that can create confusion and make the code more rigid and less reusable.
- Follow an agile software development strategy.
- Use design patterns when needed without abusing them to maintain a linear structure of the whole software.
- Follow the SOLID principles, briefly reported here for the sake of completeness:
    1. Single responsibility principle: each class should have one and only one responsibility encapsulated inside it.
    2. Open-closed principle: each software entity should be open to extension but closed to modifications.
    3. Liskov substitution principle: object in a program should be replaceable with instances of their sub-types, without affecting the correctness of the software.
    4. Interface segregation principle: many separated interfaces fulfilling one task are better than one single general-purpose interface.
    5. Dependency inversion principle: one class should depend upon abstractions, not concretions.

With this section we have concluded with specifications, design and guidelines.

6. Effort Spent

Andrea Manglaviti: 35 hours

Topic	Hours
Discussion on first part	3h
Introduction	1,5h
Overview & high-level architecture	2h
Component diagram	1h
Deployment diagram	2h
Component interface	0,5h
Architectural styles	1h
Other design decisions	1h
Discussion on second part	3h
Mock-ups	17h
Implementation and Integration	3h
Documents Revision	3h

Davide Marinaro: 30 hours

Topic	Hours
Discussion on first part	3h
Introduction	0,5h
Overview & high-level architecture	1,5h
Component diagram	1,5h
Component diagram description	3,5h
Deployment diagram description	3,5h
Component interface	3h
Design patterns	2h
Other design decisions	2h
Discussion on second part	3h
Testing	4h

Documents Revision	3h

Luca Marinello: 38 hours

Topic	Hours
Discussion on first part	3h
Introduction	1h
Overview & high-level architecture	0,5h
Component diagram	4h
Runtime view	15h
Component interface	1h
Architectural styles	1,5h
Discussion on second part	3h
Requirements Traceability	3h
Implementation and Integration	3h
Documents Revision	3h

7. References

- The diagrams have been made with: https://www.draw.io
- The mock-ups have been made with Adobe XD: https://www.adobe.com/it/products/xd.html
- Oracle Documentation: https://www.oracle.com/