A.Y. 2020-2021

Politecnico di Milano, Software Engineering 2 project

POLITECNICO
MILANO 1863

DD

Design Document

version 1.1

Cirino Duilio - 968466

Cocchia Lorenzo - 968139

Table of contents

1. INTRODUCTION

1.A. Purpose

This document aims to provide a specification for the architecture of the CLup system yet analysed in the Requirements Analysis and Specification Document. This means that this document is complementary to the RASD, but focusing on the specification on the architecture and on some information about its implementation, including the testing part of the project.

1.B. Scope

CLup is a software-based system with the aim to provide instruments to both customers and managers of groceries' shops in order to avoid crowds and more generally contact between people. This need comes from the increasing plague from 2019's pandemic given by SARS-CoV2 viral strain. Anyway, a future application could be well-suited even in situations in which the rules are not the same as those created during the pandemic plague: this system could improve efficiency of the commercial entities and customer satisfaction.

The reader can find a more detailed description of the scope of this project in the RASD(1.E).

1.C. Definitions, Acronyms, Abbreviations

In this document there are referred to the definitions, acronyms and abbreviations yet exposed in the RASD, here are listed some of them considered more important together with some other peculiar to this document.

RASD	Requirements Analysis of Specification Document
Gn.m	Goals identification n = macro-goal number m = sub - goal number
Dn	Domain assumptions identification

	n = domain assumption number
Rn	Requirements identification n = requirement number
DD	Design Document
RBAC	Role based access control
DBMS	Database Management System
HTTPS	Hypertext Transfer Protocol over Secure Socket Layer
DMZ	Demilitarized Zone
IFML	Interaction Flow Modeling Language
ER	Entity-Relationship
UX	User Experience
UI	User Interface
OSM	OpenStreetMap. An open-source collaborative project providing a maps service

1.D. Revision history

This is the version 1.1 of this document. This version has been made after a consultancy with an expert professional in contact with industries and a first stage in which it was implemented a prototype of the application.

Version	Date	New features
1.1	4/2/2021	✓ Components diagram includes DBMS ✓ ER improved

		✓ Clarifications about the information modelized in the sequence diagrams ✓ Modifications on the components interfaces while implementing
1.0	10/1/2021	✓ first version of the document

1.E. Reference Documents

- Requirements Analysis and Specification Document v1.1 (find it at this link)
- Project specification document, from Software Engineering II course A.Y. 2020/2021

1.F. Document Structure

This document is composed of 6 chapters.
The first section introduces the design document exposing the purposes and scope of the document and fixing some conventions used in the document. Here there is even information about the referenced documents and about the revision history of it.
The second section exposes the main architectural components of the system and the relationships between them and the needed services, there are even references on their dynamic behaviour and deployment.
The third section describes more specifically the user interfaces and user experience for the project already described in the RASD.
The fourth section is a direct link with the RASD: every requirement and every goal specified in the RASD is mapped with a component already described in the DD.
The fifth chapter shows a suggested plan for implementation and testing of the software-to-be accompanied with some more detail on the algorithms that should be used to accomplish such implementation.

The last chapter is composed of a table describing how much effort the components of the group paid quantified in hours.

2. ARCHITECTURAL DESIGN

2.A. OVERVIEW

This section of the document is fundamental to comprehend in depth how to actually implement the service from an high level of abstraction and in order to have a clear view on how the system will be structured in each of the following subsections:

1. Component view: which will illustrate the components of the system, how they work and how they interact between each other and with the related services;
2. Deployment view: which will illustrate the hardware resources on which the system will work on and how the different nodes must be connected and logically and physically divided;
3. Runtime view: which describes the behaviour and interaction of the system's components in some use cases;
4. Component interfaces: which takes a closer look at the the components' interfaces;
5. Selected architectural styles and patterns: which will describe and explain the architectural style chose in the previous parts and the presence of known design patterns;
6. Other design decisions: which will state any other relevant design choices which were taken while designing the platform

A.1. High-level components and their interaction

Here we can see the three PAD (presentation, application and data) on which we based our architecture:

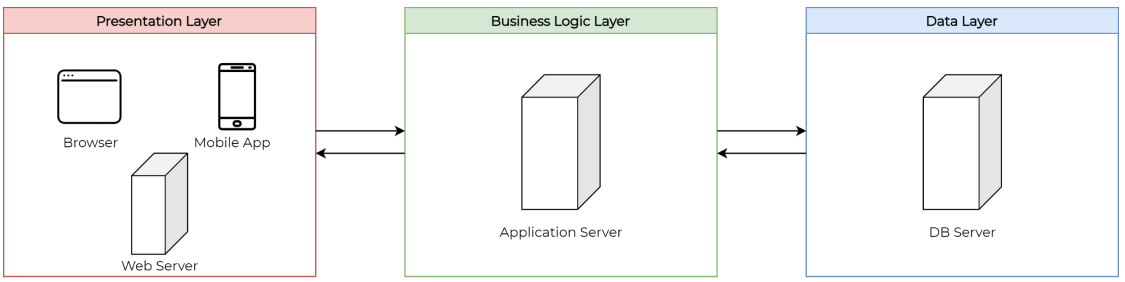
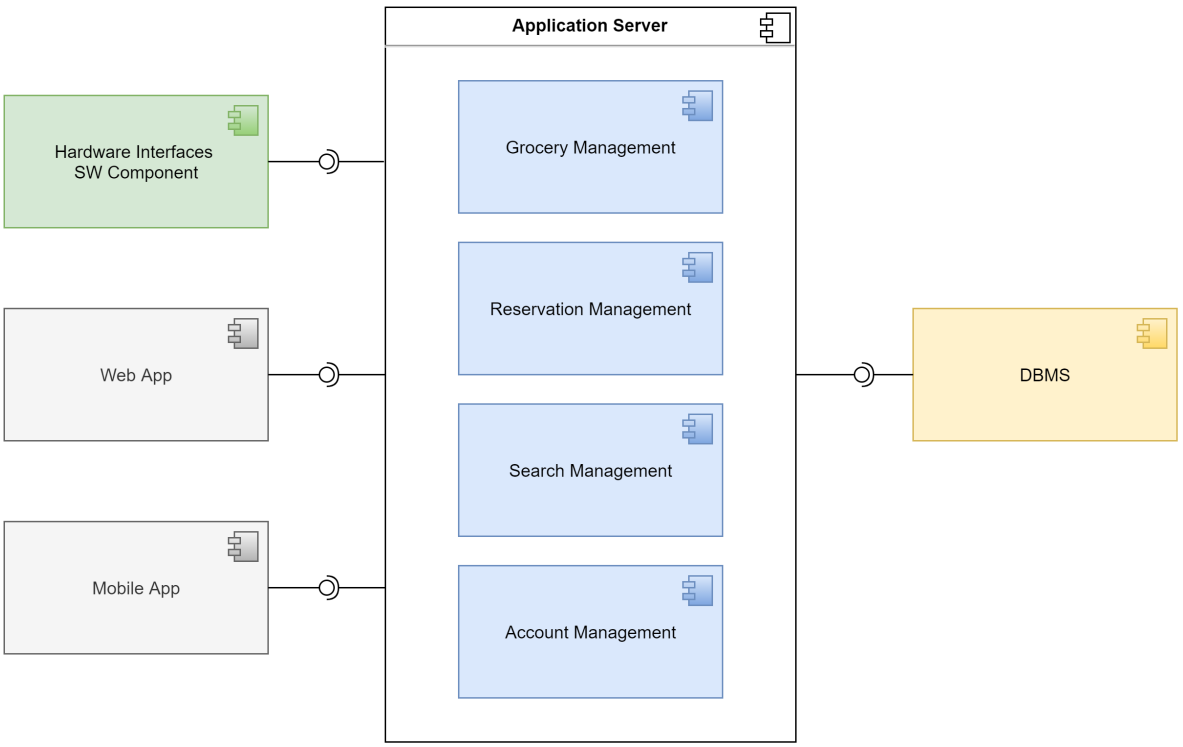


Figure 1

Starting from this structure, we decided to distribute the layer in 3 tiers plus an additional one. They are:

1. Client tier, dealing with the final user and the related presentation technology;
2. Web tier, which manages the requests made by the client through the web and forwards the information from the business logic to the client;
3. Business logic tier, which deals with business logic and is the proxy between the web tier and data tier;
4. Data tier, which permanently stores the business data needed by the software.

We chose this design in order to guarantee greater scalability and flexibility to the system. Moreover an architecture of this type improves greatly security regarding data access from the users of the service.
Given this architecture the suggested solution, both for ease of implementation and efficiency, is to divide the three layers of presentation, business logic and data, shown in figure 1, in the four tiers of the architecture as we previously stated, but also some logic is accepted on the client and is applicable if it is considered good to implement in favor of a better user experience.
We give more details on this matter in section [2.C]



In figure 2 we can see the system's high-level component architecture, and the relation between these components.
The two main client side components are (grey):

- Web App: accessible for every user of the service;
- Mobile app: accessible for every user of the service but intended for customer use.

We can also see the presence of a component not server related but that interacts with its components (green):

- Hardware Interfaces Software Components.

These generic components are optional components, store-side, that communicate with the reservation management, through their dedicated software, in order to do things such as letting people into the store.
Additional information on this component can be found in the RASD document [3.A.2 Hardware Interfaces].
While the business logic layer presents four components, each serving one of the main functionalities of the product:

- Grocery Management: component used by store managers to edit and manage every asset of their grocery;
- Reservation Management: component that enables customers to add, delete or edit every type of reservation, and gives managers and employees the same functionality;
- Search Management: component that serves the customers to find local stores;
- Account Management: component that makes every user able to add or modify account information.

Finally, we can see(yellow):

- DBMS: manages the communication with the database

2.B. COMPONENT VIEW

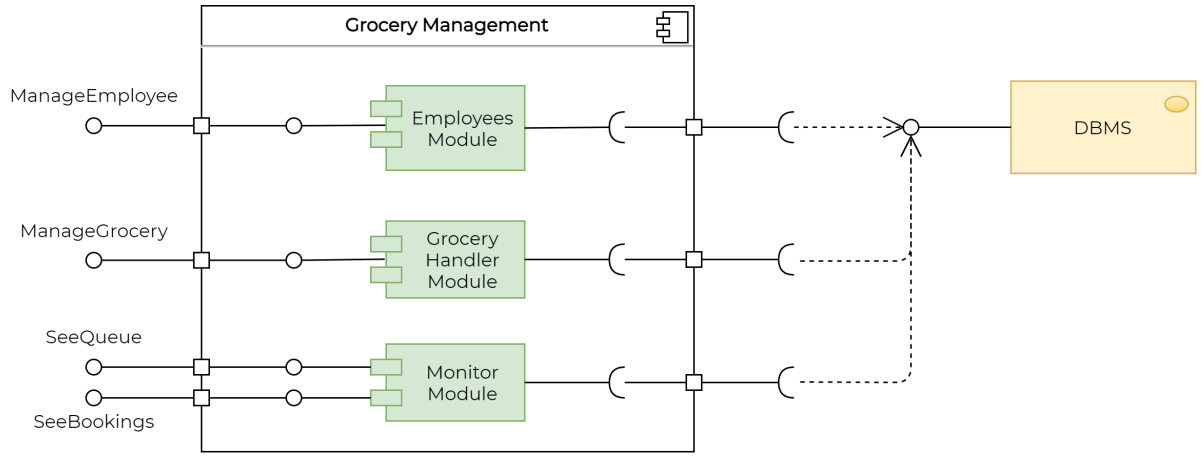
In this section we will take an in-depth look at the server-side components previously listed, see how they interact between each other and the modules they have if present.
Components will be accessible based on the user's role, so for this part we suggest the use of *role based access control (RBAC)*.

As it was visible even in the precedent section, it has been used a color notation in order to refer to different components of the architecture. From now on, it is used the sequent color notation in order to better understand the component provenience of the architectural modules and of the actors.

Colors legenda

Color	Actor or high-level module
	Customer
	Administrator privileges: employee or manager
	Client-side component (Web application)
	Grocery Management
	Reservation Management
	Search Management
	Account Management
	SMS Gateway Service
	Maps Service
	DBMS Service

B.1. Grocery Management

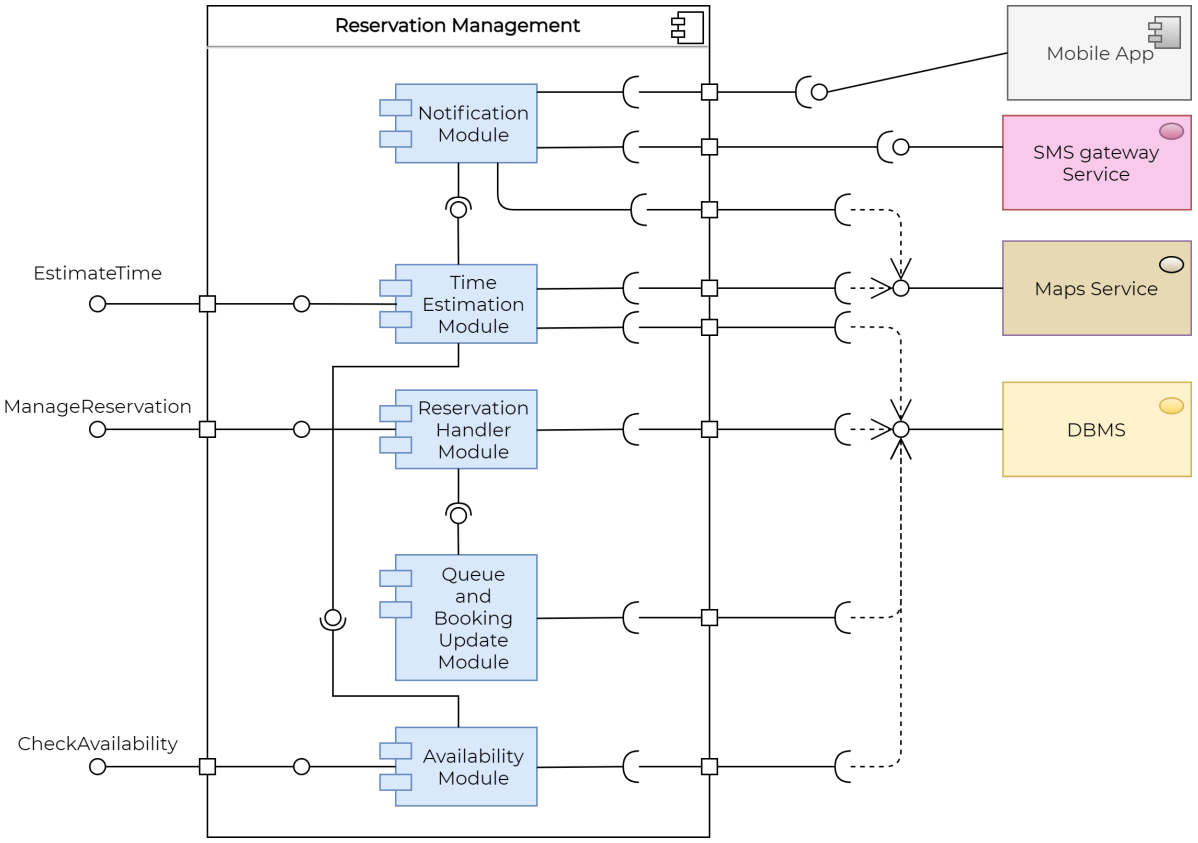


The grocery management component is made of all the modules intended mainly for manager and employees to use.

- The **employees module** is accessible for managers only and it serves the purpose of managing employees, adding or removing them, but also see a list of them and the respective information;
- The **grocery hander module** is accessible for managers only and it lets them update or add informations about the store, such as opening hours or grocery capiency, but also let the manager add a new store or deleting them;
- The **monitor module** can be used from all users, it lets the store workers take a look on the grocery's queue and bookings and the customers to see the situation about the queue;

As it regards reservations, managers and employees will make use of the reservation manager component that we will describe in the next subsection.

B.2. Reservation Management



The reservation management component serves the purpose to handle every operation regarding reservations, from their creation to their end.
It is divided in 6 modules that focus on different aspects of reservations:

- The **notification module** manages the sending of alerts to the customers both via telephone number or via mobile app push notifications;
- The **time estimation module** calculates on request or periodically the expected time to enter the store;

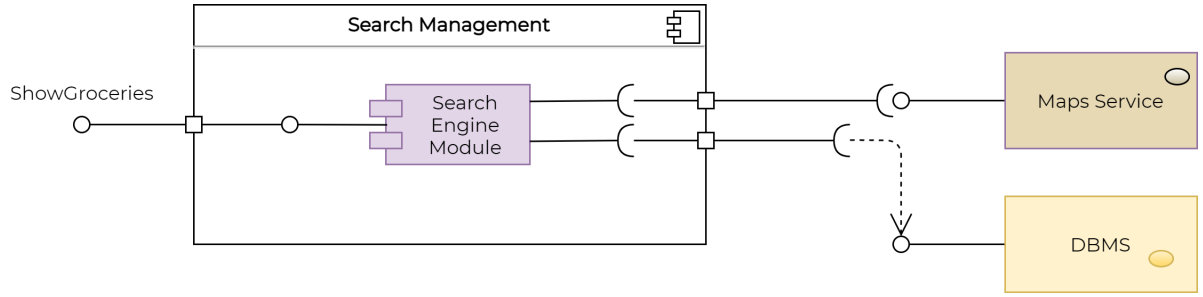
As we can see the two modules are related, because the notification module will have to send notifications to the customer for approaching the store based on his position and to do so, the two modules must be able to communicate.
Then we have:

- The **reservation handler module** offers the functionalities to add, edit and remove a reservation. This module will be used from all users, but also from the QR reader component (as we can see in the

figure 2 in D.1), in order to correctly store the entrance and the exit of any customer;

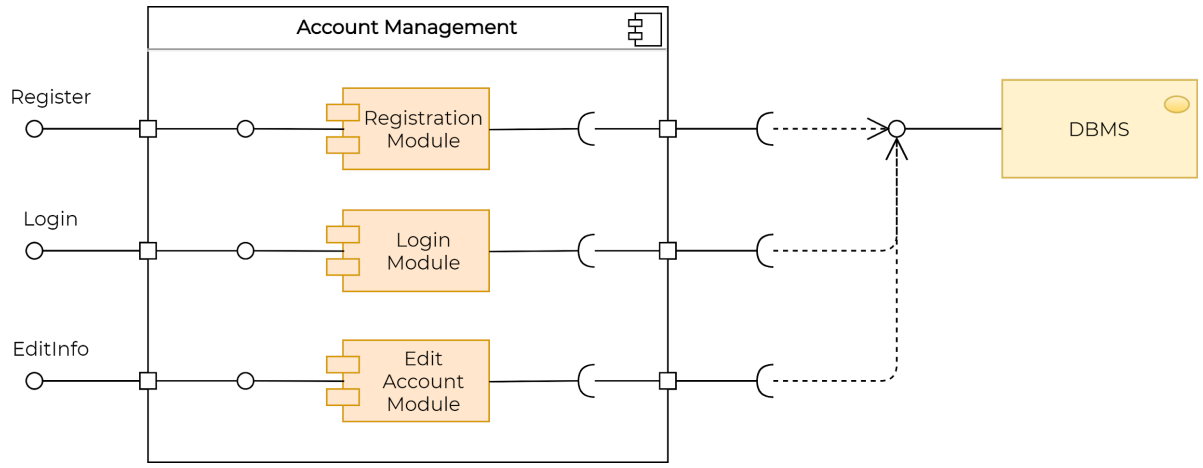
- The **queue and booking update module** is used by the server to automatically apply any changes to the data structure of a set grocery when every add, edit or close action is made by the respective modules;
- The **availability module** is used by the user in order to see the availability to make a reservation to a certain grocery. Doing so, it uses the information produced by the Time Estimation Module.

B.3. Search Management



The search management component is made only by one module that will serve the customers' need to search for a grocery near their location and for registered users it offers a list of the favourite stores of the user. It will make use of the Maps Service to localize the groceries.

B.4. Account Management



The account management component will be used by every user of the software making possible every operation needed on it thanks to its modules:

- The **registration module** will take care of every register request, be it a user or a manager (we remember that employees are created and added to the system by the store manager) and will commit the result to the database if all the required info are correctly inserted;
- The **login module** will be used to access accounts by each user of the software, by checking the correctness of the data filled to authenticate;
- The **edit account module** will be used by every user to modify some optional and changeable information of the account and successfully commit it to the database.

B.5. Services

a. Maps Service

As it regards the *Maps Service* we previously remained generic on purpose. Every third party map service API can be used for the purposes of this software if it at least provides the possibility to give to the system the capability to calculate the time that separates the position of the customer to a grocery store.
We suggest two maps API with two different approaches to the developer of the system: a commercial one offered by Google Maps API and one open-source one offered by OpenStreetMap. Further information about

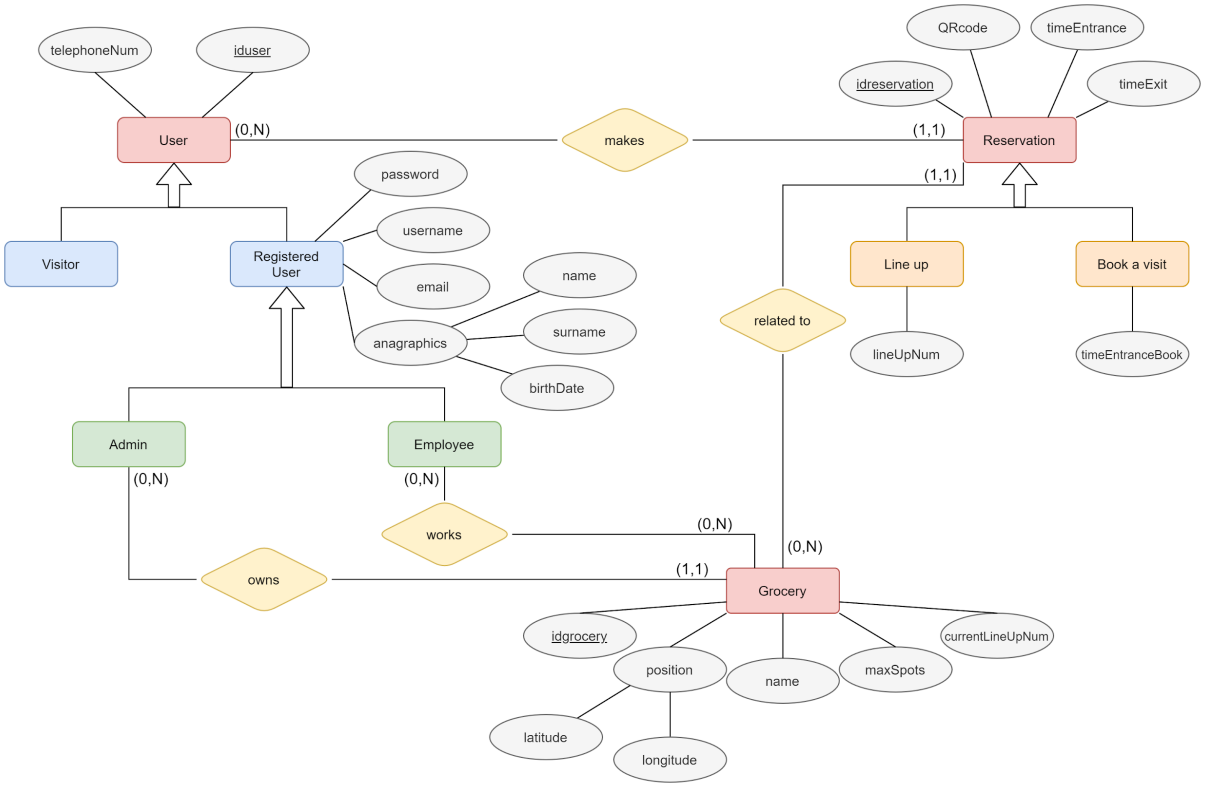
these two services and their main differences will be presented and discussed in the section 5.A of this document.

b. SMS Gateway

An SMS gateway allows a computer system to send or receive SMS to or from a telecommunications network, thus to or from mobile phones of clients. This service will be used by the CLup system to send notification about the state of their reservation to the customers.

c. Database

The database has to contain all the needed business data. Here is provided an ER model for the conceptual model of it. Here colors notation is not used and colors have the only aim to better represent entities, relationships and attributes.



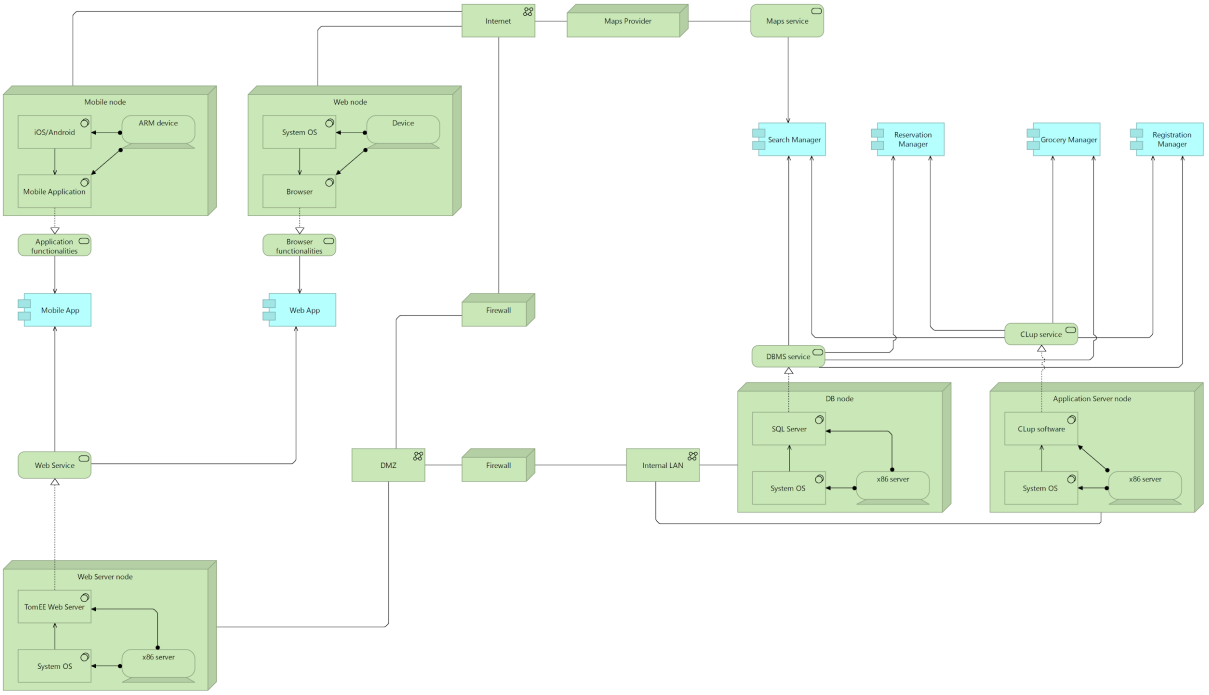
2.C. DEPLOYMENT VIEW

This section of the document illustrates the topology of the system hardware, specifying the distribution of the components in the real world, in physical nodes each offering different services to fulfill certain functionalities, highlighting how each node interacts with each other logically and physically.
As previously stated we opted to choose a 4-tiered architecture separated respectively:

- Tier 1: composed by the *web and mobile applications nodes*. This is the *client tier* and it represents one part of the *presentation layer* of the system. We do not suggest implementing application logic in this tier other than a few checks on user input correctness.
- Tier 2: contains only the *web server node* and is used by web app users to get pages and serves also the web service functionality from which users can get access to their functionalities. This tier represents the other part of the *presentation layer* of the system. The web server is placed inside a *DMZ* that is a physical or logical subnetwork that contains and exposes an organization's external-facing services to an untrusted network such as the *Internet*.
- Tier 3: composed only by the *application server node.* It represents the *business logic layer* of the system. This is the first node inside the Internal LAN of the system and it is home to the *CLup software* containing all the 4 components described in the component view. A best practise in this tier could be to divide each component in a relative software each belonging to a different node of the infrastructure to improve greatly performances.
- Tier 4: represented by the *DB node*. This tier represents the *data layer* of the system. We recommend the use of a relational database for the system.

All communications between nodes outside the internal LAN of the system will rely on HTTPS while Internal LAN communications will rely on simple TCP. Access to the internal LAN from outside is allowed through two

network nodes the Internet and the DMZ, and two firewalls separating respectively the DMZ from the Internet and the DMZ from the Internal LAN, all of this guarantees great security and avoids unauthorized access to the business logic and data layers of the system.
In this section the color notation is not valid. We developed this deployment view using ArchiMate modeling language. The green nodes represent infrastructural services and components, while the blue ones represent the application components.



The upper figure can be found even in a directory attached to the git repository (link on References).

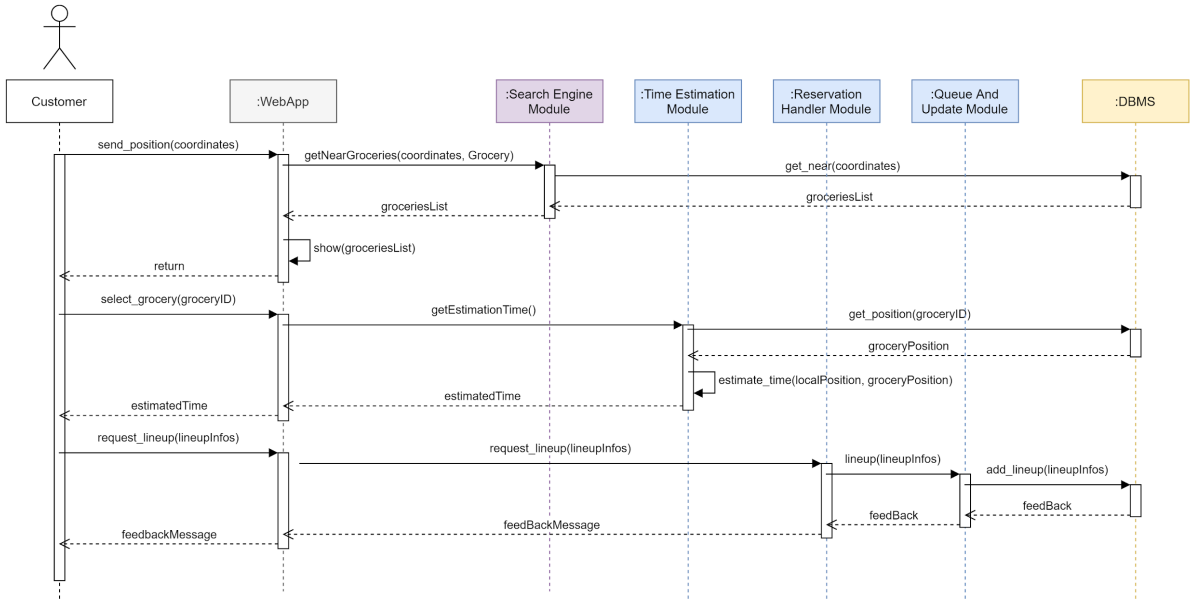
2.D. RUNTIME VIEW

In order to provide a more complete overview on the dynamic behaviour between the components of the high-level architecture described in the previous sections, in this paragraph some UML sequence diagrams are shown. These will describe some use cases already defined in RASD. With the aim to align the notation between this document and the requirements' one, the use cases will be identified with the use case number assigned in the RASD use cases definition (chapter 3.B.2). Use cases should be self-explanatory, but detailed information about use cases such as input and output conditions are exposed in the RASD. In order to maintain a graphical readability we used some name-shortcut: better explanations are provided together with the use case.
There will be described the next use cases:

DD use case number	Brief description	RASD use case number
1	A customer lines up	1
2	A customer watches the available spots in a given supermarket	3
3	A customer books a visit to the grocery	4
4	A customer uses a fallback option	6
5	A user logs in	9
6	A manager registers a store as a manager	10
7	A manager retrieves information about the grocery's data	11

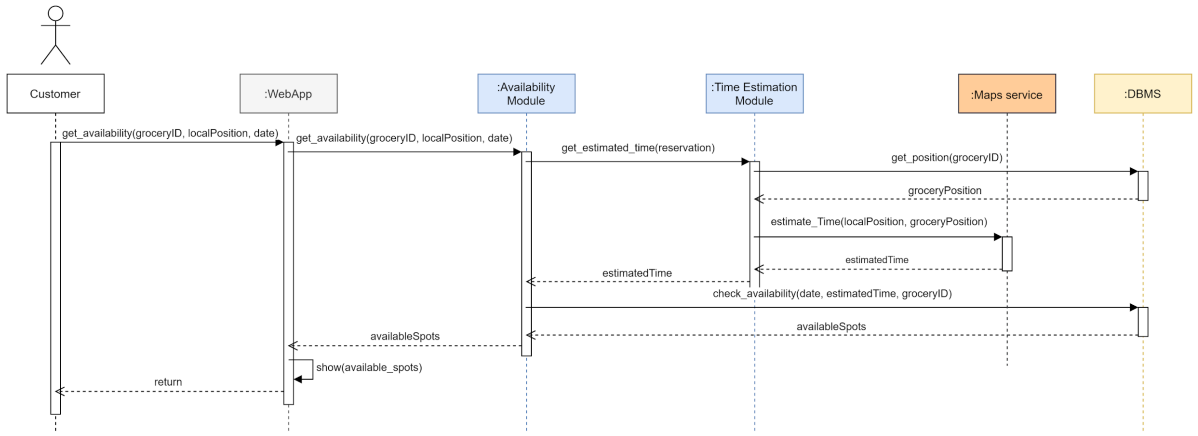
For all the sequence diagrams, original images can be found even in a directory attached to the git repository (link on References). There are some notes below the diagrams, that are referring to some type of information that can depend on the logical model database used.

D.1. A customer lines up

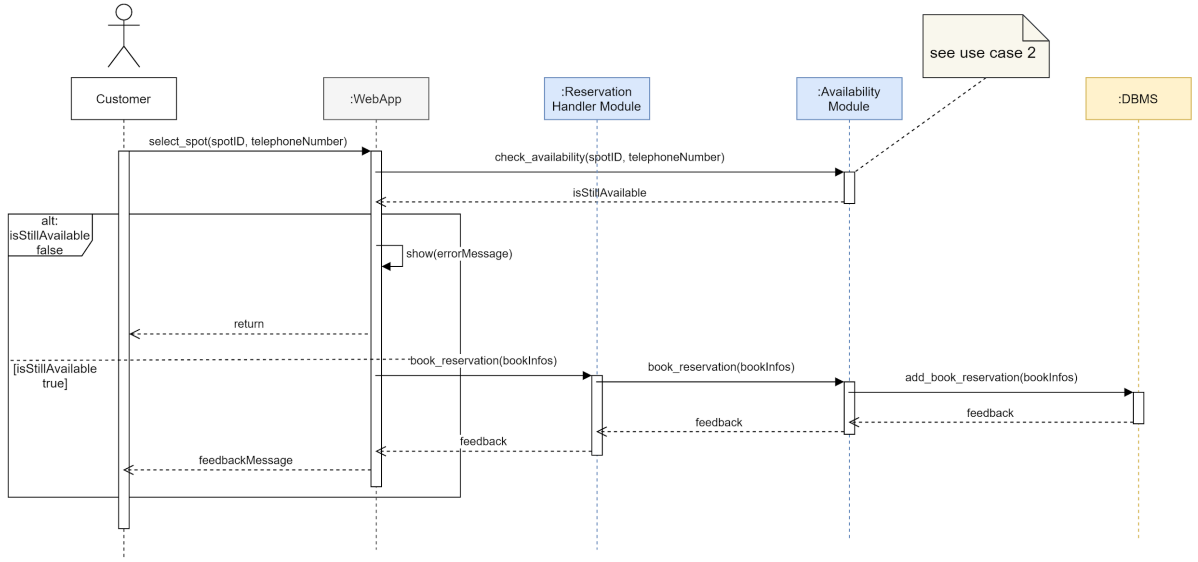


Here, lineupInfos is referred to all the information useful to make a line-up reservation.

D.2. A customer watches the available spots in a given supermarket

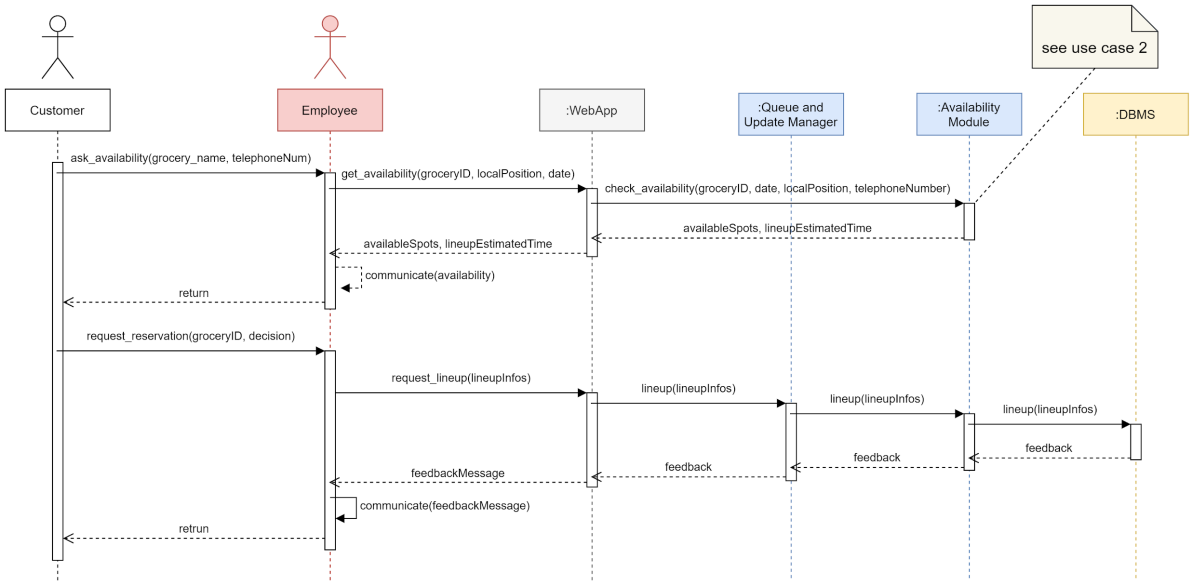


D.3. A customer books a visit to the grocery



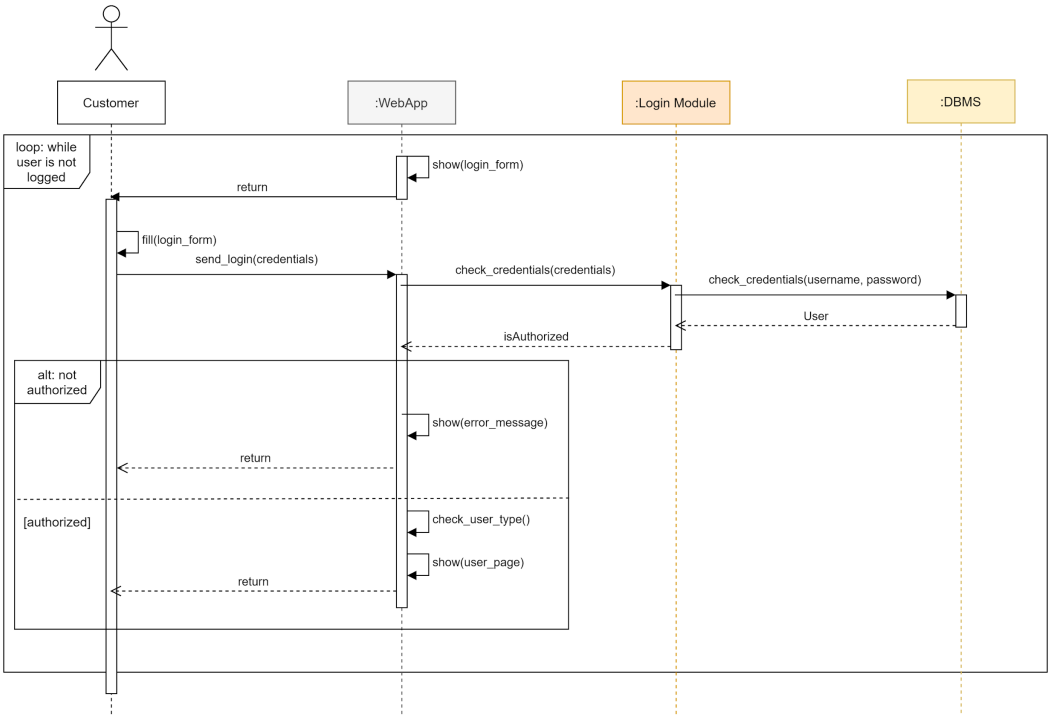
Here, bookInfos is referred to all the information useful to make a book-a-visit reservation.

D.4. A customer uses a fallback option

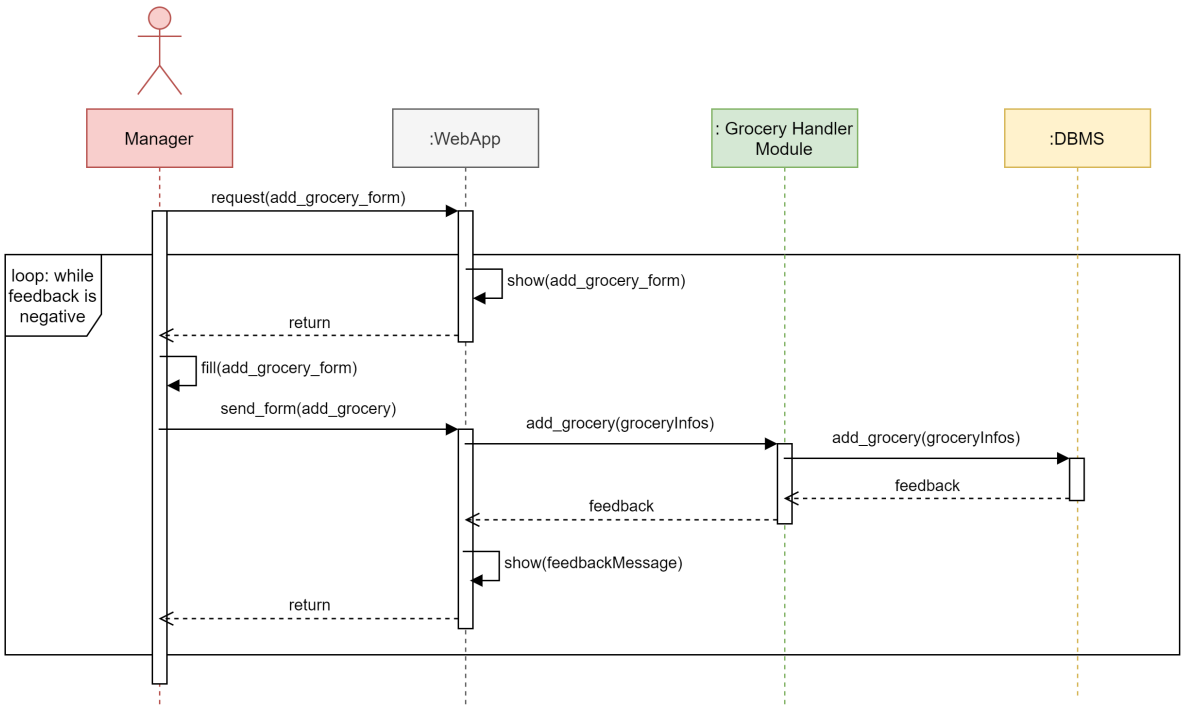


Here, lineupInfos is referred to all the information useful to make a line-up reservation.

D.5. A user logs in

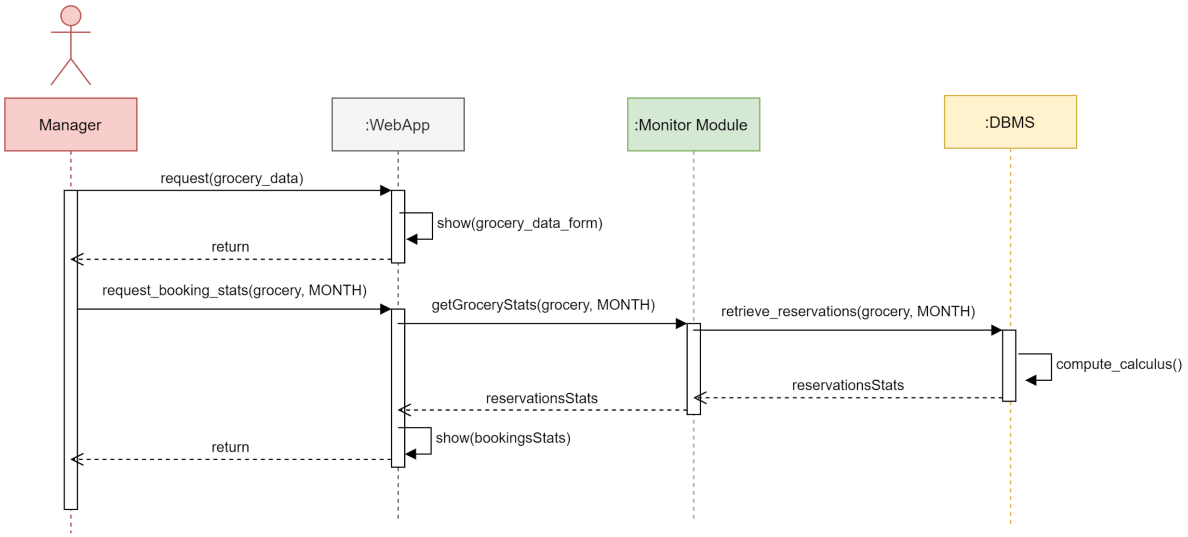


D.6. A manager registers a store as a manager



Here, groceryInfos is referred to all the information needed to define a grocery.

D.7. A manager retrieves information about the grocery's data

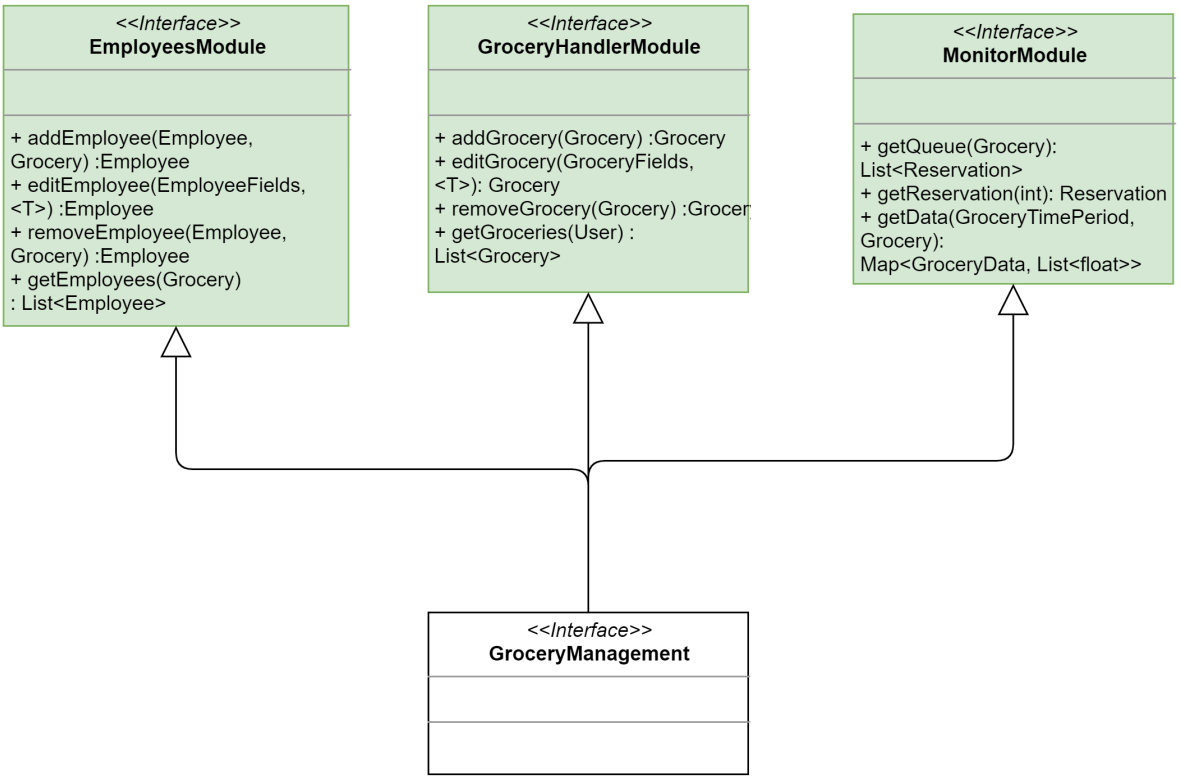


For this use case, it is shown the case in which the manager wants to see the bookings of a certain grocery in a month.

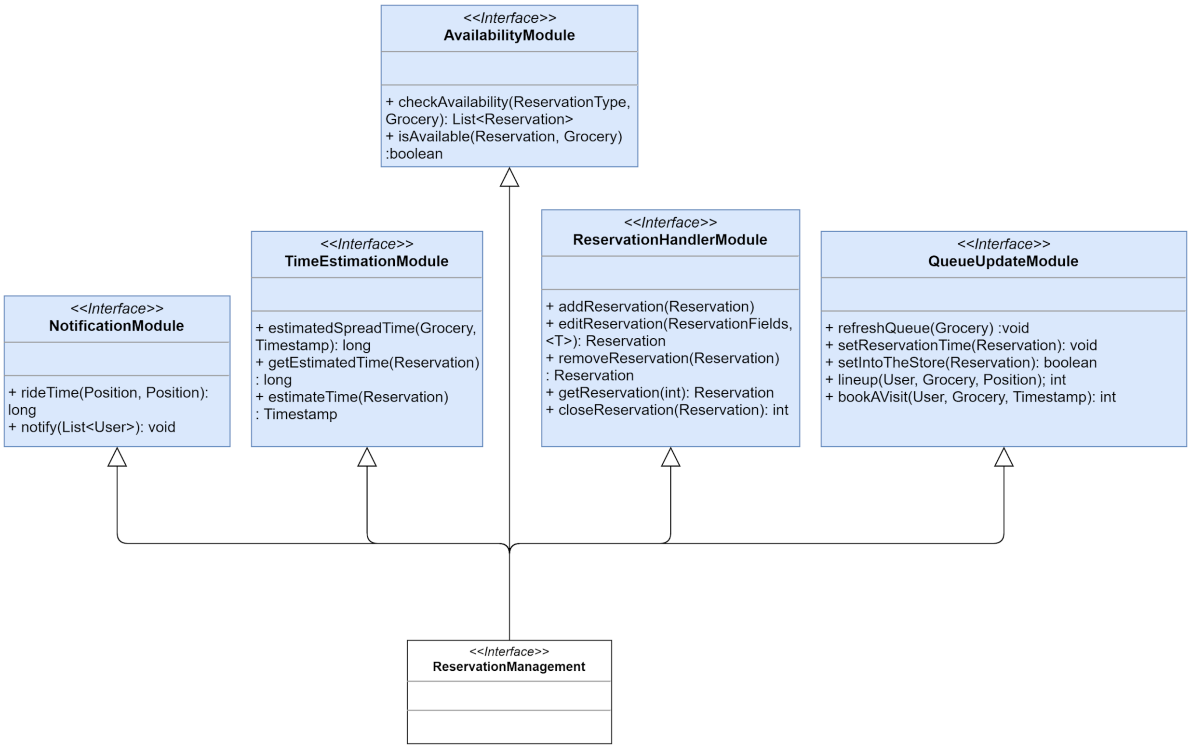
2.E. COMPONENT INTERFACES

In this section there are some UML diagrams representing the expected interfaces to be developed with some main method. Each UML class diagram represents a particular high-level component.

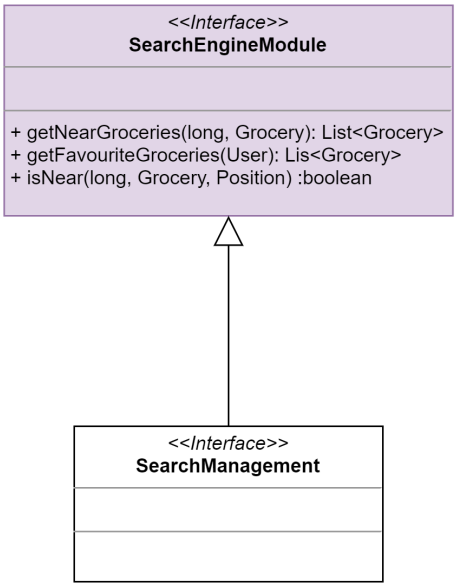
E.1. Grocery Management



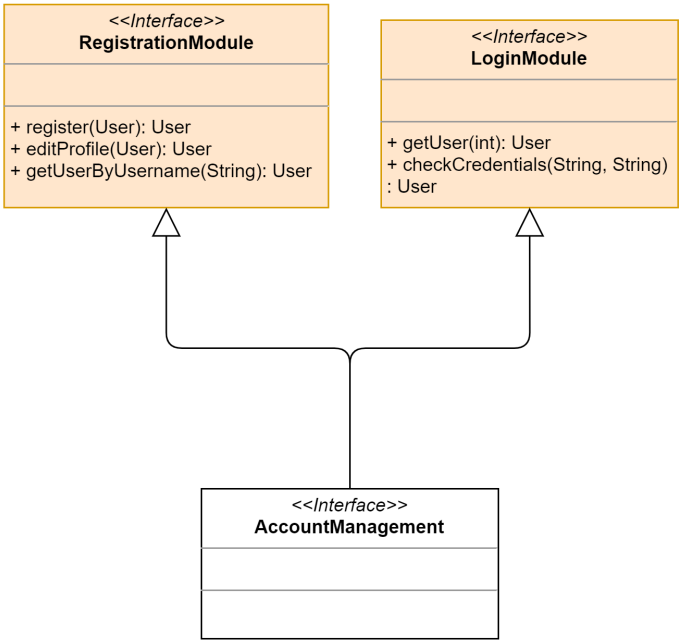
E.2. Reservation Management



E.3. Search Management



E.4. Account Management



2.F. SELECTED ARCHITECTURAL STYLES AND PATTERNS

The main architectural style chosen has been the *client-server web-based* one. Looking at the components provided in the previous sections, the architecture proposed is a “thin client” one. But, to improve the user experience and make both the mobile and web interfaces more interactive, it could be possible even to select a “fat client” approach: in which more business logic could be implemented client-side. Since the software should follow the client-server architectural pattern, it becomes natural to choose even a clear *Model-View-Controller* (*MVC*) and event-driven architectural patterns. Being a web-based architecture, there are some essential architectural rules to follow:

- Client and web server use *HTTPS* protocol to communicate.
- The presentation to the client is made through pages described through markup languages, for instance XML or HTML.

The presented architecture has been inspired by the *Java Enterprise Edition* (JEE) framework. For this reason, and for other reasons related to facilitation on the implementation and testing and managing of

non-functional requirements, we strongly recommend to use such software framework.

2.G. OTHER DESIGN DECISIONS

Design decisions should follow the experience, technical skills and resources of the implementation group.

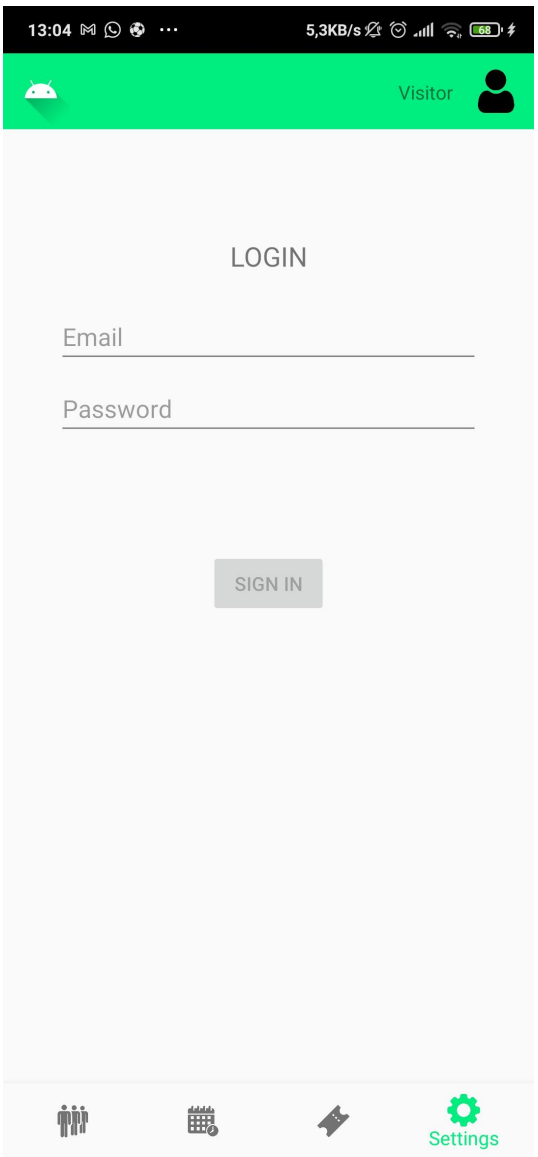
3. USER INTERFACE DESIGN

3.A. Interfaces design

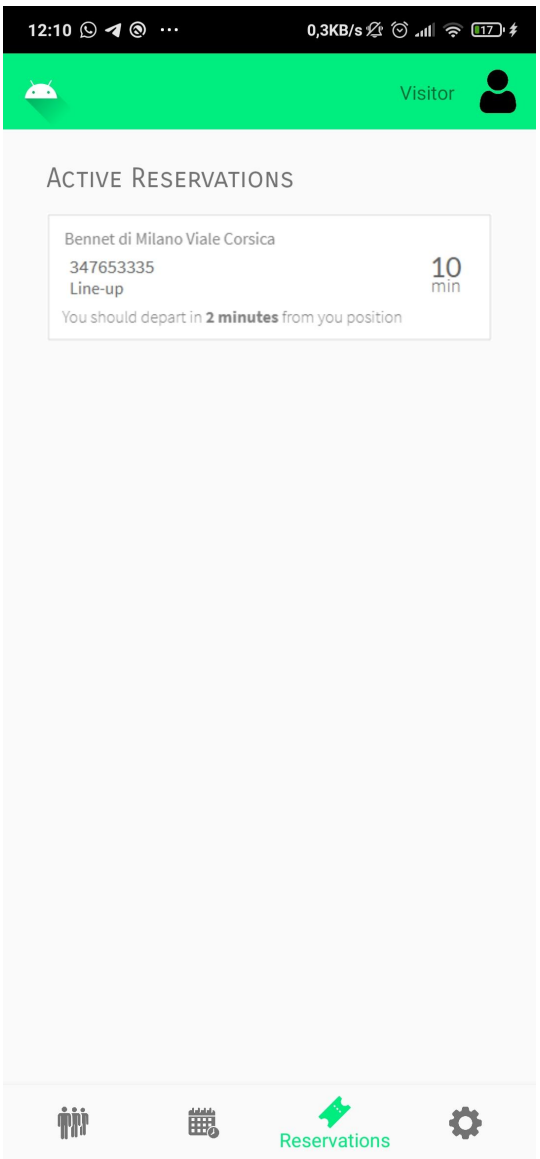
The following are intended as a suggestion on how to structure the web and mobile applications, obviously if the developers think that the views represented in section 3.2 can be displayed and organized in a better way they can go with their style.

A.1. Mobile UI

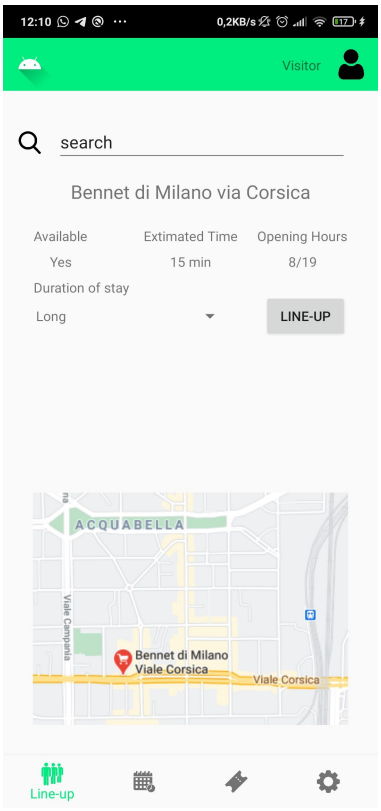
○Login



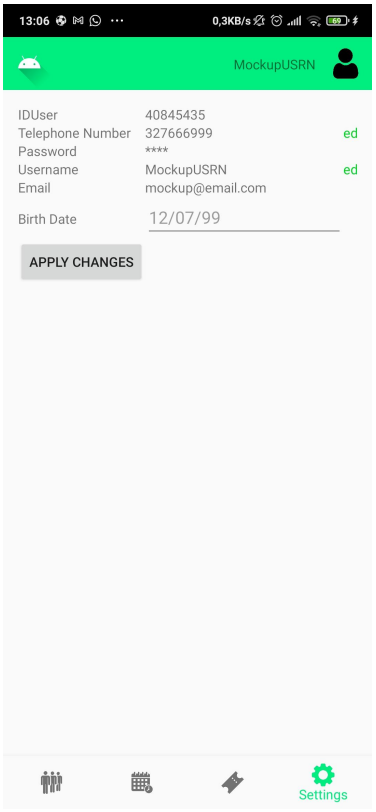
○Active Reservations



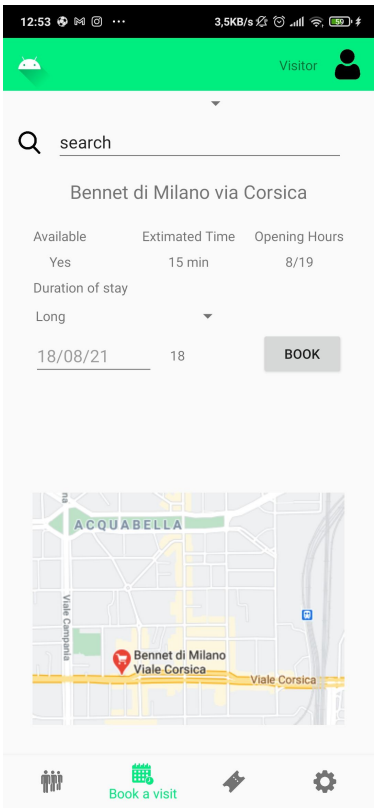
○Line-Up



○Edit Profile

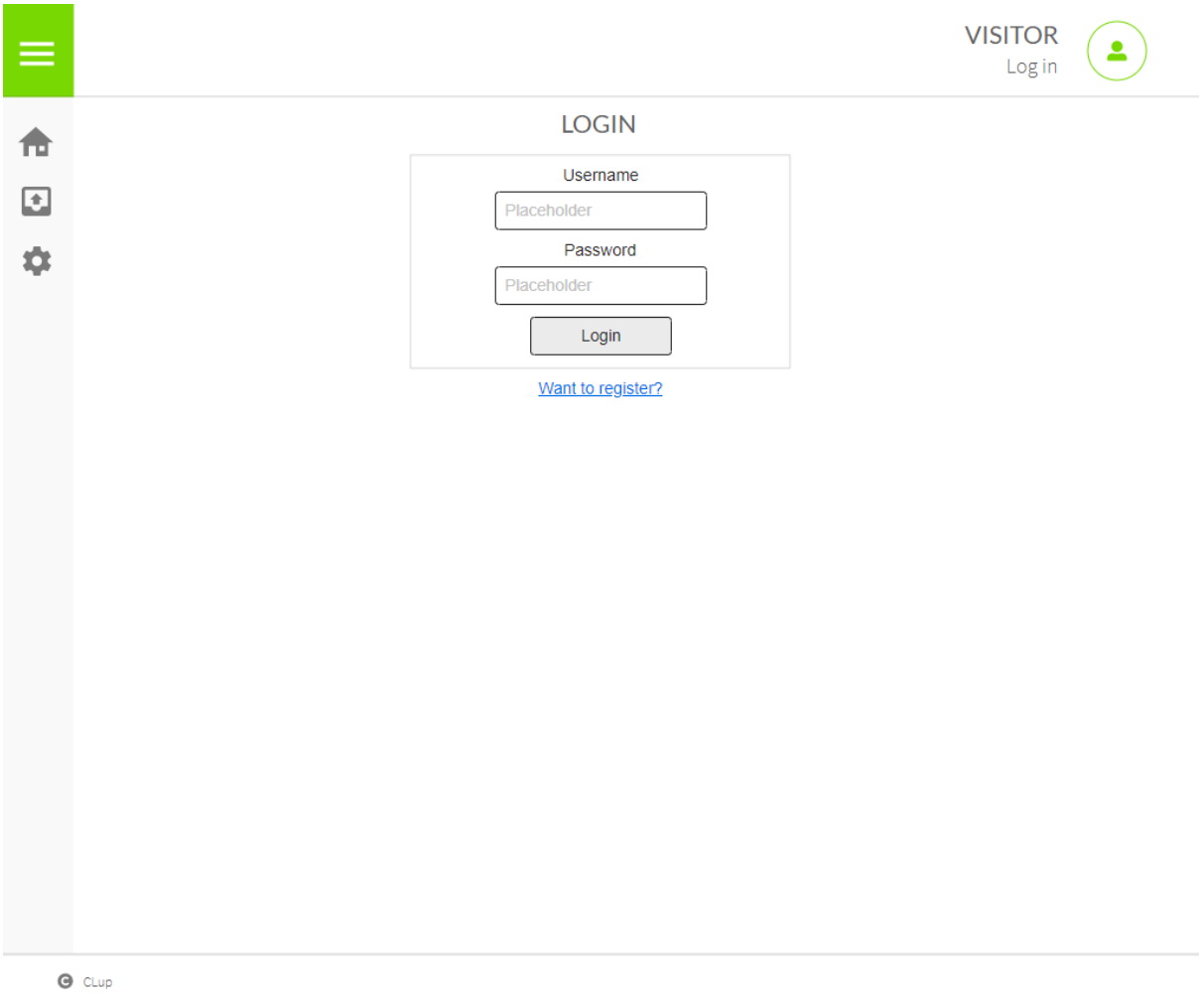


○Book a Visit

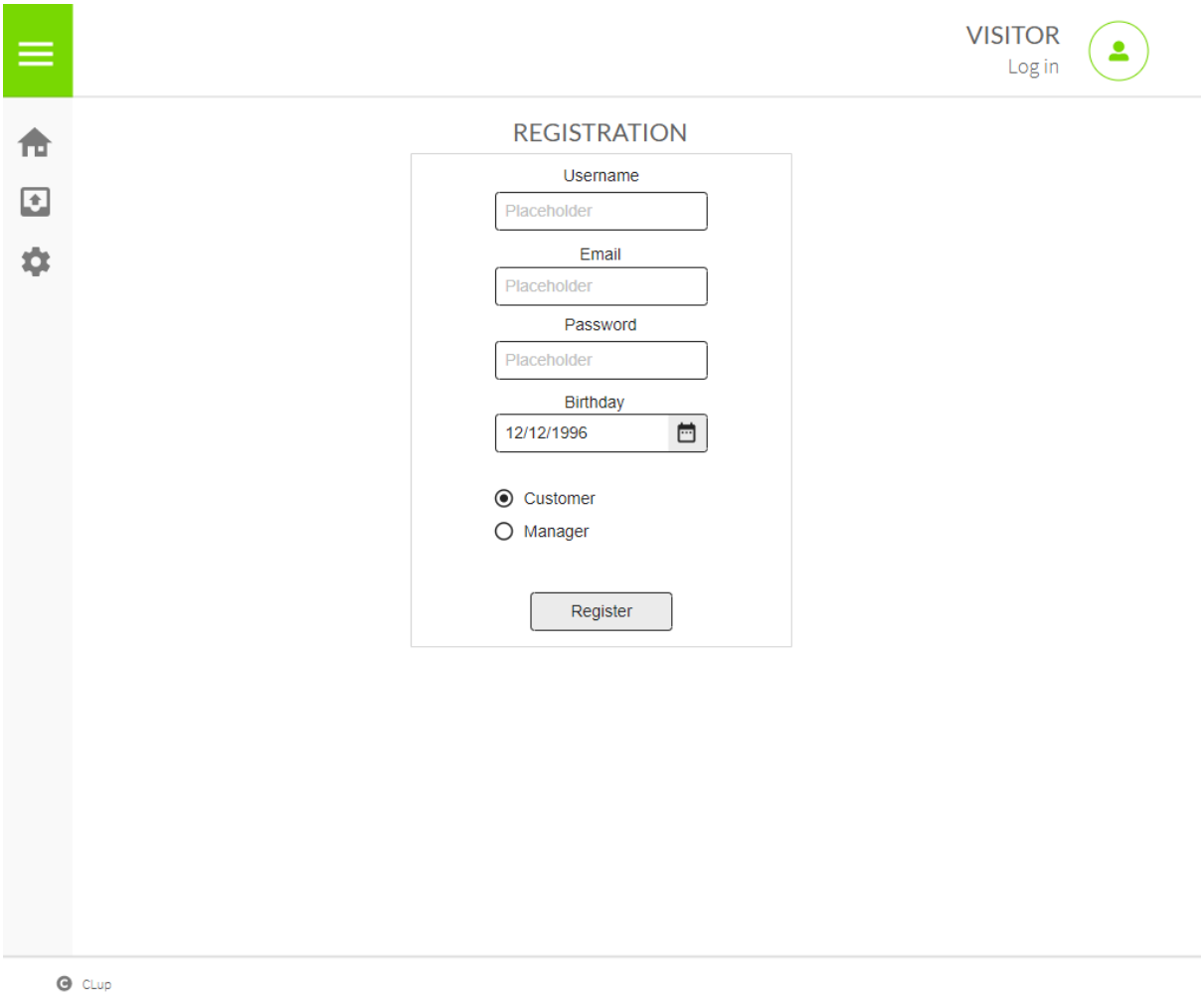


A.2. Web UI

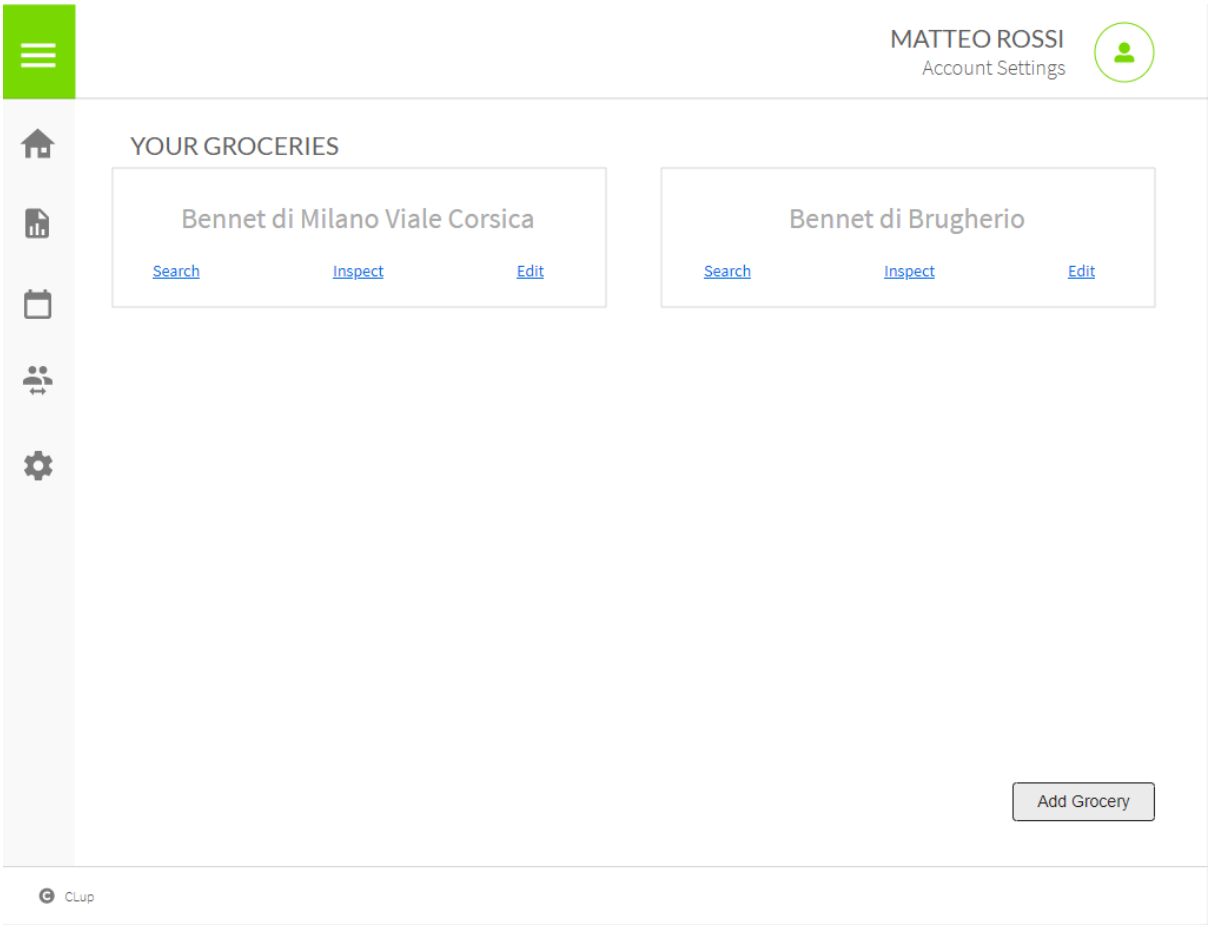
- Login



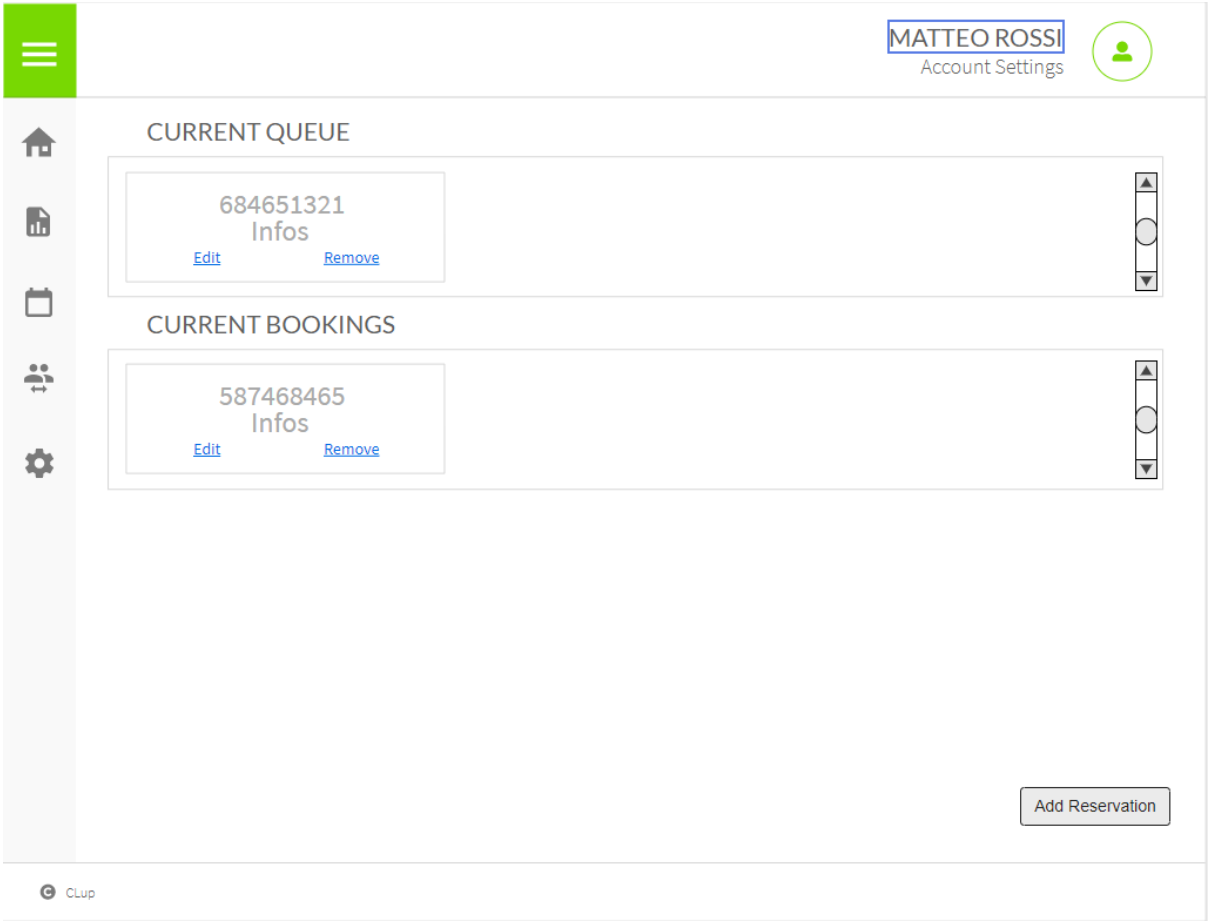
- Registration



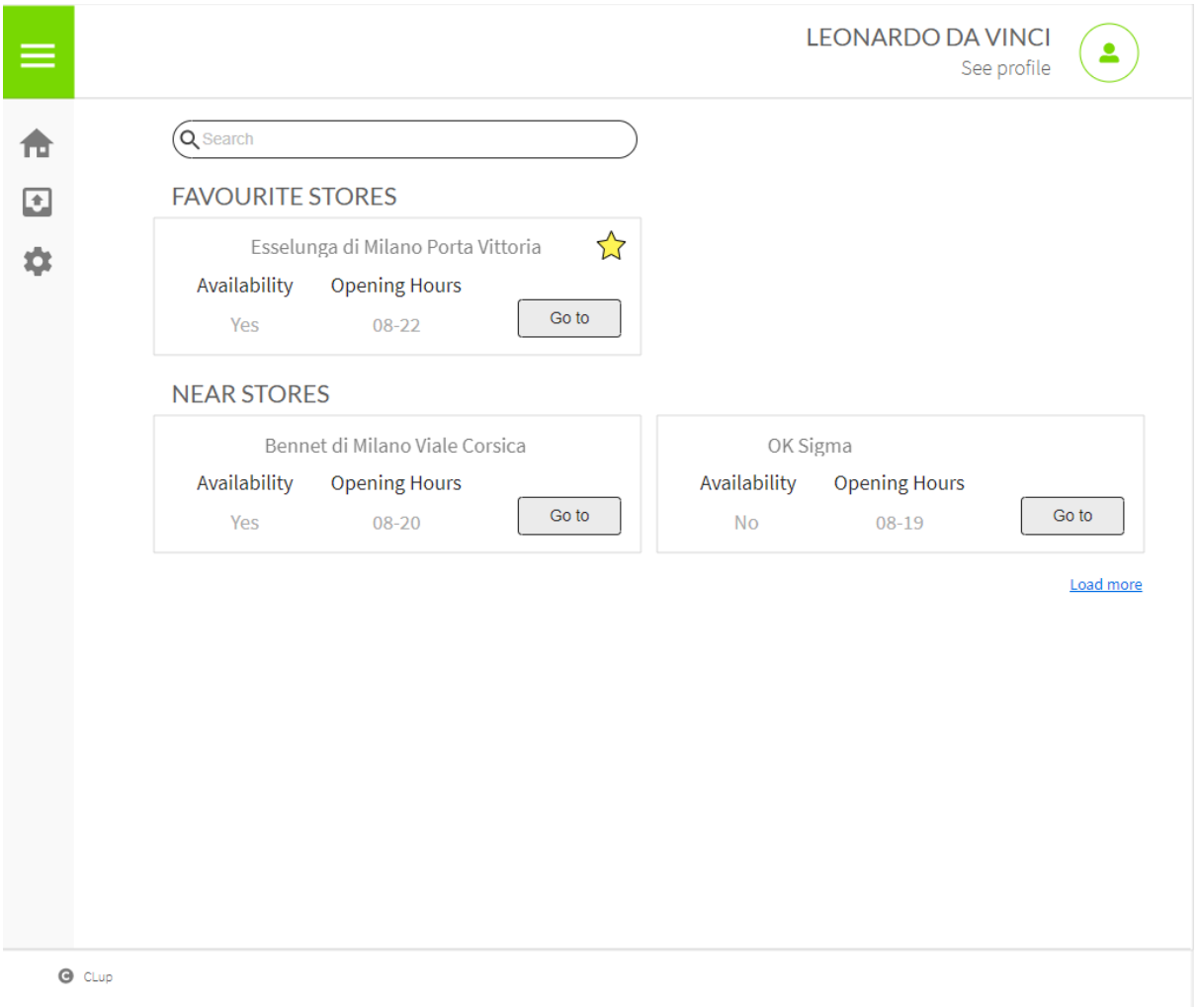
- Home Manager



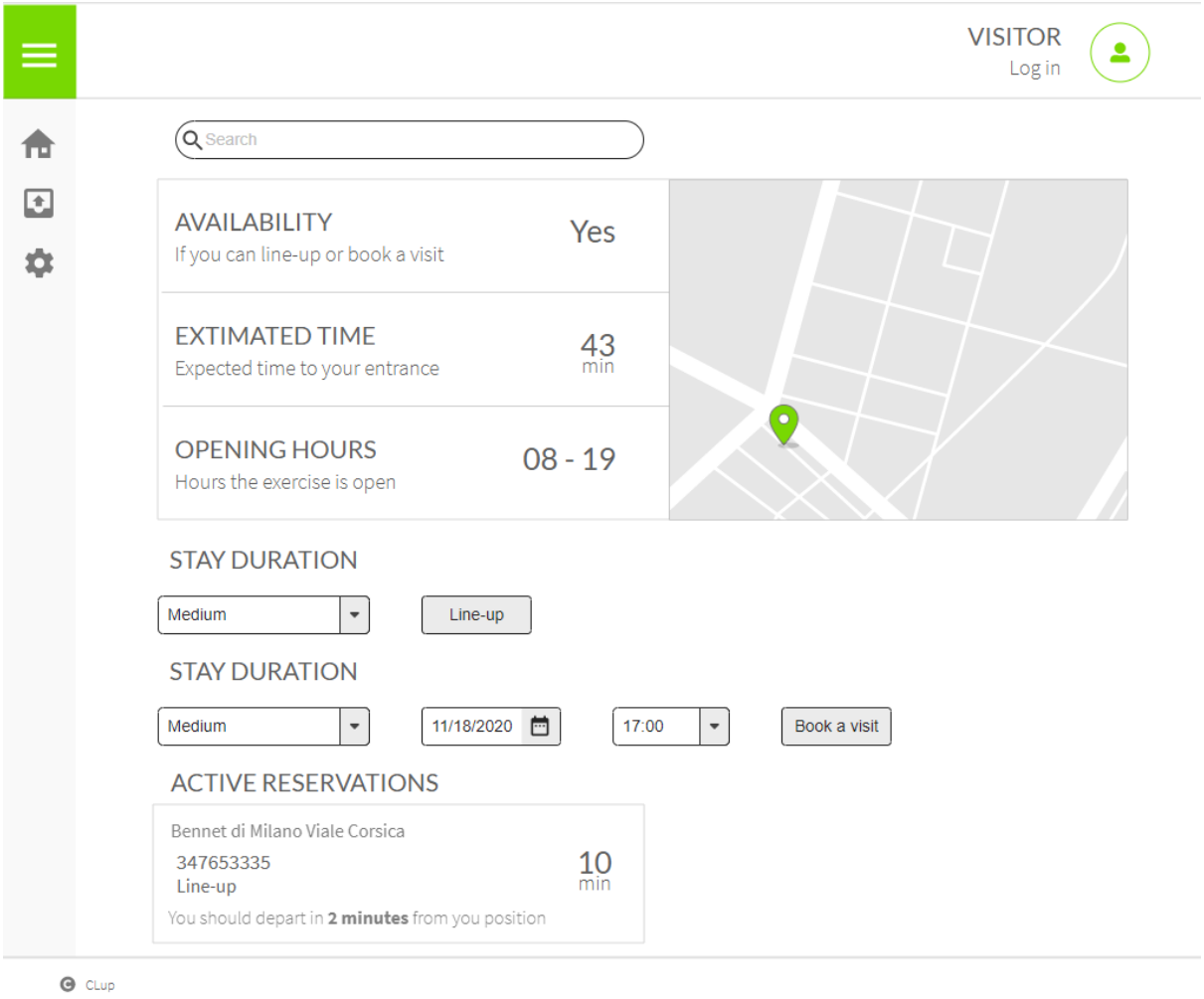
- Admin’s Reservation Page



- Customer Search



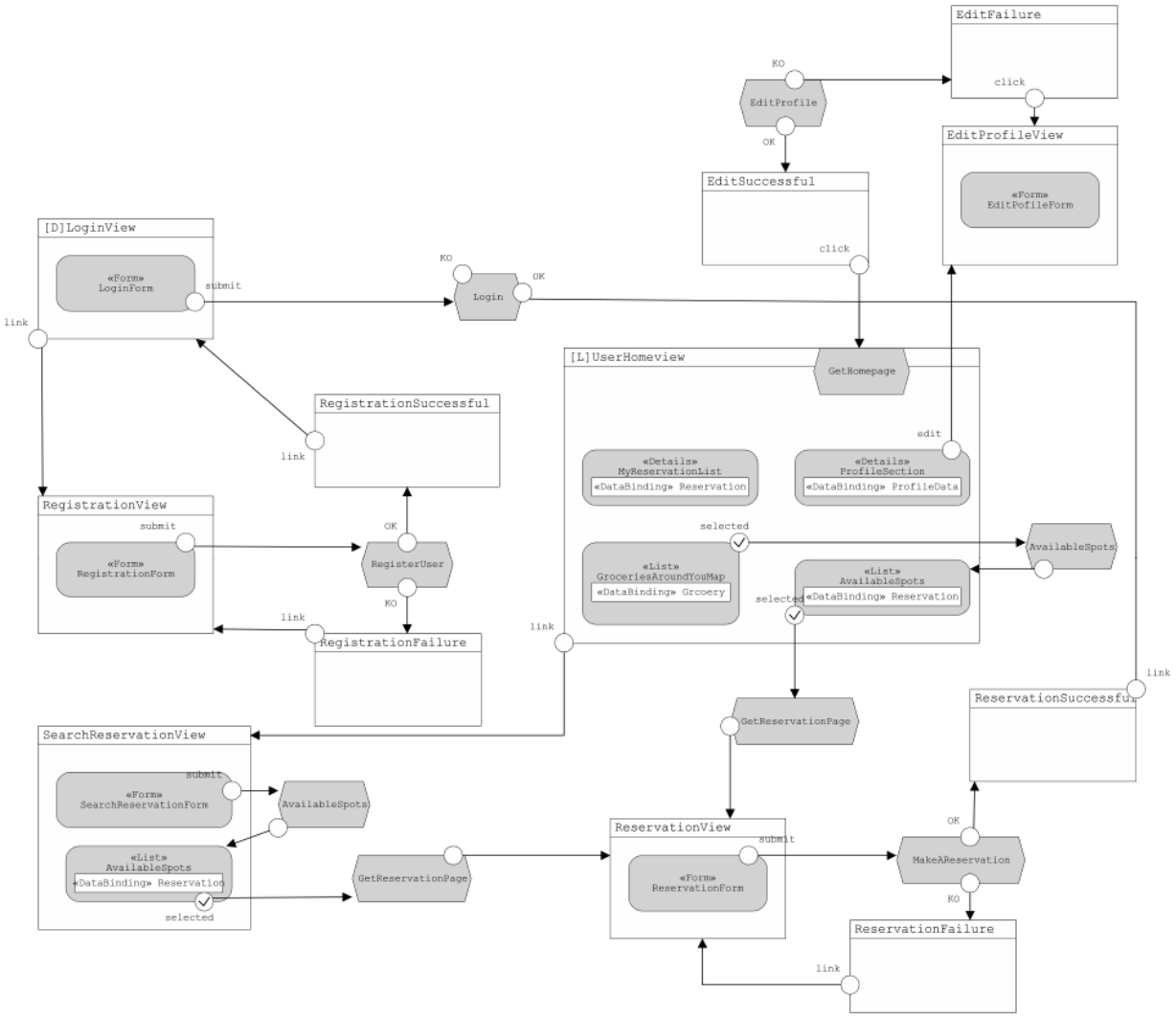
- Customer Reservation



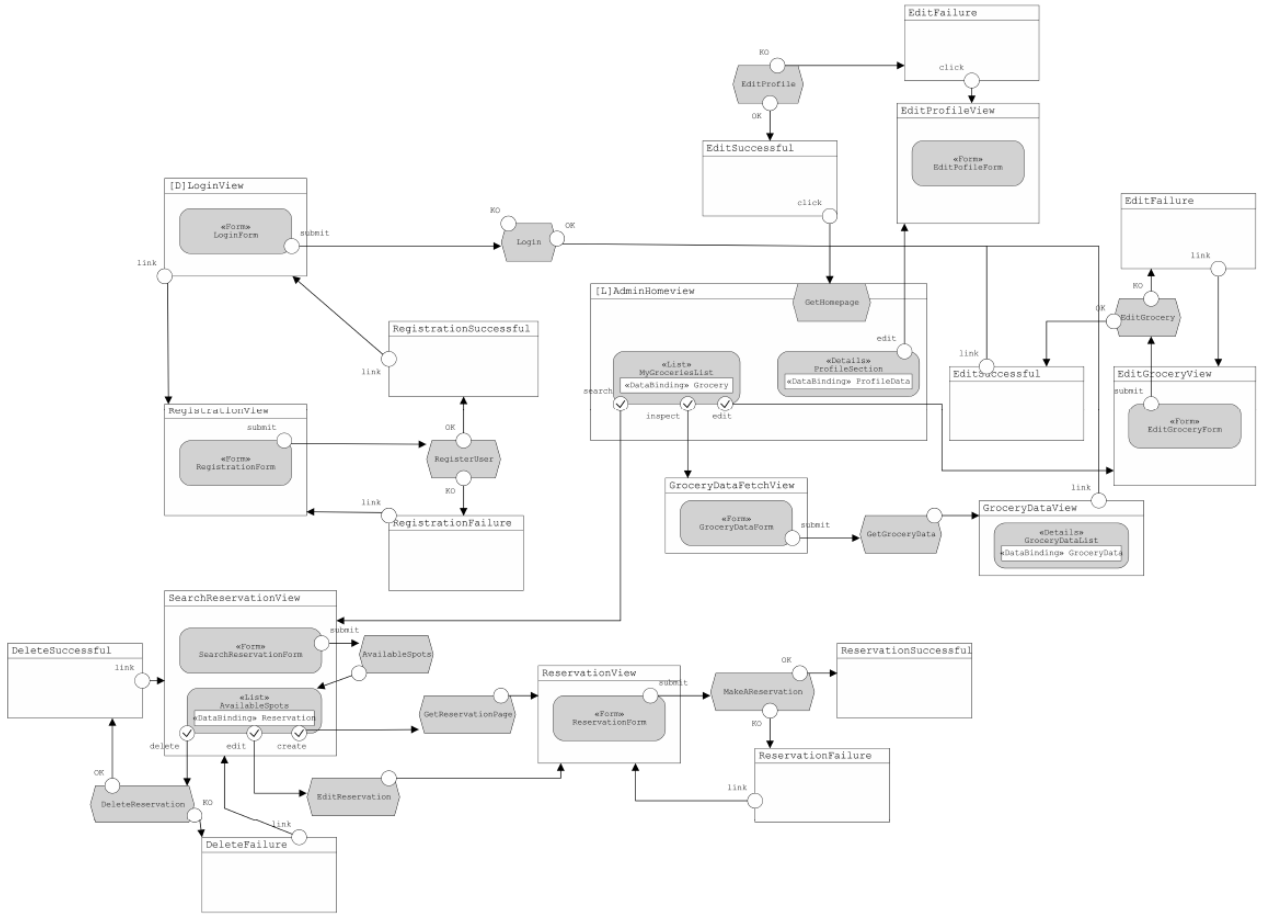
3.B. User experience design

In order to provide a better insight of the user experience, here are some UX diagrams representing it. The modeling language used is the *Interaction Flow Modeling Language* (*IFML*). We remind that the view containers are to be intended as views, so not necessarily pages but even fragments of the UI such as a pane, and this can be seen in the previous examples of interfaces, where we have two different approaches to the mobile and web applications. The original pictures of this section can be found in a directory attached to the git repository (link on References).

B.1. Customer experience



B.2. Administrator experience



This UX is a general experience, valid for both managers and employees. Some views have to be made inaccessible through access techniques such as RBAC. The following views (and those inherited by them) are to be accessible only to managers:

- GroceryDataFetchView
- EditGroceryView

4.REQUIREMENTS TRACEABILITY

The purpose of this section is to map each component described in the previous section with the requirements described in the RASD. In the first paragraph there are refreshed all the requirements, next to that list, there are all the components described in this section mapped with requirements that they should fulfill in a table.

4.A. Requirements

R1	The customer must be able to add himself to the current line
R2	The customer must be able to provide at least a telephone number
R3	The customer must be able to search for specific groceries
R4	The customer must be able to do the research basing on his position
R5	The customer must be able to make the research basing on a position provided by himself
R6	When a line-up request concludes successfully, an access code attached to a number has to be provided to the customer
R7	The customer must be able to know the current reservation number everytime, even during the opening hours of the grocery store
R8	Current reservation number must be provided in real-time and on-demand
R9	The current line-up number must be refreshed as soon as it is possible
R10	The customer must be able to know all the available booking spots in every registered grocery store
R11	Registered customers must be able to access to shortcuts in order to see their previous groceries choices
R12	The customer must be allowed to book for an available spot in a certain grocery shop
R13	When a book-a-visit request concludes successfully, an access code attached to a reservation time has to be provided to the customer
R14	Registered customers must be able to access to shortcuts in order to book same reservations made in the past
R15	The customer must be able to know the estimated time of his visit: both for the line-up and the book-a-visit functions
R16	In case of deletions, the time could need to refresh its value
R17	The estimation time must be based on the historical data
R18	In the case in which the customer has the technology to support alarms, he must be notified considering the estimation time of the previous customers
R19	For the line-up reservations, the alarm has to be provided basing on the current reservation number
R20	For the book-a-visit reservations, the alarm must be provided basing on the reservation time chosen by the customer
R21	Fallback options must include all the product functions
R22	When it comes the turn of the lined-up customer, he must be able to get into the grocery store
R23	The hardware must support the entrance of the customer
R24	Registration must include name, username, email and password
R25	Registration must be possible from all the communication technologies supported
R26	Access must be possible providing email, or username, and password
R27	Managers must be able to add a new grocery in the list
R28	Managers must be able to update informations about the store
R29	Managers must be able to register themselves to the system
R30	Managers must be able to see data mined by the application
R31	Managers must be able to add to the line a reservation
R32	Managers must be able to edit a reservation in the line
R33	Managers must be able to delete a reservation in the line
R34	Managers must be able to add employees
R35	Managers must be able to delete employees
R36	Employees must be able to log with the credentials provided by the manager
R37	Employees must be able to manage both line-up and book-a-visit reservations such as managers

4.B. Mapping table

The mapping table follows the next rules: in the columns there are represented the components, while in the rows there are the requirements. If a certain component fulfills a certain requirement then there will be a “X” in the corresponding row and column.

Since the Reservation Management component fulfills more functionalities of the application, the mapping is a bit more precise and made through its main modules (Reservation Handler Module, Time Estimation Module, and Queue and Booking Update Module) and including all the other modules in “Other modules”.

Requirements \ Components	Grocery Management	Reservation Management					Search Management	Account Management
		Reservation Handler	Time Estimation	Queue and Booking Update	Availability	Notification		
R1		X			X			
R2								X
R3							X	
R4							X	
R5							X	
R6		X				X		
R7				X				
R8				X				
R9				X				
R10		X			X			
R11							X	X
R12		X				X		
R13		X				X		
R14							X	X
R17			X					
R16			X	X				
R17			X					
R18			X			X		X
R19				X		X		X
R20						X		X
R21	X	X	X	X	X	X	X	X
R22		X		X				
R23		X	X	X		X		
R24								X
R25								X
R26								X
R27	X						X	
R28	X							
R29	X							
R30	X							X
R31	X	X						
R32	X	X						
R33	X	X						
R34	X							X
R35	X							X
R36								X
R37	X	X			X			

5. IMPLEMENTATION, INTEGRATION AND TEST PLAN

5.A. IMPLEMENTATION PLAN

A.1. Algorithms

In this section we will describe some of the algorithms that we consider at the same time more important and more complex, so that the team that will deploy this software will be helped in the job. These will be how we would intend the time estimation, how we would intend to manage the queue and some examples on how to estimate the ride time of the customer with different Maps API.

1.1. Time estimation

One of the central, and complex, functionalities of the software-to-be is the estimation time part. Here we summarize which are the minimum factors that will have to contribute to the calculation of the estimation time:

- a time factor that looks at other previously done line-up reservations and tries to avoid that too many people overlap in a single time, thus creating a large queue outside of the building (defined as `spread_time`)
- in the case in which the grocery is full, a factor concerning the average duration of shopping by other customers in the same period of the day (defined as `avg_time`);
- a fixed time to be set in order to do pessimistic estimates (defined as `fix_time`);
- necessary time to move from the position provided to the selected grocery (defined as `maps_time`).

Defined this two values, the estimated time will be given by the following formula depending on the fact that the grocery can be full or not:

$$\texttt{estimated\_time} = \begin{cases} \texttt{spread\_time + fix\_time + maps\_time}, \text{ if grocery is not full} \\ \texttt{spread\_time + avg\_time + fix\_time + maps\_time}, \text{ if it is full} \end{cases}$$

1.2. Managing the queue

The first algorithm that we will discuss is the managing of the queue. Indeed this concept is crucial to the line-up functionality of the app, and since there is some estimated time related to it, its implementation is not trivial. Any other implementation, if documented, is acceptable. Anyway, due to its complexity, we provide a possible way to implement it. Our suggestion is to create two queues:

- a first ordered list, composed of line-up reservations that will be ordered by their estimated_time illustrated in the previous section;
- a second ordered list that is populated whenever a member of the first list expires the estimated-time calculated when lining-up or when the booking time arrives if talking about the book-a-visit functionality.

The first member of the second queue is the reservation which is allowed to get its QR code scanned to get into the store.
The line-up number, already defined in the RASD(1.E), is assigned to a reservation upon entering in the second queue.
To give a further explanation:

- the first queue represents a virtual queue, which purpose is to spread out people visits as much as possible while still maintaining a time priority, based on the customer position and on other people, and efficiency regarding people that will obtain the access to the building, avoiding a store that, given other solutions, could remain empty, but at the same time avoiding the creation of a large queue outside of the building as much as possible;
- the second queue represents, in another way, the people that would actually stay outside of the building waiting for their turn; this is a queue which purpose will be very effective in case of a full grocery, because persons would get lined-up in this secondary queue waiting for someone to exit from the inside of the building and trying to respect as much as possible the estimated time given upon reserving, preserving the priority based on the time of reservation.

Given the pessimistic outlook we gave while calculating the estimated time, people who have granted the access to the store will have to enter in a short span of time, after which the entrance will not be allowed, passing to the next in the queue.

1.3. Notification time and Maps API

In this context, at least a notification will have to be sent at least a time (defined as `notification_time`) before of the estimated entrance_time calculated as:

`notification_time = estimation_time - maps_time`

It is clear that `maps_time` is a central calculation in this estimation because the customer has to be able to move from her position to the grocery and has to be able to be on time, without losing the right to get into the store. Here we provide some methods that can be used to make such calculus. These methods are strictly related to the choice made for the Maps Provider, and are based on their APIs. For this reason, we provide two methods for the two possible macro-choices that the development team can do concerning the Maps Service: a commercial one, or an open-source one.

- **Commercial** approach:

There are many ways to do the calculus with APIs provided by firms which sell the maps service. The most famous maps API is the Google one. Google is a firm, this means that the usage of the Google Maps API will require the acquisition of product keys and licences that cost depending on the number of requests to the service (more information on the Google Maps Platform page). This method is based on Google's Distance Matrix API (see References). Basically, all the needed information can be requested both by the client or by the server through an HTTP get request and retrieved by its response.

`https://maps.googleapis.com/maps/api/distancematrix/`*`outputFormat`*`?`*`parameters`*

In which:

- *outputFormat*: is the format of the response which can be "`json`" or "`xml`"
- *parameters*: are more than one and formatted such as an HTTP's get URL. There are some required parameters including information about the origin, destination, and the licence provided by Google, and some other optional but very useful

such as some related to the mode of the movement (`driving`, `bicycling`, `walking`, or `transit` that is via public transport, and it is possible even to set a preference on the public transport) or the assumption to do while calculating the traffic (`traffic_model=pessimistic` is advised)

- **Open-source** approach:

There are even many ways to do such calculus in an open-source based system. This is based on OpenStreetMap (OSM) project and provided by OpenRouteService (ORS, see References). The method is really similar to the commercial one, but in this case there are no additional costs and the ORS site provides many examples of many different programming languages. In this case the HTTP get request is the following:

```
https://api.openrouteservice.org/v2/directions/profile?query-paramters
```

In which:

- `profile`: is the way in which the movement has to be made. There are many options comprehending `vehicle`, heavy-goods vehicle(`hgv`), different types of cycling, different paces of walking and even `wheelchair`.
- `query-parameters`: these parameters are named `api-key`, `start` and `end` and have the role to authorize the request and set the origin and the arrival of the ride, expressed by means of latitude and longitude, separated by a comma.

A.2. Plan and testing

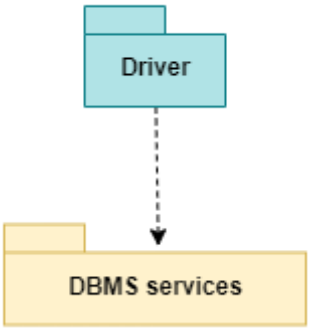
The components are logically disjointed between them, this allows a grade of freedom on the choice to the order of the implementation: this can be made both independently or sequentially. The implementation has to be supported with unit tests. To consider the modules as a whole system it is needed to do integration between modules and integration tests. This is the subject of the next section.

5.B. INTEGRATION AND TEST PLAN

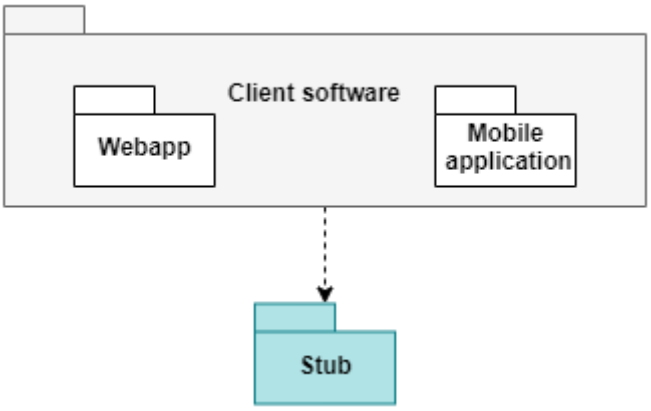
In this section, it is provided a blueprint on what concerns the integration and additional testing information is listed. In this section it is assumed

that the implementation of all the sub-modules is accomplished, together with unit testing on those components. Indeed, once that each module has been implemented it is necessary to integrate each submodule accompanying this phase with integration testing, using an incremental approach.
In order to exploit the general benefits of both incremental approaches, top-down and bottom-up, we advise an incremental hybrid approach: mixing top-down approach for the sub-modules more sensible to the functionalities of the system and bottom-up approach for the sub-modules more likely to have big consequences to eventual bugs. For this reason, our advice is to integrate the different modules chasing the following phases:

1. Test the services offered by the DBMS with a bottom-up approach, exploiting ad-hoc drivers.

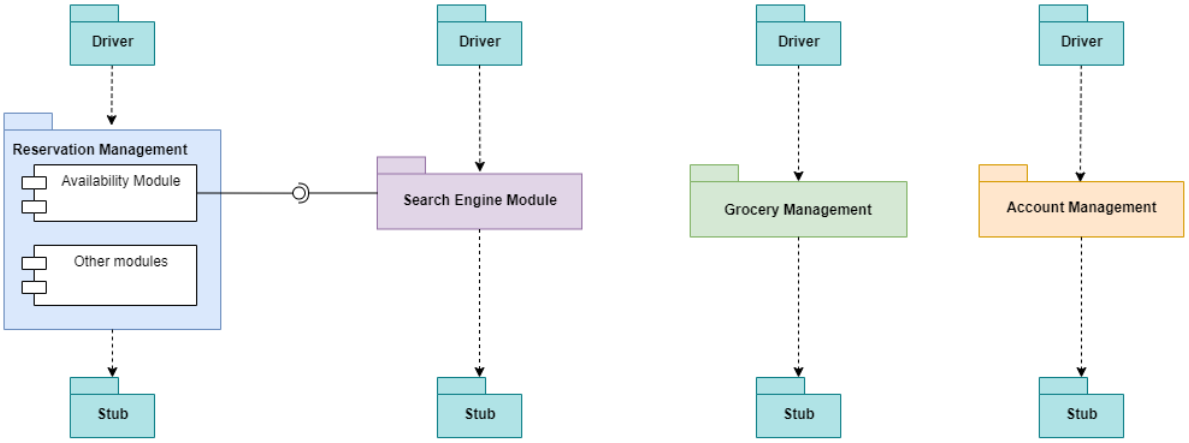


2. Test the webapp and the mobile application, providing the necessary stubs for the tests.

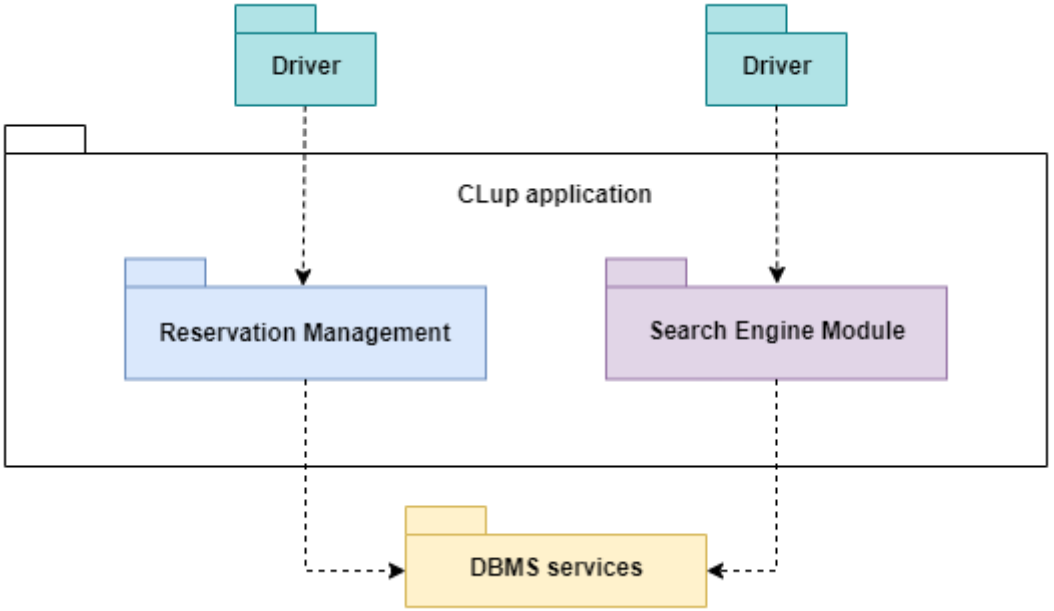


3. Integrate and test the sub-modules included in the macro-module Reservation Management and Search Management, while integrating and testing even Grocery Management and Account

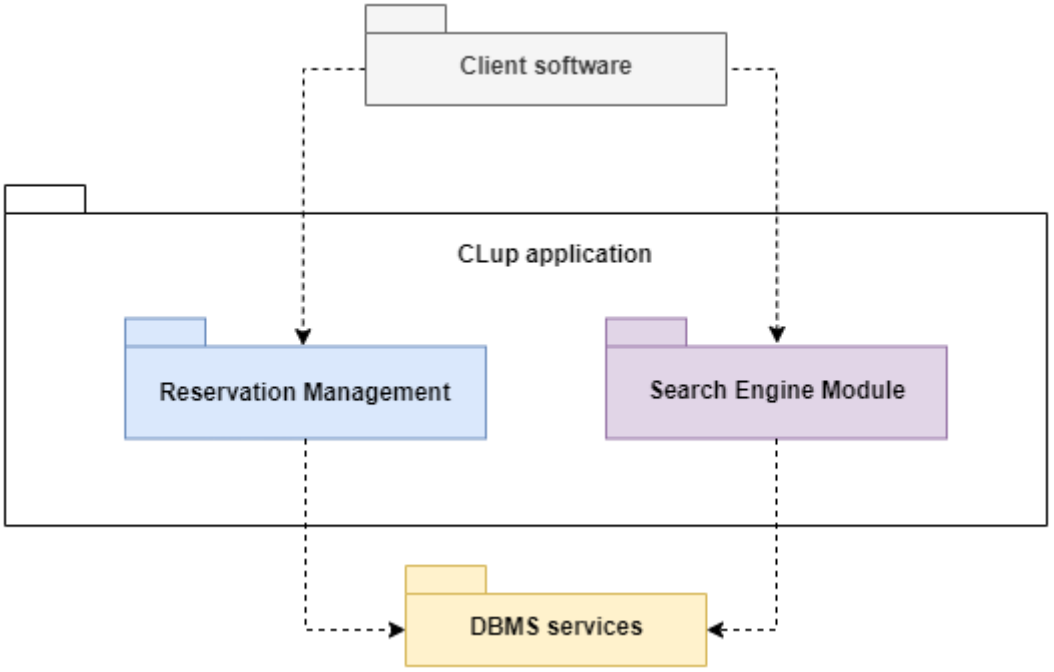
Management sub-modules. This will require creating stubs and drivers to test it.



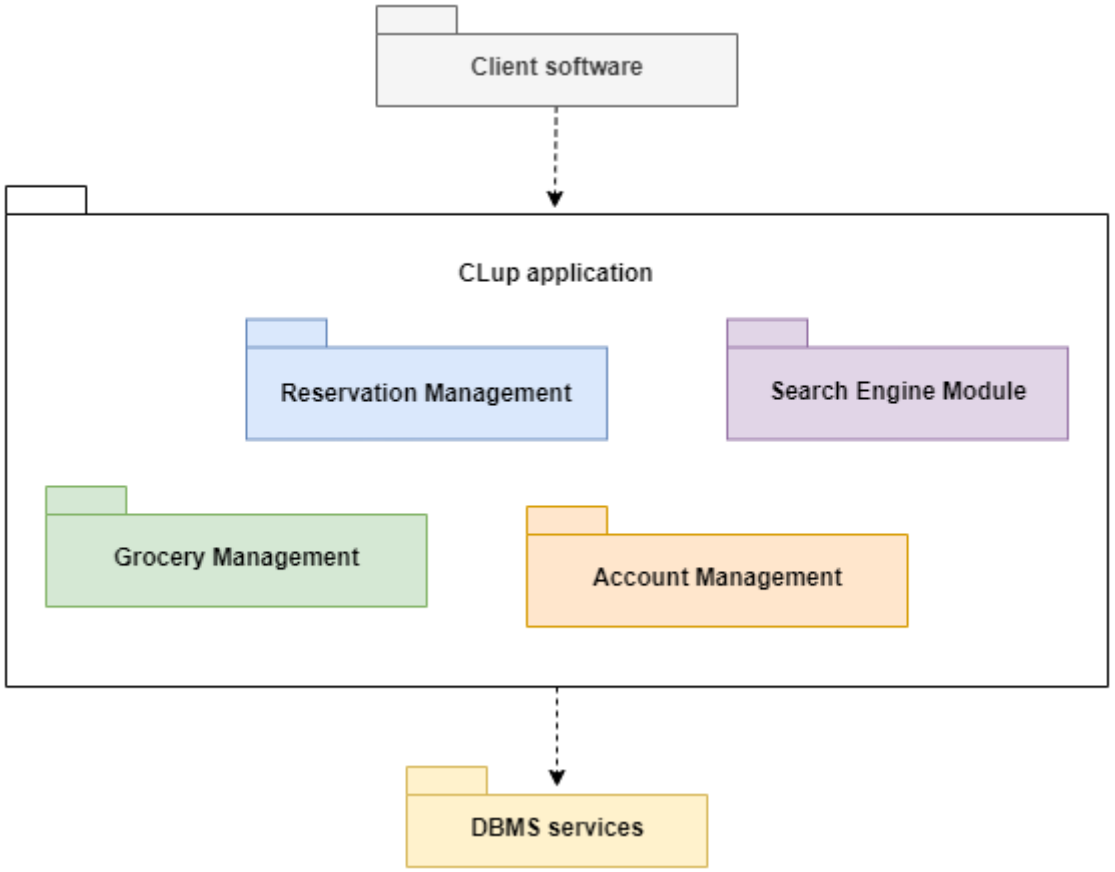
4. Substitute the driver created in the phase 1 with the Reservation Management and Search Management macro-component testing it again with the already used stub.



5. Substitute the stub created in the phase 2 with the system created up to phase 4, testing it again.



6. Integrate and test Grocery Management and Account Management with the system created up to step 5.



6. EFFORT SPENT

Duilio Cirino:

18-12	1,30	General group discussion on main characteristics of DD and scheduling
19-12	1,30	Deployment view
21-12	2	Architectural design: overview and component view
22-12	3,30	Component View
28-12	1,30	Component View adjustments and Deployment View
30-12	3	Deployment View and Interface Design
31-12	3,30	Web and Mobile Interfaces
03-01	2	Revision on interfaces and general revision

06-01	2,45	Revision after new version of RASD
08-01	2	Final check of coherency

Lorenzo Cocchia:

18-12	1,30	General group discussion on main characteristics of DD and scheduling
20-12	2	Introduction
22-12	3	Sequence diagrams
23-12	1,30	Sequence diagrams finished
27-12	1,30	Component Interfaces
28-12	2	Exploitation of architectural styles, design patterns. Requirements traceability

29-12	1,30	Update of sequence diagrams, component interfaces and requirements traceability
29-12	2	UX diagram and ER diagram design
30-12	4	UX diagram and ER diagram design, algorithms
03-02	2,30	Implementation, integration and testing plans
06-02	2,45	Revision after new version of RASD
07-02	2,30	Algorithms
08-01	2	Final check of coherency
04-02	2	Version 1.1 of the document after consultancy and implementation

7. REFERENCES

- Git repository: https://github.com/duiliocirino/CirinoCocchia
- IFML specifications: https://www.ifml.org/
- ArchiMate: The ArchiMate® Enterprise Architecture Modeling Language | The Open Group Website
- Google Maps Platform: https://cloud.google.com/maps-platform/
- Distance Matrix API: https://developers.google.com/maps/documentation/distance-matrix/overview#travel_modes
- OpenStreetMap: https://www.openstreetmap.org/
- OpenRouteService: https://openrouteservice.org/
- OpenRouteService's function used: https://openrouteservice.org/dev/#/api-docs/v2/directions/{profile}/get
- OpenRouteService documentation: https://github.com/GIScience/openrouteservice-docs#routing-options
- OpenStreetMap profiles: https://wiki.openstreetmap.org/wiki/Routing_profiles