POLITECNICO
MILANO 1863

Software Engineering 2 Project

Politecnico di Milano AA 2020-2021

Computer Science and Engineering

Design Document

Professor: Di Nitto Elisabetta

Group Components

Amato Nunzio (Matricola: ***970618***)

Ciriello Giovanni (Matricola: ***963100***)

Colombrino Fulvio (Matricola: ***966097***)

Contents

1. INTRODUCTION

1.A Purpose

The purpose of this document is to describe in a more technical and detailed way the system described in the RASD document. First of all the content of this documentation is fundamental to organize and deliver the structure of the system to the stakeholders that need to understand whether the design decisions can satisfy the user needs and their business objective. Secondly, the document helps the programmers during the development process giving them a broader view of the system architecture.
The document describes different diagrams that represent its main structure through different components underlying how they interact with each other. Moreover, there are multiple paragraphs dedicated to the specification of its possible implementation plan, integration plan, and testing plan. The documentation also depicts some of the main interfaces of the system together with the description of the main technologies that will be used during the development phase.

1.B Scope

The goal of this project is to develop a user-friendly application that not only allows store managers to regulate the influx of people in the building but moreover saves people from having to line up and stand outside of stores for hours on end, putting their health at stake. Every person that has a smartphone can book a turn in line through the mobile app, whereas whoever hasn’t the possibility to access such technology can use an automatic machine placed outside the supermarket to book a reservation.
This is the main functionality of the developed system in a nutshell. For a better description, the RASD document fully describes the main objectives and requisites that the system achieves to perform what was just described.

1.C Definitions, Acronyms, Abbreviations

1.C.1 Definitions

Confirm notification	Whenever a user books a reservation for a supermarket, he receives a confirmation message in which there is the number of the ticket, the QR code, the expected waiting time, and a precise time and day(in the case the user decides at what time and day he wants to go shopping)
Delete notification	Whenever a booked slot is canceled either by the user or by the system the user receives a message describing the cause of cancellation.
Wrong notification	Whenever a user scans his QR code but at a different time or supermarket he receives a message reminding him of his bookings (if present).
Free notification	The message sent by the system notifies the user that a new slot for the time/day and supermarket he selected as preferred is available.
Approaching notification	Whenever the turn of a user is approaching, he receives a notification. (It depends on his current position and his number in the queue)
Queue	It defines the virtual line.
To book	To take a slot in the queue of a supermarket as soon as possible or on a specific date and time.
Turn	The user slot in the queue
Ticket	A reservation for a slot in the queue
Online user	The user that makes an online reservation with an

	internet-connected device
Offline user	The user that makes a reservation outside a supermarket through the automatic ticket machine
Time slot	Range time(1: 00 H) in which you can book a turn
ASAP turn	The first available turn
Safety Margin	The number of available customer slots inside the supermarket that are used only in case of multiple delays or for finer affluence of people.
Success ring	The sound coming from the QR reader

1.C.2 Acronyms

RASD	Requirement Analysis and Specification Document
CLup	Customers Line-up
GPS	Global Positioning System
QR	Quick Response

1.C.3 Abbreviations

CONNOT	Confirm Notification
DELNOT	Delete Notification
WRONOT	Wrong Notification
FREENOT	Free Notification

APNOT	Approaching Notification
Gn	Goal number n
Rn	Requirement number n
ASAP	As soon as possible

1.D Revision history

21/12	❖ First component diagram ❖ First interfaces ❖ First Runtime
25/12	❖ Review of the previously added paragraphs ❖ Added deployment view ❖ Added new runtime view
30/12	❖ Added interface mockups ❖ Added requirements traceability ❖ Added Testing
8/12	❖ Final total revision

1.E Reference Documents

- Specification Document: “SafeStreets Mandatory Project Assignment.pdf”
- Slides of the lectures

1.F Document Structure

❏ **Chapter 1:** Here an introduction of the design document is given. This chapter contains the purpose and the scope of the document, as well as some details on used terms to provide a better understanding.

❏ **Chapter 2:** This chapter deals with the architectural design of the application. It gives an overview of the architecture and it also contains the most relevant architectural views:

● Component view;

● Deployment view;

● Runtime view;

● Interaction of the component interfaces. Some of the used architectural designs and design patterns are also presented here, with an explanation of the purpose of their usage.

❏ **Chapter 3:** this chapter specifies the user interface design. The mock-ups presented in the RASD document and UX diagrams are here visualized.

❏ **Chapter 4:** the requirements traceability is analyzed in this chapter; in detail, is described how the requirements identified in the RASD are linked to the design elements (defined in this document).

❏ **Chapter 5:** briefly describes the implementation plan, the integration plan, and the testing plan, specifying how all these phases are thought to be executed.

❏ **Chapter 6:** contains a detailed report of the effort spent by each group member while working on this project.

❏ **Chapter 7:** this is the last chapter and includes references to the documents, texts, and resources used to write this document.

2. ARCHITECTURAL DESIGN

2.A Overview: High-level components and their interaction

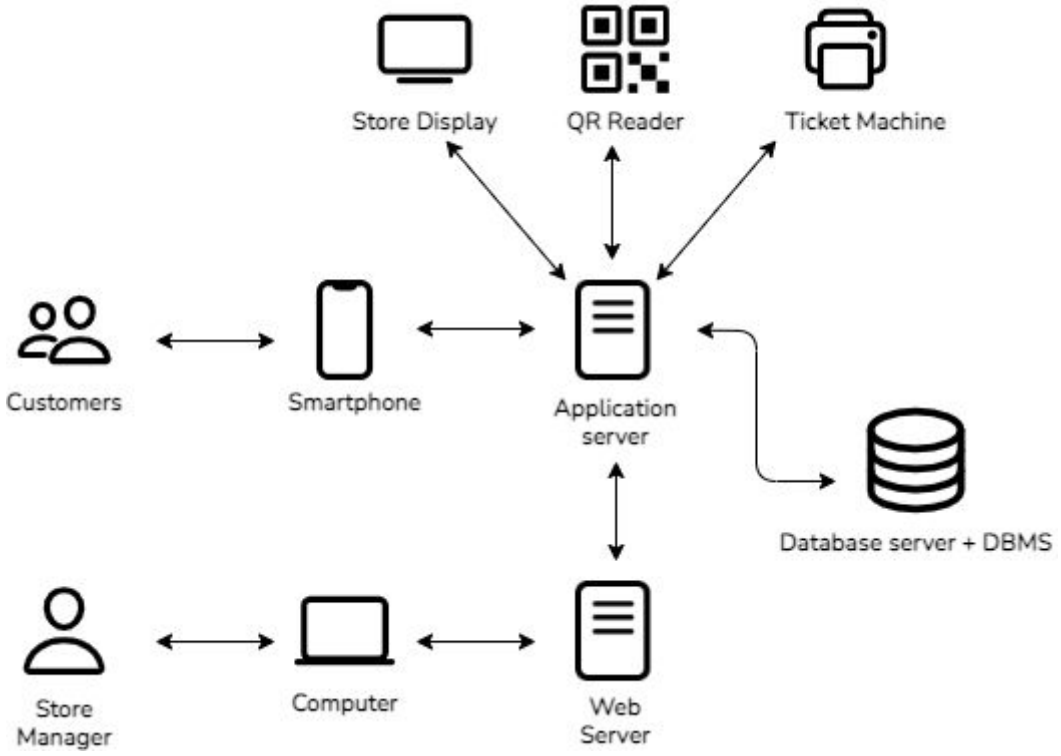

Image 0: architectural design

The architecture of the application is structured according to three logic layers:

Presentation tier: front end consisting of the user interface. This one is accessible through a web browser or web-based application and displays content and information useful to an end-user(customer or manager).

Logic tier: it mainly works as the bridge between Data Tier and Presentation Tier, making logical decisions and moving data between the other two layers.

Data-tier includes the data persistence mechanisms(database servers, file shares, etc.) and the data access layer that encapsulates the persistence mechanisms and exposes the data.

Regarding the application used by the Customer, it is defined as a thick client, i.e. it not only incorporates all the presentation logic functions but also part of the business one. This helps distribute the workload inside the system avoiding useless strain on the main server. For example, the customer client, thanks to a direct link to the Google Maps API and the GPS of the smartphone, can automatically generate the APNOT necessary to notify the user that their turn is approaching. In this case, there is no message exchange with the application server and so both the chance of failure and the load on the server decrease.

ApplicationServers encapsulate most of the CLup service logic. To provide 99.9% availability and reliability, servers must be replicated. This approach requires the adding of a load-balancing system to distribute the workload among the nodes. The replication shall follow the cloning with shared disk configuration, to distribute the computational load and to replicate data for security reasons. Both scalability properties and fault tolerance of the service are increased with this configuration. Treated data must be protected; firewalls are installed before and after the application tier to create a DMZ (Demilitarized Zone). The functionality of the firewalls to be installed is standard, so they will not be described in detail in this document.

2.B Component View

The paragraph includes a diagram that depicts the main components that form the presented system. As described previously the components are organized in the three-tier architecture where we have the thick clients, the application server, and the Database server and each one has its components.

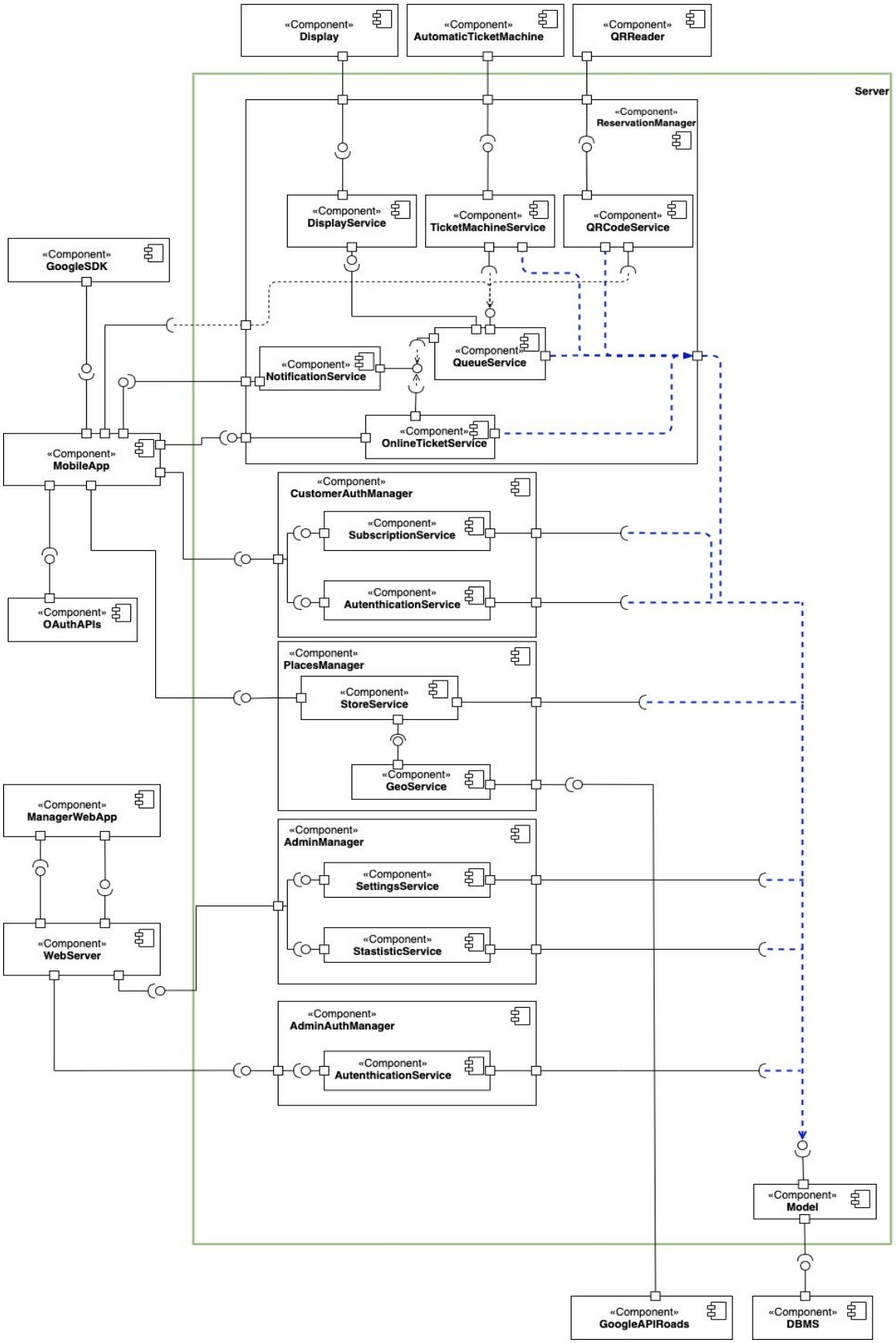

Image 1: component view

- **MobileApp:** This is the client for the customer that wants to book an online reservation. It is a thick (or flat) client. It encapsulates some of the logic functions of the system. For example, the client itself manages the dispatch of Approaching Notifications. For this reason, it needs a direct link with the services of Google maps.
- **Display Client:** This is the client used by the display to ask for the numbers to show to the customers outside the supermarket.
- **QR Reader Client:** This is the client used by the QR Reader that notifies the system any time a valid code is scanned at the entrance or the exit of the store. In this way, the system can actively update the queue and manage more finely the influx.
- **Ticket Machine Client:** This is the client used by the automatic ticket machine to generate ASAP reservations for all the offline users that interact with it. It asks the Reservation manager for the first available turn in line together with all the necessary information (QR code, number in line, precise turn time).
- **Model:** The components of the system that allows every other component in the system to interact with the chosen DBMS. It is the application's dynamic data structure, independent of the user interface It directly manages the data, logic, and rules of the application.
- **PlacesManager:**
    - GeoService: This component functions as an intermediate interface between the system and the chosen map API. In this case, if in future development, there is a need for a change of Map API this will be the only component that needs modification.
    - StoreService: This component has the functionality to retrieve all the supermarkets that are nearby the area where the user wants to book a reservation.
- **CustomerAuthManager:** This is the component used by the mobile app to let customers sign in or to access the system. Moreover, it functions as an authentication layer for the system avoiding malicious users interact with the system.

- **AdminAuthManager:** This is the component used by the manager web app of the stores to access the control panel of their stores. As the previous component, it also functions as an authentication layer.
- **AdminManager:** This component is used by the manager client to modify and control the main settings of the system regarding the assigned supermarket.
    - SettingsService: This component allows the manager web app to set parameters like the max quantity of customers that can shop at the same time and the Safety Margin. Moreover, it has the functionality of analyzing blueprints of the supermarket to estimate a proper max quantity of customers that can be suggested to the manager any time he decides to modify the dispositions of the shelves. Lastly, it enables the manager client to specify which and how many departments are present inside the supermarket to support the managing of the influx more finely.
    - StatisticService: This component is used by the manager web app and post-process the analytics produced by the DB.
- **ReservationManager:** This component allows the online and offline customer to book a turn in a supermarket and manages the influx of customers inside the store. It takes care of updating the screen outside the store with the numbers of the reservation that can enter the building and checks whether a scanned reservation can enter the building or not. Moreover, the component manages all the main reservations that need to be sent to the online customer apart from the APNOT that is under the MobileApp responsibility.
    - QueueService: the main function of this component is to check, whenever a user wants to book a turn in a supermarket if there is an available time turn in the online queue managed by the Model. Therefore, it has the role of managing the queue such that there is no possibility to determine an overcrowded supermarket.

2.C Deployment View

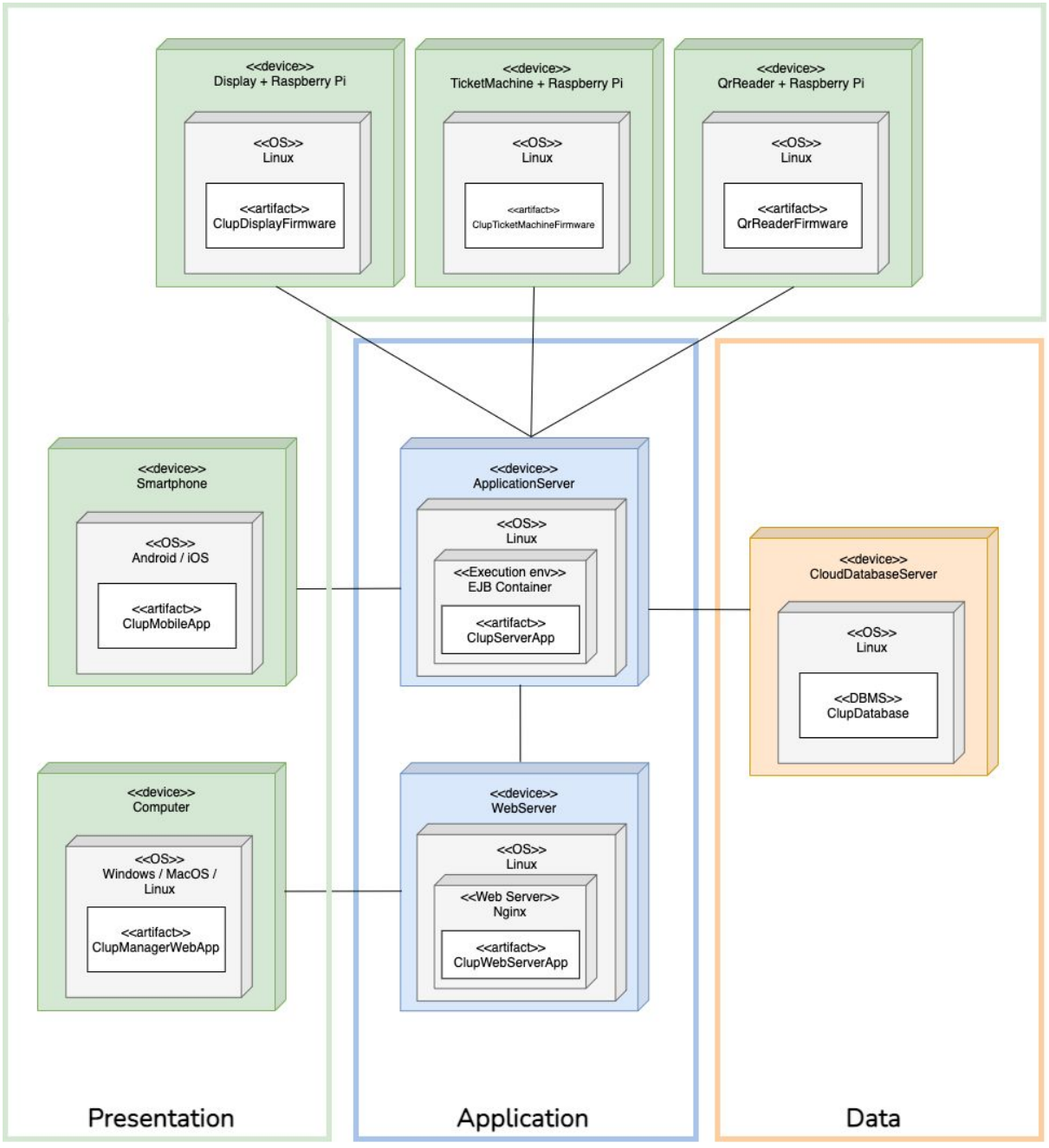

Image 2: deployment view

The Deployment diagram above shows the architecture of the system as the deployment of software components. Below is a more detailed description of them:

Smartphone

The smartphone node represents a key client node in this system. All customer requests to the ApplicationServer are delivered by the mobile application that runs on the customer Smartphone.

This node communicates also with GoogleMaps API (not represented in this diagram) thanks to the relative SDK.

Computer

Store Managers log into the Clup Manager WebApp through a generic browser on any supported desktop operative system.

Display, TicketMachine & QrReader + Raspberry Pi

Here we have the clients that are part of the store that customers interact with during the shopping process. They are connected to a network-connected mini-computer like a Raspberry pi that can communicate with the ApplicationServer.

ApplicationServer

This node contains all Clup service logic, concerning Tickets, queues, Stores information. The requests arrive from the client nodes or the webserver. Whenever a request arrives, this node accesses the cloud database to make all his actions persistent.

WebServer

This node provides access to services to the Store Managers through a Web Application. No business logic is found in this node.

CloudDatabaseServer

The Application layer relies on the DataServer to make all his actions persistent. The data layer deals with data manipulation, insertion, and deletion of all types of data.

2.D Runtime View

This paragraph collects the runtime views that describe the main functionalities of the developed system. That is, they describe in detail the different interactions made among the different components of the system.

2.D.1 Retrieval of the list of supermarkets the customer can visit

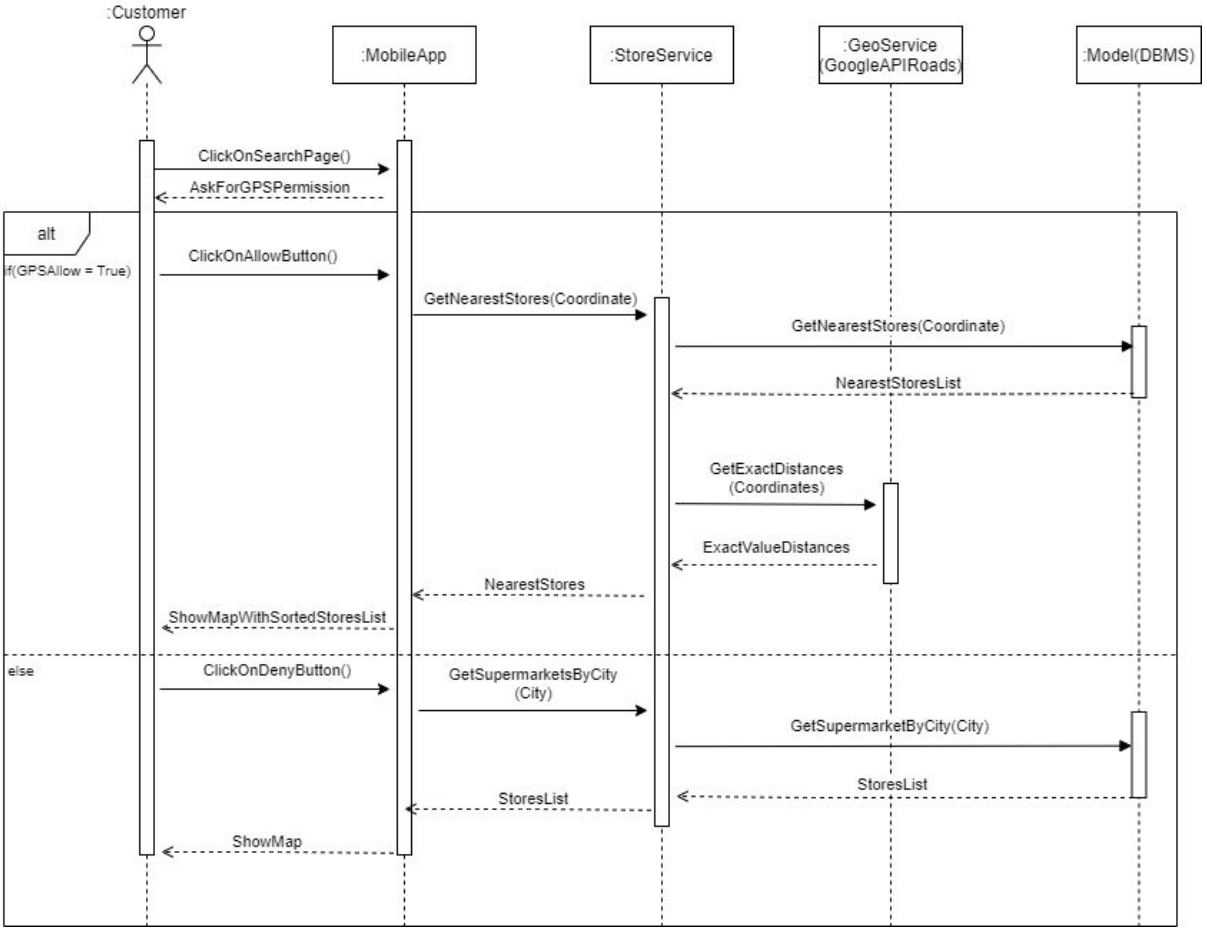

Image 3: runtime view 1

This graph represents the interactions needed to retrieve from the database all the supermarkets that are in the proximity of the user or his city. Everything starts whenever a user wants to book a reservation but wants to choose the

supermarket first. The application asks the customer whether it can use the smartphone GPS to find the nearby stores. In case of permission is given then the mobile app uses the coordinates of the user to search for the nearest stores, in a radius of 500m, inside the database. The latter gives back the coordinates and info of these supermarkets. They are then paired, together with the coordinates of the customer, to the GeoService interface to calculate the necessary data needed to add markers of the locations on the map of the mobile app. The latter is then shown to the customer that can choose one of the suggested supermarkets for his grocery shopping.

2.D.2 Book a reservation through the mobile app

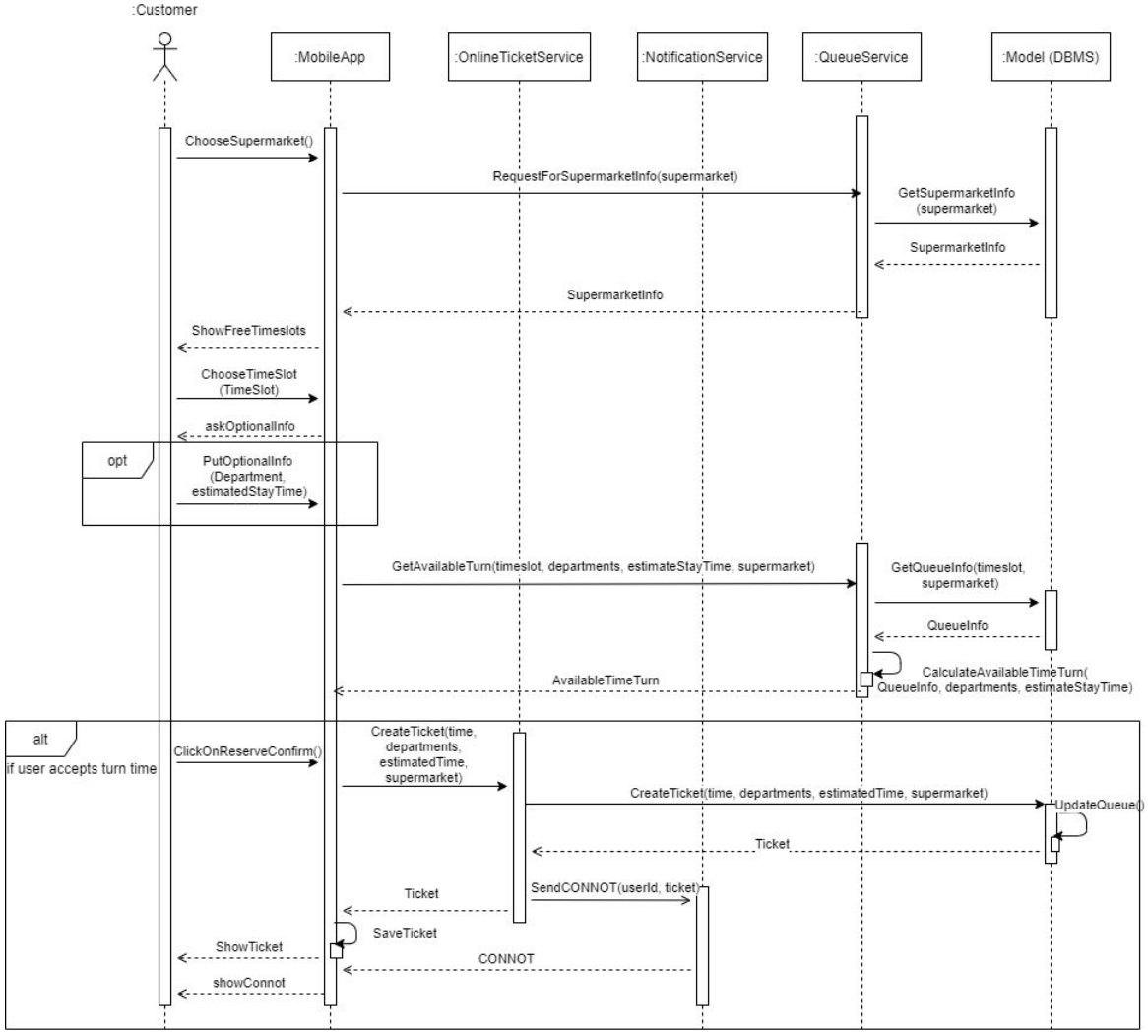

Image 4: runtime view 2

This graph represents the interactions needed to successfully book a reservation through the mobile app. After the selection of the supermarket, as explained in the previous runtime view, the mobile app retrieves the necessary info on the store through the QueueService. The data is composed of the available time slots and the different departments that are present inside the store. The info is then shown to the user that first needs to indicate either a preferred time slot or, in case there isn't any, to book the first available one(ASAP reservation). After this, he can choose the departments he intends to visit and the approximate time he will spend shopping. Once all the parameters of the booking are collected they are sent to the TicketService that will create the ticket and will make sure it is stored in the system. Once the reservation is stored, both in the DB and in the smartphone, the mobile app shows the customer its ticket and the related CONNOT.

2.D.3 Book a turn offline

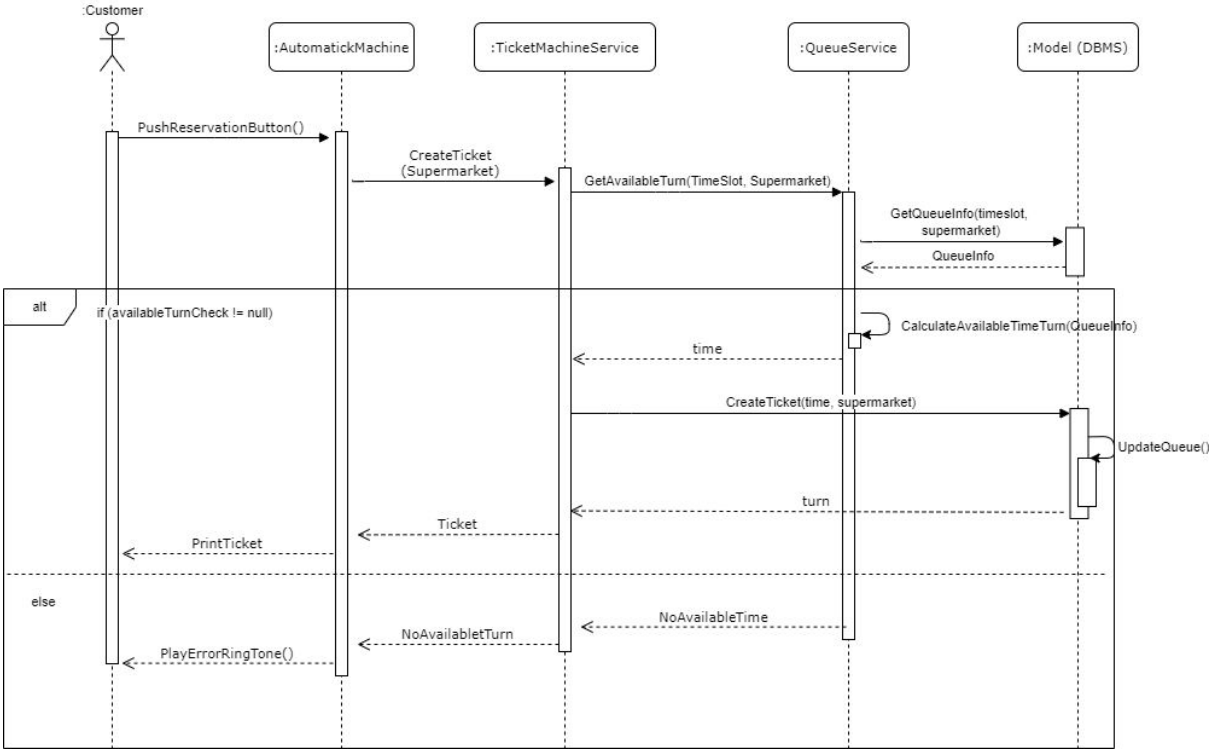

Image 5: runtime view 3

This graph represents the necessary interaction to book a reservation without the mobile app. First of all, the user interfaces with the automatic ticket machine outside the shop and requests a ticket. The ticket machine routes the request to the TicketmachineService that asks the Queue service for the first available turn in line.

If there is an available turn before the closure of the supermarket, then the TicketMachineService creates the tickets and orders the TicketMachine to print them out for the customer. In case the queue is full, the ticket machine will only play an error ringtone.

2.D.4 Sign up

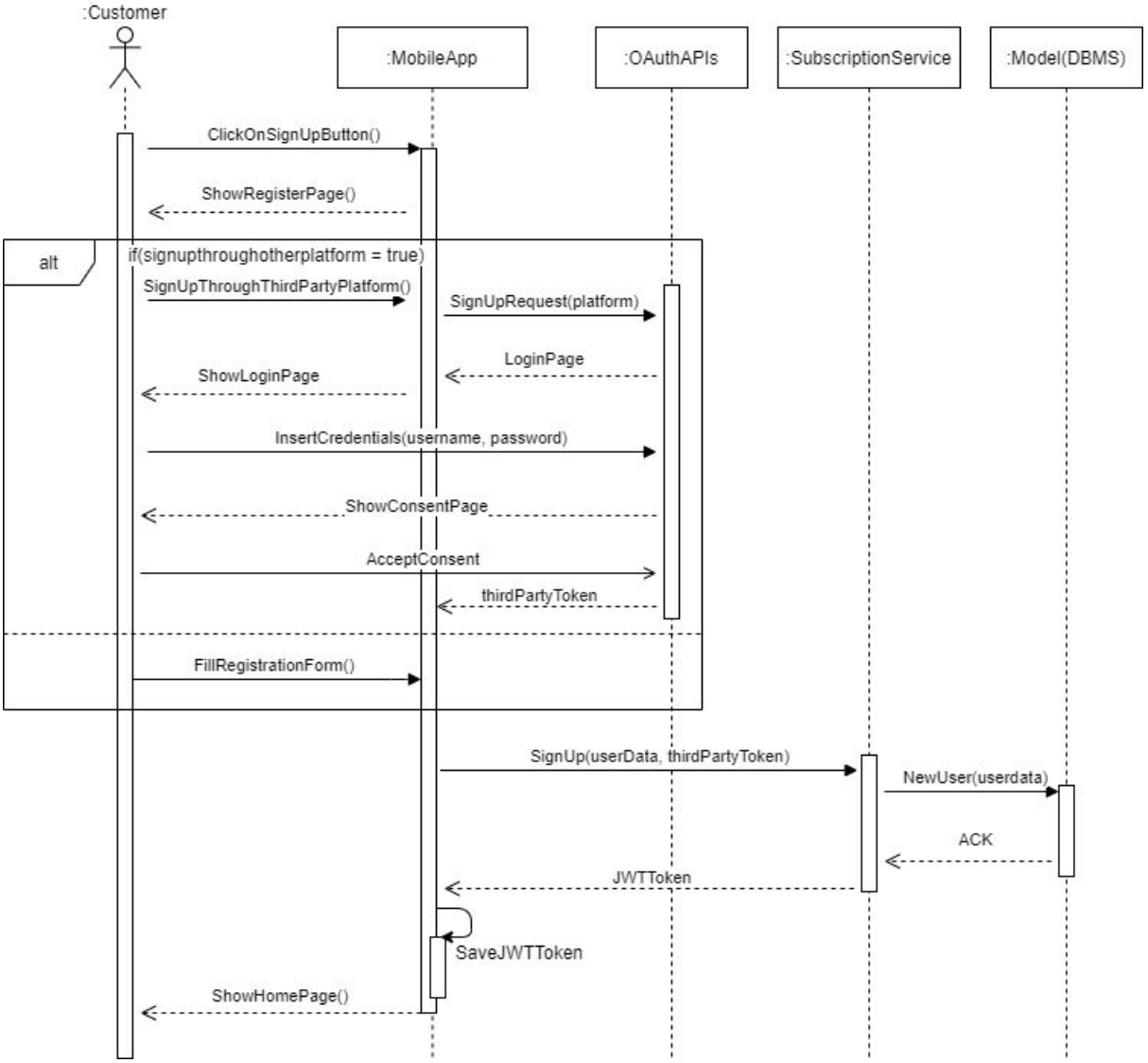

Image 6: runtime view 4

This graph represents the interactions that are needed for a customer to register on the mobile app. The first step is for the user to go to the signup page and decide how he wants to insert his personal data. The customer can either sign up through a different platform (Google, Apple, Facebook, etc.) or through the insertion of data inside a given form. In the former case, the mobile app interacts with the OAuth interface to open a browser page and present the customer with the sign-in page of the chosen platform. Once the customer accepts the use of the data by the CLup application, the info is sent to the MobileApp which in turn will send it to the SubscriptionService that will take care of creating the user profile and making sure it is stored inside the

database. In the other case, the customer can fill in the signup form given by the application with his personal data. In both cases, after the creation of the profile, the system gives back a fresh validation token that will be used to authenticate its interactions with the system.

2.D.5 Receiving an approaching notification(APNOT)

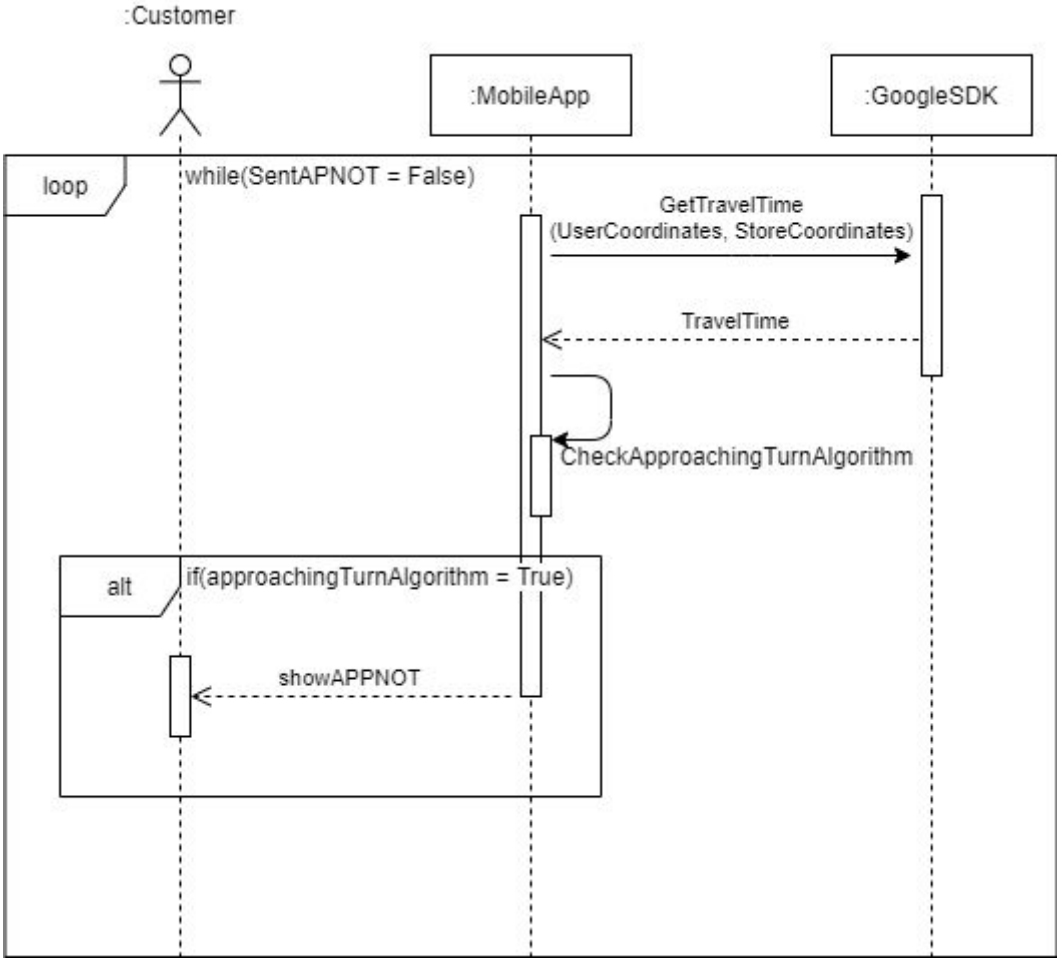

Image 7: runtime view 5

In this sequence diagram, it is shown the process that allows the customer to receive an APNOT, a notification made by the mobile app to notify the user that his turn to go shopping is approaching. This process is defined through an algorithm used by the mobile app that will be described in the next sections.

2.D.6 User starts shopping

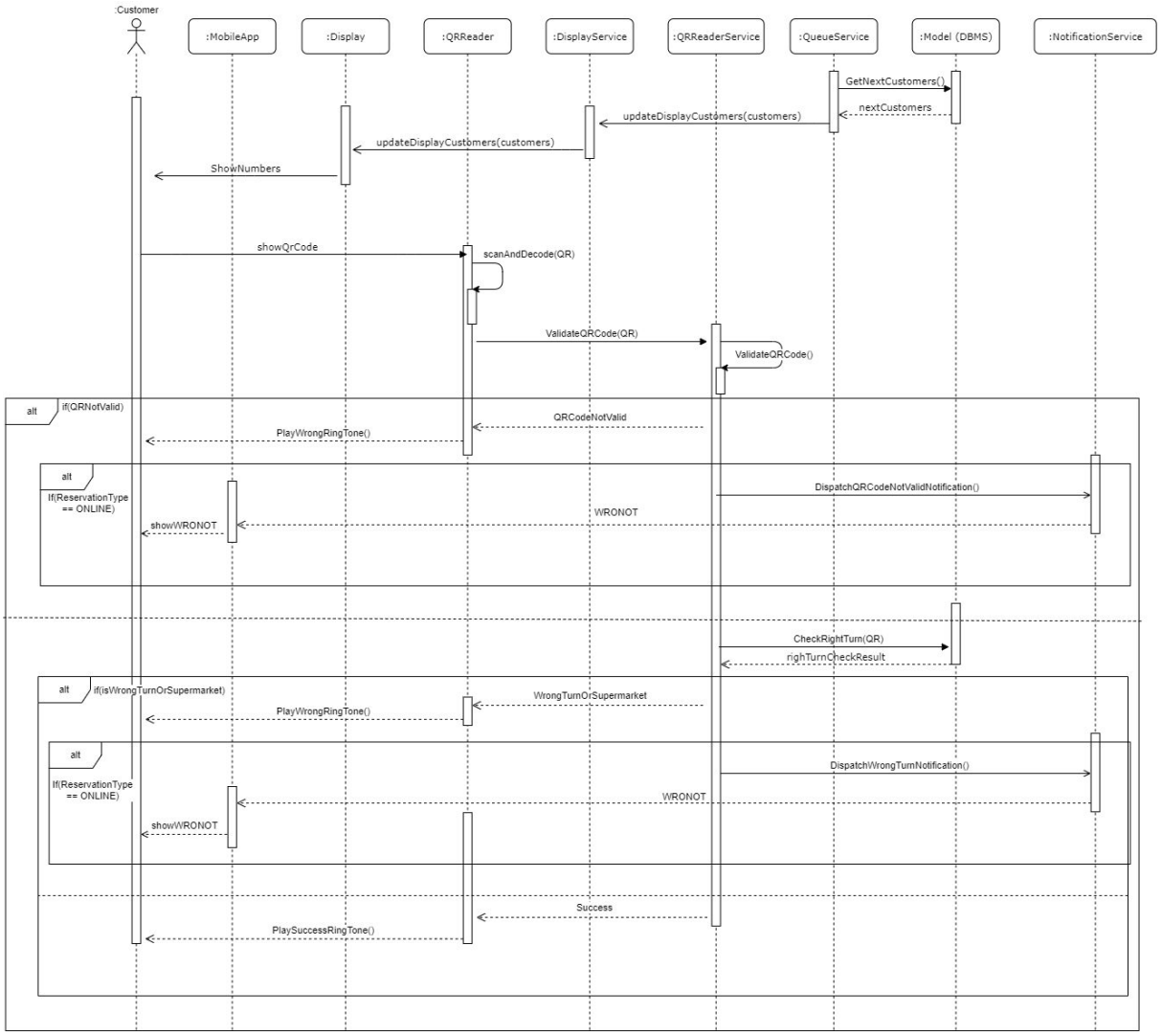

Image 8: runtime view 6

This sequence diagram illustrates what happens when the customer enters the store to do grocery shopping. Firstly, the Display receives a message from DisplayService to show the new incumbent numbers. At this point, the customer outside the supermarket sees the display and notices that he can enter the store. Therefore he goes to the entrance and scans the QRCode that is on his ticket. So the QRReader decodes the QRCode and sends the result to the QRReaderService. This one has the role to request the Model if the QRCode is related to the right reservation (i.e. if the customer has chosen the right supermarket to go shopping and the time at which he arrived respects the time on the reservation). After receiving the response from the Model, in case

the QRCode is valid, the QRReaderService sends an asynchronous message to the QRReader that notifies it that it can make a success ring. In case the QRCode is not valid, the QRReaderService delegates the NotificationService to send a WRONOT to the QRReader that makes an error ringtone.

2.D.7 User ends shopping

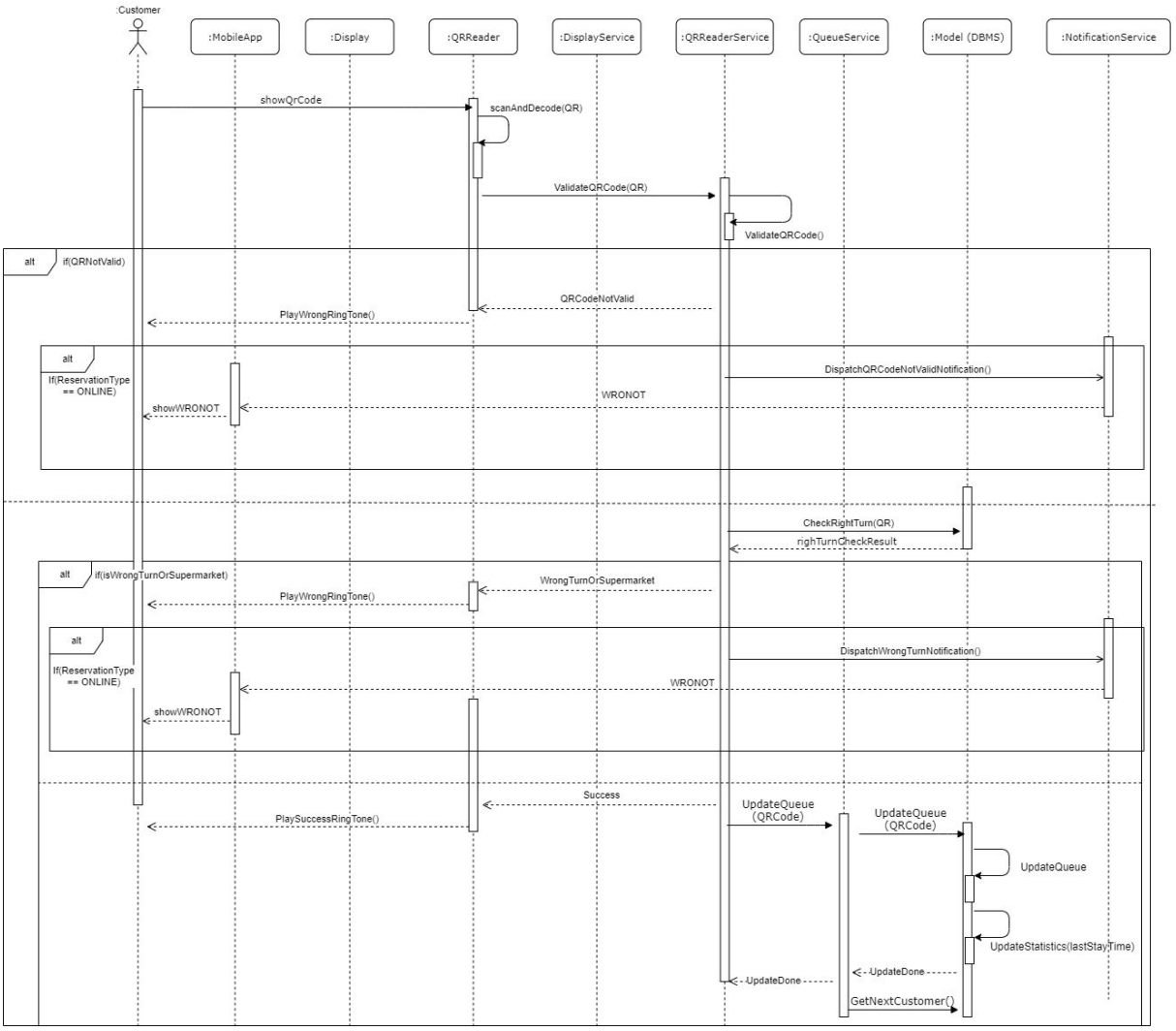

Image 9: runtime view 7

In this sequence diagram, it is shown what happens when an online user ends his shopping. Firstly, he scans the QR code that is on his ticket. Then, the QRReader requests QRReaderService to validate the QRCode. After doing that, the QRReaderService requests the QueueService to update the queue to allow other customers outside the supermarket to enter the store. At this point, the

QueueService contacts the Model that updates the queue and the statistics of the user as well(i.e. the average time spent inside the supermarket).

2.D.8 Cancellation of a reservation

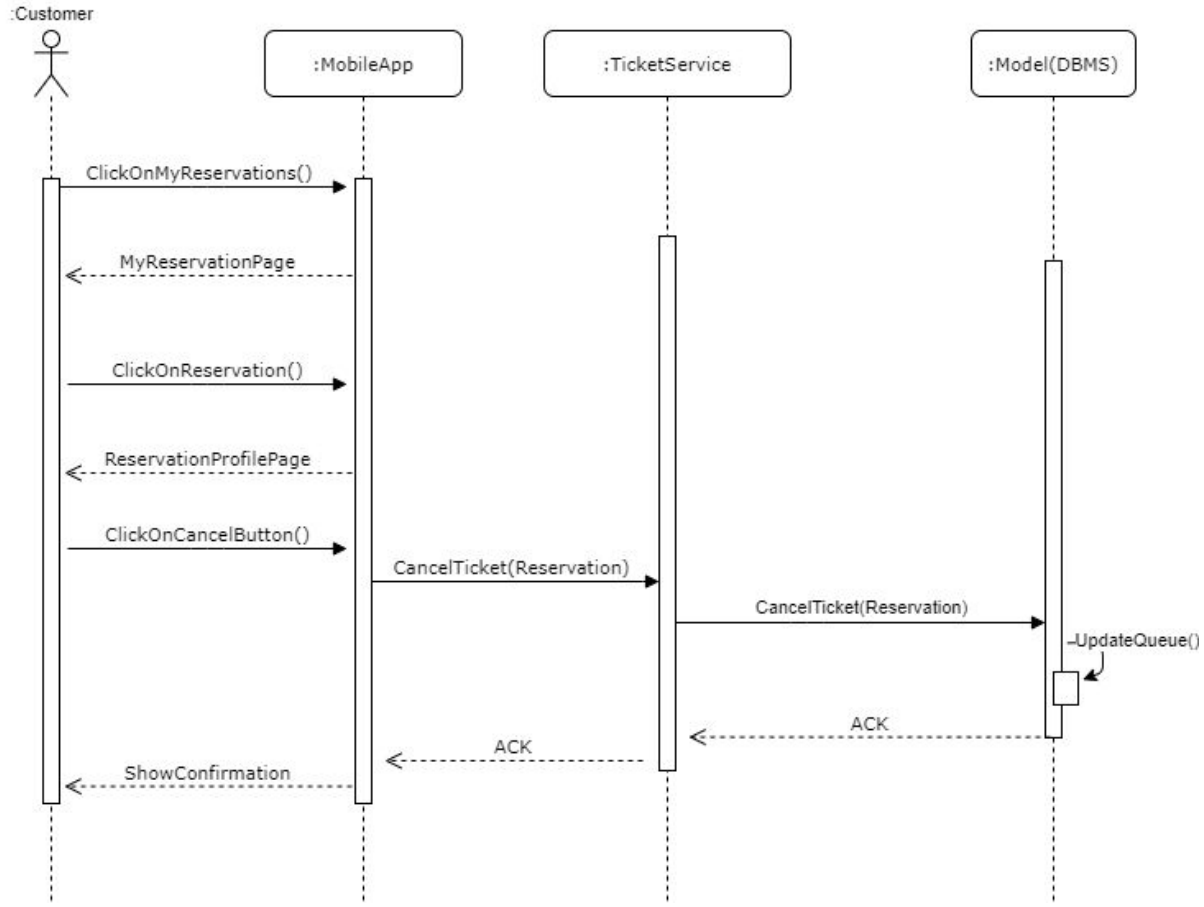

Image 10: runtime view 8

This sequence describes the steps to cancel a reservation. Firstly, the online customer chooses a ticket to cancel and clicks on CancelButton. Then, the mobileApp sends a request to the TicketService to cancel the selected reservation. The TicketService delegates the Model to cancel the reservation. When the reservation is canceled, the MobileApp receives an ACK from the TicketService and shows a message that confirms the ticket has been deleted.

2.D.9 Setting and receiving periodic notifications

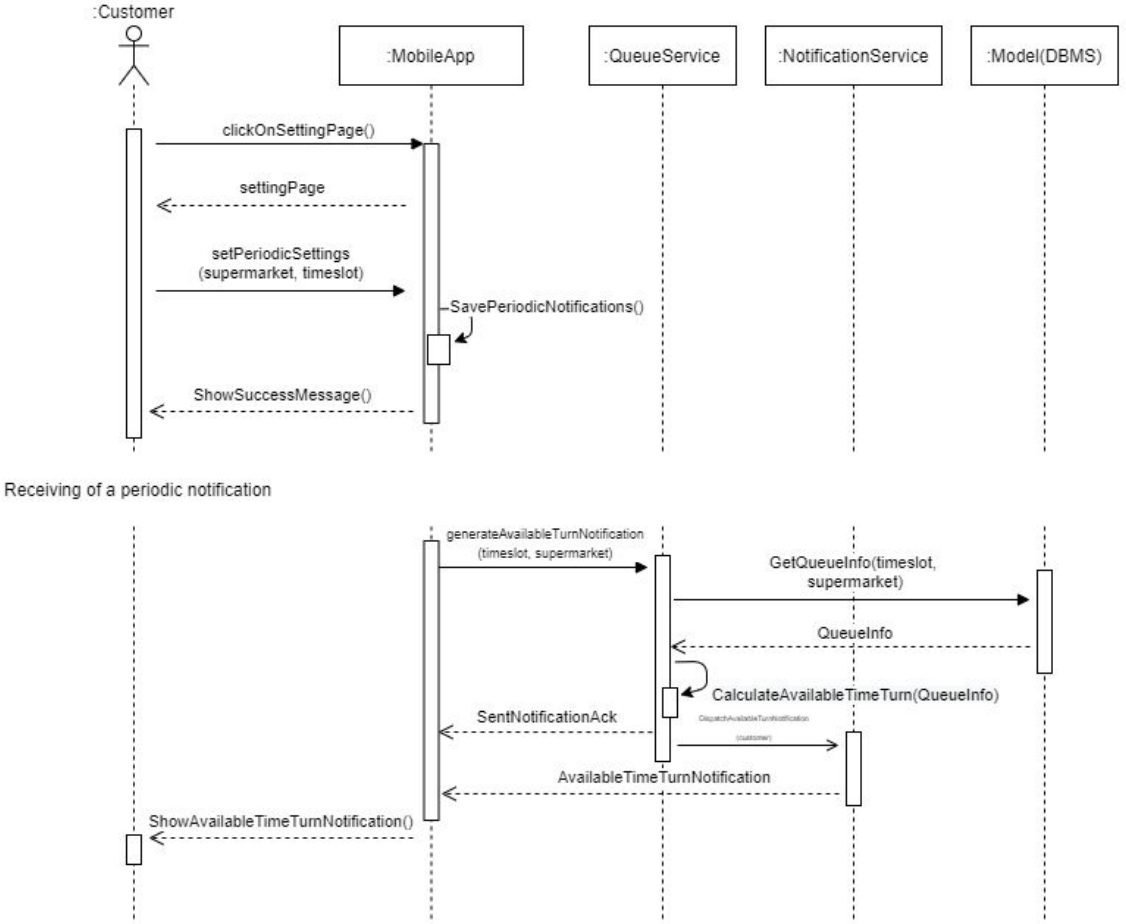

Image 11: runtime view 9

This sequence shows the interaction between components when an online customer chooses to receive periodic notification about the availability of a turn in the line of a specified supermarket in a specific time slot. When the user sets a periodic notification, the mobileApp saves the information about the preferred store/s and time slot on the smartphone. 24 h before the selected time the mobileApp requests the QueueService to check if there is an available turn inside the time frame. The QueueService uses the information that obtains from the Model to check the availability and delegates the NotificationService to send a notification that informs the user of the free turns found.

2.D.10 Setting of parameters for the supermarket by the manager

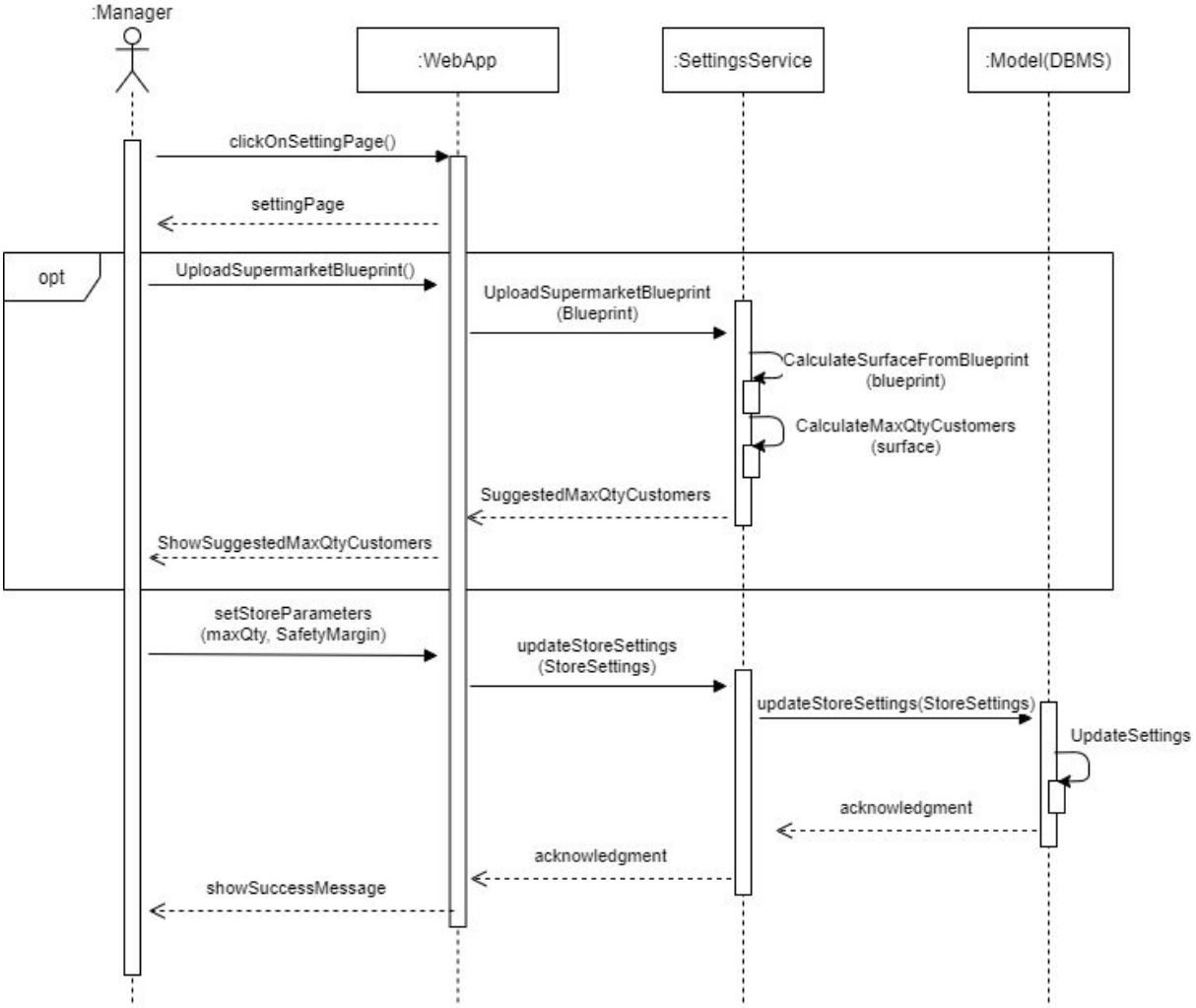

Image 12: runtime view 10

This sequence shows the interaction between components when the store manager sets the main parameters of the store queue(max quantity of people that can enter the store and the safety margin). Before setting the parameters, he can firstly upload a blueprint of the supermarket to receive some suggestions about the number of customers that can visit at the same time the store. If he uploads a blueprint, the mobile app sends it to the SettingService that processes the data and responds with an estimated quantity. After the manager sets the parameters, the mobileApp requests the SettingService to update these quantities inside the database. Finally, when all data is stored, the manager receives a message of confirmation.

2.E Component Interfaces

This graph contains a graph that shows the interfaces of the different components defined previously. The names of the methods defined in the interfaces are self-explanatory and reasons for their use can be understood thanks to the sequence diagrams in section 2.D. One thing to point out is that all the methods shown are not meant to be followed precisely by the developers. They are just a simplified representation of what the different components can offer and how they interact with each other. That is, these graphs help the stakeholders to understand how the structure of the system works but it also gives a simple guideline for the developers nonetheless.
For sake of readability, the main graph has been divided into two subgraphs. The former represents the interfaces used by the manager whereas the latter represents the ones used by the customer.

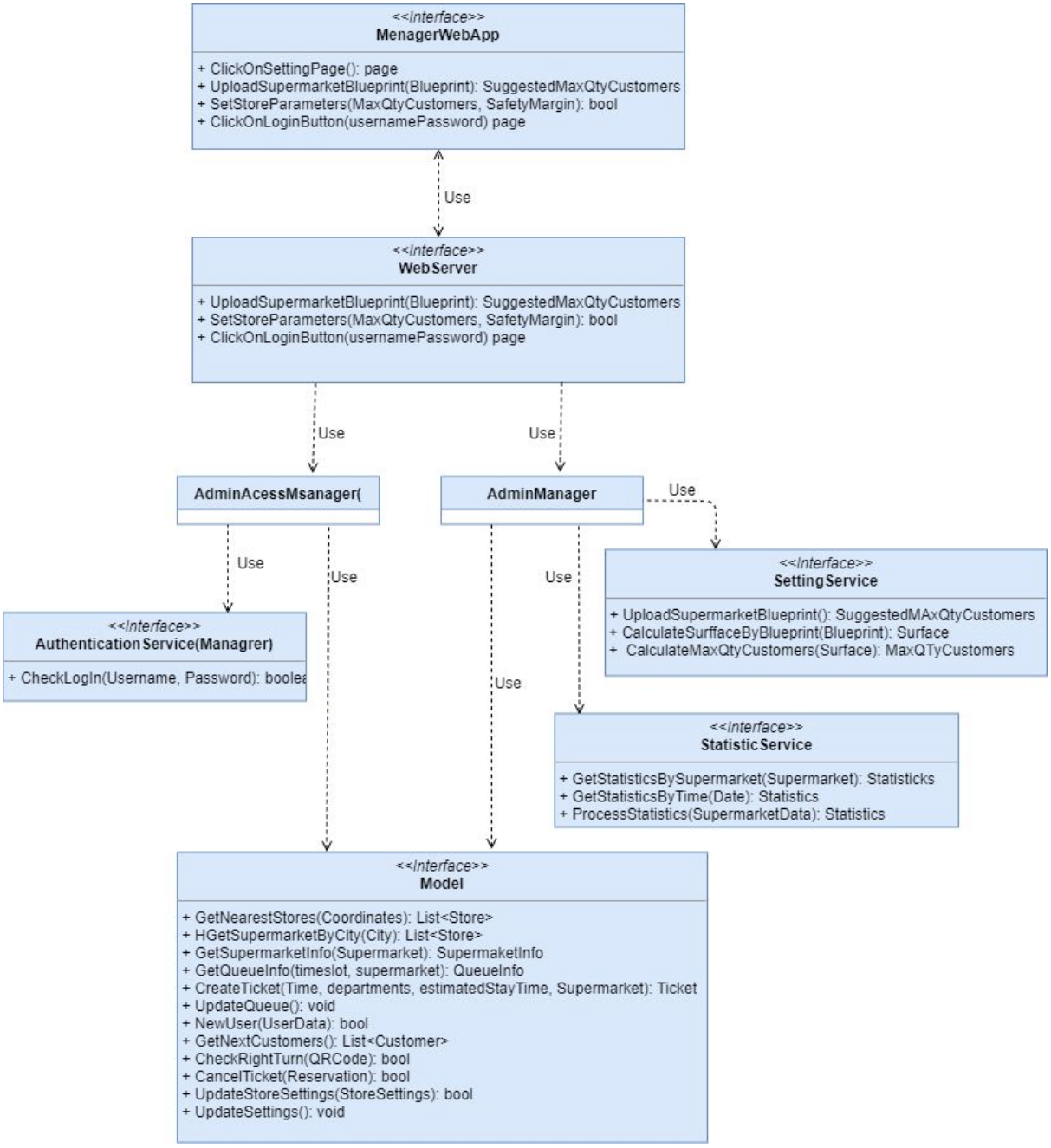

Image 13: component interface for manager

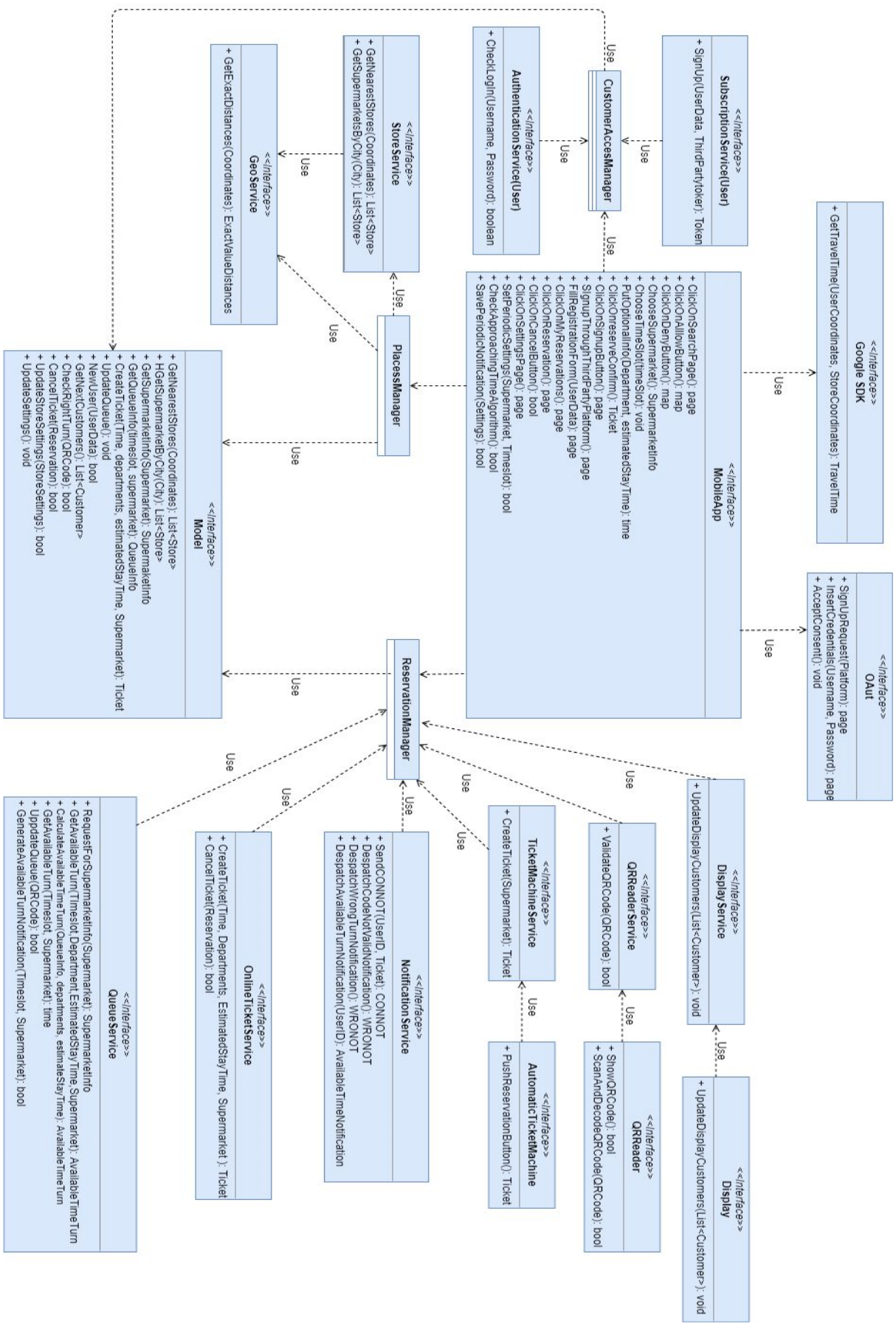

Image 14: component interface for customer

2.F Selected architectural styles and patterns

RESTful architecture

The architecture that must be used is RESTful, the goal is to reduce the coupling among client and server components as much as possible. In order to be a truly RESTful API, the system must adhere to the following six REST architectural constraints:

- Resources should be uniquely identifiable through a single URL, and only by using the underlying methods of the network protocol, such as GET, POST, PUT, DELETE with HTTP
- Client-server based. as explained so far, there should be a clear delineation between the client and the server. UI and request-gathering concerns are the client's
- Stateless operations. All client-server operations should be stateless, and any state management that is required should take place on the client, not the server.
- RESTful resource caching. All resources should allow caching unless explicitly indicated that caching is not possible.
- Layered system. REST allows for an architecture composed of multiple layers of servers.
- Code on demand. the server will send back static representations of resources in JSON format. However, when necessary, servers can send executable code to the client.

Model-View-Presenter Pattern (for the smartphone APP)

The MVP pattern allows separating the presentation layer from the logic. Everything about how the UI works are agnostic from how it is represented on the Android/IOS device. In this way, the code is easily extensible and maintainable.

Model-View-Controller (for the web app)

For the implementation of the application, the main design pattern that has been followed is the MVC one. That is, it is one of the most used patterns nowadays for the production of applications of different kinds. The pattern is base on the 3 main blocks that are:

Model: defines the structure of the stored data of the application and the operations that can be done upon them.

View: manages the presentation logic of the data and represents the main interface used to interact with the Model

Controller: implements the control logic of the application translating the interactions between the user and the View inactions on the Model and the data it stores.

2.G Other design decisions

GPS Permission

The system should ask the user for permission for the use of his smartphone GPS and should use it only to show the nearby supermarkets and to estimate a time arrival.

Data Protection

All the personal customer data should be preserved and protected on encrypted databases and in case of password recovery, all the exchanged data between the user and system should be encrypted as well. Also, the system should be protected against any attack that aims to steal the private information of both customers and supermarkets.

JWT Authentication

In order to authenticate the requests from clients (mobile app, web app) we chose to use a token-based authentication, more precisely, the JWT (JSON Web Token) authentication.
JSON Web Token is a standard used to create access tokens for a client.

It works this way: the server once received credentials and validated them, generates a **JWT token** that certifies the user identity, and a **refresh token** (that will be useful later) and sends them to the client.
The client will send the JWT token back to the server (into the request header) for every subsequent request, so the server knows the request comes from a specific identity.

The provided token from the server has a short duration of 360 minutes; after its expiration, the immediately following request to the server will respond with a 401 status (unauthenticated). This means that the client automatically has to call another endpoint sending the refresh token to get back a fresh JWT token.

2.H Algorithms

This paragraph contains the algorithms of some of the main functionalities of the system. They are developed through a Java-like language and it can help both the stakeholders and the future developers to better understand the logic behind some of the functionalities incorporated in the system.

2.H.1 ApproachingNotification

```
public boolean function CheckApproachingTurnAlgorithm
    (Int travelTime, LocalStorageReservation reservation){

  // defining now variable
  LocalDateTime now = new LocalDateTime();
  Boolean notifictionSent = false;
  Int threshold = 10;

  // If the mobile app hasn't send this notification yet
  if(!reservation.getSentAPNOT()){

    // if the reservation time is before (now + (calculated travel time) + (threshold))
    if(reservation.getDatetime() <= now.addMinutes(travelTime).addMinutes(threshold)){

      // send the APNOT notification
      notificationService.dispatchApproachingNotification(reservation.getUser().getId())
      notifictionSent = true;
      reservation.setSentAPNOT(true);

    }

  }

  return notificationSent;

}
```

Image 15: approaching notification Algorithm

The code above describes the main logic behind the control that the CLup mobile application performs every time that it needs to check whether to send an APNOT to the customer or not. In this case, we use the travel time from the user position to the supermarket and the details of the reservation. Whenever the check is done the system registers the present time(time of the check) and sums to it the travel time, calculated before, and a threshold of ten minutes. The latter is added to consider a delay between the user receiving the notification and him approaching the store. Once the addition has been done, the result is compared to the time of the reservation and the system understands whether it's time for the customer to move toward the supermarket to arrive in time for his turn or not.

This operation begins one hour before the reservation and the check is done every ten minutes. Following this schedule and thanks to the threshold(that is put equal to the check interval), the system can notify with a modest advance

the user about his approaching reservation avoiding, for the most part, late notifications.

2.H.2 CalculateAvailbaleTimeTurn

```
public LocalDateTime function CalculateAvailableTimeTurn(Date date, TimeSlot timeslot, Int estimatedStayTime,
Suprmarket supermarket){

  // take reservations in chosen time slot from database
  List<Reservation> reservationInTimeSlots = Reservation.getByTimeSlot(supermarket, date, timeslot);

  // TimeSlot: 1 hour
  // TimeCell: 5 minutes

  List<Integer> numCustomersInTimeCells;


  // for each 5-minutes-cell into the time 1-hour-slot
  for(timeCell : timeslot.getTimeCells()){

    // Setting the initial number of customer for 5-minutes-cell to 0
    numCustomersInTimeCells[timeCell.getNumber()] = 0;

    // for each reservation inside the 1-hour-slot
    for(Reservation reservation : reservationInTimeSlots){

      // if reservation occupies the current 5-minutes-cell
      if(reservation.getStart() <= timeCell.getStart() && reservation.getEnd() >= timeCell.getEnd()){

        // Incrementing the number of customers inside the sumerket in that 5-minutes-cell
        numCustomersInTimeCells[timeCell.getNumber()]++;

      }

    }

  }

  // Calulcate numTimeCells in the customer estimated time for this reservation
  // Of course estimatedStayTime is multiple of 5
  Integer numTimeCells = estimatedStayTime / 5;

  // Calculating the max number customers allowed in sumpermarket at the same moment
  maxCustomersSimultaneously = supermarket.getMaxCustomer() - supermarket.getSafetyMargin();

  // Now I have to find the first consective 5-minutes-cells that have a number of customers less than
maxCustomersSimultaneously
  // (mimimum numTimeCells nuber of cells)

  int index = 0;
  List<integer> takenCells;
  // for each 5-minuts-cell, retrieve the num customers previously populated
  for(int numCustomers: numCustomersInTimeCells){

    if(numCustomers < maxCustomersSimultaneously){
      takenCells.add(index);
    }else{
      // if the consecutiveness is break, i have to start from an empty list
      takenCells.clear();
    }
    index++;

  }

  // if number of collected 5-minutes-cells is higher then the required number of time cells
  if(takenCells.size() >= numTimeCells){

    // return the available time turn
    // 16:00 + (3 + 5)minutes
   return timeslot.start().addMinutes((5 * takenCells[0]));

  }else{

    // return negative result
    return null;
  }

}
```

Image 16: calculate available time turn algorithm

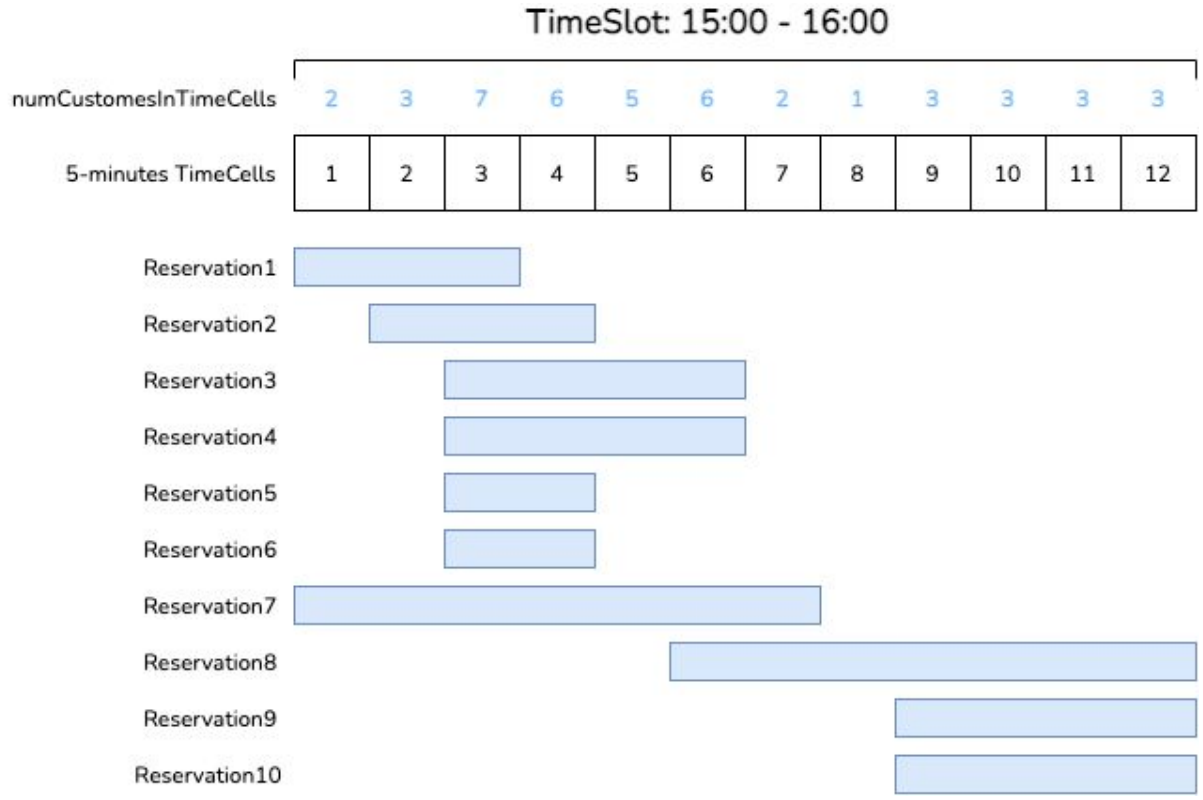

Image 17: a representation of an example of timeslot and how it is divided in 5-minutes-cells by the algorithm, in order to find the less-crowded time for the reservation.

The algorithm above describes the logic behind the research for an available turn in line at any time that an online or offline customer wants to book a reservation.

The check is done utilizing the many information given by the customer like the chosen supermarket, the selected time slot and estimated stay time. Thanks to this data the system can retrieve the number of customers that are present in the chosen time slot inside the store. This is possible by dividing the time slot of 1h into 12 time 5-minutes cells each and for every reservation that occupies one of them, their internal counter (the number of customers present in the shop at that time) is increased. Once the number of customers in each time cell is defined, the system can look for a row of consecutive cells that can host another customer. In case the row is found the beginning time of the first cell is sent to the customer to show him at what time his reservation is.

Of course, the selection of cells is also based upon the different parameters previously mentioned. As a matter of fact, if the user inserted also the time of

his stay and the departments he intends to visit the system could also select time cells that are at the limit of capacity. In this case, the system would take advantage of the Safety Margin that has been introduced in the system also for this reason. However, for sake of readability and comprehension, the code represents only the general case to allow the stakeholders to understand the main logic behind this procedure.

3. User interface Design

3.A Customer Mobile App

3.A.1 Splash Screen

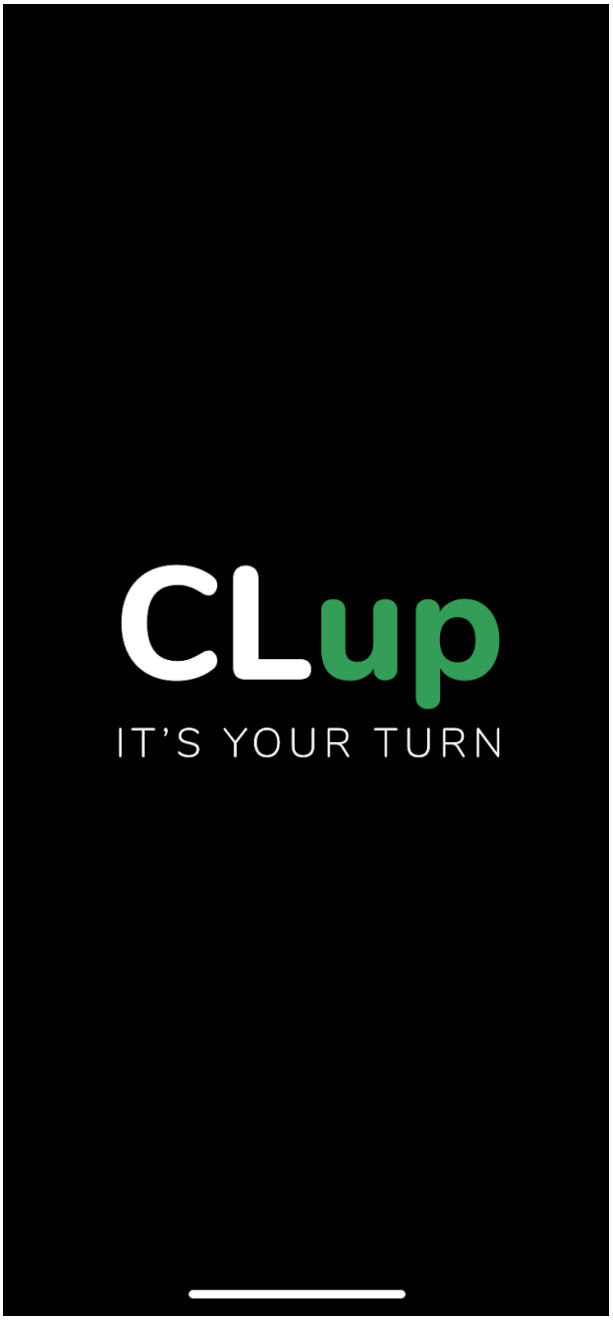

Image 18: splash screen

3.A.2 Book a turn flow

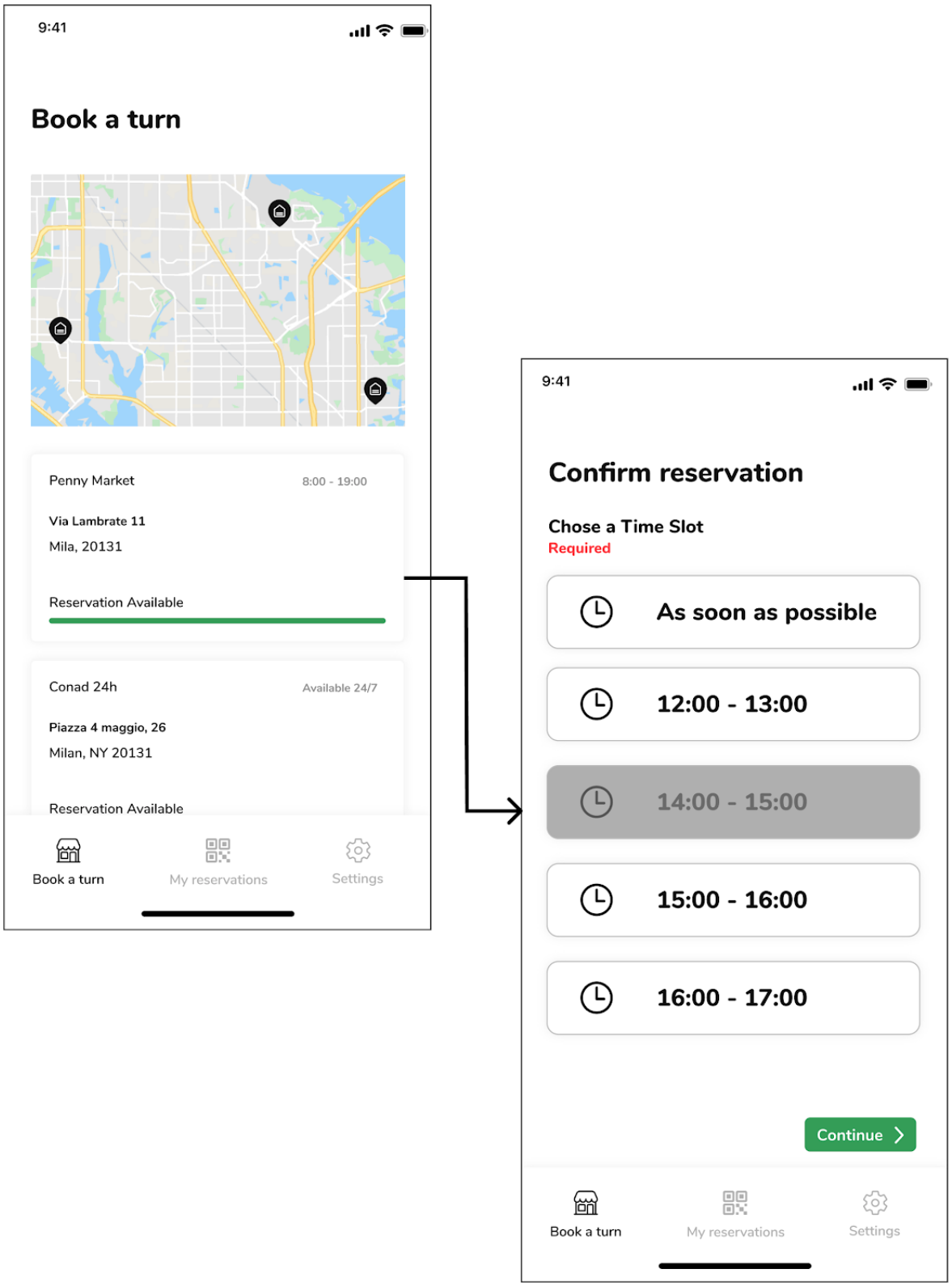

Image 19: book a turn flow part 1

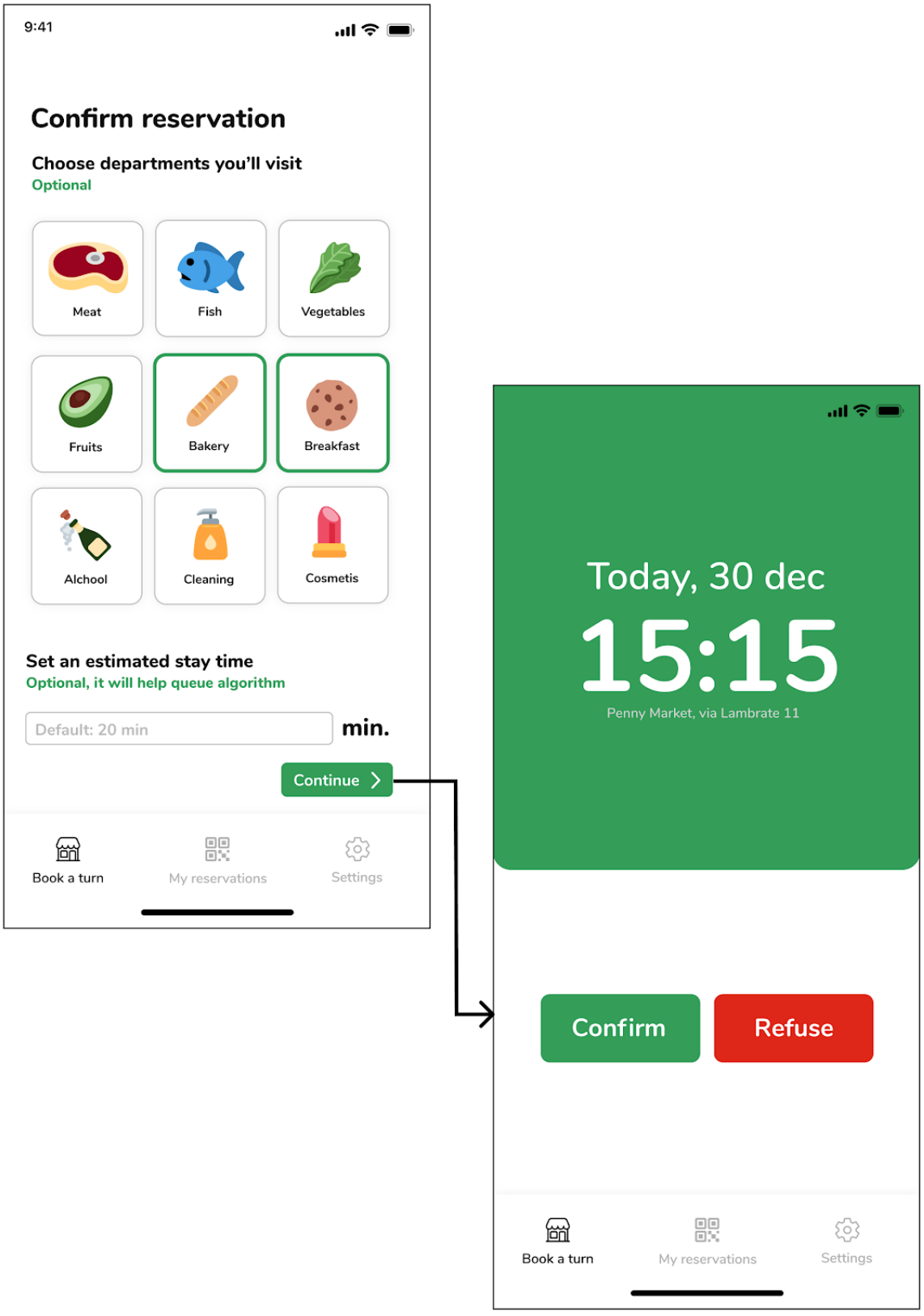

Image 20: book a turn flow part 2

3.A.3 My reservation screen

Image 21: my reservation screen

3.B Manager Web App

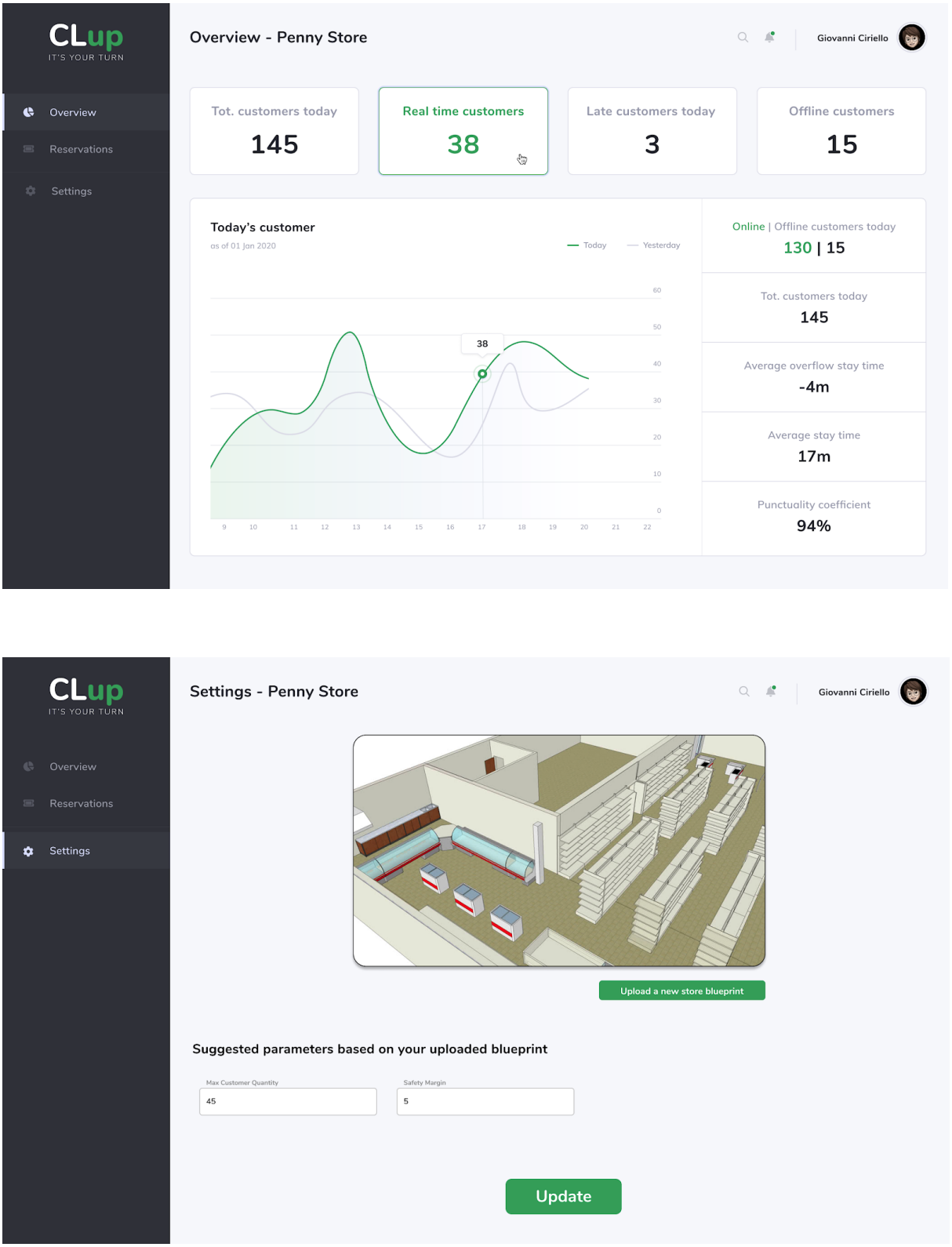

Image 22: manager webapp

4. REQUIREMENTS TRACEABILITY

The entire design must ensure that the system can enforce all the requirements defined in the previous document (RASD) and consequently that the system can achieve the goals that have been established. This chapter shows a mapping between the established goals and the design components illustrated in the DD. For a better mapping between components and requirements, the reader should refer to the RASD document that has the list of the latter.

	Goal	Requirements	Components
G1	Allow store managers to regulate the influx of customers inside the supermarket	**R1, R5 ,R7, R8, R10, R11, R15, R16, R17, R18, R19, R21, R24, R25, R26, R27, R29, R30**	• ManagerWebApp • WebServer • SettingService • Model (DBMS) • StatisticService • MobileApp • OAuthAPI • SubscriptionService • AuthentucationService • QueueService • OnlineTicketService • NotificationService
G2	Let people avoid to line up and stand outside of stores creating assemblage through the creation of a virtual line	**R1, R2, R6, R7, R8, R9, R12, R13, R14, R15, R16, R17, R18, R21, R22, R23, R24, R25, R29**	• MobileApp • OAuthAPI • SubscriptionService • AuthenticationService • QueueService • OnlineTicketService • Display • QRReader • AutomaticTicketMachine • DisplayService • QRReaderService • AutomaticTicketMachineService • NotificationService • Model (DBMS)

G3	Ensure social distancing between customers into different store departments	**R11, R12, R19, R26, R27, R29, R30, R31**	• MobileApp • OAuthAPI • SubscriptionService • AuthenticationService • OnlineTicketService • ManagerWebApp • WebServer • QueueService • SettingService • Display • QRReader • DisplayService • QRReaderService • Model (DBMS)
G4	Allow customers to book a reservation to do shopping in a store	**R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R12, R13, R14, R15, R19, R20, R21, R22, R23, R25, R27, R28**	• MobileApp • OAuthAPI • SubscriptionService • AuthenticationService • OnlineTicketService • QueueService • AutomaticTicketMachine • AutomaticTicketMachineService • StoreService • GeoService(GoogleAPIRoads) • GoogleMapsSDK • Model (DBMS)
G5	Distribute the flow of the customers in multiple stores avoiding overcrowding them	**R1, R2, R3, R11, R12, R13, R14, R15, R15, R17, R19, R20, R26, R27, R30**	• MobileApp • OAuthAPI • SubscriptionService • AuthenticationService • GoogleMapsSDK • StoreService • OnlineTicketService • QueueService • Model (DBMS)

5. IMPLEMENTATION, INTEGRATION, AND TEST PLAN

5.1 Implementation plan

The entire system, with its relative sub-systems, has to be implemented, but also tested and integrated exploiting a bottom-up approach. This approach is chosen both for the server-side and the client-side, which will be implemented and tested in parallel. Using bottom-up, the system could be done incrementally so that also testing can proceed in parallel with the implementation.

The implementation has to be done from the lower components up to the top because in this approach the implementation is gradual. Some components rely on some others so a priority among the components is present. For example, every component, described in the Component diagram, interacts with the Model that is the bridge between the DBMSService and the Application Server. The first step is to implement the Model, which is the component that implements all methods that allow access to the Database and perform queries and updates on it.

After that, we can proceed with the implementation of the other sub-systems with their respective components. The main subsystem is the ReservationManager, which allows the online and offline customer to make a reservation in a supermarket. The components inside the ReservationManager that interact with the Model are:

- QueueService
- QRReaderService
- TicketMachineService
- OnlineTicketService

Since the QrReaderService, TicketMachineService, and OnlineTicketService use the interface provided by the QueueService we should implement the latter to keep using the bottom-up approach. Anyway, the QueueService depends on the implementation of NotificationService and DisplayService.

Therefore we have to start with the implementation of these latter two. After that, we continue with the implementation of QueueService, TicketMachineService, QrReaderService.
Then we can proceed with the implementation of the CustomerAuthManager and the PlacesManager.

Finally, we define the AdminManager and the AdminAuthManager.

The order in which we implement our components is:

1. **Reservation Manager**
    a. NotificationService
    b. DisplayService
    c. QueueService
    d. OnlineTicketService
    e. TicketMachineService
    f. QRReaderService
2. **CustomerAuthManager**
3. **PlacesManager**
4. **AdminManager**
5. **AdminAuthManager**

5.2 Integration strategy

In order to implement and test the different functionalities of the system, a bottom-up approach has been used. The following diagrams describe how the process of implementation and integration testing takes place.

1. At first, the ReservationManager is implemented.
   The components of the ReservationManager we chose to implement firstly are DisplayService and NotificationService. DisplayService needs to be tested using a driver and a stub to cope with the lack of communication with Display. The same is done for NotificationService that uses a driver and a stub. The addition of these stubs goes in contradiction with the bottom-up approach but is needed in order to proceed with the integration in the application server.

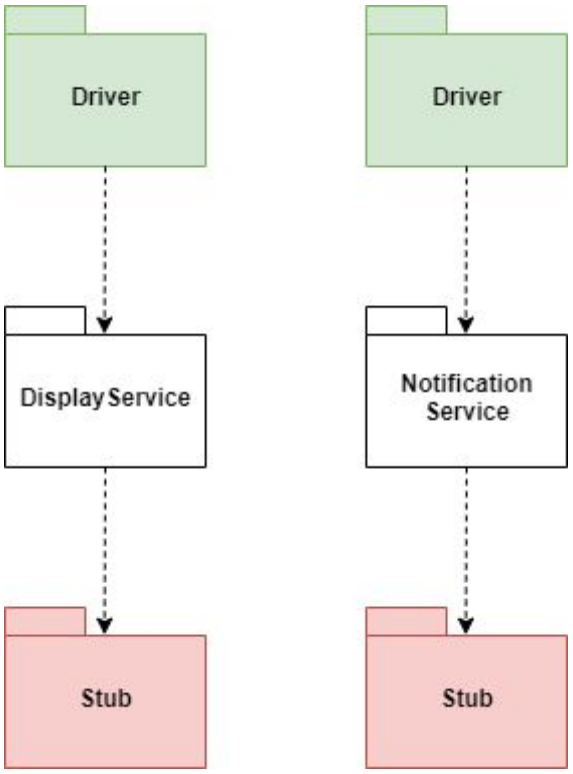

Image 23: Integration strategy 1

2. At this point we can substitute the two Drivers with the main important component of the ReservationManager, that is the QueueService. This

one is piloted by a Driver and interacts with the DisplayService, Notification Service, and the Model(the first component implemented).

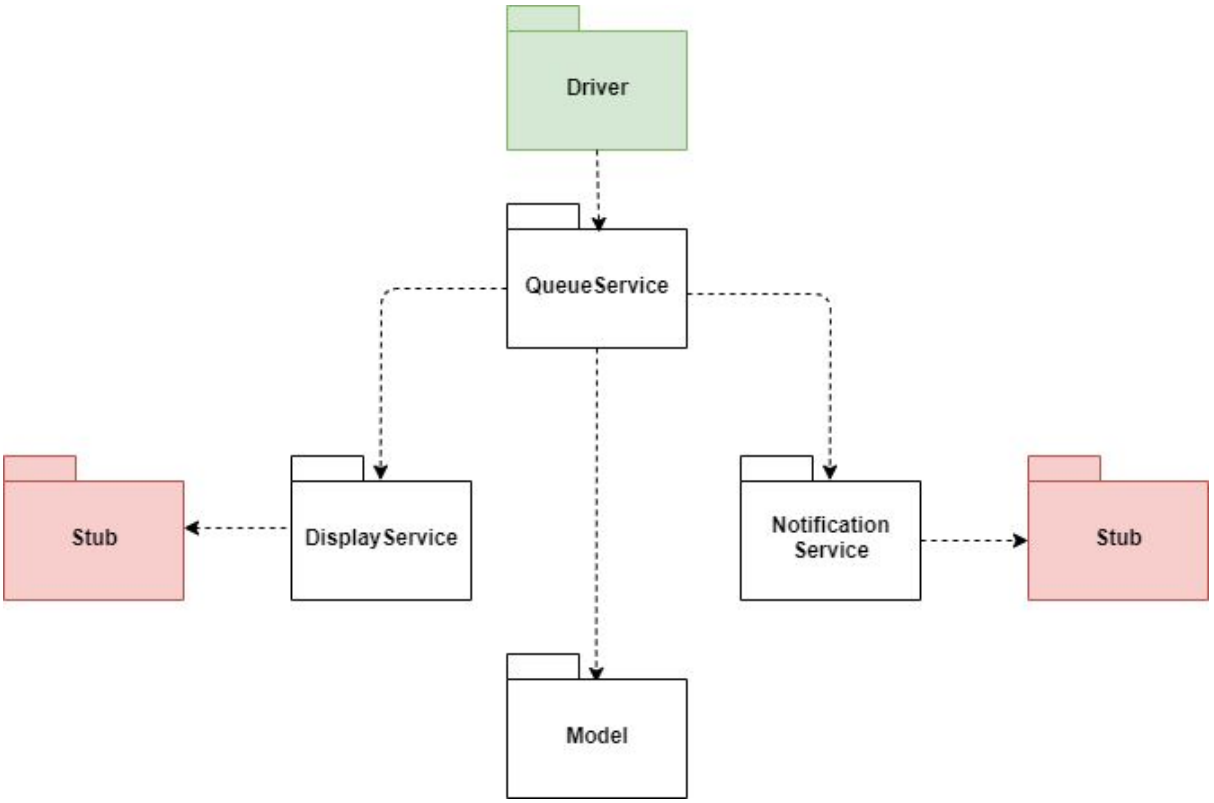

Image 24: Integration strategy 2

3. Then the driver is substituted with the two components that use the interface provided by the QueueService, which are TicketMachineService and MachineService. These are piloted by two Drivers and can be tested in parallel because there are no direct dependencies between them. Then we also add the OnlineTicketService.

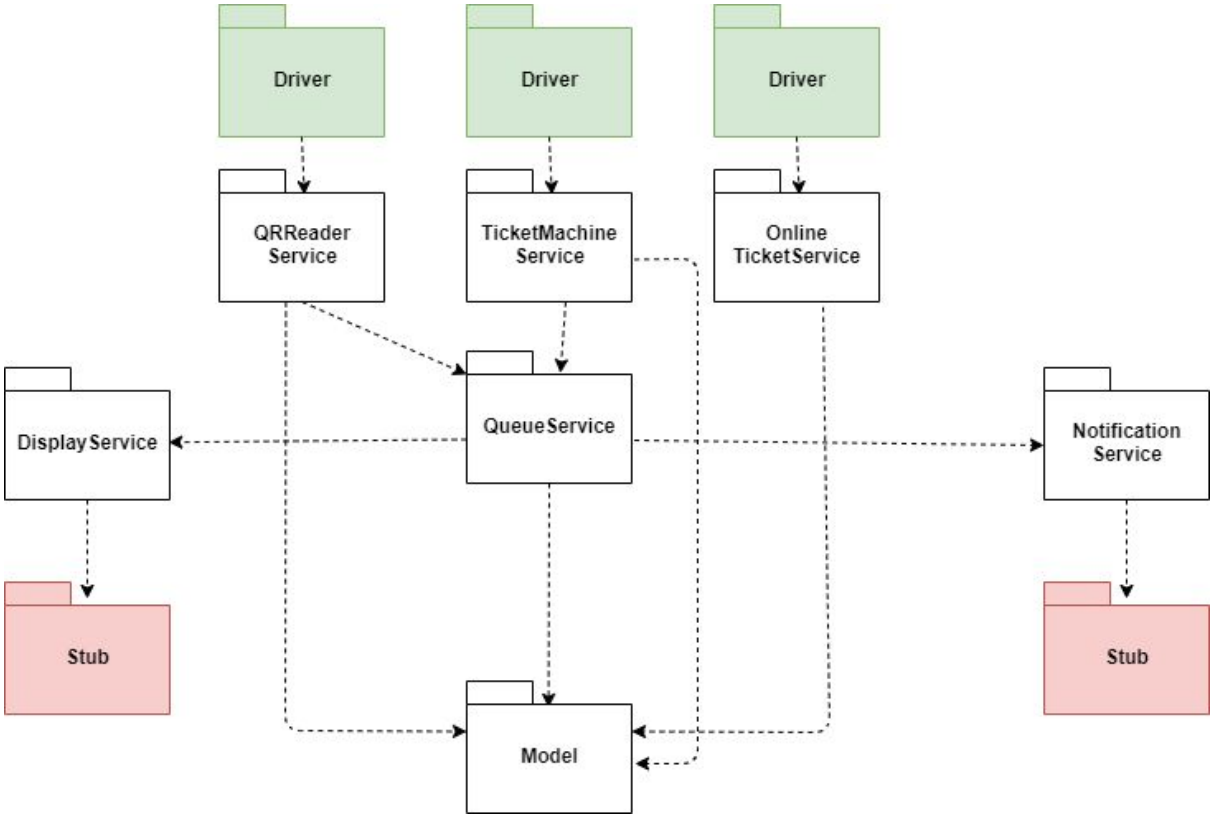

Image 25: Integration strategy 3

4. At this point, the ReservationManager is tested and we can substitute two of the Drivers with QRReader and TicketMachine. The other one will be substituted only at the end when the MobileApp will be implemented. We also substitute the Stub invoked by DisplayService with Display. The stub used by NotificationService will be replaced at the end with the MobileApp.

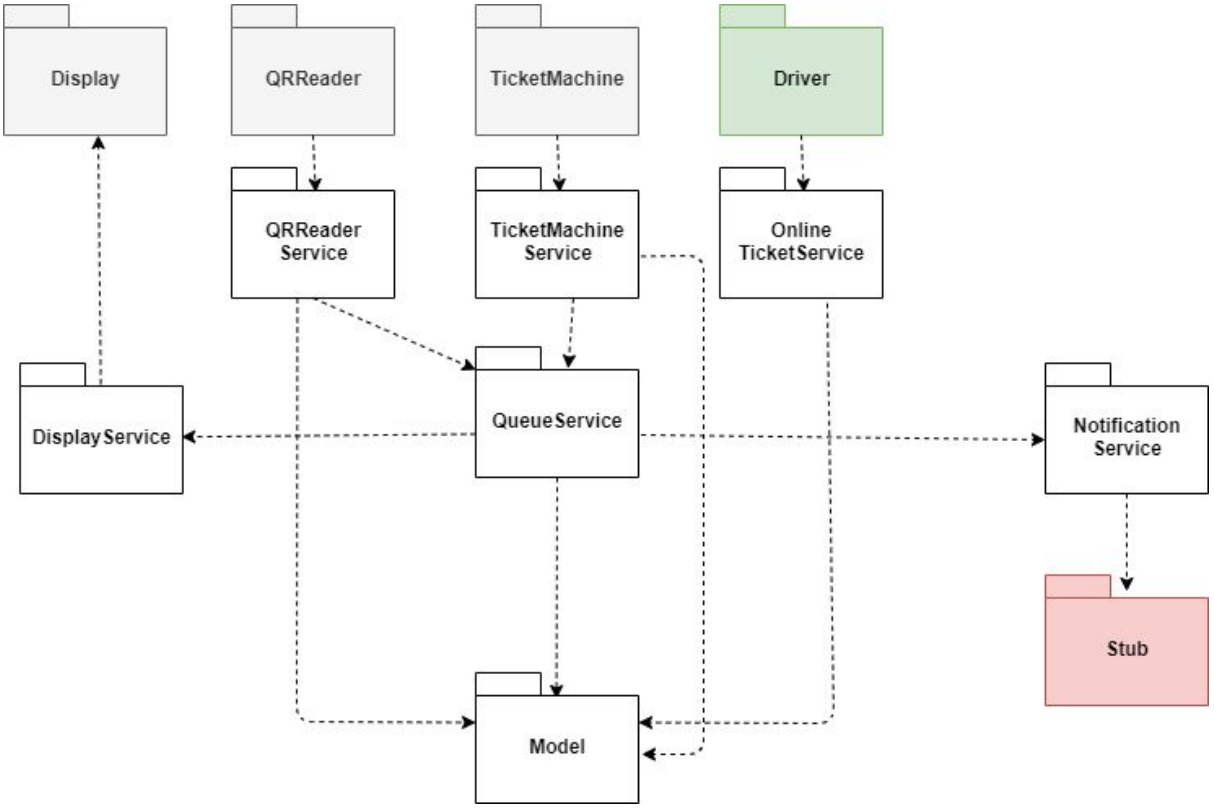

Image 26: Integration strategy 4

5. The following image is a high-level perspective of the integration of the ReservationManager.

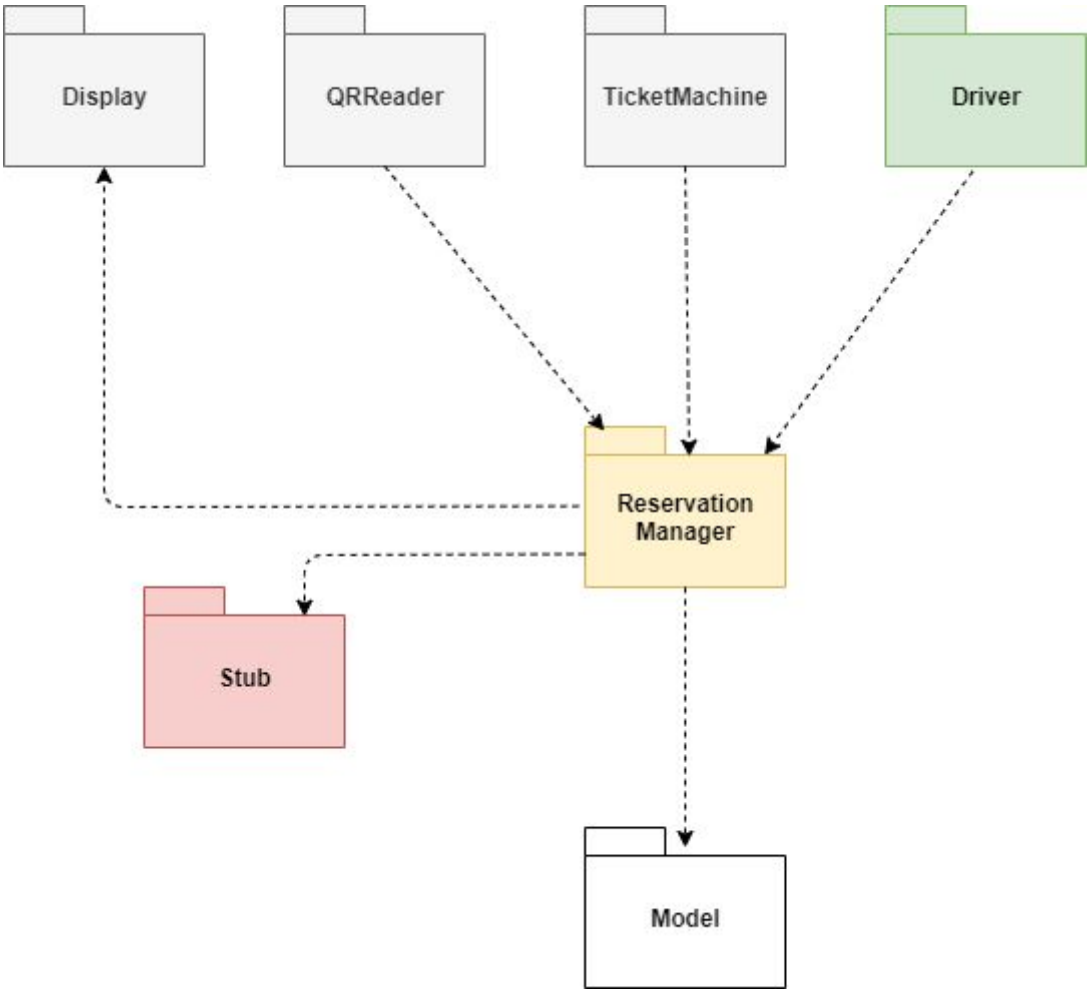

Image 27: Integration strategy 5

6. Then we can proceed with the implementation and integration of CustomerAuthManager and PlacesManager. They can be tested in parallel because there are no direct dependencies between them. In this step, we can also integrate GoogleAPIRoads without testing because it is an external service offered by a trusted provider.

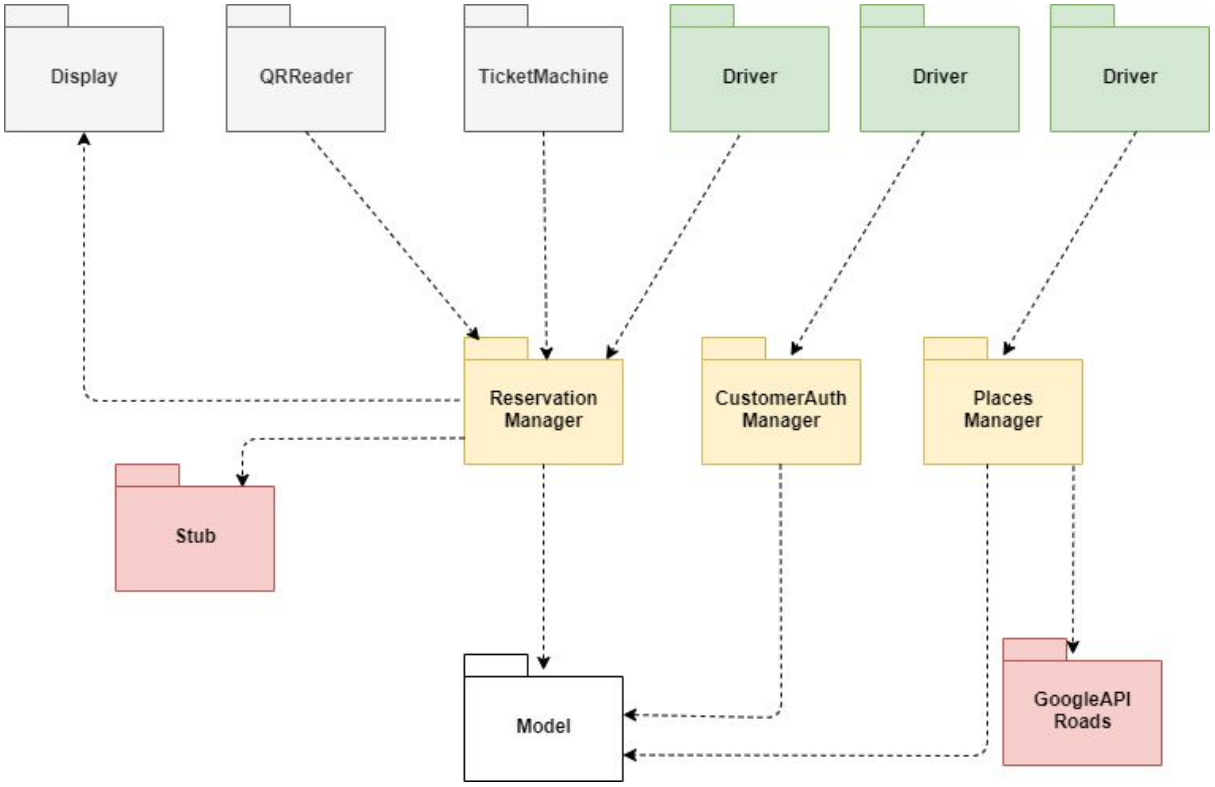

Image 28: Integration strategy 6

7. Once we have defined the functionalities of the application server related to the mobile app, we can substitute the drivers and the stub with the mobile app. We can also integrate GoogleMapsSDK and OAuthService without testing them because they are external services offered by trusted providers.

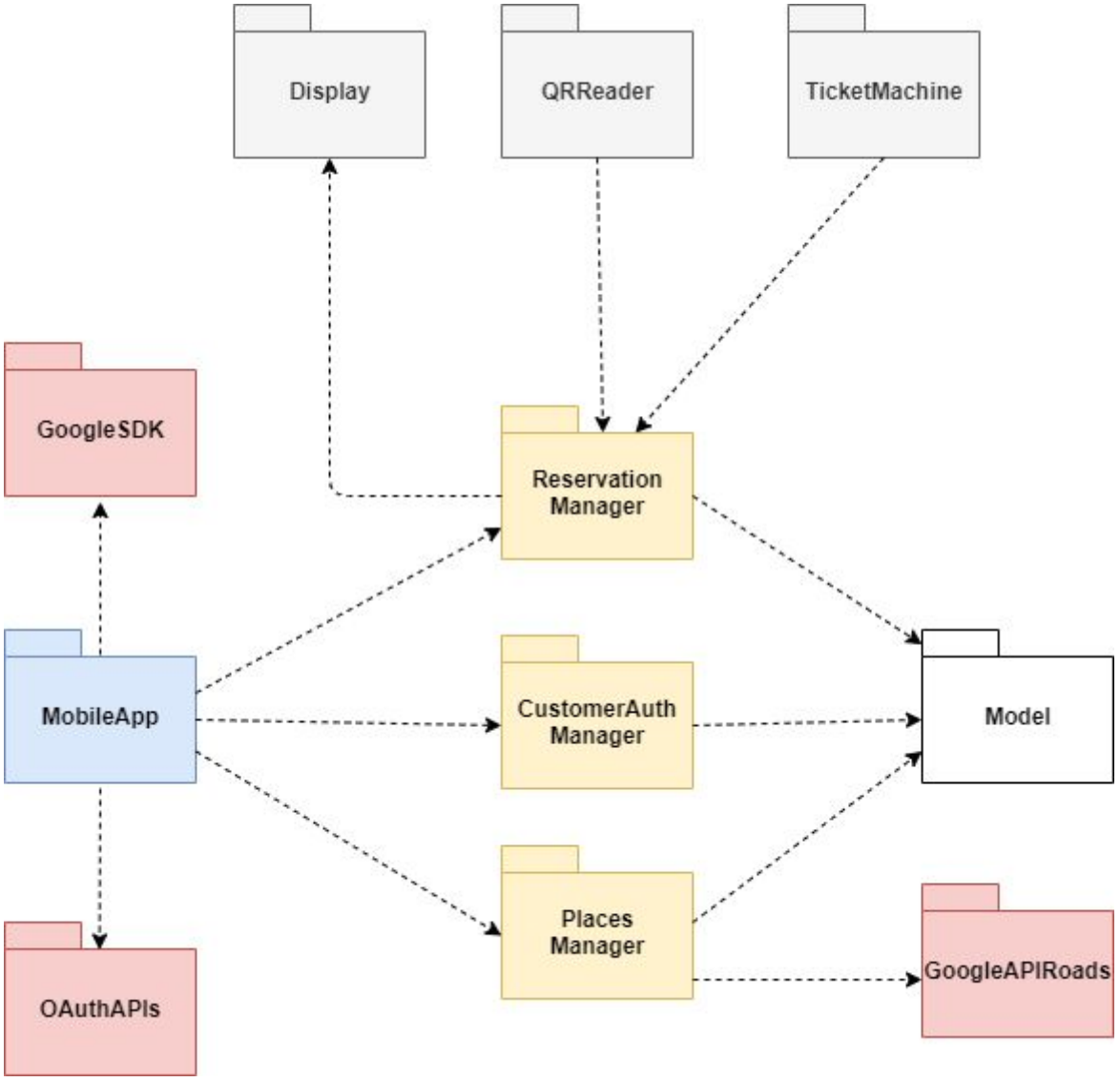

Image 29: Integration strategy 7

8. Then we can implement and test the part of the application server related to the manager of the supermarket.

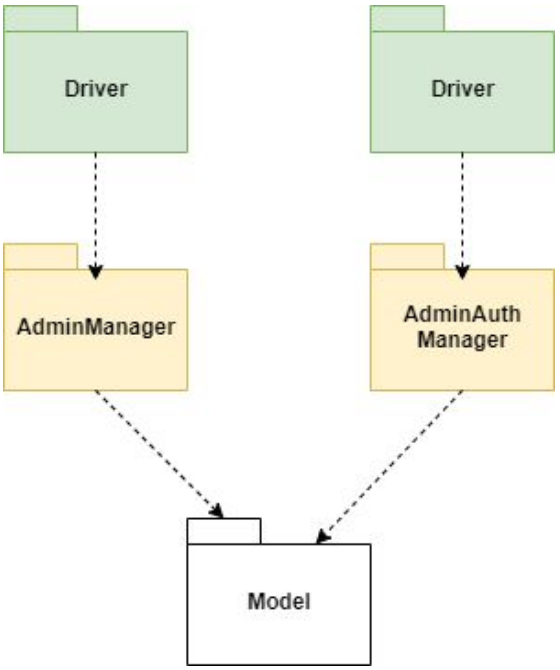

Image 30: Integration strategy 8

9. Finally, the remaining part characterizing the WebServer and WebApp can be implemented, unit-tested, and integrated into the system.

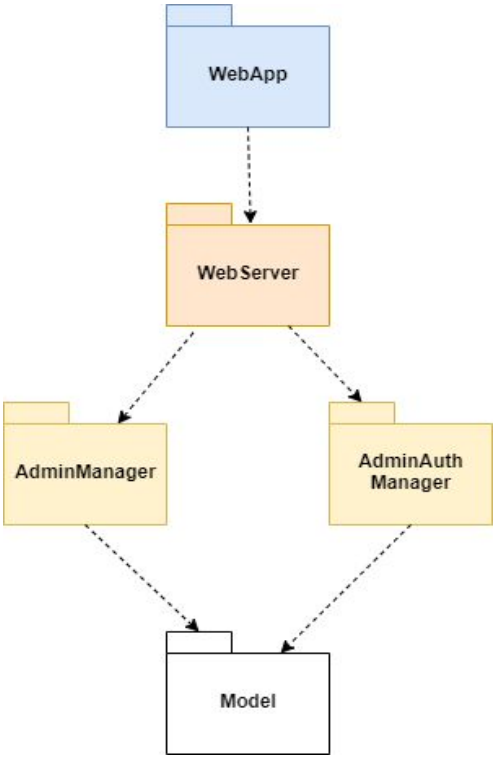

Image 31: Integration strategy 9

5.3 System testing

The Verification and Validation phase must begin as soon as the development phase begins, this means that the Unit Tests must be done in parallel with the implementation of the components, using a white-box approach. Moreover, the Integration Tests must start at the moment in which the integration itself begins, in this case using a black-box approach. This incremental testing plan will ensure that the majority of bugs will emerge in the early phases of production so it will be easier to correct them.

Once the system is completely integrated, it must be overall tested for verifying that functional and non-functional requirements hold. The system should go under the three main kind of testing that is:

- **Performance testing** to identify bottlenecks and possible algorithm optimization
- **Load testing** helps find bugs and memory leaks
- **Stress testing** to observe the resilience of the entire system under heavy load.

All the tests in general are fundamental for studying the behavior of the system. For example, performance testing helps understand whether the algorithms explained in the chapters above work efficiently. The load testing allows developers to observe if the system could support all the many bookings and interactions made by the thousands of customers that could use the application. Stress testing is important because it stimulates the possible hardware and software failures procured by faulty hardware and tests the ability of the system to recover from them.

As mentioned previously all the system testing needs to emulate an environment as close as possible to the production environment.

6. EFFORT SPENT

Amato Nunzio

Topic	Hours
Discussion on the second part	1 h
Runtime view	4 h
Component diagram	3 h
Description of component diagram	1 h
Deployment diagram	2 h
Requirements Traceability	1 h
Mockups	4 h
Algorithm	1 h
Implementation, integration and testing	7 h
Component interface	1 h
Document Revisions	2 h

Ciriello Giovanni

Topic	Hours
Discussion on the second part	1 h
Runtime view	2 h
Component diagram	3 h
Description of component diagram	1 h

Deployment diagram	4 h
Requirements Traceability	2 h
Mockups	10 h
Algorithm	1 h
Implementation, integration and testing	2 h
Component interface	1 h
Document Revisions	2 h

Colombrino Fulvio

Topic	**Hours**
Discussion on the second part	1 h
Runtime view	3 h
Component diagram	2 h
Description of component diagram	2 h
Deployment diagram	2 h
Requirements Traceability	1 h
Mockups	5 h
Algorithm	2 h
Implementation, integration, and testing	2 h
Component interface	4 h
Document Revisions	3 h

7. REFERENCES

- Runtime view, component diagram, and other diagrams made using: https://app.diagrams.net/
- Interface Mockups design made using Figma: https://figma.com
- Java code is written with Intellij
- Wikipedia https://www.wikipedia.org/
- Software engineering, Global edition, Ian Sommerville

Image Index

Changelog

- 11/01/2021
    - Minor layout improvements