POLITECNICO
MILANO 1863

Customer Line Up DD

Design Document

Etion Pinari 10619348
Giorgio Romeo 10608778
Cristian Sbrolli 10607290

AY 2020/2021

Table of contents

CLup

Customer Line-up

1) INTRODUCTION

A) *PURPOSE*

The purpose of this document is to build a more concrete foundation of what the system-to-be will be. It will also define the general behavior and specific limitations of the system. This document is primarily addressed to the programmers and mostly includes technical language.

B) *SCOPE*

The scope of the design document is to define the system's behavior in general cases and some critical scenarios, and to design the architecture of the system-to-be so as to provide a time-efficient, logical allocation of the components and the interaction between these components.
The document is not only limited to the architecture and behavior of the components, but it also extends in part to the implementation and testing plan, where one possible course of action is explained, user interface design of user applications and requirements traceability relating to the Requirements and Specifications Document (RASD).

C) *DEFINITIONS, ACRONYMS, ABBREVIATIONS*

FCM: Firebase Cloud Messaging

D) *REVISION HISTORY*

19/12/2020: High level definition of the architecture of our system
Duration: 2hrs

21/12/2020: Components' logical definition
Duration: 2.5hrs

26/12/2020: Component interaction

Duration: 1.5hrs

28/12/2020: Sequence diagram definition
Duration: 2.5hrs

30/12/2020: Timeslot method definition
Duration: 2hrs

03/12/2021: Notifications and implementation definition
Duration: 1.5hrs

06/12/2021: Review of used methods
Duration: 2hrs

10/12/2021: Closing remarks
Duration: 2hrs

E) *REFERENCE DOCUMENTS*

- Interactive mobile app mockup: *https://customerlineup.bubbleapps.io/*
- Interactive web app mockup: https://customerlineupmanager.bubbleapps.io/ *(login with user: admin@mail.it password: admin)*

2) ARCHITECTURAL DESIGN

A) *Overview: high-level components and their interaction*

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

- *Presentation Layer (P)* manages the presentation logic, namely the interaction with the user. It comprises a GUI (Graphic User Interface) that makes the application's functionalities more understandable to the user.
- *Business Logic* or *Application Layer* (*A*) handles all the functions to provide to the user and manages the exchange of information between the user interface and the data source.
- *Data Access Layer* (*D*) provides access to the stored data. The implementation of the access logic should be both easy and structurally robust to guarantee a correct abstraction from the specific database and provide a model easy to use.

In order to guarantee as much flexibility and scalability as possible, the system is based on a 4-tier architecture (Client, Web Server, Application Server, Database Server) with a thin client. Since the application should be easy to use and executable in several different devices, the use of a thin client prevents a heavy computation load client side, carrying out all the heavy operation at server side. As represented in figure 2.1 the user, through a smartphone or a tablet, and the physical dispenser can directly communicate with the application server, while the store manager can access the functionalities devoted to him through a web application communicating with the web server. The use of a web server for the store manager's functionalities is based on the fact that, in general, the web apps are quicker and easier to build, maintain and update and less expensive than the mobile apps, even if slower. The application server communicates with the data server to store the needed information. A more detailed description of the architectural design is given in the next section.

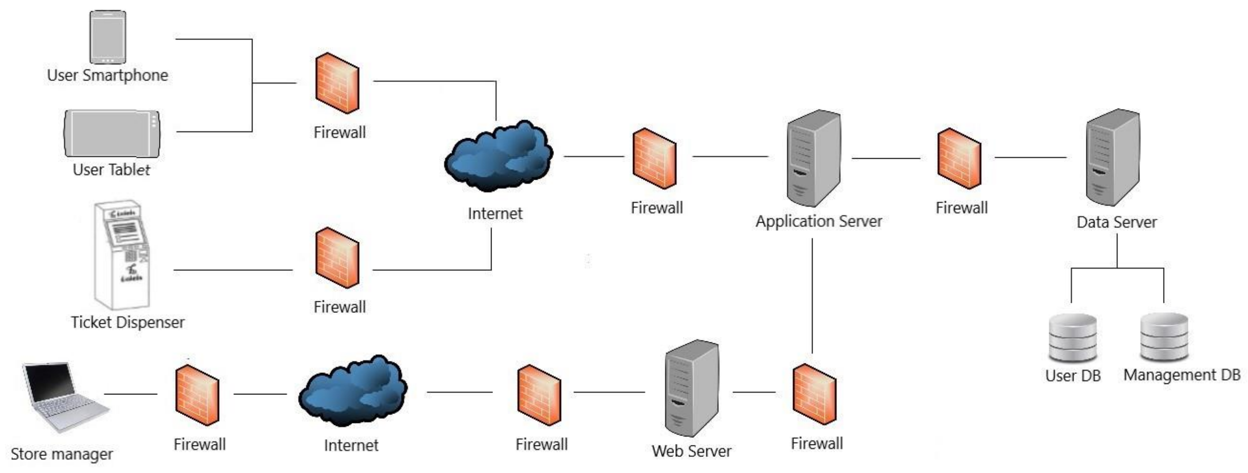
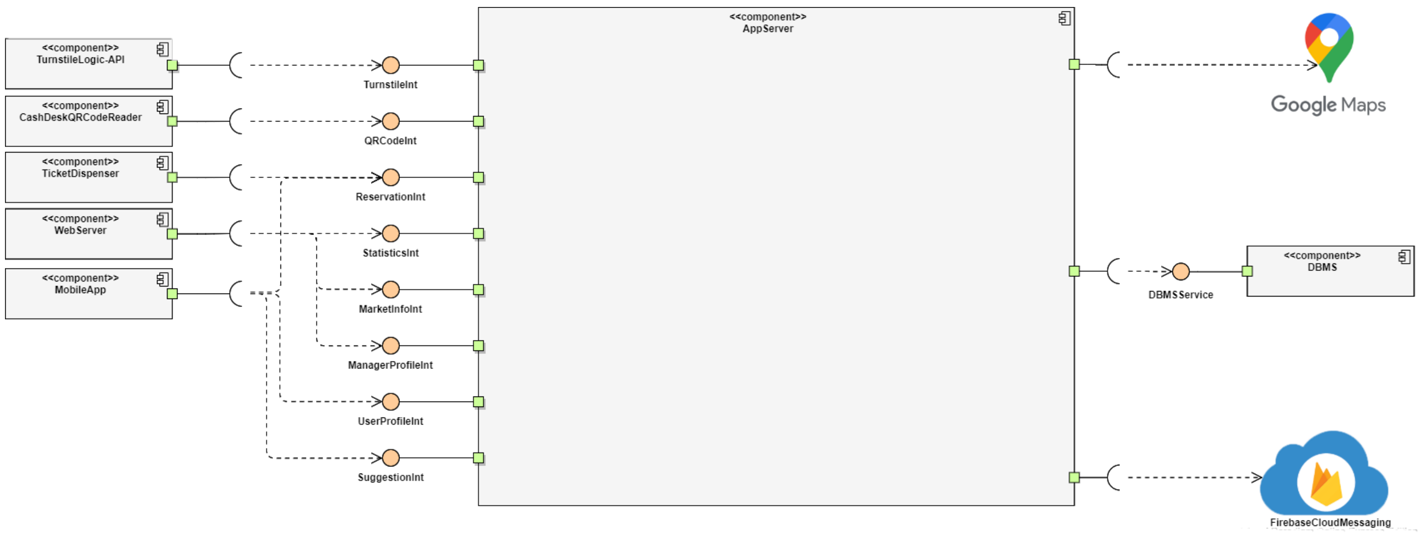


FIGURE 2.1 – HIGH LEVEL ARCHITECTURE

B) *Component view*

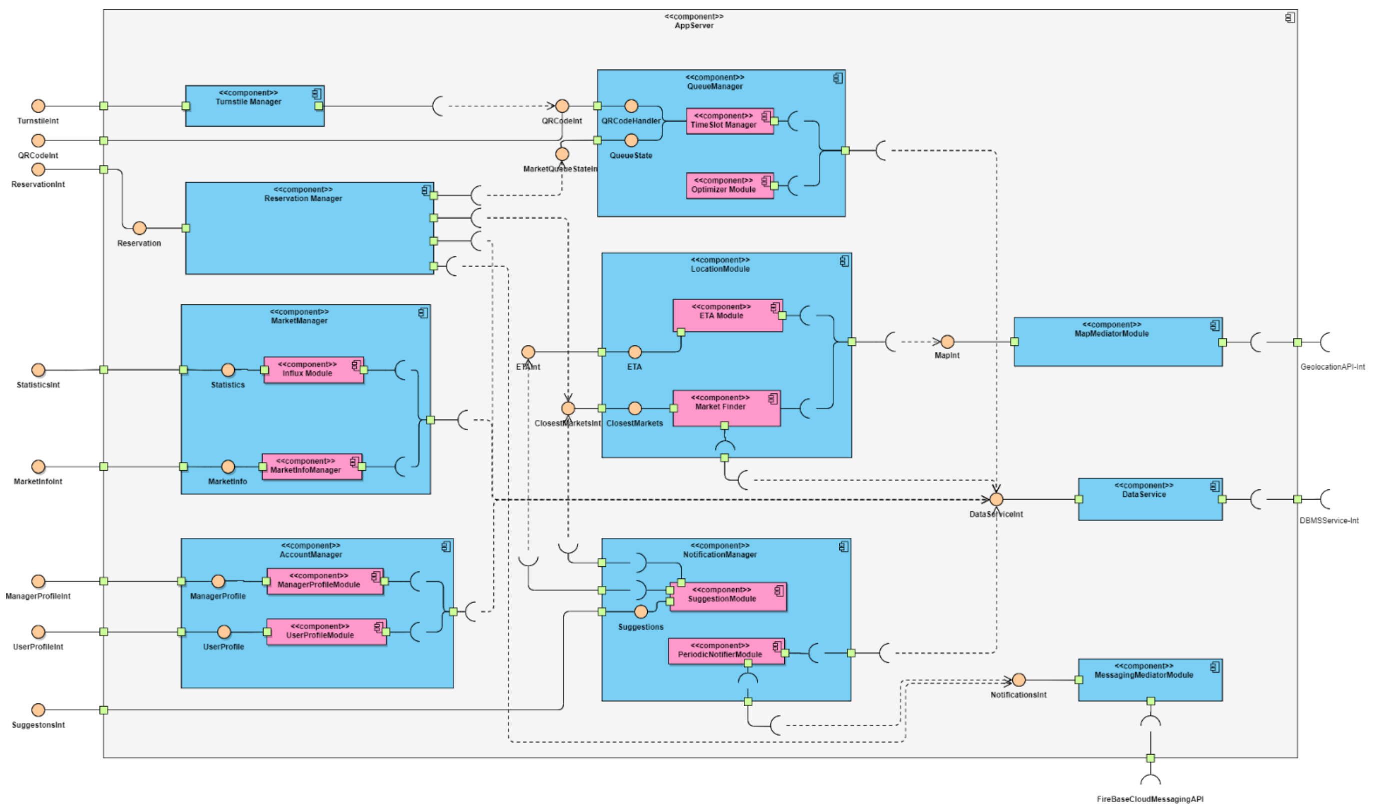
B.1) *General component view*



The above image represents the general component representation of the system. The application server architecture is not shown here (see next paragraph).

Here are shown the interfaces with the application server needed by the other components of the system: turnstiles have an interface to handle QR codes checking; the physical dispenser interacts with the application server as it was a normal device getting a ticket, but with "restricted" degrees of freedom (e.g. it cannot choose store, but it is linked to the one it is in front of, it cannot specify shopping list…); Web Server will interface with application server to get all information on statistics it needs to allow the correct usage of the web app; Mobile App, having a thin client, will interface with the system to do basically everything it needs: account and ticket managing, subscribing to notifications, getting new tickets and visits. It is also shown that the application server communicates with external DBMS to maintain permanent data and with a geolocation service: Google Maps, that must provide services allowing retrieving distances between locations (both in terms of space and time) and retrieving info on nearby places given a type of place (in our case, shops). Google, under the Maps category in its API library, offers interesting APIs for our case: "Places API" allows lookups for specific types of places (again, in our case, shops), with useful parameters as the radius of search, while "Distance Matrix API" provides travel distance and time from locations to locations. Furthermore, Firebase Cloud Messaging is used to provide push notification services (see details in section H-additional specifications).

B.2) *Application Server component view*



<<component>> AppServer
<<component>> Turnstile Manager
<<component>> QueueManager
QRCodeHandler
<<component>> Time Slot Manager
QueueState
<<component>> Optimizer Module
TurnstileInt
QRCodeInt
ReservationInt
QRCodeInt
MarketQueueStateInt
Reservation
<<component>> Reservation Manager
<<component>> LocationModule
<<component>> ETA Module
ETAInt
ETA
<<component>> Market Finder
ClosestMarketsInt
ClosestMarkets
MapInt
<<component>> MapMediatorModule
GeolocationAPI-Int
<<component>> MarketManager
<<component>> Influx Module
Statistics
StatisticsInt
<<component>> MarketInfoManager
MarketInfo
MarketInfoInt
DataServiceInt
<<component>> Data Service
DBMSService-Int
<<component>> AccountManager
<<component>> ManagerProfileModule
ManagerProfile
ManagerProfileInt
<<component>> UserProfileModule
UserProfile
UserProfileInt
<<component>> NotificationManager
<<component>> SuggestionModule
Suggestions
<<component>> PeriodicNotifierModule
NotificationsInt
<<component>> MessagingMediatorModule
SuggestionsInt
FireBaseCloudMessagingAPI

The image above shows the internal architecture of the Application Server and all the interfaces that are provided to the outside world, and all interfaces required.
Starting from what are conceptually two mediators, the system will contain two components called:

- ***Map Mediator Module***
  This component's job is to communicate with the external API, by modifying the API's information so that it can be comprehensible by the Application server, and adapting the requests to the API's protocol, required by other modules of the server. It provides information regarding the local map with all the relevant stores in the map as well as an estimation of the time it takes to reach one store from a certain location.
- ***Turnstile Manager***
  The turnstile manager, as opposed to the previous component, provides an interface for the turnstiles so that they can be opened or closed through the scans of the right QR codes. It is meant to also manage the possible (and probable) different requests caused by different types or brands of turnstiles.

For the authentication of an account, be it user or store manager, the following component will be of help:

- ***Account Manager***
  This is the component which is responsible for the authentication of Store Managers and of any user that wants to log in or register, allowing also account modifications and general account managing. Furthermore, it provides to all users the possibility to fetch all of theirs tickets.

Once a user is logged in the system, the application server must provide functionalities to the user that allow him to book visits, get tickets, be reminded of upcoming visits and be suggested different stores in case the requested ones are full. The following components allow users to:

- ***Reservation Manager***
  It provides the interfaces that allow booking of tickets and visits. It communicates with different components to provide the functionalities required by tickets and visits requests i.e., check the relevant, available timeslots, gets the info that will allow the presentation layer to build map of the available shops around the user's current location, handle visits additional info…(See interface diagram for a more detailed list)

- ***Notification Manager***
  This component has the job of handling notifications of two types:
  - reminding users when it is time to leave their current location so to reach in time their destination; to do this, the component interfaces with the map provider, getting info on ETA. Push notifications are handled through Firebase Cloud Messaging.
  - provide suggestions of relevant available stores and timeslots when the requested one is full or the user is inactive.
- **Messaging Mediator Module**
  It is a mediator for the cloud messaging service , it handles the communication with FCM, providing services as subscription and push requests for notifications.

Instead, functionalities related to the store managers are provided by the following component:

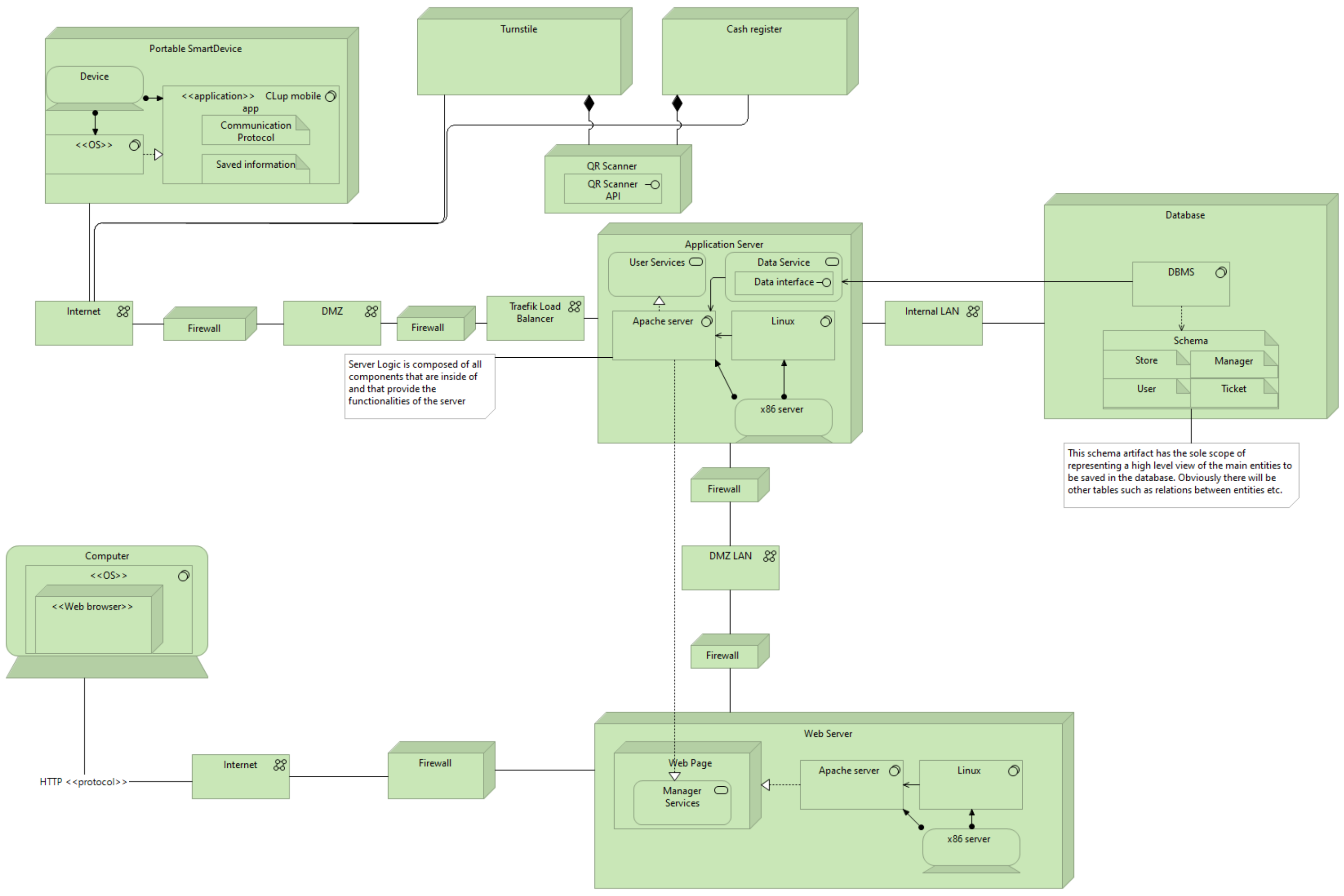
- ***Market Manager***
  Issues, through its interfaces, the functionalities of controlling how many customers can enter in the store and how many people are inside of said store. It also provides the statistics as average entrances per time range, average duration of shopping, most selected items in shopping lists (for visit only).

The components which connect everything together and provide the logic of the dispensing of tickets are the following:

- ***Queue Manager***
  This is the most critical and important component of the system, as it defines and handles timeslots and the queue. Through the timeslot manager subcomponent, timeslots are searched to see or update their availability on new tickets requests. It also generates QR codes for new tickets and visits, assigning them to available timeslots, and checks QR codes on scan, so to correctly update the state of the queue in that timeslot. The other component is a timeslot optimizer, that is a periodically activated component, which handles aspects of the optimization of timeslots, as the maximum number of people that can enter considering info about user visits that declared their shopping list or departments they will visit; another possible optimization is the duration of timeslots, performed once in a greater period of time with respect to the previous one, that could optimize the duration of timeslots based on the average shopping duration. To have a clearer understanding of how the timeslot system works, see the referenced section (Additional specifications).
- ***Location Module***
  It provides interfaces that allow the search of the closest stores in a range around a given location, filtering on the stores that are registered and use CLup; it also provides the ETA services, needed by other components to have an estimation of the user distance (quantified in time) from the given location. To provide these services, it interfaces with the Mediator, that will then process and forward requests to the external API.
- ***Data Service***

  It provides access to the external interface of the database, doing queries and generally interacting with the database for needed operations.

C) *Deployment view*

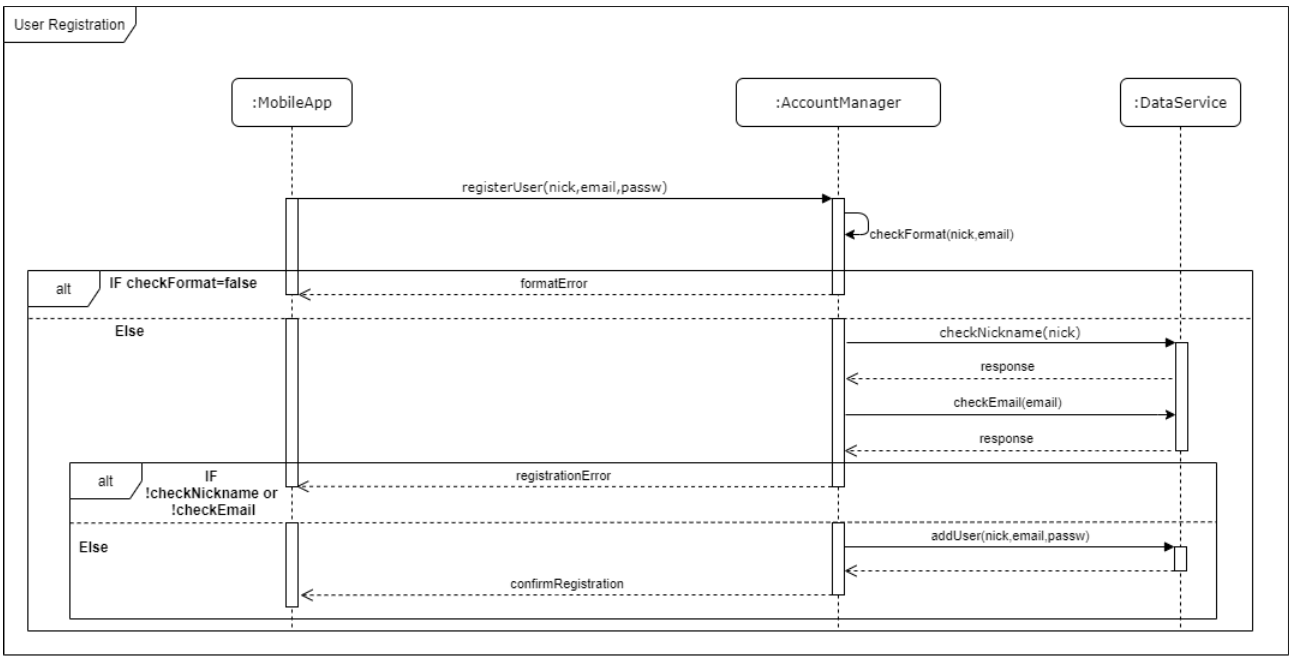


Some important aspects to highlight are the following:

- With Portable SmartDevice we mean any mobile device such as tablets and smartphones. With OS we intend either IOS or Android. In computer, any OS and Web browser can communicate with the web server as long as the HTTP protocol is being used.
- The CLup mobile app will store information in the client side such as upcoming tickets data, for faster data retrieval and user information for identification. It furthermore uses a specific communication protocol that the programmers might see fit.
- The installation of the CLup mobile app is done through the mobile phone's application store and is not been shown in the deployment view.
- The application server is connected to the database through an internal LAN with no firewalls, for increased connection speed.
- Turnstiles and cash registers need to forward information regarding QR codes through the internet.
- Traefik has been chosen as an open-source load balancer, referenced in the link in point 7) References.

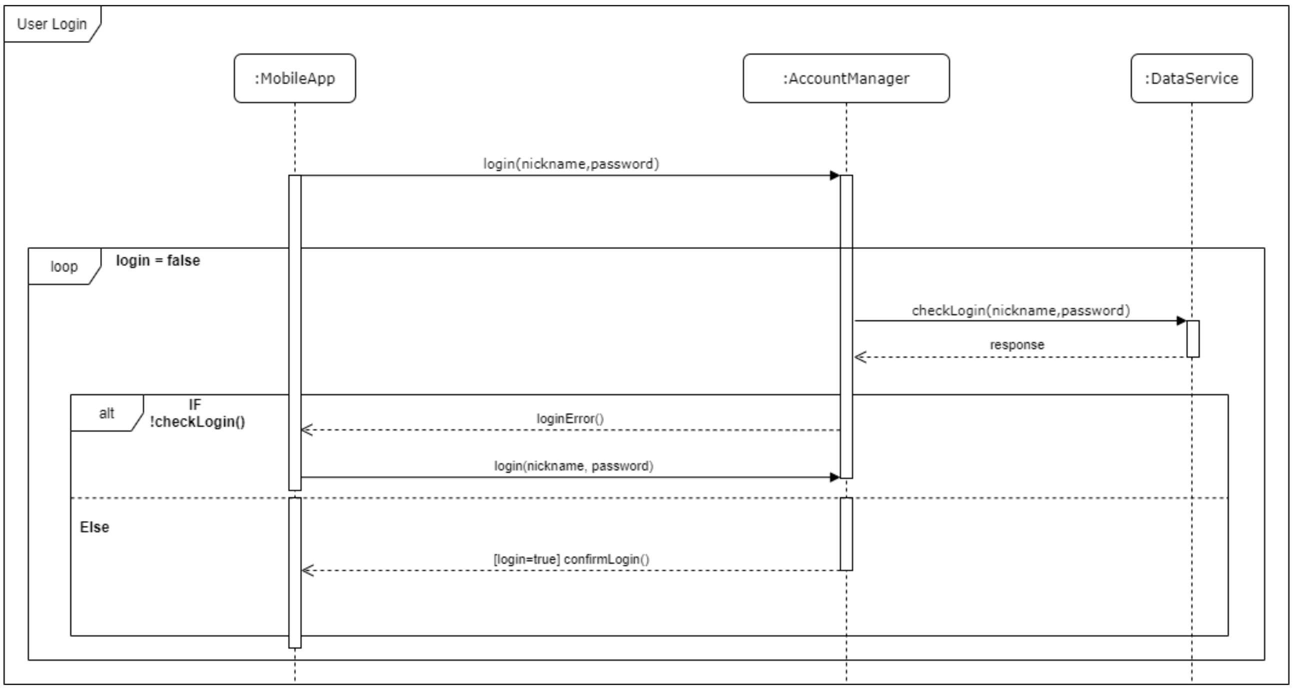
D) *Runtime view:*

D.1) User Registration



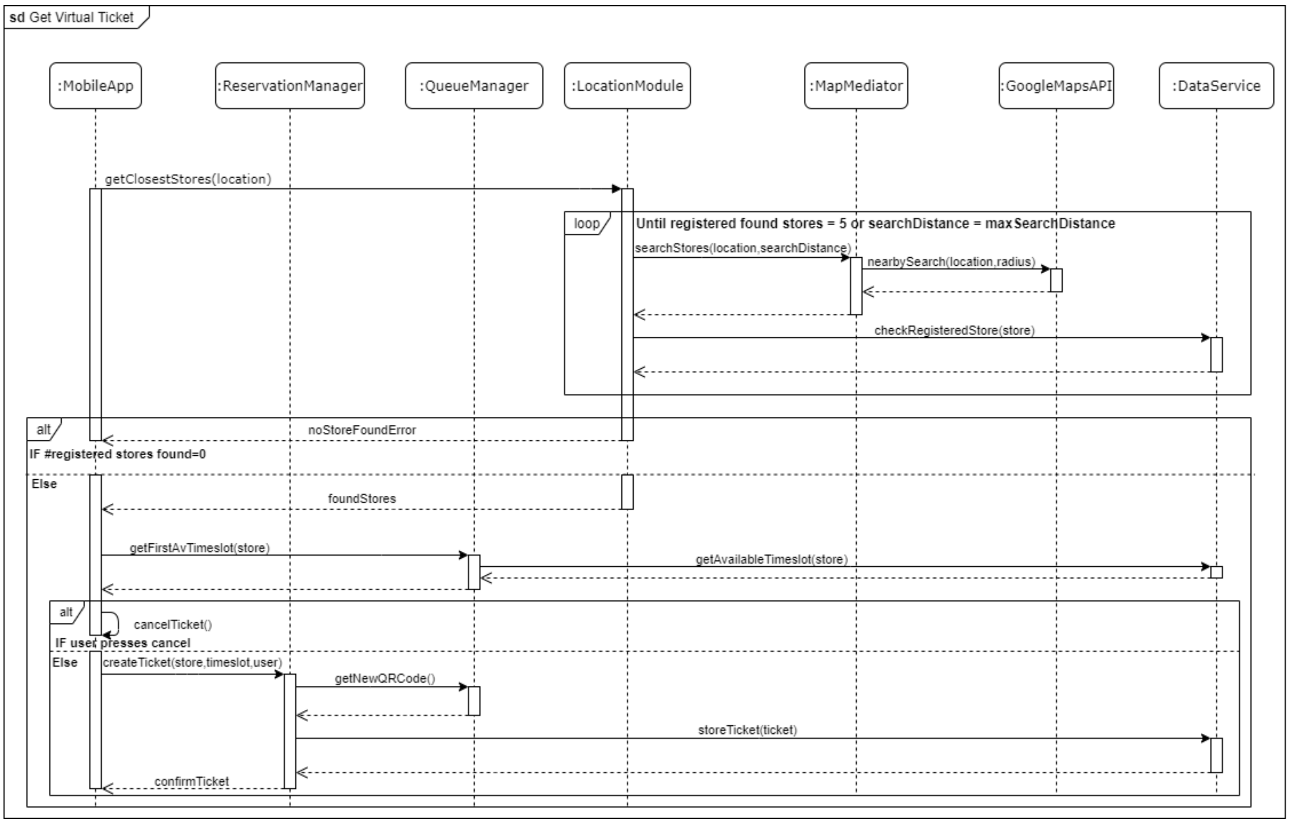
User registration is the usual, with checks on registration fields. User interactions have been omitted as trivial and superfluous to the understanding of the interaction.

D.2) User Login



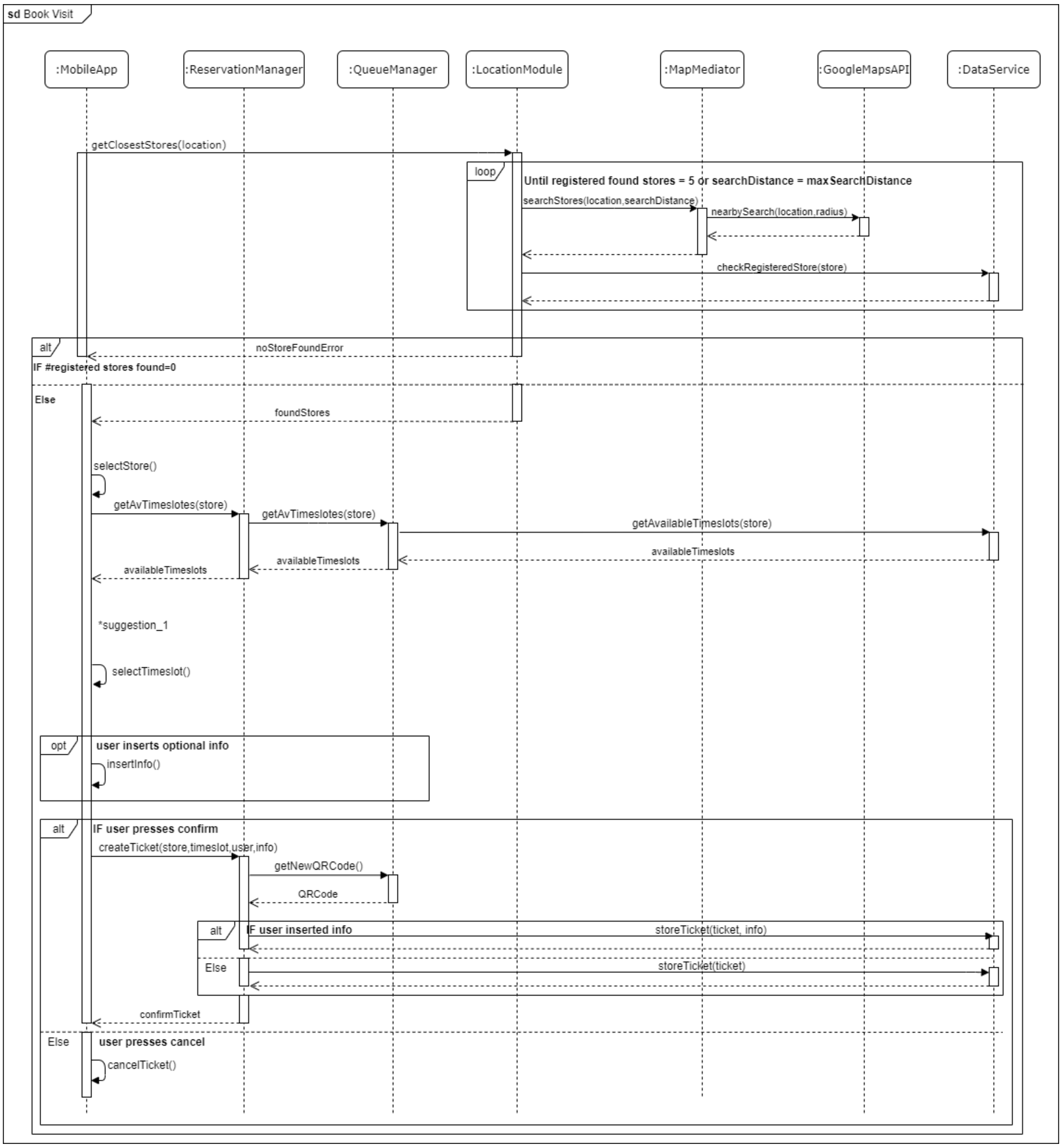
The user, whose representation has been considered superfluous, attempts to login in the application through the Mobile App and his access is granted only after a credential check through the Data Service component. If the credential check is not passed, then the application notifies the user, who can retry the procedure, that an error occurred due to either a username or password error.

D.3) Get Virtual Ticket


sd Get Virtual Ticket
:MobileApp
:ReservationManager
:QueueManager
:LocationModule
:MapMediator
:GoogleMapsAPI
:DataService
getClosestStores(location)
loop
Until registered found stores = 5 or searchDistance = maxSearchDistance
searchStores(location,searchDistance)
nearbySearch(location,radius)
checkRegisteredStore(store)
alt
noStoreFoundError
IF #registered stores found=0
Else
foundStores
getFirstAvTimeslot(store)
getAvailableTimeslot(store)
alt
cancelTicket()
IF user presses cancel
Else
createTicket(store,timeslot,user)
getNewQRCode()
storeTicket(ticket)
confirmTicket

This sequence diagram represents the interaction between components to search for an available shop to get a virtual ticket. As shown, the system handles cases where the user has no close markets by trying to increase the radius if no close markets or too few have been found, eventually stopping when a max distance of search has been reached. This allows to give some degree of choice to all users. The final ticket confirmation, as detailed in section H-Additional specifications, contains info that allows the MobileApp to build a scheduled notification, used then to remind the user of its ticket.

D.4) Book visit

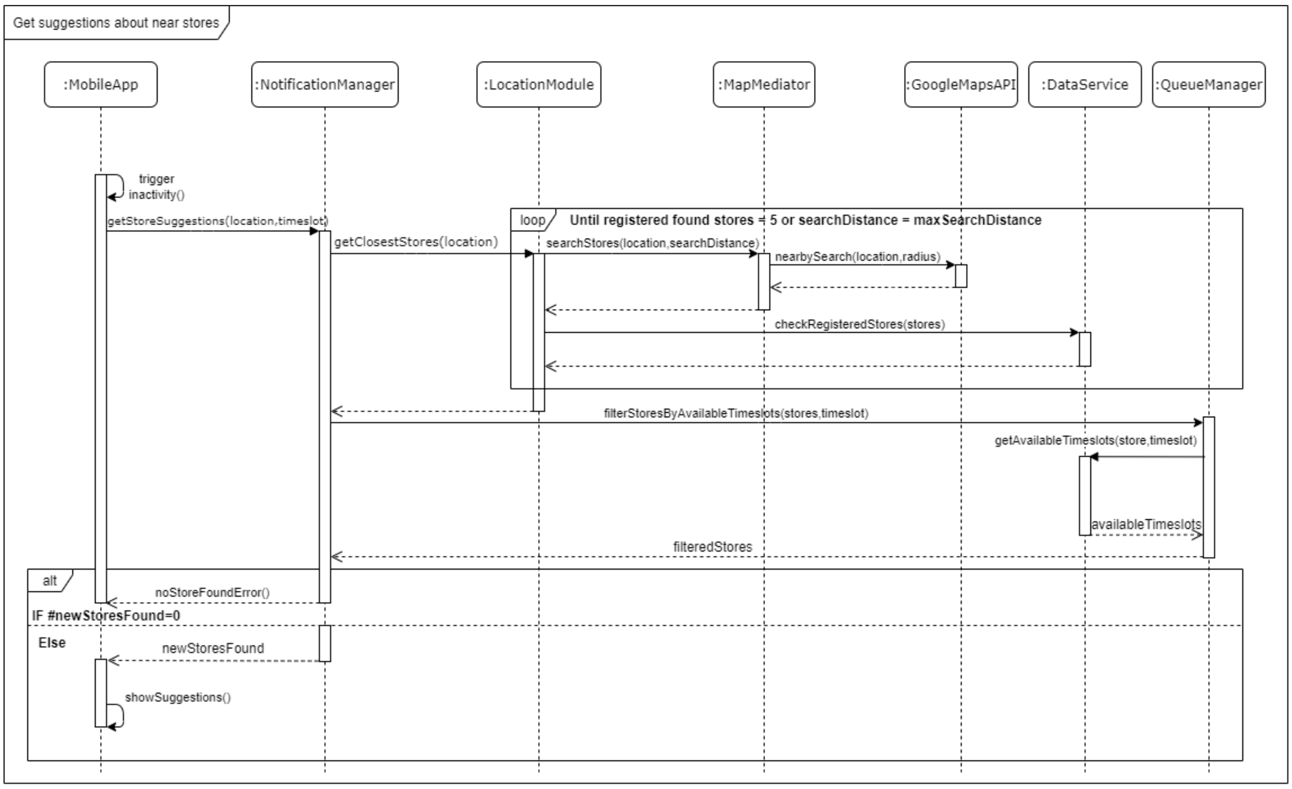


This sequence diagram represents the procedure to book a visit. The differences respect to the "Get Virtual Ticket" sequence diagram are:

- The user can either selects the desired timeslot or store as first option, while in getting a virtual ticket he can only selects a store, accepting or rejecting the first available timeslot of the chosen store. Note that in this case the user selects a store as first option and then a timeslot between those ones provided by the application. The case in which the user selects a timeslot as first option is symmetrical to this sequence diagram respect to the getStores/selectStore and getAvailableTimeslots/selectTimeslot parts. Moreover, in the second case, *suggestion_1 would be substituted *by suggestion_2,

- The user can, optionally, fill a form where he inserts the estimated duration of the shop trip, and either the specific items or the categories of goods he intends to purchase. Other than the new ticket, the additional information, if provided by the user, will be stored in the database through the Data Service Component.
- The *suggestion_1 and *suggestion_2 represent the cases in which the user is inactive for 15 seconds while selecting, respectively, a store and a timeslot (for more information see the sequence diagram below).

D.5) *Get suggestions about other available near stores (suggestion_1)*

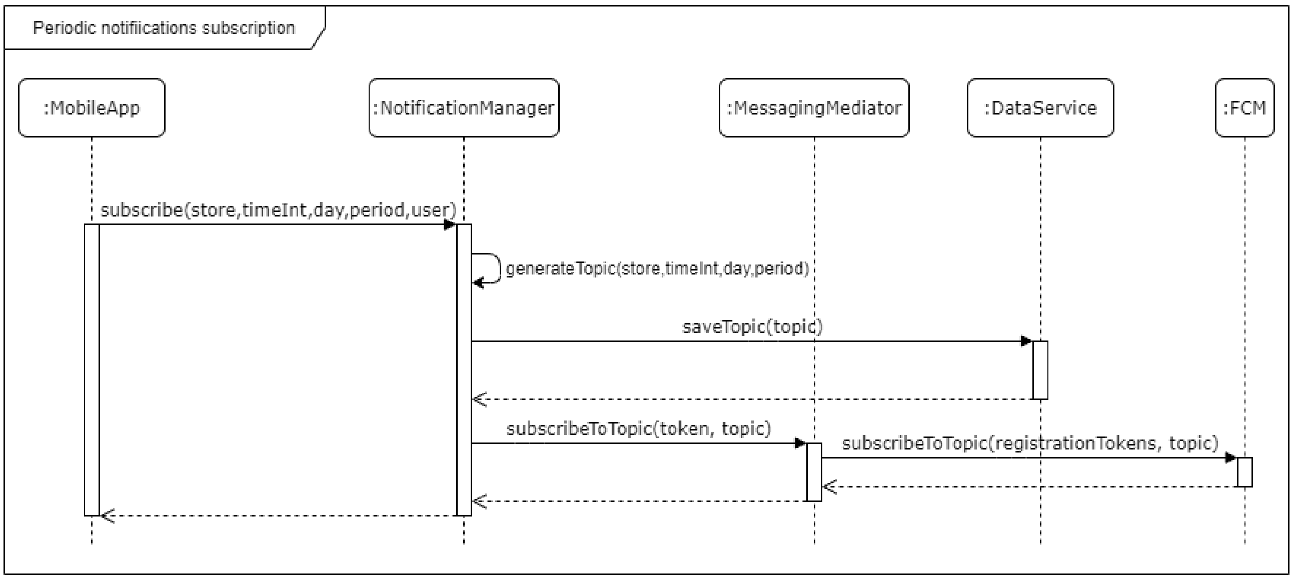

Get suggestions about near stores
:MobileApp
:NotificationManager
:LocationModule
:MapMediator
:GoogleMapsAPI
:DataService
:QueueManager
trigger inactivity()
getStoreSuggestions(location,timeslot)
loop
Until registered found stores = 5 or searchDistance = maxSearchDistance
getClosestStores(location)
searchStores(location,searchDistance)
nearbySearch(location,radius)
checkRegisteredStores(stores)
filterStoresByAvailableTimeslots(stores,timeslot)
getAvailableTimeslots(store,timeslot)
availableTimeslots
filteredStores
alt
noStoreFoundError()
IF #newStoresFound=0
Else
newStoresFound
showSuggestions()

This diagram shows the interactions providing the first type of suggestion given to inactive users during a ticket/visit booking. As shown, the system searches for further available stores near the selected one. After the suggestion is shown to user, if he selects on it, he will start browsing through the suggested stores, and the diagram is the same of the ticket standard booking from the point of selecting a store.

The "Get suggestions about available timeslots (suggestion_2)" sequence diagram is not represented to avoid redundancy. Indeed, the difference between these two are:

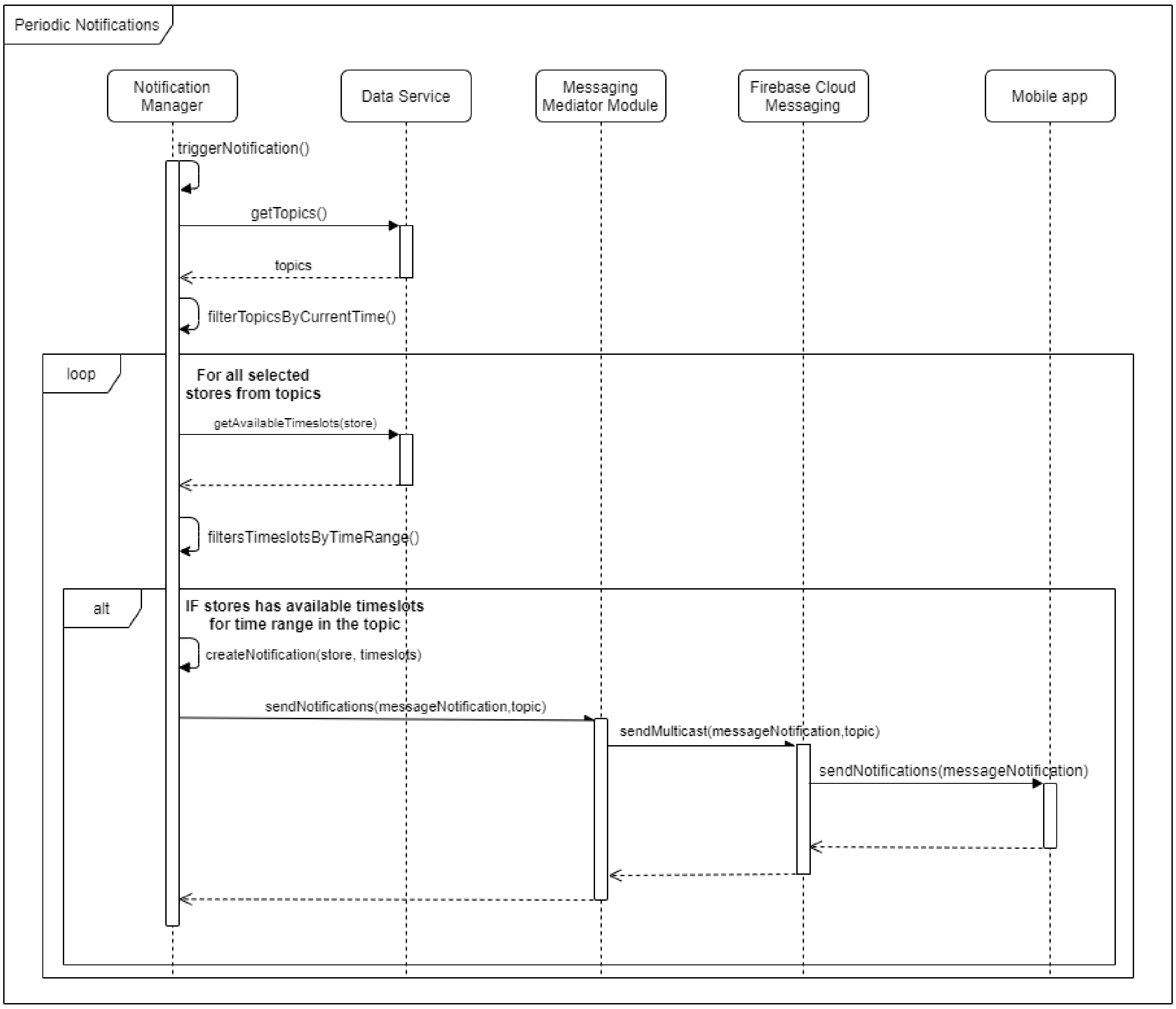
- After the user being inactive for 15 seconds, Suggestion_2 is triggered while the user is selecting a timeslot (Suggestion_1 is triggered while the user is selecting a store).
- Suggestion_2, unlike Suggestion_1, provides further stores near the user's location filtering them through their first available timeslot. Since the user has already selected a timeslot, it is likely that the user would want to select a store with an available timeslot close to the selected one.

D.6) Periodic notifications subscription


Periodic notifiications subscription
:MobileApp
:NotificationManager
:MessagingMediator
:DataService
:FCM
subscribe(store,timeInt,day,period,user)
generateTopic(store,timeInt,day,period)
saveTopic(topic)
subscribeToTopic(token, topic)
subscribeToTopic(registrationTokens, topic)

Whenever the user decides to activate notifications for a store, he can select a time interval, a weekday to be notified about, and the period and weekday of the notifications. Once received the request, the notification manager must create the subscription, that will translate the requested subscription to a topic. If the topic is not already present, it will be added to the database. Then the user will be subscribed to the generated topic through Firebase Cloud Messaging (mediated by Messaging Mediator Module), that offers the handling of topics; as for the database, if the topic is already present, Firebase will add the user to the subscription list of that topic, else it will create the new topic and subscribe to it the user. For more information about the generation of topics, see Paragraph H.2 – Generation of topic.

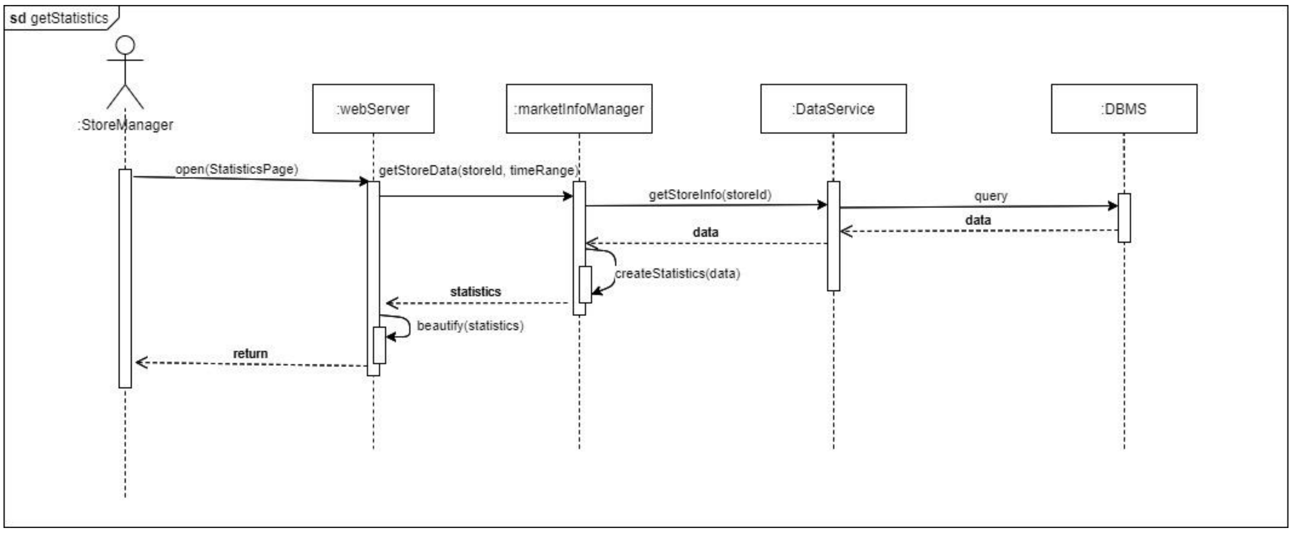
D.7) Periodic Notifications



Periodic Notifications
Notification Manager
Data Service
Messaging Mediator Module
Firebase Cloud Messaging
Mobile app
triggerNotification()
getTopics()
topics
filterTopicsByCurrentTime()
loop
For all selected stores from topics
getAvailableTimeslots(store)
filtersTimeslotsByTimeRange()
alt
IF stores has available timeslots for time range in the topic
createNotification(store, timeslots)
sendNotifications(messageNotification,topic)
sendMulticast(messageNotification,topic)
sendNotifications(messageNotification)

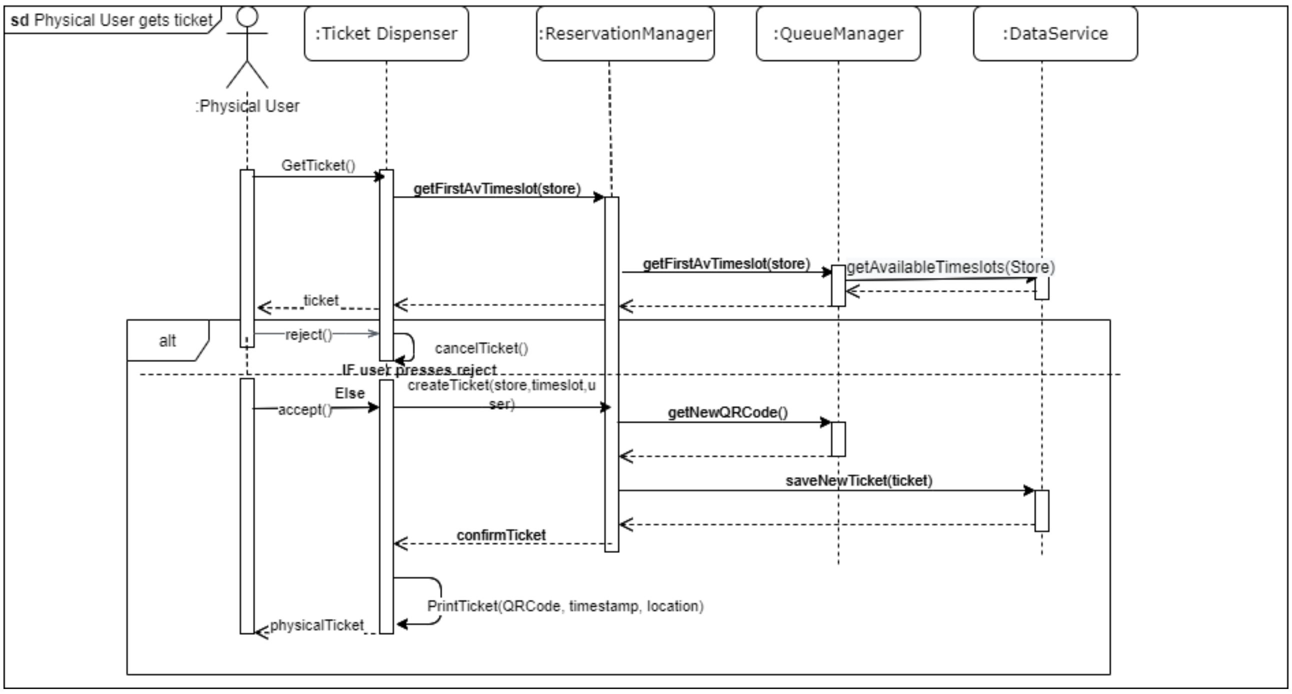
This sequence represents how notifications are sent to the users thought Firebase Cloud Messaging. Periodically, a trigger activates the Notification Manager to carry out the notifications sending. The Notification Managers first retrieves all the topics from the database through the Data Service, then filters them though the weekday and the notification period indicated into the topic string (for more information see Paragraph H.2 – Generation of topic). Once the topic are filtered, the Notification Manager retrieves the available timeslots of the stores included in the topics and filters them though the time range period, included into the topic string too. Eventually, if there are available timeslots after the filtering, Notification Manager through the Messaging Mediator Module informs Firebase Cloud Messaging to send notifications to the users subscribed to the topic.

D.8) Store Manager statistics



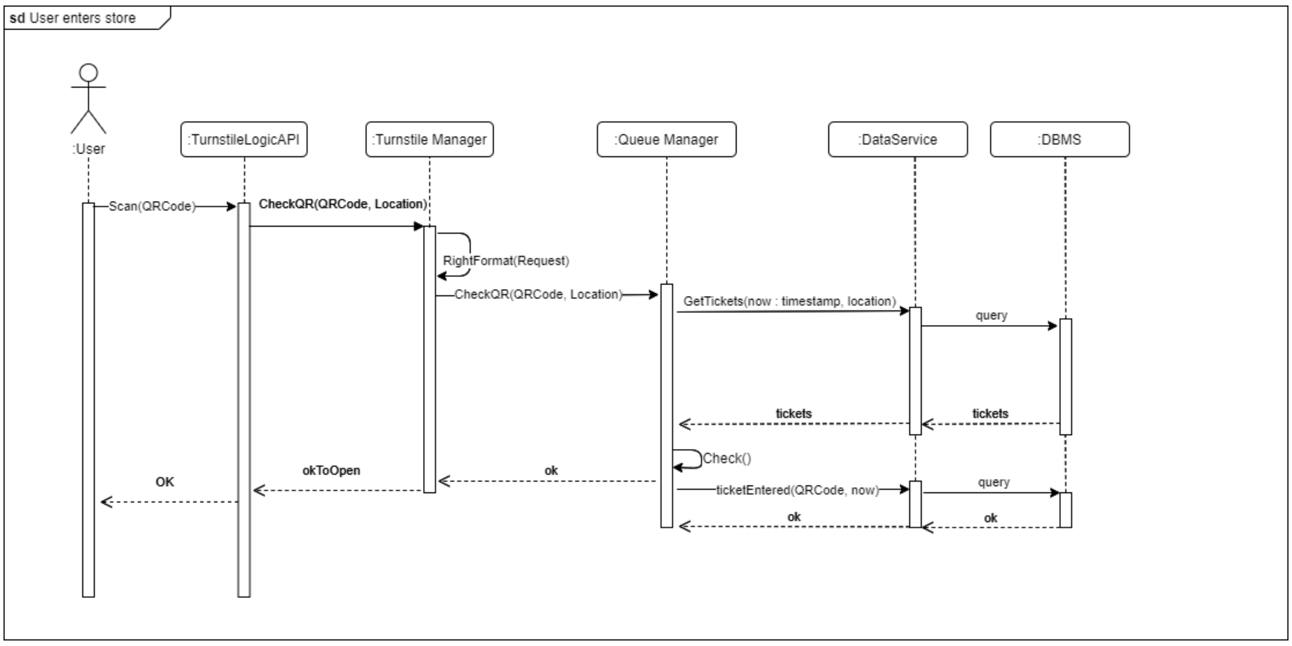
Once the store manager, through his *browser* opens the statistics page, the web server requests all the data regarding the store's entrances. Instead of requesting all statistics to be created on the DBMS, the Data Service component retrieves only the data regarding that store and the influx module creates all the statistics internally. This so we can ease the computational workload of the DBMS and increase it for the Influx Module, a component which is going to be used way less in comparison.

D.9) *Physical ticket acquisition*



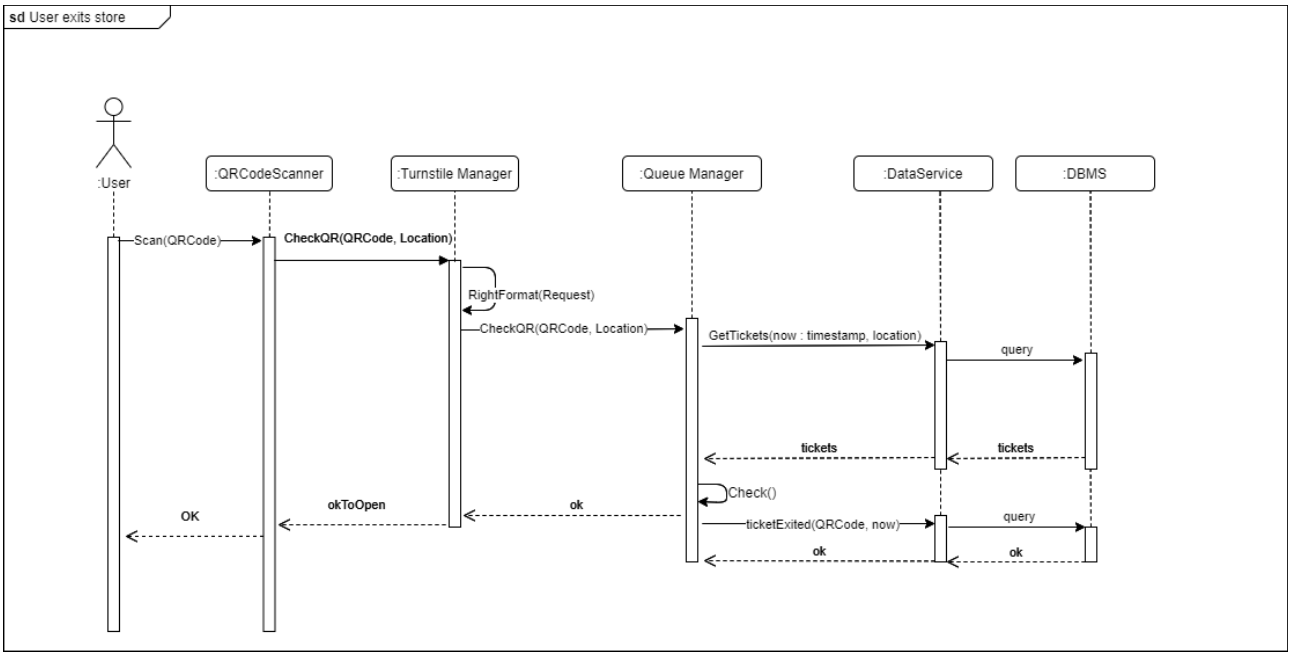
The user who goes physically to the store, requests a ticket and the ticket's dispenser internal logic communicates with the system, asking the first available time slot of that one store. Just like the virtual ticket, the execution of the methods is the same.

D.10) User enters store



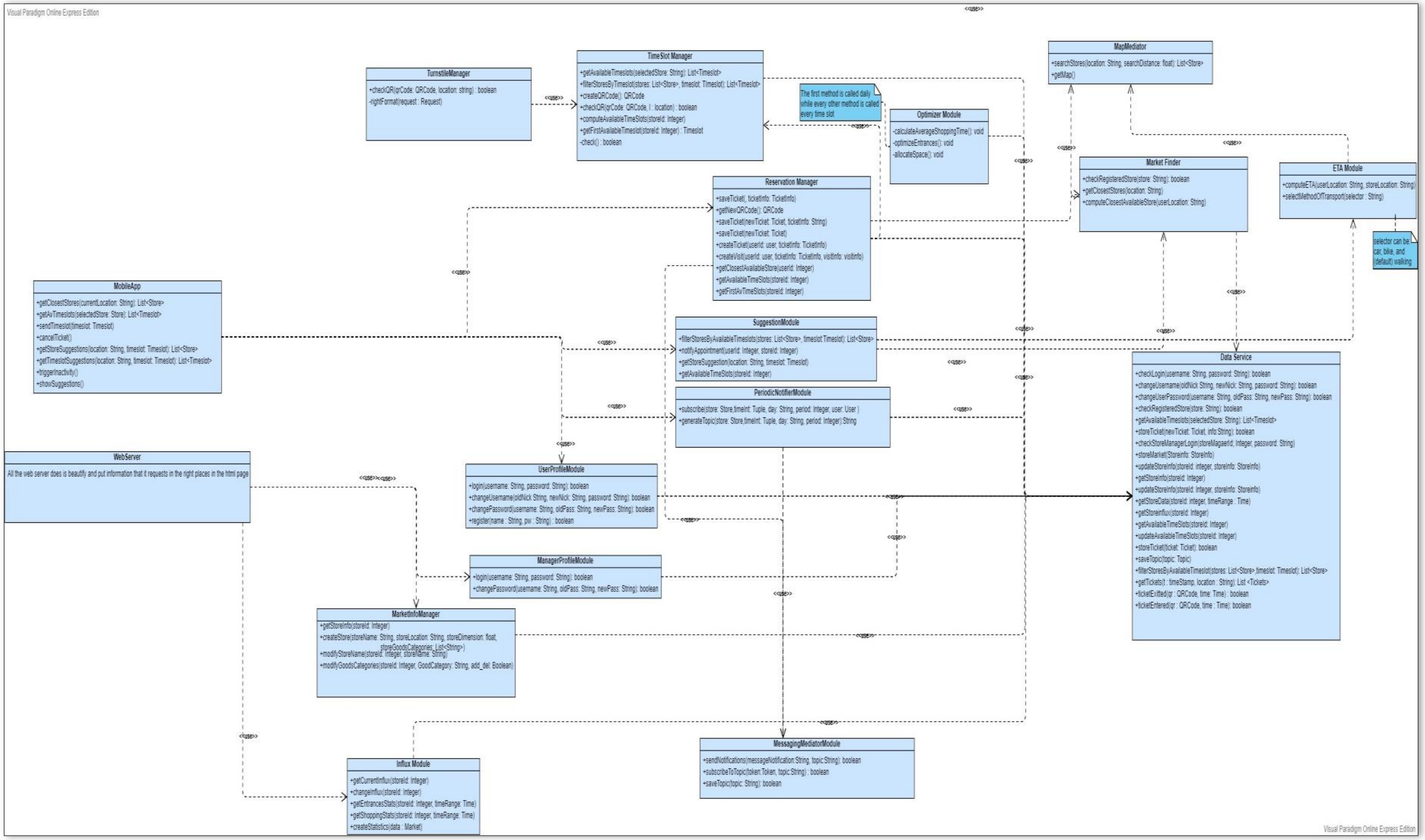
A user that initially scans the code activates the turnstile's internal logic which will invoke the method that checks if the scanned QR code is correct with respect to the location of the turnstile. The mediator turnstile manager will translate the request so that it is compatible with the internal server protocol. After the check has been completed, the queue manager will save the data regarding the ticket that has just been scanned as a valid entrance, without waiting for a confirmation from the turnstile logic API that the user has actually pushed the turning mechanism. Instead in the case the scanned QR is early, late or in the wrong store, there is no data to be saved and our system will send a NOK response that will not allow to unlock the turnstile.

D.11) User exits store



The only difference from the "User enters store" sequence diagram is that in this case, there might be a QR code scanner inside the cash register that communicates with the system's mediator. The scan can be done by the cashier or by the system's user through a default turnstile where all users who have not bought any products go to.

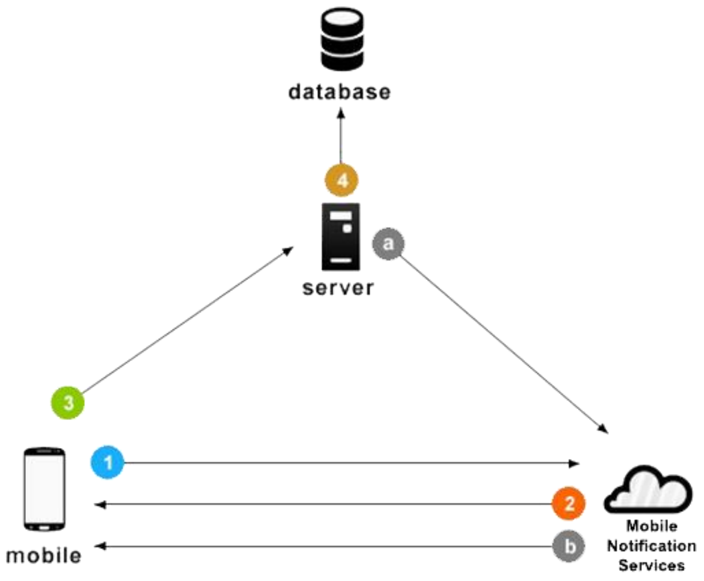
E) Component interfaces


Visual Paradigm Online Express Edition
TurnstileManager
+checkQR(qrCode: QRCode, location: string) : boolean
-rightFormat(request : Request)
TimeSlot Manager
+getAvailableTimeslots(selectedStore: String): List<Timeslot>
+filterStoresByTimeslot(stores: List<Store>, timeslot: Timeslot): List<Timeslot>
+createQRCode(): QRCode
+checkQR(qrCode: QRCode, l : location) : boolean
+computeAvailableTimeSlots(storeId: Integer)
+getFirstAvailableTimeslot(storeId: Integer) : Timeslot
-check() : boolean
The first method is called daily while every other method is called every time slot
Optimizer Module
-calculateAverageShoppingTime(): void
-optimizeEntrances(): void
-allocateSpace(): void
MapMediator
+searchStores(location: String, searchDistance: float): List<Store>
+getMap()
Market Finder
+checkRegisteredStore(store: String): boolean
+getClosestStores(location: String)
+computeClosestAvailableStore(userLocation: String)
ETA Module
+computeETA(userLocation: String, storeLocation: String)
+selectMethodOfTransport(selector : String)
selector can be: car, bike, and (default) walking
Reservation Manager
+saveTicket(, ticketInfo: TicketInfo)
+getNewQRCode(): QRCode
+saveTicket(newTicket: Ticket, ticketInfo: String)
+saveTicket(newTicket: Ticket)
+createTicket(userId: user, ticketInfo: TicketInfo)
+createVisit(userId: user, ticketInfo: TicketInfo, visitInfo: visitinfo)
+getClosestAvailableStore(userId: Integer)
+getAvailableTimeSlots(storeId: Integer)
+getFirstAvTimeSlots(storeId: Integer)
MobileApp
+getClosestStores(currentLocation: String): List<Store>
+getAvTimeslots(selectedStore: Store): List<Timeslot>
+sendTimeslot(timeslot: Timeslot)
+cancelTicket()
+getStoreSuggestions(location: String, timeslot: Timeslot): List<Store>
+getTimeslotSuggestions(location: String, timeslot: Timeslot): List<Timeslot>
+triggerInactivity()
+showSuggestions()
SuggestionModule
+filterStoresByAvailableTimeslots(stores: List<Store>, timeslot:Timeslot): List<Store>
+notifyAppointment(userId: Integer, storeId: Integer)
+getStoreSuggestion(location: String, timeslot: Timeslot)
+getAvailableTimeSlots(storeId: Integer)
PeriodicNotifierModule
+subscribe(store: Store,timeInt: Tuple, day: String, period: Integer, user: User)
+generateTopic(store: Store,timeInt: Tuple, day: String, period: Integer):String
Data Service
+checkLogin(username: String, password: String): boolean
+changeUsername(oldNick String, newNick: String, password: String): boolean
+changeUserPassword(username: String, oldPass: String, newPass: String): boolean
+checkRegisteredStore(store: String): boolean
+getAvailableTimeslots(selectedStore: String): List<Timeslot>
+storeTicket(newTicket: Ticket, info:String): boolean
+checkStoreManagerLogin(storeMagaerId; Integer, password: String)
+storeMarket(Storeinfo: StoreInfo)
+updateStoreInfo(storeId: integer, storeInfo: StoreInfo)
+getStoreInfo(storeId: Integer)
+updateStoreInfo(storeId: Integer, storeInfo: Storeinfo)
+getStoreData(storeId: integer, timeRange : Time)
+getStoreinflux(storeId: Integer)
+getAvailableTimeSlots(storeId: Integer)
+updateAvailableTimeSlots(storeId: Integer)
+storeTicket(ticket: Ticket): boolean
+saveTopic(topic: Topic)
+filterStoresByAvailableTimeslot(stores: List<Store>,timeslot: Timeslot): List<Store>
+getTickets(t : timeStamp, location : String): List <Tickets>
+ticketExitted(qr : QRCode, time: Time) : boolean
+ticketEntered(qr : QRCode, time : Time): boolean
WebServer
All the web server does is beautify and put information that it requests in the right places in the html page
UserProfileModule
+login(username: String, password: String): boolean
+changeUsername(oldNick String, newNick: String, password: String): boolean
+changePassword(username: String, oldPass: String, newPass: String): boolean
+register(name : String, pw : String) : boolean
ManagerProfileModule
+login(username: String, password: String): boolean
+changePassword(username: String, oldPass: String, newPass: String): boolean
MarketInfoManager
+getStoreInfo(storeId: Integer)
+createStore(storeName: String, storeLocation: String, storeDimension: float, storeGoodsCategories: List<String>)
+modifyStoreName(storeId: Integer, storeName: String)
+modifyGoodsCategories(storeId: Integer, GoodCategory: String, add_del: Boolean)
Influx Module
+getCurrentInflux(storeId: Integer)
+changeInflux(storeId: Integer)
+getEntrancesStats(storeId: Integer, timeRange: Time)
+getShoppingStats(storeId: Integer, timeRange: Time)
+createStatistics(data : Market)
MessagingMediatorModule
+sendNotifications(messageNotification:String, topic:String): boolean
+subscribeToTopic(token:Token, topic:String) : boolean
+saveTopic(topic: String): boolean
<<use>>

F) *Selected architectural styles and patterns*:

Notification Push Pattern

To provide the required periodic notifications functionalities, a push pattern has been chosen, so to have more efficiency and scalability. This decision is mainly influenced by the expected high number of users using the system: with a pull approach, the server would risk getting flooded by periodic notifications requests. The general approach with this pattern has 3 layers:
- **Mobile notification services** (MNS) are components that allow third-party app developers to send information from their servers to the user's devices
- **Application Server** (push Server) is the server creating push requests handled by MNS, targeting directly specific users, or group of users, based e.g., on a topic they have subscribed to.
- **Client Application** registers to the notification service, getting uniquely identified. The client will then keep a background connection, nor data or power expensive, that will be used to receive notifications by MNS.
Upon server push requests, the MNS uses the connection with the user to send the notifications.



1) device registers to MNS
2) Upon successful registration, MNS server issues registration token to client device.
3) Client device sends registration token to app server
4) App server stores client registration token for future notification uses.
a) Server sends push notification request to MNS identifying the device(s).
b) MNS Server sends notification to device(s).

The decision is to use Google's FCM, that provides a cross platform (Android, IOS) full handling of push notifications (sending, routing, queuing). It is also implementing subscriptions to specific topics, with no limits on the number of topics nor subscriptions to a topic (important given our system expected clients and variety of topics, that can be each timeslot or store or a combination of both).

Mediator Pattern

Mediator helps in establishing loosely coupled communication between objects and helps in reducing the direct references to each other. This helps in minimizing the complexity of dependency management and communications among participating objects. It also facilitates the interaction between objects in a manner in that objects are not aware of the existence of other objects.

In our system, the mediator pattern is used through the Map Mediator Module and the Messaging Mediator Module. The former enables the communication between the Location Module and the Google Maps service, while the latter enables the communication between the Notification Manager and Firebase Cloud Messaging service. In the case of a change of either the geolocation Service or the push notification service to use, the decoupling provided by the mediator will allow us to modify just the Map Mediator Module or the Messaging Mediator Module, without making any modification on the Location Module or Notification Manager, respectively.

G) *Other design decisions*

As by RASD specifications, we need to provide an availability of 99.9% and a response time of 0.5 seconds or less. For that reason, we need to replicate the application server, and introduce a load balancer which will allow to achieve the required goal. The web server might be replicated too, but it is less critical because the number of users is significantly lower than that of the application server.

Data regarding stores and upcoming tickets need to be replicated for security reasons and so that if one of the databases malfunctions critical data do not get lost and response time is higher through load reduction of each database.

H) *Additional Specifications*

H.1) *Timeslots method*

We have decided to manage queues in the following way:

- Each timeslot, called τ, is divided in fractions of an hour (1/4 hour by default) in which only a fraction of the maximum capacity of people allowed inside the store*, called m, can enter. In this way, we can divide entrances into smaller groups making queues thinner.
  - The timeslot is divided in two parts, one being the entrance time called ε (10 minutes by default, 1/6 hour) and the other will be the preparation time called ρ (5 minutes by default, 1/12 hour, or equivalently $\rho = \tau - \varepsilon$).
    - The entrance time allows people to enter in a 10-minute period time which, following RASD specifications, considers the human error.
    - The preparation time instead, considers all people of the successive timeslot who might come in front of the store 5 minutes earlier than their timeslot, for personal reasons (such as wanting to be first to enter or because they might be worried, they might not arrive in time). This makes most likely that people in two different timeslots will not get to see one another in a queue.
- Furthermore, an important variable to define is the number of timeslots it takes for a grocery shopping trip**, called σ. If the average trip lasts 59 minutes, then σ is 4 ($= \lceil 59\text{min} / 15\text{min} \rceil$)
  - The general formula then is:
    - $\sigma = \lceil \text{critical\_shopping\_time} / \tau \rceil$
  - In the general case, for markets which we have no information on, we can consider the average shopping time as the time defined in the study reported by the American journal (*Pew Research Center*, references).

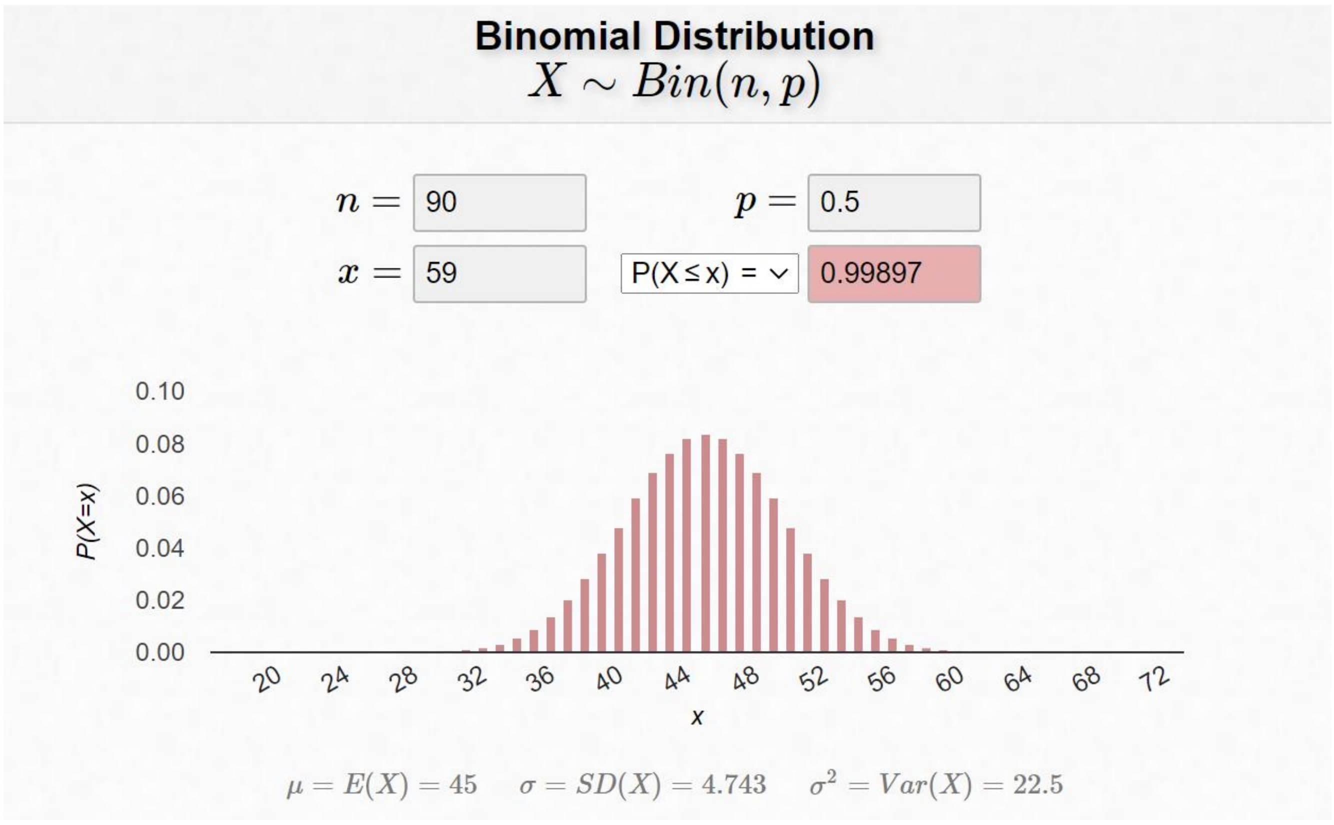
*this number is defined by law, in Italy as 13m^2 of free space per person.
**this number is taken by statistics. average grocery shopping trip is considered the time it takes a client (user) to enter and exit the store through our system.

- Lastly, we introduce a δ variable to take into account all entrances that might last longer than our considered average shopping trip.
  - This is going to be a fraction of the average shopping time.

- In the general case, we have found a binomial distribution to be a good representation of the shopper's time habit. With this reasoning we can define δ as the 99.9 highest percentile of cases in which a customer shops and stays no longer than expected.
- The average time of shopping mentioned above (59) represents the time (in minutes) for which P(X <= x) ≈ δ = 99.9%. The critical shopping time is defined as
  - γ = E(Shopping Time) + 3*SD(Shopping Time)

E(x) := Expected value of X

SD(x) := Standard deviation of x



- Finally, we can define the number of people that can enter inside the store per each timeslot π , that assures, each time, the maximum amount of people inside of it while keeping the social distancing as:
  - π = ⌈ m / σ ⌉

Understanding the following rules is crucial to the correct and optimized functioning of queues. We provide the following example:

To calculate the maximum number of people allowed for each timeslot we can use the following formula:

$$\pi = \lceil m / \sigma \rceil$$

The market "Ellelunga" can, by law, only host at most 100 people inside of it (1), with timeslots long 15 minutes (2) and critical shopping time lasting 56 minutes (3).

(1) $m = 100$ people
(2) $\tau = 15$ minutes / timeslot
(3) $\gamma = 56$ minutes

Having these 3 variables we can calculate the number of timeslots needed per shopping trip as:

$$\sigma = \lceil \gamma / \tau \rceil = \lceil 56 / 15 \rceil = 4 \text{ timeslots}$$

Lastly, we can find the maximum number of people allowed inside the store per timeslot simply as:

$$\pi = \lceil m / \sigma \rceil = \lceil 100 \text{ people} / 4 \text{ timeslots} \rceil = 25 \text{ people / timeslot.}$$

This formula allows people to enter the stores in convenient times, while keeping safety for the personnel and clients and allowing the shop to be filled completely with an interval of certainty equal to 99.9%. Since the occurrence of a client, who has not yet exited the store after their expected time of exit has passed, is expected to happen once every 1000 costumers (0.01%) and for a time no longer than 5-10 minutes, it can be neglected.

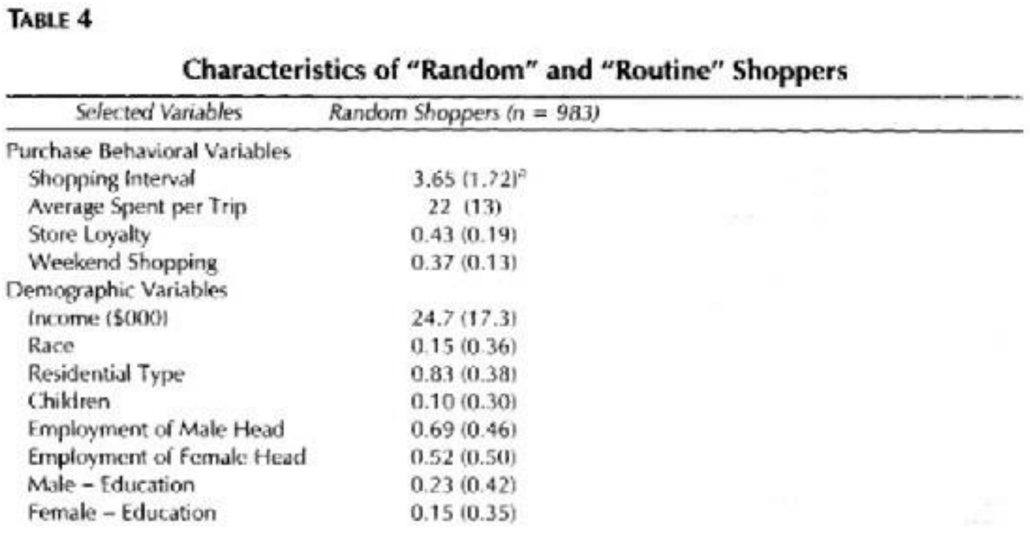


TABLE 4

Characteristics of "Random" and "Routine" Shoppers

Selected Variables	Random Shoppers (n = 983)
Purchase Behavioral Variables	
Shopping Interval	3.65 (1.72)[a]
Average Spent per Trip	22 (13)
Store Loyalty	0.43 (0.19)
Weekend Shopping	0.37 (0.13)
Demographic Variables	
Income ($000)	24.7 (17.3)
Race	0.15 (0.36)
Residential Type	0.83 (0.38)
Children	0.10 (0.30)
Employment of Male Head	0.69 (0.46)
Employment of Female Head	0.52 (0.50)
Male – Education	0.23 (0.42)
Female – Education	0.15 (0.35)

FIGURE 1 IN PARENTHESIS IS STANDARD DEVIATION WHICH WE DO NOT CONSIDER. REFERENCE IN THE BOTTOM OF THE DOCUMENT. (AUTHORS: BYUNG-DO KIM, KYUNGDO PARK)

In more common cases, when users leave before the critical shopping time, π might increase its value for successive time slots up to (by default) k = 25%. This coefficient needs to be calibrated through successive testing and in-field studies. The input of the users is important because it allows the system to plan finer shopping trips and calculate their shopping time independently of other clients who have not expressed any details and so their critical shopping time would be lower in number of timeslots. In this case we would have a critical shopping time for random shoppers equal to

$\gamma = 22 + 3 * 2.345$ minutes ≈ 27 minutes or equivalently $\sigma = 2$ timeslots. This allows the system to allocate future spaces to other users who might be interested in visiting the store.

It is important that the system calculates the average time it takes to shop for groceries, for each shop and for each category of foods in that shop so it can allocate time more efficiently in the basis of change of customer behavior. This is the job of the component called optimizer module which will periodically wake up during ρ preparation time and control if the number of entrances is indeed equal or less to what was expected.

In the end, the variables of timeslots τ, might be changed if the future requires it. In this current moment we found no benefits in changing the timeslots period, even though further study might show the contrary to be true.

H.2) *Generation of topics*

Here is explained how topics that handle users' subscriptions to store notifications are generated and handled to send notifications.

But first, subscription request has:

-StoreId: Id of the store to be notified about
-Selected Time interval: specified as a couple of starting hour-end hour, meaning the interval of time the user is interested in.
-Selected weekday: the weekday to be notified about
-Period of notification: couple of two values: first is the weekday on which user would like to receive notifications, the second is the period, that can be a multiple of a week up to a month.

"Selected time intervals" could be empty, meaning that every hour in that day is admissible, same goes for "Selected weekdays".
Now, we want to create a topic, that is a String, that encapsulates all this information so to efficiently store and then retrieve the topics concerning a day.
A string representation of these parameters is easily implementable and using the same order of parameters and of elements inside parameters (intervals, weekdays and period can have multiple elements) leads to an efficient representation for topics, as the generated topics for the same parameters will be the same.
Here is reported a naïve approach to the said string generation:
Let us consider the request: Store1, <15,19>, <Monday>, <Saturday,2 Weeks>
It can be encoded into string s = (IdOf(Store1))+"M"+"S"+"2".

This makes sending the notifications more efficient, since topics, that are stored in the database, can be filtered based on their substrings representing the weekday they refer to and the day on which they should be fired out to users subscribed. For handling the period of notification (the one expressed in weeks) it is just needed a field in the tuple of the topic stored into the database, indicating the last day the notification was sent for that topic.
The notification manager, to send out each day the correct notification, must filter the topics of the current weekday as said before, checking also that the period to send the notification has passed since the last one sent, then for each retrieved topic it will check the availability of tickets in the topic specified store and time interval. In case of available tickets, it builds and sends the notification to subscribed users, sending the request for topic notification to Firebase Cloud Messaging. This procedure leaves a lot of space for code optimizations, and here it has just been drafted so to give the idea of how conceptually works.

H.3) Ticket reminders notifications

To lower the probability of users forgetting their tickets and visits reservations, the system manages reminder notifications, sent to user when its turn is about to come, with a reasonable advance. To implement this, when a ticket is confirmed, the info returned by the server allow the Mobile App to build a scheduled notification to be shown e.g., 1 hour before the reservation time. This has no impact on the performance of the mobile app and allows to reduce the computation that otherwise would be necessary on the Application Server to send, before each timeslot and for each store, notifications to users with tickets in that timeslots. Furthermore, if the user gives the mobile app the necessary permissions, it can, just storing minimal info about the next upcoming ticket, check periodically, in the few hours before the ticket's time, the estimated time from the current location to the store, showing a reminder notification when the current time plus the estimated time is close to the reservation time.

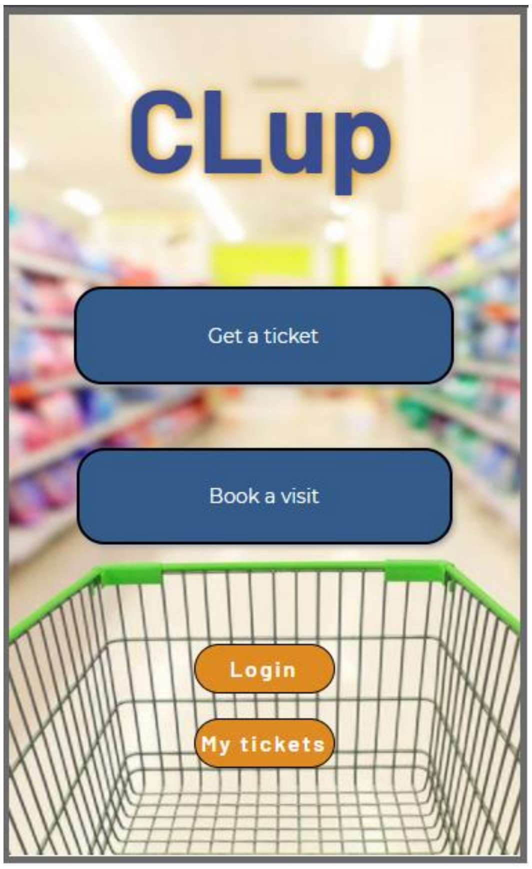
3) *USER INTERFACE DESIGN*:

Here we include some draft mockups about how the mobile applications and the web application should look like. As they are drafts, these are not to be intended as strictly constraining designs: backgrounds, styles, text formatting and page structure are simplified and rough, as we want just to give an idea of the general schema of the pages.

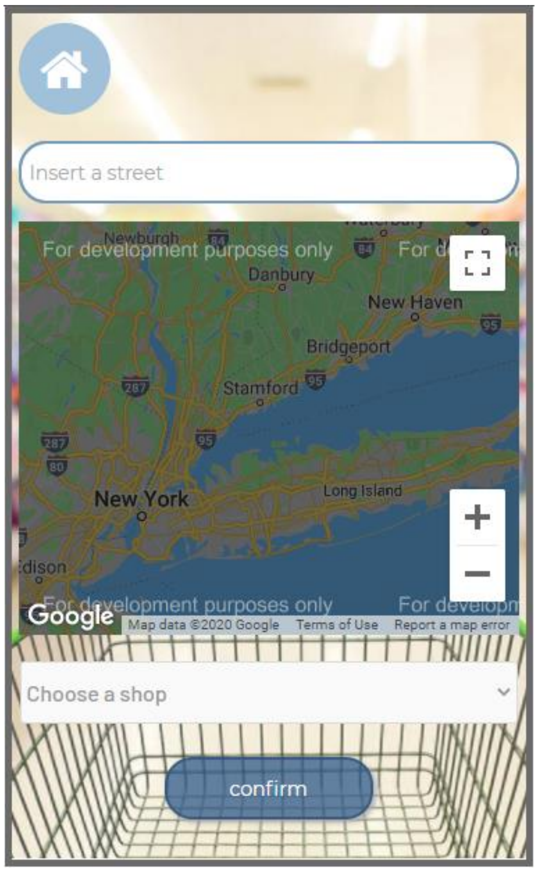
3.1) Mobile App Mockups

As already mentioned in RASD, the mobile app must have simple and intuitive interfaces, as to be easily usable by users that may not be practical with technology (as may be elderly people).

The app will allow user to define first either the shop or the date and time, in the case of "Book a visit", it is not shown in this demo because it is similar and irrelevant.

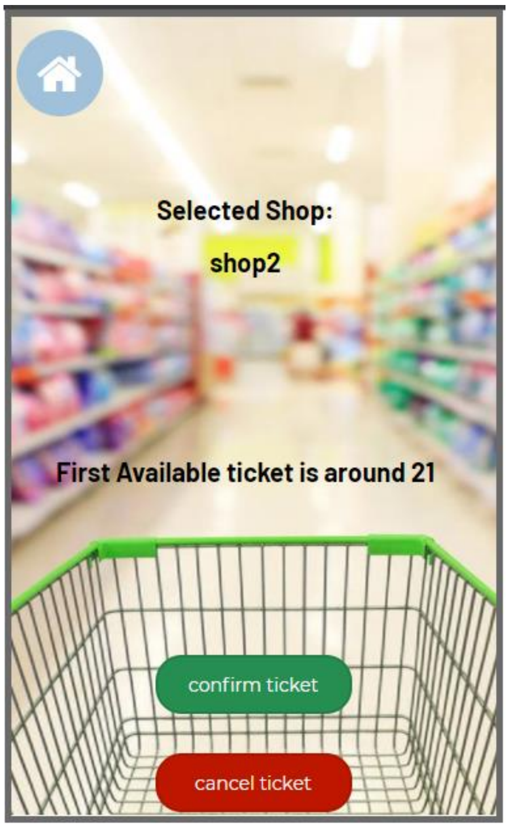


HOME PAGE OF THE APP

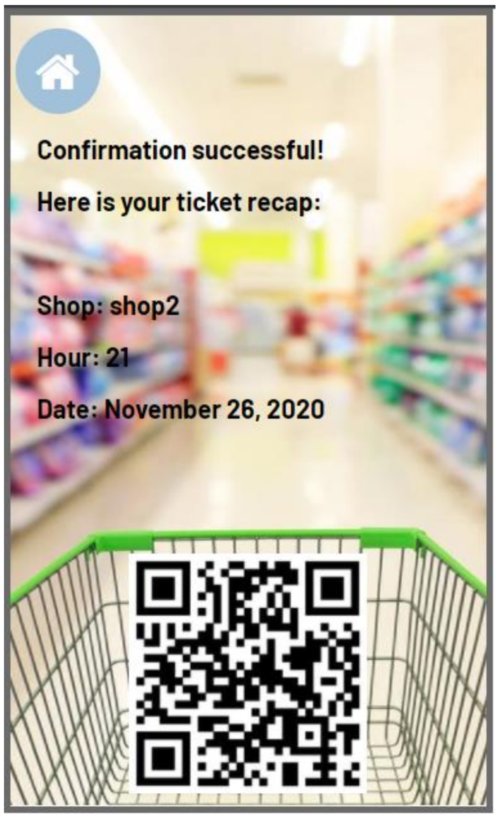


SHOP SELECTION

**Mockups are taken from an interactive graphic demo with graphic-driving purposes only, linked in the "reference documents" paragraph.*

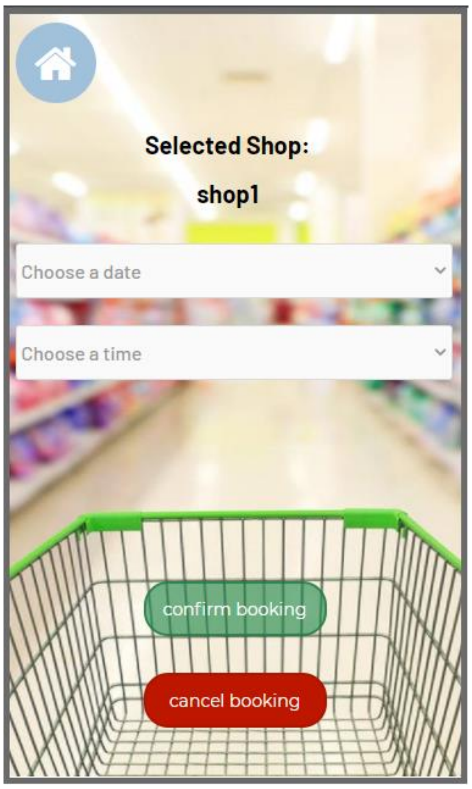


TICKET PROPOSAL

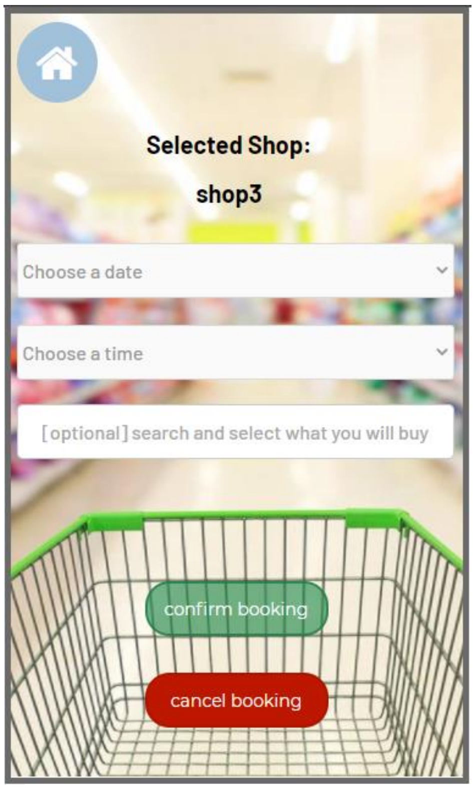


TICKET CONFIRMATION

In the case of "book a visit", after the shop selection, the app will allow registered users to select their shopping list:



BOOK A VISIT AS GUEST

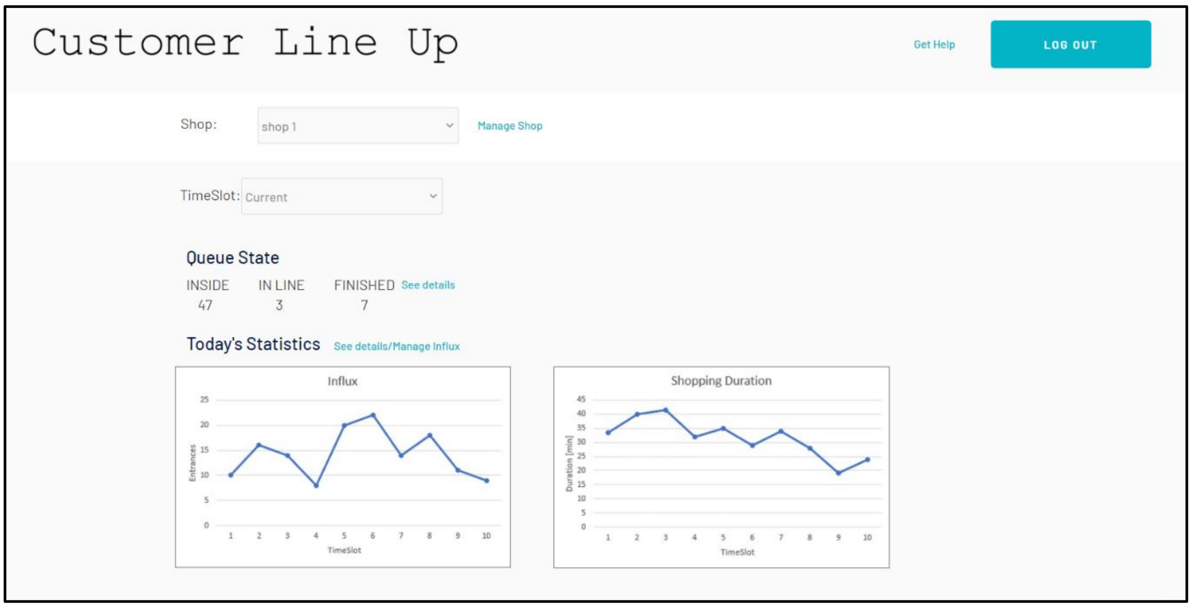


BOOK A VISIT AS REGISTERED USER

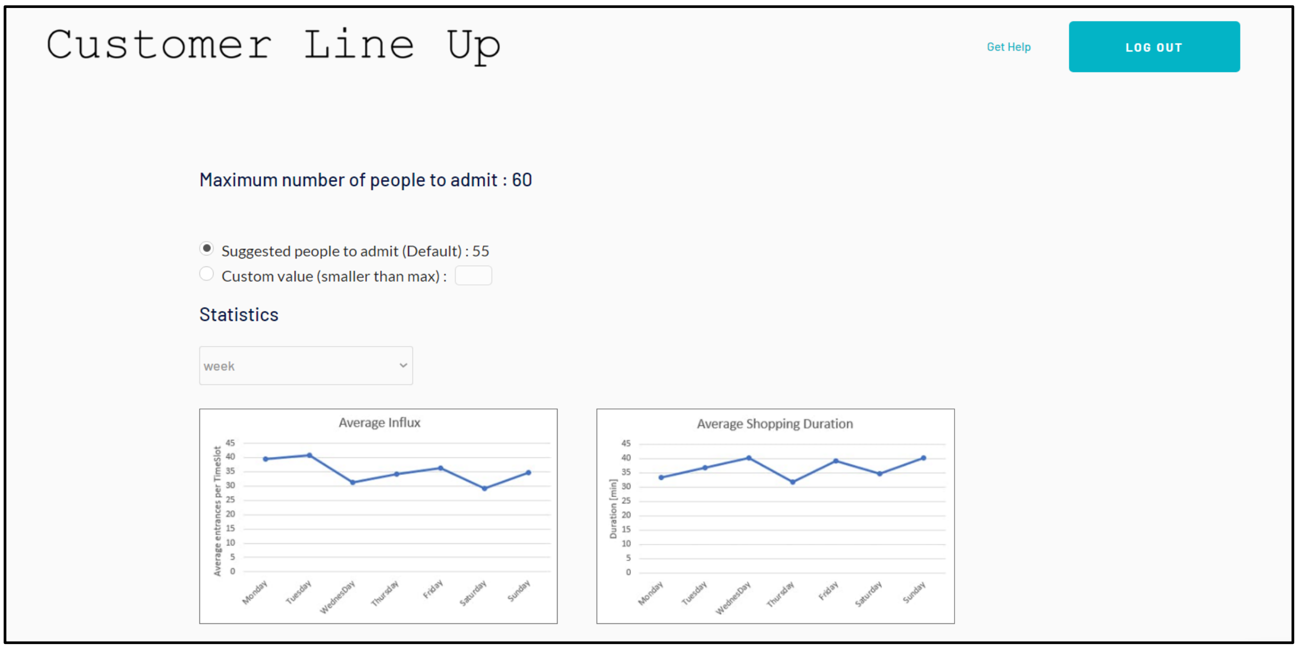
3.2) Shop Manager Web App mockups

The web app for managers will allow only to login with an authorized account. For security purposes, such account cannot be registered directly through the Web App, but it has to be set up by CLup Systemstaff through certified communication with the market/market chain.
The web app must allow managers to trace entrances, to see the queue state of any timeslot, and to see statistics as average time of shopping and the average influx in a given period.

In the real app other statistics may be implemented, as the most declared shopping items or departments (given by user that specified these elements in their visits). Graphics in the real app may be interactive, allowing more interactivity, usability, and utility.



WEB APP HOMEPAGE


Customer Line Up

Get Help

LOG OUT

Maximum number of people to admit : 60

● Suggested people to admit (Default) : 55

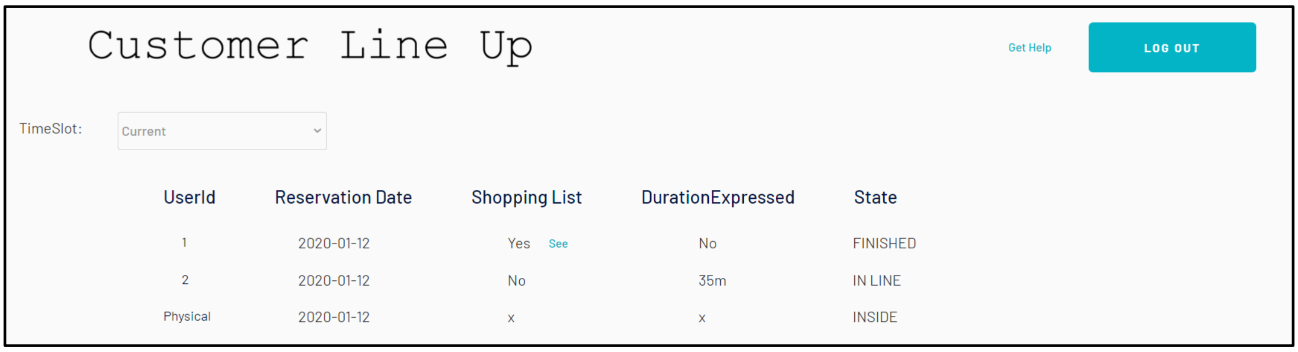
○ Custom value (smaller than max) :

Statistics

week

WEB APP STATISTICS PAGE

The “Manage Shop” page is not shown here, it simply allows managers to add or update logistics info for the shop, as product positions or departments. It just lists departments and their items.


Customer Line Up

Get Help

LOG OUT

TimeSlot: Current

UserId	Reservation Date	Shopping List	DurationExpressed	State
1	2020-01-12	Yes See	No	FINISHED
2	2020-01-12	No	35m	IN LINE
Physical	2020-01-12	x	x	INSIDE

WEB APP QUEUE STATE

The page shown upon clicking the “See” button next to the declared shopping list is not shown here, since it is unnecessary as it simply shows the list of declared items. As shown in the image, physical users have no Id, but they still are tracked and considered correctly in the queue state.

4) REQUIREMENTS TRACEABILITY

R1	The system shall allow users to get a ticket with a date and time that shows when to go to a certain store, virtually
R3	The system shall allow users to book a visit virtually with their desired store.
	Components: • Reservation Manager • Queue Manager • Location Module • Map Mediator Module
R2	The system shall allow users to get a ticket with a date and time that shows when to go to a certain store, physically
	Components: • Reservation Manager • Queue Manager
R4	The system shall allow users to look up on a map available registered stores where to go to
	Components: • Map Mediator • Location Module
R5	The system shall ask users how much he or she thinks the trip to the store will last
	Components: • Reservation Manager
R6	The system shall allow users to be identified by their device unique ID
R7	The system shall allow users to be identified by a username of their choosing
	Components: • Account Manager
R8.1	The system shall notify the user who is inactive while on the confirmation page of booking a visit on other available stores he could go to and other timeslots of the same store
R8.2	The system shall notify the user who is inactive while on the confirmation page of getting a ticket on other available stores he could go to
	Components: • Notification Manager: Suggestion Module • Location Module: Market Finder • Map Mediator

	• Queue Manager: Timeslot Manager
R9	The system shall allow its users to insert information about which categories or items they want to buy
	Components: • Reservation Manager
R10	The system shall infer how long it will take for “Book a Visit” customers to buy the expressed shopping list
	Components: • Queue Manager: Optimizer Module
R11	The system shall store data about registered virtual users' visits durations and expressed shopping lists
	Components: • Reservation Manager • Turnstile Manager • Queue Manager: Timeslot Manager
R12.1	The system shall inform users periodically of the available time slots in the store for which he subscribed to the service
R12.2	The system shall allow its users to select which store(s) to get informed about
R12.3	The system shall allow its users to select which time slots he is interested to get informed about
R12.4	The system shall allow its users to select how often to get notified
	Components: • Notification Manager: Periodic Notifier Module • Messaging Mediator Module
R13	The system shall allow the user to scan its QR code in entrance through the turnstiles
R14	The system shall allow the user to scan its QR code in exit through the turnstiles or cash register
R15.1	The system shall unlock turnstiles after a unique QR code scan in entrance
R15.2	The system shall unlock turnstiles after a unique QR code scan in exit
R16	The system shall map the QR code of a scan to the virtual user who owns the QR code
	Components: • Turnstile Manager • Queue Manager: Timeslot Manager
R15.3	The system shall lock the turnstiles after a push has occurred
	Components: • Turnstile Manager
R17	The system shall send a reminder to the user when it is time for him to leave so that he can arrive at the store in time

	Components: • Reservation Manager
R18	The system shall calculate the time it takes the user to go to a shop in which he has an appointment
	Components: • Notification Manager. Suggestion Module • Location Module: ETA Module • Map Mediator Module
R19	The system shall manage how many people can enter inside the store for each timeslot, as to not exceed the maximum number of people allowed inside the store
R20	The system shall calculate the maximum number of people allowed inside each store, as to allow for social distancing to take place
	Components: • Queue Manager: Optimizer Module
R21	The system shall count the number of entrances and exits each day for each market
R22	The system shall store the number of daily entrances and exits for each market
	Components: • Queue Manager: Timeslot Manager
R23	The system shall manage how many people can enter inside the store for each timeslot, as to not exceed the maximum number of people allowed inside the store hourly, without exceeding the maximum calculated by the system
R24	The system shall allow store managers to see how many customers have entered the store in any day.
	Components: • Market Manager: Influx Module • Web Server
R25	The system shall allow store managers to register their stores
R26	The system shall allow store managers to input the store's location
R27	The system shall allow store managers to input what categories of goods and what products are contained in the store
R28	The system shall allow store managers to input the dimensions of the store
	Components: • Market Manager: Market Info Manager • Web Server

Traceability table:

	MM	WS	*TM*	*AM*	*RM*	*NM*	*QM*	*LM*
R1					×		×	×
R2					×		×	
R3					×		×	×
R4								×
R5					×			
R6				×				
R7				×				
R8.1-2						×	×	×
R9					×			
R10							×	
R11			×		×		×	
R12.1-4						×		
R13			×				×	
R14			×				×	
R15.1-2			×				×	
R15.3			×					
R16			×				×	
R17					×			
R18						×		×
R19							×	
R20							×	
R21							×	
R22							×	
R23	×	×						
R24	×	×						
R25	×	×						
R26	×	×						
R27	×	×						
R28	×	×						

Legend of components:

- *Turnstile Manager: TM*
- *Account Manager: AM*
- *Reservation Manager: RM*
- *Notification Manager: NM*
- *Market Manager: MM*
- *Queue Manager: QM*
- *Location Module: LM*
- *Data Service: DS*
- *Web Server: WS*

Note that, in the table, the two mediators have been omitted, as they just encapsulate the interactions of the internal and external components. They are anyway reported in the full previous table. Also note that, as shown in the component and in the sequence diagrams, each component interacts with the database through Data Service component, that was omitted here in both tables to avoid repeating it in every requirement.

5) *IMPLEMENTATION, INTEGRATION AND TEST PLAN*

5.1) *Implementation Plan*

The implementation of the CLup system will be done component by component. The order in which the implementation will be carried out depends on a number of factors such as the complexity of the component (e.g., number of provided functionalities) and the dependence on the other components being implemented. In parallel, unit tests should be carried out in order to discover flaws early, make easier the changes and guide the design. The implementation's order of the components is the following:

1) Data Service, Map Mediator Module, Messaging Mediator Module
2) Location Module, Queue Manager, Account Manager, Market Manager
3) Reservation Manager, Notification Manager, Turnstile Manager

Note that components belonging to the same group are independent, so they can be implemented in parallel. Moreover, the DBMS service, the Geolocation API and the Firebase Cloud Messaging API are assumed to work properly since are external services.

The **Data Service**, the **Map Mediator Module** and the **Messaging Mediator Module** components should be implemented as first. All the remaining components of the application server depend directly or indirectly on the Data Service since it handles the communication with the database to store and retrieve data. Map Mediator Module and Messaging Mediator Module components are independent from all the other ones because the former just communicates with the external geolocation service, by modifying the API's information so that it can be comprehensible by the Application server, and adapting the requests to the API's protocol, while the latter communicates with Firebase Messaging Computing to allow sending notifications to the users.

Queue Manager, **Account Manager** and **Market Manager** are grouped together because they directly exploit the Data Service module to retrieve information, store data or carry out consistency and security checks. Moreover, **Location Module** belongs also to the second group because it depends on Data Service, to retrieve information about the registered stores, and Location Module to find the closest stores in a range respect a location and compute the ETA.

Eventually, in the third group belongs the **Turnstile Manager**, communicating with the Queue Manager to check the validity of a QR code; the **Notification Manager**, depending on Queue Manager and Location Module to retrieve information about time slots and stores and on Messaging Mediator Module, that enables the communication with Firebase Cloud

Computing, to send notification to the users; the **Reservation Manager**, that communicates with the Queue manager, Location Module and Data Server to handle the creation of a ticket.

5.2.1) *Integration Strategy*

Considering both the overall system's architecture and the implementation plan, the chosen integration strategy is the **bottom-up approach.** System integration begins with the integration of the lowest level modules and uses test drivers to drive and pass appropriate data to the lower-level modules. As and when the code for the other module gets ready, these drivers are replaced with the actual module.

This approach allows to start the integration and testing without necessarily waiting for the completion of the development and the unit testing of each system's component. Being the low-level modules and their combined functions often invoked by other modules, it is more useful to test them first so that meaningful effective integration of other modules can be done. Moreover, starting at the bottom of the hierarchy means that the critical modules are built and tested first and therefore any errors in these modules are identified early in the process.
Each integration in the same level (defined by the groups of the previous section) is independent and there is no specific order in which to complete them. In this way, the integration process and its testing are more flexible.

5.2.2) *Integration and testing*

In this section it is defined the order of the integration between components. Test drivers will be used to simulate higher components not yet implemented. As already stated, since using the bottom-up approach we should begin to integrate the lowest module, the first components chosen for integration are the Data Service (figure 5.1), the Map Mediator Module (figure 5.2) and the Messaging Mediator Module (figure 5.3).

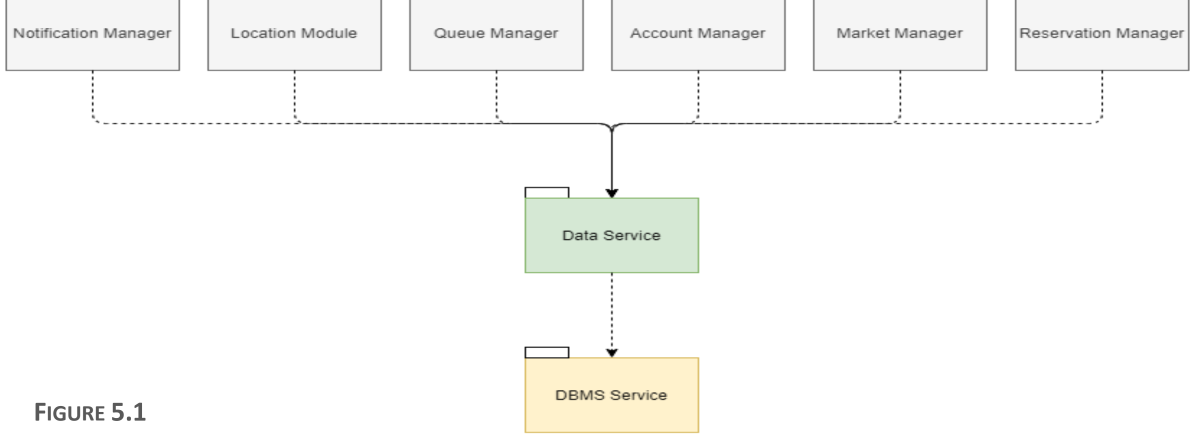


FIGURE 5.1

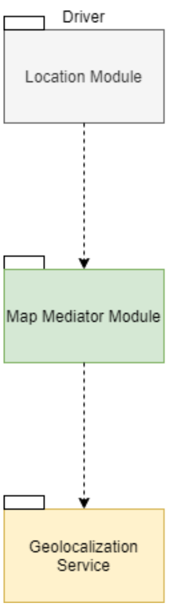


FIGURE 5.2

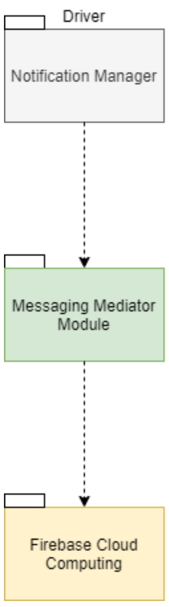


FIGURE 5.3

After that Data Service and Map Mediator Module have been implemented and tested, the integration of the components depending on them can start, namely, the Account Manager (figure 5.4), the Market Manager (figure 5.5)and the Queue Manager (figure 5.6), relying only on the Data Service, and the Location Module (figure 5.7), relying on the Data Service and the Map Mediator Module. Note that Queue Manager and Location Module should be preferably implemented, integrated, and tested before Account Manager and Market Manager because they are more critical and so more likely subject to flaws.

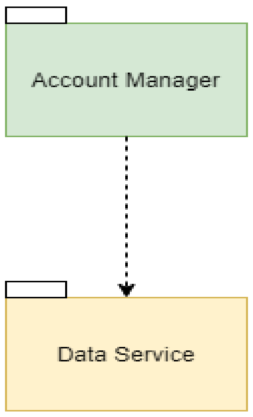


FIGURE 5.4

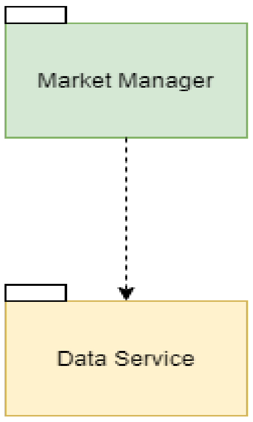


FIGURE 5.5

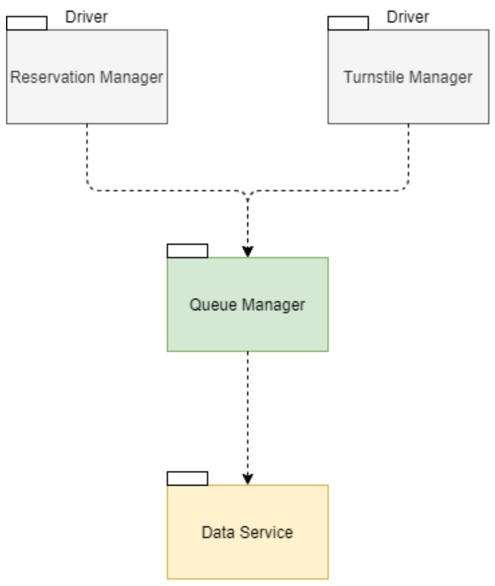


FIGURE 5.6

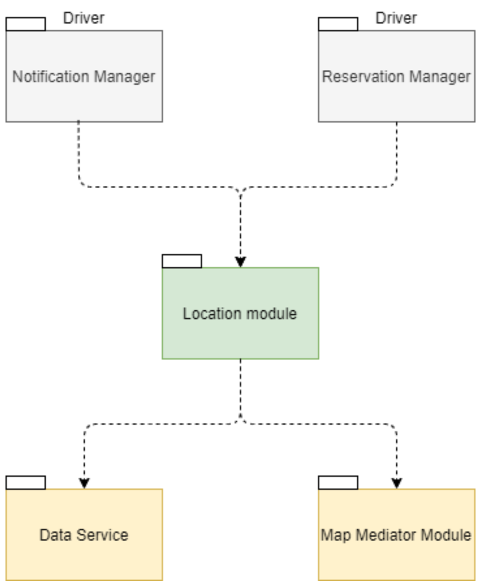


FIGURE 5.7

Eventually, the integration plan involves the Reservation Manager (figure 5.8), depending on the Queue Manager, Location Module and Data Service; the Notification Manager (figure 5.9), depending on the Location Module and Data Service; the Turnstile Manager (figure 5.10), depending on the Queue Manager.

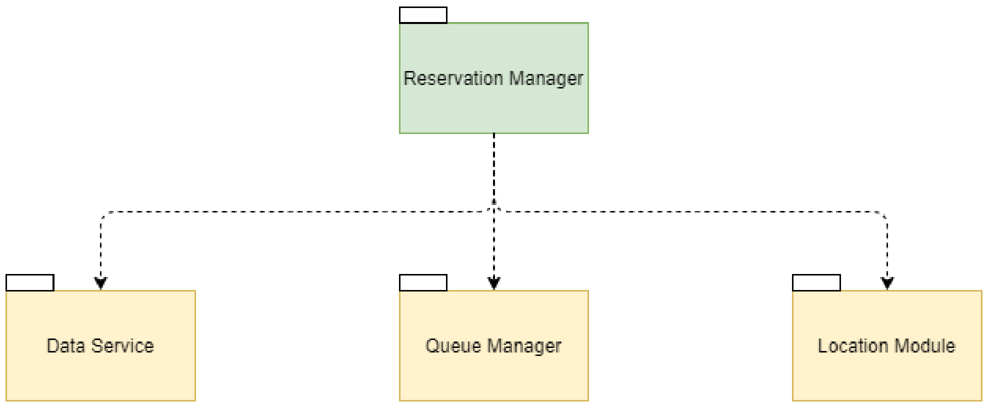


FIGURE 5.8

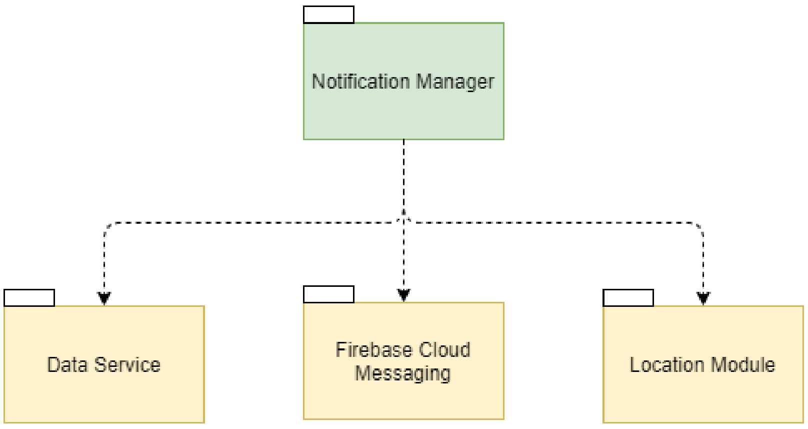


FIGURE 5.9

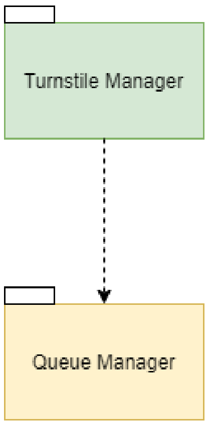


FIGURE 5.10

5.3) *System testing*

Once the whole server has been integrated there need to be carried out both stress and load testing. The scope of these is to achieve data regarding availability. Load testing is especially important because of the high expect demand of users that will use our system. On the same note, performance testing would allow us to know the response time of the most frequent operations and requests such as getting a ticket virtually or physically, booking a visit, searching for stores etc. Performance testing is a key aspect to fulfilling the RASD non-functional requirement of needing a response time of 0.5s. Moreover, the testing should consider important algorithms such as the filtering of topics, the retrieval of data from the database (queries and insertions), optimizer algorithm, or response time from external APIs. The testing will follow the integration plan.

6) *EFFORT SPENT*

- As a group*:*

Date	*Topic*	*#Hours*
19-12-2020	*High level definition of the architecture of our system*	*2*
21-12-2020	*Components' logical definition*	*2.5*
26-10-2020	*Component interaction*	*1.5*
28-11-2020	*Sequence diagram definition*	*2.5*
30-11-2020	*Timeslot method definition*	*2*
03-11-2020	*Notifications and implementation definition*	*1.5*
06-12-2020	*Review of used methods*	*2*
10-12-2020	*Closing remarks*	*2*
NA	*Total*	*16*

- Etion Pinari

Topic	*#Hours*
Introduction	1
Overview: high-level components and their interaction	0.5
Components diagram	5
Deployment view	3.5
Runtime view	4.5
Components interface	1.5
Architectural styles and patterns	1
Additional specifications	3.5
User interface design	1
Requirements traceability	0.5
Implementation, integration, and test plan	*1*
Research on various topics	4
Writing on Word and formatting	1
Total	28

- Giorgio Romeo

Topic	*#Hours*
Introduction	0.5
Overview: high-level components and their interaction	1.5
Components diagram	5.5
Deployment view	0.5
Runtime view	5
Components interface	2
Architectural styles and patterns	1.5
Additional specification	1.5
User interface design	1
Requirements traceability	0.5
Implementation, integration and test plan	4
Research on various topics	3
Writing on Word and formatting	1.5
Total	28

- Cristian Sbrolli

Topic	*#Hours*
Introduction	0.5
Overview: high-level components and their interaction	0.5
Components diagram	7
Deployment view	0.5
Runtime view	6
Components interface	1.5
Architectural styles and patterns	2.5
Additional specification	1
User interface design	3
Requirements traceability	1.5
Implementation, integration and test plan	1
Research on various topics	3
Writing on Word and formatting	1
Total	29

7) *REFERENCES*

- Web App to generate mockups: https://bubble.io/home
- Web app for components diagram: https://online.visual-paradigm.com/diagrams/
- Web app for sequence diagrams: https://app.diagrams.net/
- Average time spent grocery shopping for parents, *Pew Research Center*: https://www.pewresearch.org/fact-tank/2019/09/24/among-u-s-couples-women-do-more-cooking-and-grocery-shopping-than-men/
- Random and routine shoppers, *Byung-Do Kim, Kyungdo Park* (Table 4, pg.510): https://www.sciencedirect.com/science/article/pii/S0022435997900324
- Traefik installation guide: https://github.com/traefik/traefik/blob/master/README.md