



# **POLITECNICO**

## **MILANO 1863**

Software Engineering II

# **CLup - Customers Line-up**

**Design Document**

*Authors:*

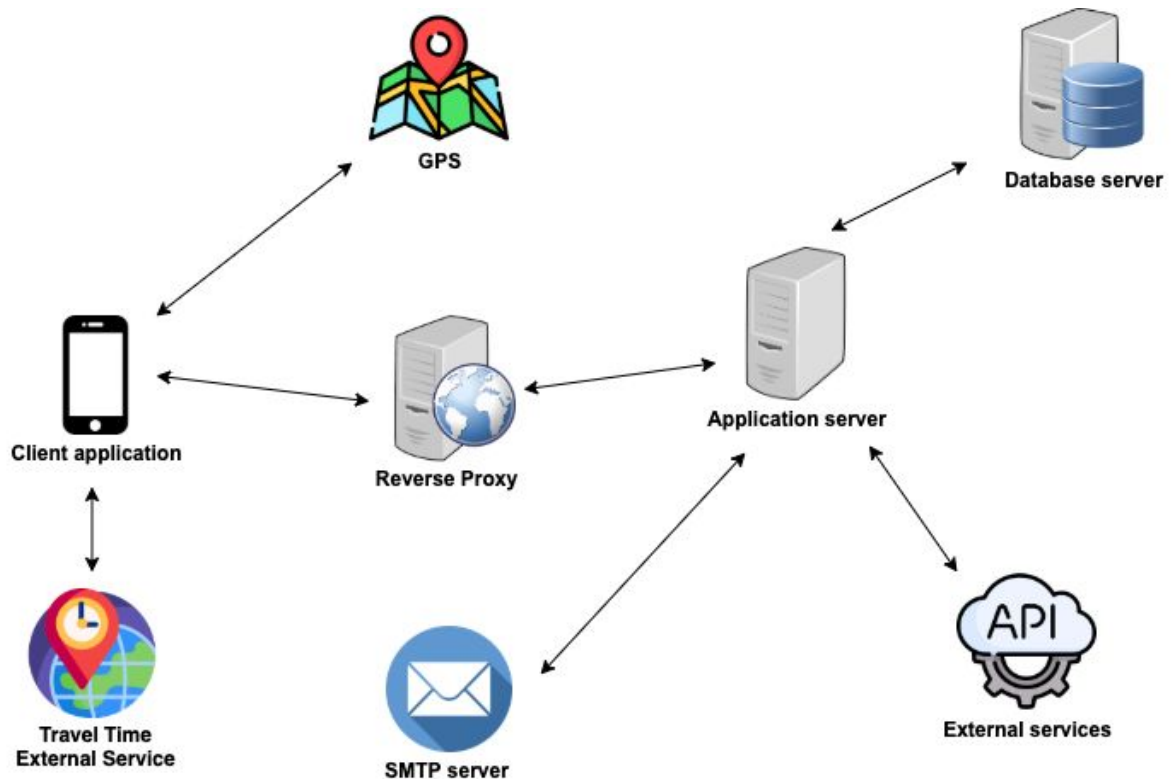
*Cosimo Sguanci*

*Roberto Spatafora*

*Andrea Mario Vergani*

## ARCHITECTURAL DESIGN

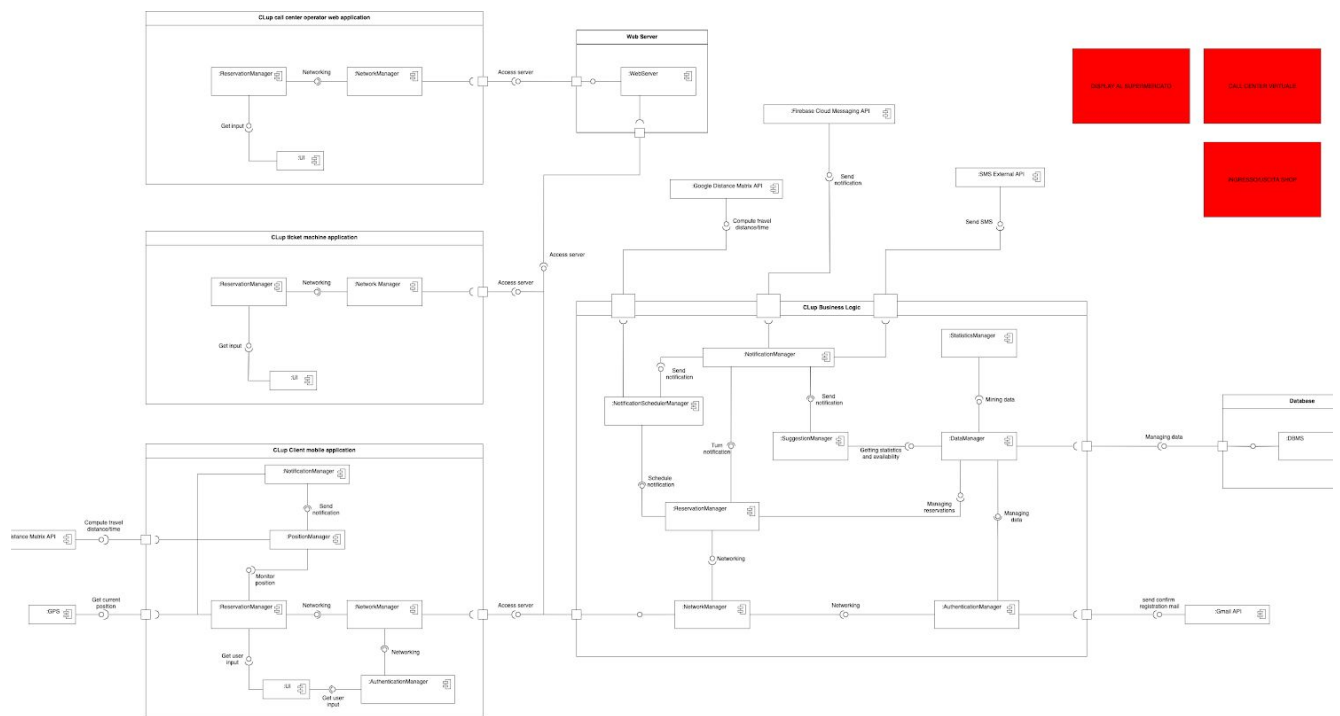
### 2.A Overview



The picture above shows a high level overview of the *CLup* system, which is composed of a three-tiered client-server architecture. Presentation, Application and Data layers are placed on separate machines. Speaking about the presentation layer, it is represented by the “Client Application” component, which includes all types of clients that can be identified in the *CLup* architecture: mobile application, ticket machine and call center operator web application.

Moreover, the call center operator interface interacts with a Web Server (not modeled here for simplicity) in order to get the HTML pages for the Web Application that the operators use to create reservations for call center users.

## 2.B Component View



The Component View shows an internal representation of the layers that make up the system. In particular, this diagram highlights the distinction between presentation, application and data layers.

Regarding the presentation layer, all the three types of client that can be encountered in the system are described, together with their functional differences. In addition to the three clients, the presentation layer also uses a Web Server to provide web content (pages and code) to the call center operator Web App. This server also acts as a *forward proxy*, forwarding the Web App's requests to the Application Server.

The second layer is mainly composed by the Application Server, which is the core of the system and contains all the business logic.

The data layer is represented by the Database server, which handles all the queries sent by the DataManager component of the Application server.

### CLup Business Logic Components

#### DataManager

The DataManager component is the one that handles all the interactions between the Application Server and the Database Server. Interfaces for all the necessary operations (mainly user data management and reservations management) are offered to the other components which make up the business logic.

## **ReservationManager**

The ReservationManager is the component in charge of handling the client's requests regarding adding/deleting/updating reservations (both tickets and visits). In addition, it is up to this module the scheduling of notifications to let the user know when it's its turn to enter the shop. These notifications are *dynamic*, meaning that they cannot be planned in advance: a user that has a reservation for 17:00 may have to wait some more minutes if the ReservationManager detects that the probability of crowding inside the shop is too high, according to reservation data stored in the database.

This component implements the core functionality of the system, and for this reason it interacts with most of the other modules.

## **AuthenticationManager**

This component handles the requests sent by the clients when a new user is registering and when an existing user is logging in. This means that this module actually authenticates users. In addition, it is responsible for the protection of users information, validating the access token sent by the client to get access to user private resources. To achieve these tasks, an interaction with the DataManager is necessary to store and retrieve user data to/from the database.

## **StatisticsManager**

The StatisticsManager module contains the algorithms that are scheduled to run periodically in order to build statistics and mine the data stored in the database. More specifically, the statistics which are built by this component are used to:

- propose visits in advance to users, based on historic information regarding past visits;
- show different availabilities to distinct users, based on habitual duration of visits and the categories of items which are usually bought (to avoid too many people crowding inside the shop in the same areas).

## **SuggestionManager**

The SuggestionManager runs periodically and makes use of the information extracted by the StatisticsManager from raw data. In fact, this component determines, for each user, if there's a suitable suggestion to make, in order to let the user reserve his favorite slot to go to the grocery shop.

## **NotificationSchedulerManager**

This component keeps track of scheduled jobs used to notify users in two cases:

- when a mobile app user or a call center user finalizes the booking of a visit or a ticket, the ReservationManager module interacts with this component in order to schedule a reminder notification to be sent half an hour before the time of the reservation;
- when a mobile app user provides the starting location for a visit, the ReservationManager invokes the NotificationSchedulerManager. This component then computes the travel time using the Google Distance Matrix external API, and schedules a notification to be sent to the user when it's time to leave.

This component has been thought to reduce responsibilities given to the ReservationManager, to which, however, is still delegated the handling of turn notifications.

## **NotificationManager**

This component uses third-party APIs (in particular, Firebase Cloud Messaging API for push notifications) to send notifications to specific users. There are many cases in which a notification is needed:

- When the turn of the user has come;
- When the user is suggested to leave from home (only for mobile app users);
- When there are available suggestions for the user, for an upcoming reservation in its habitual slot of the week.

## **NetworkManager**

The NetworkManager module acts as a *router*, receiving requests by the clients (or the Web Server) and forwarding them to the right components that are designated to handle them. In practice, the NetworkManager will receive authentication and reservation management requests, and it will send them respectively to the AuthenticationManager and the ReservationManager.

## CLup Client Components

Speaking about the client-side architecture of the system, all the three applications share a common structure, composed of a **ReservationManager**; a **NetworkManager** and a component responsible to build the User Interface.

The client's **ReservationManager** has interesting features especially regarding the mobile application. In the mobile app, this component takes the user input when a reservation is being booked, validates it and sends the acquired data to the **NetworkManager**, waiting for a response to be returned to the user. Moreover, this component is responsible for handling the case in which a user wants to delete or modify a reservation, and for storing reservations details offline. This allows the user to be informed about his reservations also if there is not an available internet connection.

The Call Center operator Web App and the mobile application also make use of an additional component, the **AuthenticationManager**. In the Web App, this module is used as an helper to authenticate operators, who have privileged permissions and can therefore make reservations on behalf of standard users. In both applications, this component acts as an intermediary with the **NetworkManager**, appending the access token to HTTP requests. The access token is used by the Application Server to authenticate users (both operators and customers), and it's therefore necessary to gain access to private user resources on the database (like user data or simply to book a reservation).

The ticket machine client application has a different behaviour, as it comes with a preinstalled secret key in an encrypted area of memory. This key is then whitelisted on the backend so that the ticket machine itself can act as a "virtual operator" and book tickets for the ticket machine users.

Another component of particular interest is the **PositionManager** of the mobile app, which runs the background service to send a notification (through the local **NotificationManager**) when it's time to approach the shop, in case the user hasn't provided his starting position when the visit was booked. The algorithm used by the service is described in section 2.G.1.3.

## **EXTERNAL SERVICES [API esterne troppo specifiche?]**

### **Google Distance Matrix API (reference)**

This external API is used to get information about the travel distance from the mobile app user's location to the selected grocery shop for the booked visit. The system makes use of this API on the backend if the user provided its starting location for the visit, otherwise the service is used directly on the client, to avoid to track real time user position in CLup's servers. In the latter case, the mobile app schedules a job to run one hour before the visit, and it starts monitoring the user position. By crossing user real time location and the remaining time before the visit, the mobile app can notify users at the most appropriate time to go to the shop. More detailed specifications about the algorithms that will be used are shown in section 2.G.1.3.

### **Firestore Cloud Messaging API / SMS External API (reference)**

These external services both have the same purpose: notify specific users about their reservations. The Firestore Cloud Messaging API is used only for mobile app users, while the SMS service can be useful for both mobile app users and call center users.

In particular, the situations that require a notification to be sent to users are the following:

- The user's turn to enter the supermarket has come;
- Based on information provided by the user (location and/or means of transport) the system detected that the customer should leave to get to the grocery shop.

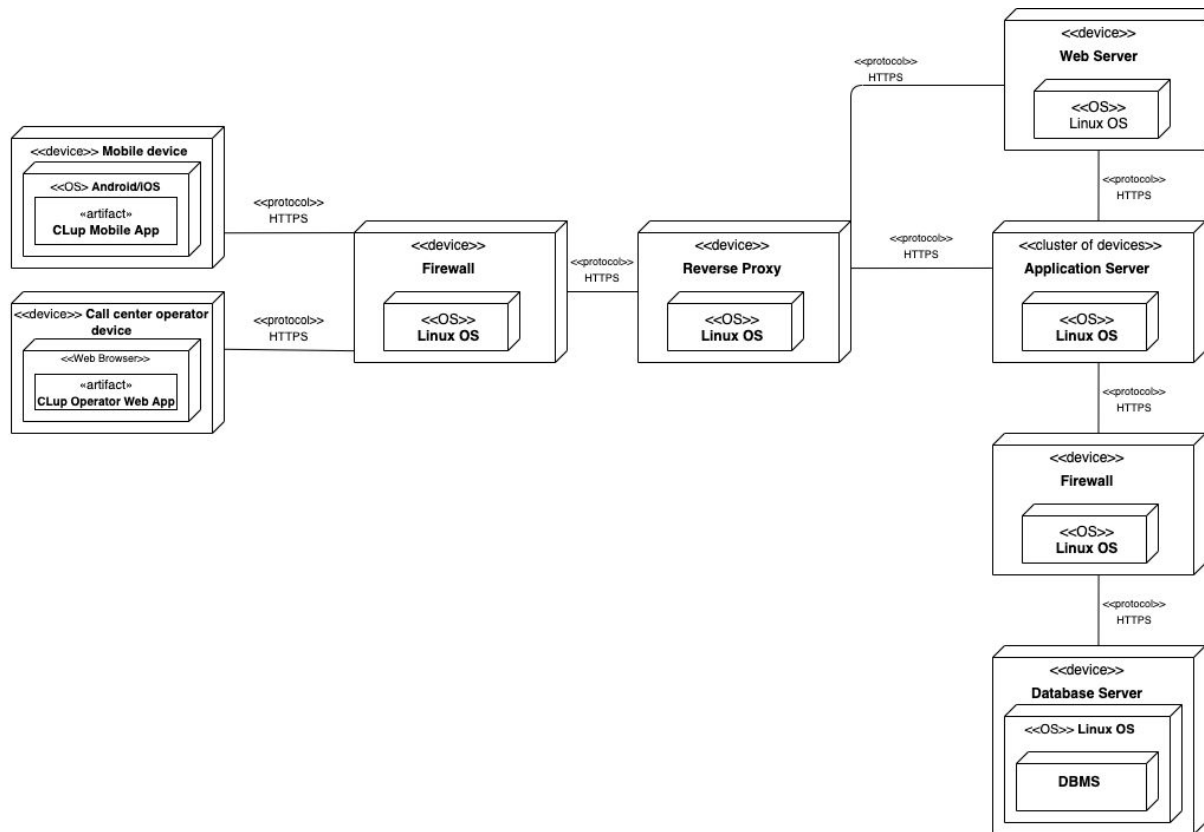
In addition, the system schedules notification at predefined times to remind users about their upcoming reservations.

### **Gmail API (reference)**

The Gmail API allows the system to send emails to users when they sign up using the mobile app, and to reset credentials in case of password loss.

In addition to the external APIs presented, the CLup mobile application makes use of the GPS utilities offered by Android and iOS, in order to get the user current GPS location used for notifications.

## 2.C Deployment View



- **Mobile device**  
This device is used by mobile app users to make reservations and see/modify information about visits and tickets which have already been booked. It is used exclusively by customers, and communicates with CLup's backend services using the HTTPS protocol.
- **Call Center Operator Device**  
This is the PC used by human call center operators to make reservations for call center users through the CLup Operator Web App, using a Web Browser.
- **Firewall**  
The Firewall is used to filter packets sent from the Internet to CLup backend services in order to protect both the Application Server Cluster and the Database Server.
- **Reverse Proxy**  
The Reverse Proxy component has mainly two purposes:
  - increase availability by acting as a load balancer, to distribute the load across the Application Server Cluster;
  - increase security by mitigating DDOS attacks.



- **Web Server**

The Web Server node is used only by the call center operators Web App and it's used to retrieve the web HTML pages and the frontend JavaScript code. Moreover, this component acts as an additional proxy between the Web App and the Application Server, forwarding requests and responses in both directions.

- **Application Server Cluster**

This is a cluster of replicated devices and services which contains the core business logic of CLup. In conjunction with the load balancer (reverse proxy) this improves the parallelism and performance of the system.

- **Database Server**

This is the node containing the core data that the CLup system must store and use to offer its service to users. A Linux OS is installed on the device, as well as a relational DBMS.

## **PAGINE DI ANDREA**

### **2.F SELECTED ARCHITECTURAL STYLES AND PATTERNS**

#### ***Client-server architecture, HTTPS and REST***

The choice for a client-server architecture is quite natural for *CLup*: in fact, the application is distributed. Since the model is quite straightforward, no further details are needed to specify.

The communication protocol for all messages between client and server is HTTPS: it guarantees a high level of security, confidentiality and data integrity.

HTTPS is enriched with the usage of REST architectural style, which provides a stateless protocol ensuring reliability, reusability and scalability.

#### ***Three-tier architecture***

The client-server architecture follows a three-tier pattern: in particular, we can distinguish a Presentation tier, a Logic/Business tier and a Data tier.

The division into three layers allows to separate tasks and increases the level of reusability and decoupling; in addition, the whole architecture results to be more flexible and maintainable (a single tier can be internally modified, fixed, ... without consequences on other tiers).

More specifically, the tier division is the following:

- Presentation tier is client application's business (mobile application, call center web application); it allows the interaction between user and *CLup*
- Logic/Business tier is present both on the client and server sides: servers' logic layers control application functionalities; client logic is related to tracking mobile app user position, in order to send specific notifications \*
- Data tier includes the database and all mechanisms for storing data

*\* client logic is better described in section named "Thick client" (which immediately follows)*

### **Thick client**

With reference to the mobile application, a thick client configuration is adopted: this implies that, in addition to the presentation layer, the mobile app client also incorporates a part of the business layer.

More in detail, *CLup* specification highlights the need to calculate the user's distance to the grocery shop, in order to send proper notifications for arriving on time and not missing the reservation; furthermore, the application's RASD describes how the customer can insert, in optional fields, his starting position and transportation means, so that the system can more easily deal with its computations (server-side, in general). However, the user may not add this additional information: to solve the situation, the decision is that of making the client itself responsible for sending proper notifications.

The main cause behind this design decision is that server, not having an estimate about when to start checking user's position (because the optional fields were not filled), would have to continuously track him (or something similar, at least for a time period): however, this is costly and would require special permissions for privacy reasons. For this reason, the logic regarding position monitoring for notifications (in case no additional information is provided by the customer) is left to the client, thus leading to a thick client design.

### **Model-View-Controller**

Integrated with the three-tier architecture, Model-View-Controller design pattern guarantees a clear separation (but also interaction) between the different layers of the application.

In detail:

- Model lies on the server, in particular on the business logic component and the database
- View lies on the client and corresponds to the Presentation layer of the three-tier architecture
- Controller is both client-side and server-side: on client some simple controls about inputs are performed, together with the more consistent part regarding the check of user's position (as described in "Thick client" section, which immediately precedes the current one); on server all simple controls are

replicated (for robustness), with the addition of all the necessary for the correct system functioning

Model-View-Controller pattern particularly fits the development of a mobile application and Web application.

Together with MVC, the usage of an Observer design pattern guarantees a flexible interaction between layers.

### ***Relational DBMS and SQL***

The DBMS managing data in the database is built upon the relational model. The choice of a relational DBMS is widely used and there is no particular reason for deviating from it in *CLup* application; all commercial DBMSs guarantee ACID properties, which in turn ensure good features (efficiency, concurrency, ...).

SQL is used for querying and managing data; the choice for SQL follows the standard for relational DBMSs (so there is no need to further comment on it).

## **2.G OTHER DESIGN DECISIONS**

### ***Replication***

*CLup* must guarantee at least 99% of availability, as discussed in the *Requirements Analysis and Specification Document*. In order for this feature to be fulfilled, server business logic and database are replicated: in case of problems to one of the units, the others can still serve clients and provide all the application services.

The choice for “two nines” (99%) availability means that the average downtime period must not be greater than 3.65 days in a year. Supposing that a grocery store is open for 12 hours a day (every day of the year), which is a quite reasonable assumption, the downtime that really influences the possibility to manage entrances affects 1.825 days.

This line of reasoning follows the fact that downtimes outside supermarket’s opening hours are not a big issue: customers do not have the possibility to book or manage reservations, but in reality this is not a very big problem. The critical part, instead, is about server unavailability during opening hours: entrance management becomes impossible and every grocery shop should put some “manual” solutions into practice, thus causing crowds to form outside the shops themselves. Since this kind of situation happens less than 2 days in a year (on average) and the nature of the application is not extremely critical (it does not concern emergency situations, since keeping distance in a line for one day can be done, hopefully without serious consequences), 99% of availability has been chosen.

**Proxy**      *descrivere l'utilizzo e motivare la scelta*

**Firewall**      *descrivere l'utilizzo e motivare la scelta (sicurezza)*

## **2.G.1. Algorithms**

The aim of this section is to provide, rather than just a high-level overview, a more precise guide to whom will handle the implementation part, in order to better understand the objectives of the following features and implement them following specific guidelines.

### **2.G.1.1. StatisticsBuilder**

The application takes advantage of an algorithm that periodically (once a week) runs in order to build and update current customers' statistics. Statistics are widely used in *CLup* application: customers' time in the shop estimations and categories of purchased products traceability are used to allocate slots in the best possible way. StatisticsBuilder algorithm uses collected data to:

- *update individual customer profiles*: their average spent time in the grocery store, the most purchased product's categories and the most frequent days of the week for shopping (used to provide the best possible recommendations and better allocate time-slots for visits and tickets)
- *update global customer behaviors*: the average spent time inside the shop and the most purchased product's categories. Moreover, taking into account these factors, StatisticBuilder is able to generalize consumers' habits, for better handling visit and ticket requests.

## 2.G.1.2 ReservationManager handles a “Book a visit” request

```
findVisit (AppUser user, GroceryShop groceryShop, Date date, Time time,
Optional<Integer> duration, Optional<List<Product>> products,
Optional<Location> startingLocation, Optional<Transport> transport)
{
    Integer durationForAvailability = computeDuration(user, duration);
    List<Category> categoriesForAvailability = computeCategories(user,
        products);

    Visit visit = DataManager.checkForAvailability(groceryShop, date, time,
        durationForAvailability, categoriesForAvailability);

    if(visit != null) {
        if(startingLocation.isPresent) {
            PROXY.handle(user, visit, startingLocation, transport);
        }
        else {
            Ask to the client to handle the part about notifications
            PROXY.handleOnlyPredefinite(user, visit);
        }

        return visit;
    }

    else {
        Datamanager.save(groceryShop, date, time, duration, products); //saving
                                                                    real preferences in the DB

        Collection<Suggestion> suggestions = new Collection();

        Compute queryFilters, as explained in RASD

        for all queryFilters {
            Suggestion suggestion = DataManager.getVisitSuggestion
                (groceryShopFilter, dateFilter, timeFilter
                durationForAvailability, categoriesForAvailability);
            suggestions.add(suggestion);
        }
        return suggestions;
    }
}
```

The proposed algorithm is a sketch of how ReservationManager handles a visit request.

First of all, since slots management is performed also basing on users' habits and/or provided information (for example, the system does not show availability if a person would like to go shopping for 2 hours, but the grocery store is almost overflowing from a certain moment on, so that only short visits are allowed), the component should get this information. This part is not fully described in the algorithm because it is delegated to other methods (computeDuration() and computeCategories()): in practice, data coming from user's habits (got through a database query) and declared duration/categories of items to buy are combined in a sort of weighted average, in order to be faithful to both (for example, a person who declares 30 minutes of visit, but usually spends 3 hours in the shops, will likely be a bit later than his declaration). If a person does not declare some of the parameters, the algorithm only counts on "history"; in case of a customer's first visit and no declaration, data coming from the average of all users is taken as a habit (in order to have a reference).

After this first computation, the algorithm asks DataManager for visit availability, passing all known information: this call results in a query to the DB, and the result comes back to ReservationManager.

If the visit is found, of course there is little more to do: simply managing the part about notifications, and return the visit. Another method (the caller of this algorithm) will confirm the visit, both to the user and to the database.

Instead, in case the visit is not found, preferences are saved (they can be useful for statistics) and some suggestions are returned; suggestions include slots for:

- same supermarket, same day and different hour ( $\pm 2$  hours)
- same supermarket, same hour and different day ( $\pm 2$  days)
- same day, same hour, different supermarket (among the five closest to the selected one)

Of course, availability for suggested slots is checked through a proper call to DataManager.

### 2.G.1.3 CLup Mobile App notifies user in case of “starting location” not selected when the visit was booked

```
// Called by the Client's ReservationManager when a visit
// without starting location has been just booked

scheduleNotification(Date date, Time time, GroceryShop shop, MeansOfTransport meansOfTransport) {

    scheduleTime = time - 1 // 1 hour before the visit (time is in hours)
    scheduleJob(monitorPositionAndNotify(date, time, shop, meansOfTransport), date, scheduleTime)

}

monitorPositionAndNotify(
    Date date,
    Time time,
    GroceryShop shop,
    MeansOfTransport meansOfTransport) {

    actualTime = time + 1
    done = false

    while(!done) {

        remainingTime = getCurrentTime() - actualTime

        if(remainingTime < fromMinutesToHours(15)) {

            sendNotificationToUser("Your Visit at " + shop.name +
                " is in 15 minutes, please get to the shop!")

            done = true
        }
        else {

            if(computeTravelTime(getCurrentPosition(), shop.getLocation(), meansOfTransport) >=
                (remainingTime + fromMinutesToHours(5))) {

                sendNotificationToUser("Your Visit at " + shop.name + " is in " +
                    fromHoursToMinutes(remainingTime) + " minutes, you should leave now!")
                done = true
            }
        }
    }
}
```

When a mobile app user books a visit from the *CLup* mobile app, he has the possibility to select his starting location and the means of transport that will be used. If the customer decides to provide this information, the Application Server can compute the travel time and schedule a notification, to allow the user know when to leave, in order to avoid waiting too much outside and thus risking forming crowds.

However, if the user does not provide this information, since it's still necessary to provide users with this kind of notification, a client-side mechanism for monitoring position with respect to the booked visit is necessary. A precondition for this algorithm to work is that the user at least has selected the means of transport he plans to use to go to the shop. If this is the case, immediately after the confirmation regarding the reserved visit has been received from the backend services, the mobile app's **ReservationManager** can schedule a background service, represented here by the method **monitorPositionAndNotify**.

This method is scheduled to run one hour before the time of the visit, and, in practice, what it does is monitoring the user location and send a notification in the following two cases:

- The time remaining before the visit is 15 minutes or less, in this case a notification is sent to the user and the algorithm ends;
- The travel time between the user's current GPS position and the shop's location is greater than or equal to the remaining time before the visit plus five minutes. To give an example, if the user has a visit at 18:00 and he is in a location which is 15 minutes away from the shop, the background service would send a notification at 17:40 and then the algorithm would end. The travel time can be computed by using the Google Distance Matrix external API.



## REQUIREMENT TRACEABILITY

### Business Logic

Reservation manager	
R1 fino a R12: tutti	
R15	
R18	
R20	
R22	
R23	
R24	

Notification manager	
R13	
R14	
R19	

<b>R25</b>	
------------	--

<b>PROXY</b>	
<b>R16</b>	
<b>R17</b>	
<b>R19</b>	

<b>Suggestion manager</b>	
<b>R25</b>	
<b>R</b>	
<b>R</b>	

<b>Statistics manager</b>	
<b>R21</b>	
<b>R25</b>	
<b>R</b>	

<b>Data manager</b> uguale DBMS	
<b>R1-R25 tutti</b>	
<b>R28</b>	

<b>Authentication manager</b>	
<b>R28</b>	

## Network manager descrivere a parole

### Mobile app

Reservation manager	
R1	
R4	
R8	
R16	
R22	

Notification manager	
R13	
R19	
R25	

Authentication manager	
R28	

Position manager	
R19	
R	
R	

UI	
R1	
R4	
R8	
R16	
R22	

**Network manager** descrivere a parole

## Ticket machine

Reservation manager	
R3	
R6	
R8	
R17	
R27	

UI	
R3	
R6	
R8	
R17	

**Network manager** descrivere a parole

**Web server** descrivere a parole

## Call center

Reservation manager	
R2	
R5	
R8	
R26	

UI	
R2	
R5	
R8	

**Network manager** descrivere a parole

**Web server** descrivere a parole

Firebase Cloud Messaging API	
R13	
R19	
R25	

SMS External API	
R13	

Google distance matrix API	
R19	

GPS	
R19	

Gmail API	
R28	

## Runtime-view suggestion diagram description

The suggestion runtime-view shows the call and method invocations from the different components involved in the suggestion of a slot process.

Periodically, the SuggestionManager component analyzes the statistics built by StatisticsManager with the objective of being able to recommend to users slots that they are more likely to reserve. When data has been analyzed from this component, it invokes the checkAvailability method of the DataManager component, passing the computed slots as parameters. At this point, the DataManager component will query the Database for the availability of the preferred slots and the result is carried back to the first caller. Only in the case in which the preferred slots are available the SuggestionManager component will notify the user for a possible favourite slot available through the NotificationManager and the use of external API: Firebase Cloud Messaging that will handle the sending of the notification.

## Runtime-view authentication diagram description

The following/above diagram represents the sequence of calls among the components involved in the mobile app user authentication process. The sequence of invocations starts with an HTTP POST in which access data is encrypted and then sent to the server. The authentication process will continue only in case of correct data from the syntactical point of view (all the fields have been correctly filled in: consistent email format).

At this point, the AuthenticationManager component will invoke the authenticate method of the DataManager passing the received access data as parameters. The DataManager component queries the database and once received the result, if the login was successful, the AuthenticationManager generates an access token that will be associated to the specific user, stores it into the database (through the DataManager component) and confirms back to the Client Mobile Application the successful login. Finally, the user is redirected to the home page.

In the other case, in which the AuthenticationManager receives a login error message from the DataManager due to wrong data or BLOCKED user condition ([link to BLOCKED user definition da mettere nelle due parole](#)) the login would fail and the user would be notified about it.

## Runtime-view book a visit from mobile app description

The following/above diagram represents the sequence of calls among the components involved in the mobile app user book a visit process. After that the user requests to book a visit by clicking the specific button, the client mobile application, through a HTTP GET requests for the list of all the possible shops to the server. This

request is at first taken into account by the ReservationManager component that forwards it to the DataManager. This last component queries to the Database and the result is carried back to the client application. Once the user sets his preferences for the visit a sequence of calls, from the Client, through the server (ReservationManager and DataManager), to the Database is invoked to check for availability for the user's preferred slots. In case of successful response, it is confirmed back to the client and it will confirm the visit to the user. In the unfortunate case in which the requested slot is not available, the system will anyway store the user's preferences (in order to be able to build statistics based on customers' actual preferences). Thus, the ReservationManager invokes the save() method of the DataManager component, passing the user's preference as parameter. The DataManager will INSERT those preferences on the database. The ReservationManager, computes some filters for possible visits based on the last expressed user's preference. Then, it invokes the DataManager to query the database for those slots availability and reserve them for that specific user for a limited period of time. When the ReservationManager receives the response, it sends the response back to the client mobile application. At this point the user has the opportunity to accept one of the suggested slots or decline them.

- ❖ If the user accepts one of the slots, the ReservationManager confirms the selected one to the DataManager that will UPDATE (unlock) the other pending tickets and then the client will notify the user of the success.
- ❖ On the other hand, if the user declines the proposed alternatives: a sequence of invocations is used to unlock all the pending tickets.

## **Runtime-view book a visit from call center description**

The reported diagram shows the sequence of calls and invocations among components involved in booking a visit from the call center process. The diagram starts after the user has expressed his preferences for the visit (either with the IVR or directly speaking with a call center operator). A series of calls from the CallCenterWebApp, through the WebServer and the Server (ReservationManager and DataManager), to the Database is invoked to query from the database what are the slots that best suit the user's preference. Those results are sent back to the CallCenter that will propose them to the user. Proposed slots' tickets are set pending as in the book a visit from the mobile app process. At this point the user will have the possibility to accept one of the proposed slots or discard them.

- ❖ In the case in which the user accepts one of the slots (the CallCenter receives it as a feedback from the user), the ReservationManager confirms the selected one to the DataManager that will UPDATE (unlock) the other pending tickets and then the CallCenter will ask for the user's phone number. When the user inserts its phone number (the CallCenter receives it as a feedback from the user), passing through the WebServer, with a HTTP POST asks the ReservationManager to save the phone number related to the ticket.



The ReservationManager invokes the DataManager to store the number, coupled with the ticketID, into the database. When it receives the successful confirmation of the INSERT, the ReservationManager invokes the sendSMSNotification method of the server NotificationManager that will request (with a HTTP POST) to send the SMS to the user at an SMS external API.

- ❖ On the other hand, if the user declines the proposed alternatives: a sequence of invocations is used to unlock all the pending tickets.

## **Runtime-view get a ticket from TicketMachine description**

The following/above diagram represents the sequence of calls and methods invocations among the different components involved in the get a ticket from a ticket machine process. It is not reported the alternative scenario in which no more tickets are available for the current day because it would not represent the typical ticket machine usage and also would be similar to the book a visit from the mobile app diagram reported above. When the user clicks on the get a ticket button on the TicketMachine, the TicketMachine application through a HTTP POST asks the server for a ticket. The ReservationManager will manage the incoming request by invoking a specific DataManager method. Then, the DataManager will query the Database for the first available ticket and set it as pending. The result of the query contains the expected waiting time and this is sent back to the TicketMachine application that will show it to the user. At this point the user will have the possibility to accept or discard the proposed ticket.

- ❖ In the case in which the user accepts the ticket, the ReservationManager confirms the selected one to the DataManager that will UPDATE the Database with the new ticket. The result is carried back to the TicketMachine application that will print the QR code and the queue number to the user.
- ❖ On the other hand, if the user declines the proposed alternative: a sequence of invocations is used to unlock the pending ticket.