POLITECNICO
MILANO 1863

Software Engineering II

CLup - Customers Line-up

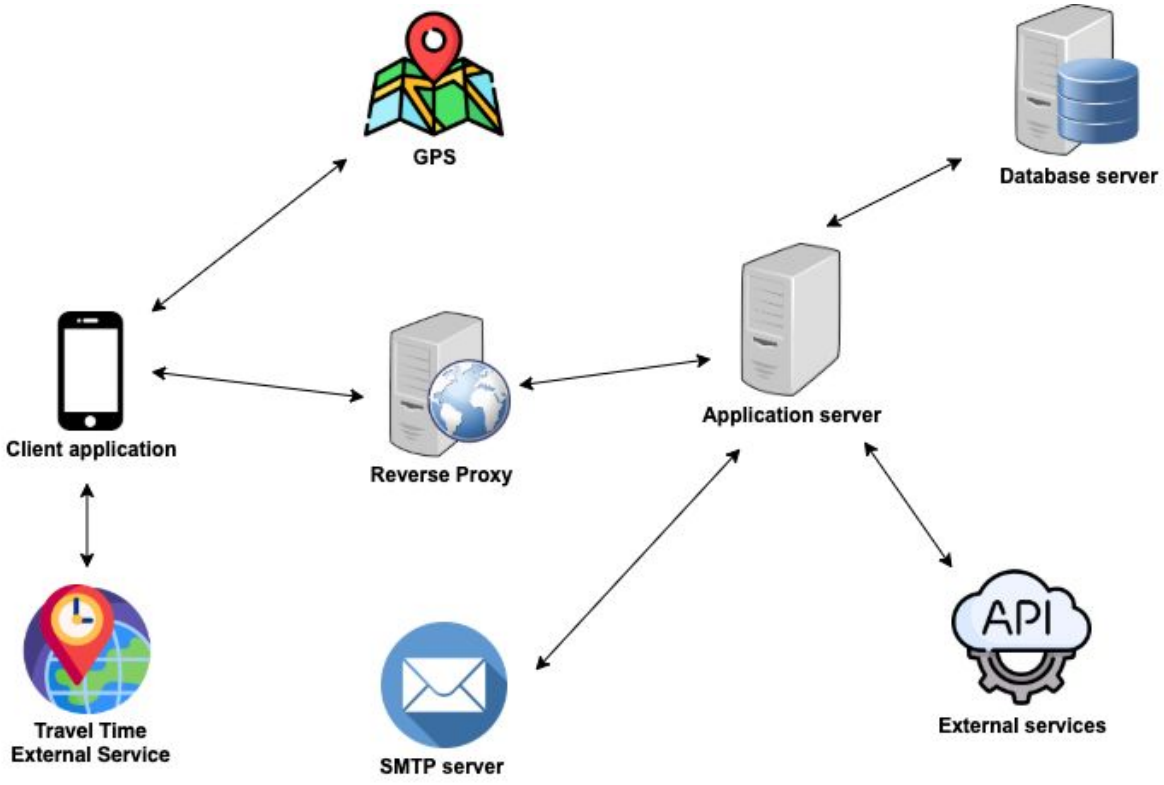
Design Document

Authors:
Cosimo Sguanci
Roberto Spatafora
Andrea Mario Vergani

PAGINE DI COSIMO

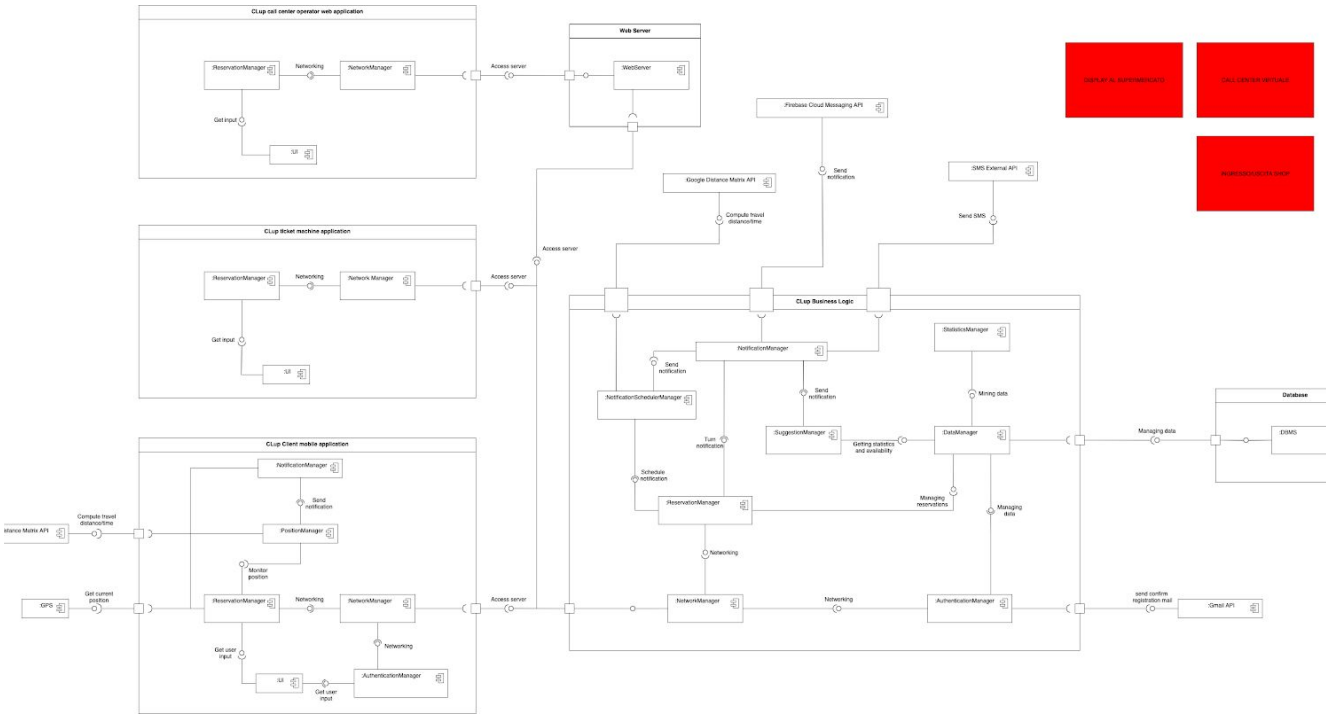
ARCHITECTURAL DESIGN

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, so 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 its favorite slot to go to the grocery shop.

NotificationSchedulerManager
This component keeps 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 always 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.

[TODO: Client Components brief description]

EXTERNAL SERVICES [TODO: 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 [TODO]

Firebase Cloud Messaging API / SMS External API (reference)

These external services both have the same purpose: notify specific users about their reservations. The Firebase 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:

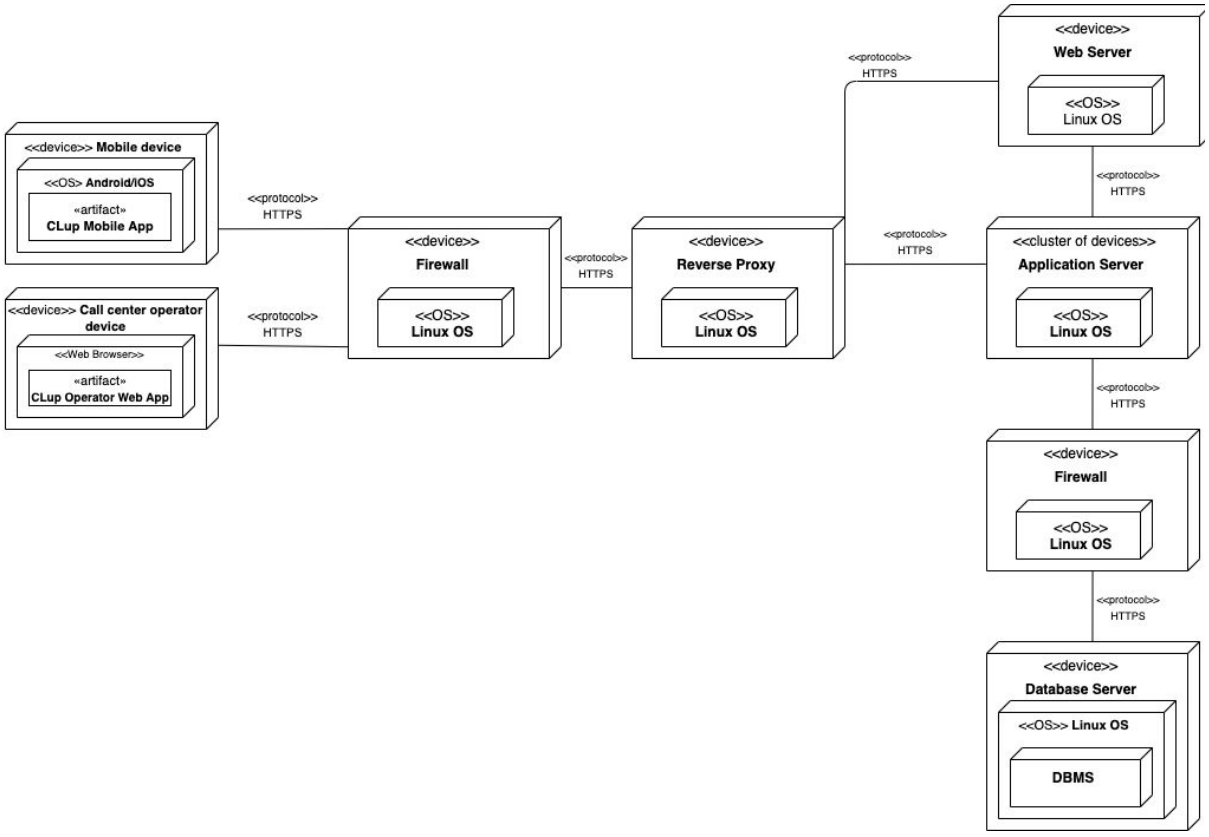
- The user's turn to enter the supermarket has come;
- Based on information provided by the user (location and/or means of transport) the system detected that the customer should leave to get to the grocery shop.

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 though 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 conjuction 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 DMBSs 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 twice: 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 *descriverne l'utilizzo e motivare la scelta*

Firewall *descriverne 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 and the most purchased product's categories (used for provide the best possible recommendations and better allocate time-slots for visits and ticket
- *update global customer behaviors*: the average spent time inside the shop. Moreover, taking into account time and most purchased products, StatisticBuilder is able to generalize consumer habits and better handle 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).
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 (±2 hours)
- same supermarket, same hour and different day (±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) {
    sendNotificationToUser("Your Visit at " + shop + " is in an hour!")
    actualTime = time + 1
    done = false
    while(!done) {
        remainingTime = getCurrentTime() - actualTime
        if(remainingTime < fromMinutesToHours(15)) {
            sendNotificationToUser("Your Visit at " + shop + " is in 15 minutes, please get to the shop!")
            done = true
        }
        else {
            if(computeTravelDistance(getCurrentPosition(), shop.getLocation(), meansOfTransport) >=
              (remainingTime + fromMinutesToHours(5))) {
                sendNotificationToUser("Your Visit at " + shop + " is in " + fromHoursToMinutes(remainingTime) +
                  " minutes, you should leave now!")
                done = true
            }
        }
    }
}
```

REQUIREMENT TRACEABILITY

Business Logic

Reservation manager	
R1 fino a R12: tutti	
R15	

R18	
R20	
R22	
R23	
R24	

Notification manager	
R13	
R14	
R19	
R25	

PROXY	
R16	
R17	
R19	

Suggestion manager	
R25	
R	
R	

Statistics manager

R21	
R25	
R	

Data manager uguale DBMS	
R1-R25 tutti	
R28	

Authentication manager	
R28	

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 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.