POLITECNICO
MILANO 1863

Software Engineering II

# CLup - Customers Line-up

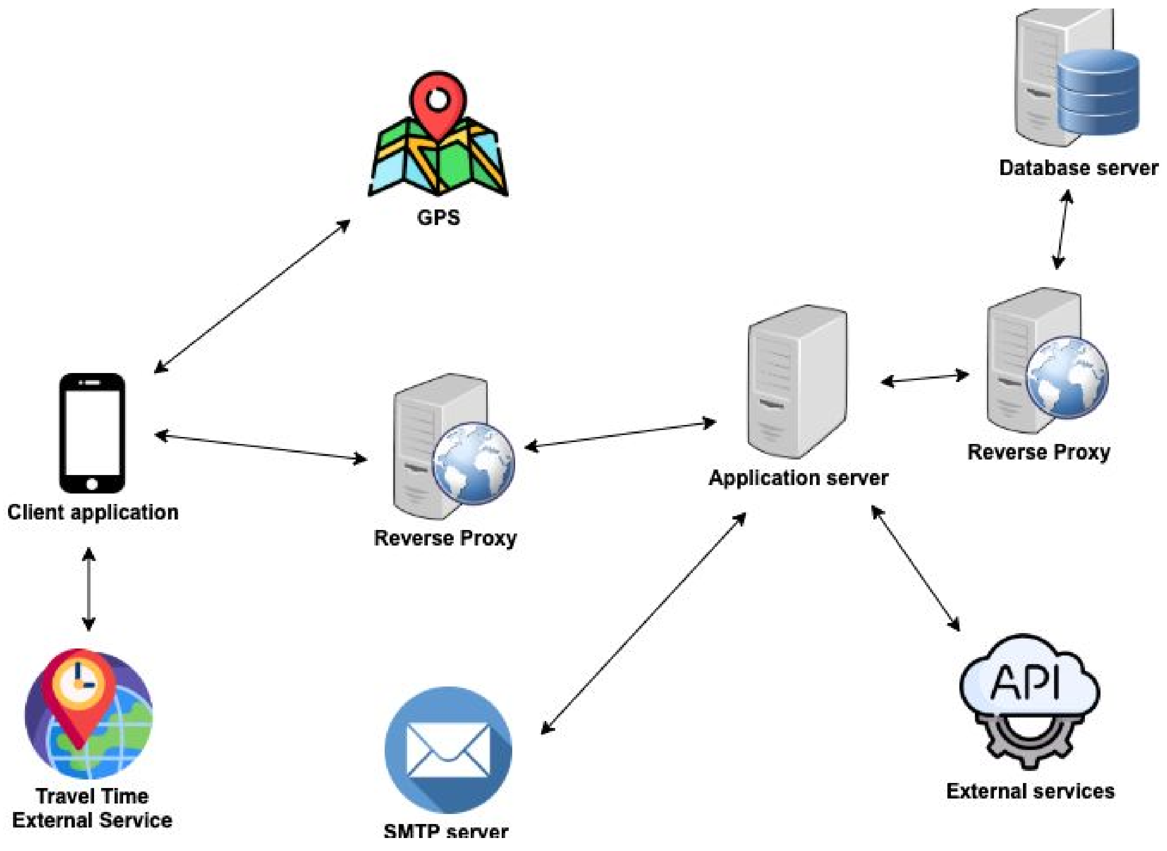
**Design Document**

*Authors:*
*Cosimo Sguanci*
*Roberto Spatafora*
*Andrea Mario Vergani*

**PAGINE DI COSIMO**

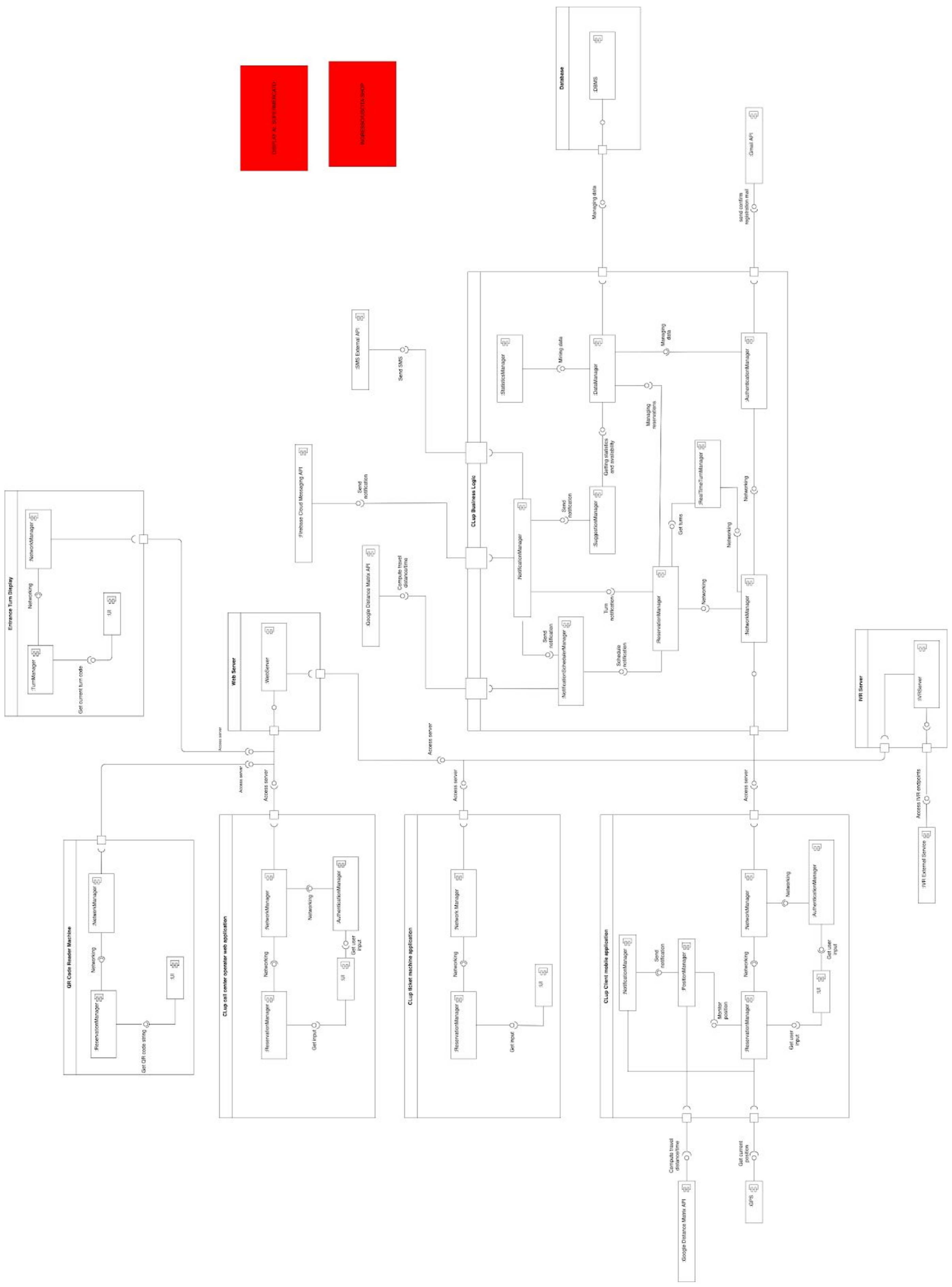
# 2. ARCHITECTURAL DESIGN

## 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, that will be described in detail in the following sections.
Moreover, a Web Server (not modeled here for simplicity) is needed for the clients which are implemented as Web Applications, in order to get the HTML pages.
Another physical distinct node is represented by the IVR Server, needed to successfully integrate with the external IVR service. In this representation it has been merged with the Application Server.

## 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 types of client that can be encountered in the system are described, together with their functional differences. Both the clients typically used by customers (e.g. mobile app, ticket machine) and those needed as utility for the system are included in this view.
In addition to the clients, the presentation layer also uses a Web Server to provide web content (pages and code) to the Web Applications. 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 most of 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.

**RealTimeTurnManager**
The RealTimeTurnManager interacts with the turn displays and the QR code reader machines located at each shop, using the WebSocket protocol to communicate with clients and provide real time updates regarding current turns. In addition, a service is implemented to check if a QR code provided by a user is valid for entrance.

**BoughItemsFetcher**

In order to better implement the statistics builder functionality, it is necessary that the system has continuously updated data about customers coming from the shop's database. To achieve this, the system is meant to periodically interact with the already existing shop's database (REFERENCE ALLA SEZIONE DEL DOCUMENT SCRITTA DA ANDREA).

In particular, the system should be able to build statistics for individuals, based on their shopping habits. Mobile app users, from the moment of registration, have a fidelity code associated with them. The system, by making use of the BoughtItemsFetcher component, will fetch purchased items from the shop's database during the last **period of interaction** days for each fidelity card, in order to associate them with users, and to be able to show different available reservation slots, taking into account the customer purchase habits and trying to avoid crowds in the same area of the shop.

**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.

**IVR Server**
The interactive voice response (IVR) Server represents the interface offered to allow call center users to make and update their reservations by interacting with virtual call center operators. For this purpose, *CLup* makes use of an external IVR service, which will maintain the physical infrastructures needed in order to accept incoming calls from users. The external service will be configured to forward user's inputs to the IVR Server, which will examine the response in order to correctly handle the flow of the operation that has been requested by the user.

**CLup Client Components**
Speaking about the client-side architecture of the system, all the applications offered to the customers share a common structure, composed of a **ReservationManager**; a **NetworkManager** and a component responsible to build and show 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.

The following sections describe the clients that are not used directly by customers to book reservations, but are still necessary to show current turns (maintaining a queue for ticket machine users) and to allow users whose turn has come to enter the shop.

## QR code reader machines

The customer flow in and out of the shop has been regulated through the use of QR code reader machines. These machines are placed at the entrance and at each cash register of any shop that uses the CLup system. From the moment in which users are notified for entrance (their reservation code is ready to be scanned), customers have a limited period of time in which they can get in.

The use of QR code reader machines allows the system to regulate the flow of entrances into the shop. These machines need to know whose turn it is, in order to accept people in turn reservations. Moreover, those machines would allow the system to implement the important functionality of building statistics about time, both for individuals and general ones.

QR code readers are meant to be implemented as a web application, used to interact with the application server. The interaction takes place through an indirect real-time communication using the WebSocket protocol (reference), connecting the reader web app with the web server, which acts as a proxy between the two (QR code reader and application server). The interaction with the application server allows the QR code readers to know what are the QR codes allowed for entrance and then update the reservation status.

Entrance QR code readers would change the reservation status from PLANNED to ACTIVE and then the customer will be able to enter the shop (in the case in which the user does not approach the shop’s entrance in time, the reservation status would be set to EXPIRED).

Exit QR code readers are placed at each cash register of the shop. The customers will be asked to scan their reservation code immediately before making the payment. The interaction with the application server would change the reservation status from ACTIVE to EXPIRED.

Status changes monitoring is used for managing entrances in case of full shop.

**Display**
Display is the component which allows ticket machine users waiting for their turn (outside the grocery shop) to monitor entrances and receive notifications when it is time to get in. It is basically a client that shows a web page on the screen: a new page is sent by the WebServer everytime a customer is called for entrance.
Because of possible frequent updates of the displayed content, the connection between Display module and WebServer is based on WebSocket communication protocol (so, not HTTP): this choice goes in the direction of efficiency because it allows to have a persistent and real-time channel, with a lower overhead and without the need to open the connection again multiple times.
In practice, ReservationManager (component which handles entrances and availability of shops) makes a proper call to notify that a ticket machine user's turn has come; this information reaches (after a set of proper calls) the WebServer, which sends to the Display the web page to show; finally, the content is shown thanks to Display's web browser.

## EXTERNAL SERVICES

**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.

**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.

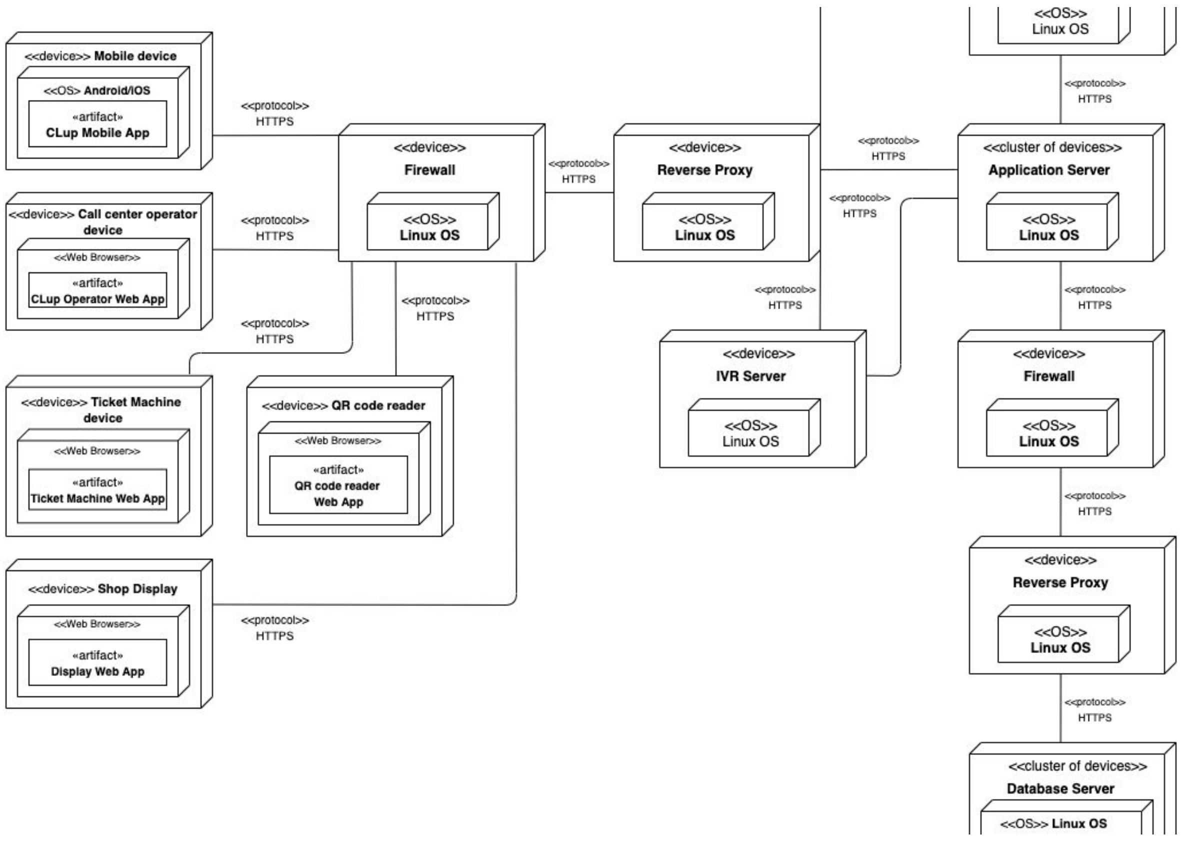
In addition, the system schedules notification at predefined times to remind users about their upcoming reservations.

**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.

**External IVR Service**
*CLup* also offers the possibility to make and delete reservations using a virtual call center, making use of an IVR system. To implement this feature, faced with a choice of *make or buy*, the latter was found to be the most convenient solution. What will have to be implemented is a set of endpoints (located at the IVR Server) that will be used by the external IVR service, to which the management of the physical infrastructure of the call center is delegated. Following this path, customer's responses will be directly sent to our server, which will handle the operations flow and will interact with the other components of the business logic.

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.

- **Ticket Machine device**
  This device is used by ticket machine users to get grocery shop tickets directly in place (if available).

- **Shop Display**
  The display is used by ticket machine users to monitor turns in real time, in order to know when it's time to approach the shop's entrance.

- **QR Code Reader**
  This device is placed at each shop entrance and at each cash register to be able to allow the users that are currently in turn to enter the shop. In addition, it's up to this machine to update the user's reservation status when it gets in and out of the shop.

- **Firewall**
  The Firewall is used to filter packets sent from the Internet to CLup backend services in order to protect both the Application Server and the Database Server.

- **Reverse Proxy**
  The Reverse Proxy component has mainly two purposes:
  - increase availability and performance of the system by acting as a load balancer (it distributes the load across the replicated back-end services);
  - increase security by hiding back-end physical nodes identity.

- **Web Server Cluster**
  The Web Server cluster is used by the Web Applications involved in the system (call center operator, QR code reader, shop display) 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 Cluster**
  This is the set of replicated devices containing the core data that the CLup system must store and use to offer its service to users. A Linux OS is installed, 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

### *2.G.1. 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, the server business logic and the database server are replicated: in case of problems to one of the units, the others can still serve clients and provide all the application services. The system can be implemented as a componentized application using an *elastic infrastructure,* to be able to scale horizontally and also achieve fault tolerance.
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.

### *2.G.2. Reverse Proxy*

A reverse proxy (e.g. *Ngnix*) is placed in the system as a common point of access to the backend services. Its main tasks are:

- hiding the identity of the physical nodes that make up the backend, thus improving security;
- implementing an *elastic load balancer* to distribute the workload to the cluster of nodes, regarding the Web Server, the Application Server and the Database Server.

### *2.G.3 Firewall*

Firewalls in the system are placed both between the reverse proxy and the public network (*Internet*) and behind the Database Server. In both cases the tasks that the firewall must fulfill are:

- implementing anti-DDOS policies to incoming network packets;
- filter packets based on source IPs, this is particularly important in the context of the Database Server: only whitelisted IP addresses (those corresponding to the Application Server nodes) are allowed to make requests and get responses;
- filter packets to allow only valid requests that can be made to the Application Server and the Web Server.

### *2.G.4. Integration with grocery shops' systems*

The part about integration with an external system (owned by grocery shops) is certainly critical: the best design choice is the one which does not alter the existing technology and application, but integrates it by adding the fewest possible set of interfaces and/or functionalities.

*Customers Line-up* needs to integrate with the grocery shops' systems in order to retrieve information about purchased products; in fact, this knowledge is an important part connected with statistics and safety improvement (through proper reservations management).

Following best-practises, *CLup*'s integration with the supermarket chain database is as "transparent" as possible. First of all, an important remark is that this solution works under the following assumptions:

- the grocery store chain allows customers to have a fidelity card
- the grocery store chain has a DB which periodically registers products bought by every customer (identified by the fidelity card code, if shown) in every supermarket; data in DB may also be periodically deleted
- every mobile app user has a fidelity card (already explained in RASD)

The first two assumptions cover most of the cases of nowadays supermarket chains. The presence of a database is extremely likely in order to monitor daily incomes, sold products, … (in practice, only very small traditional supermarkets, mainly chain independent, do not have it; but probably this category is not interested in *CLup*); also assuming a centralized DB for the whole chain is very reasonable, even if not mandatory. Instead, for what regards the presence of the fidelity card, most of the grocery stores have it.

If one of the two assumptions is not respected, an ad-hoc solution is about to be adopted. In fact, the integration part highly depends on the legacy system and so it can not be totally general; however, *CLup*'s main idea works well in most of the cases, as explained before.

The point is that, since grocery store systems already store information about products bought by customers, it is not necessary for *Customers Line-up* to implement this functionality from scratch: it would be very expensive, invasive and not necessary. The idea is that *CLup* periodically retrieves data from chain's DB (the "period" depends on how frequently the chain updates and deletes information on its database itself): in this way, the application can automatically know which products mobile app users (identified by fidelity card code) have bought and can later build statistics. This design strategy allows to interfere at the minimum level with the already existing grocery store system: the registration of products, frequency of updates and deletions, … are completely unaffected; it is only necessary to add an interface for queries with *CLup* architecture.

## 2.G.5. 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.5.1. 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.5.2 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 (±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.5.3 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 | |

| | |
|---|---|
| **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 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.

## 2.E Component interfaces

This section shows and explains the interface methods between different components of *CLup* system. The described interfaces represent the most useful and essential methods to handle the most important calls and information flow; however, it is not excluded that some other interface methods may be added in later phases of development, for different reasons.

Another important aspect is about data types: the diagram shows them, but in some cases they are only suggestions and may be improved for the sake of efficiency and reusability (code optimization, design patterns, best-practises, …). All data types present in the diagram refer to the ones defined in *Requirements Analysis and Specification Document*, Section 2.A.2 (“UML Class Diagram”); however, some exceptions are, for example, *Date* and *Time*, which are trivially supposed to already be provided by some library. In some other cases, data types are not specified, because the decision is better to be taken during a later phase (for example, the type of data returned by *DataManager.getDataForMining()*).
Component interfaces diagram is depicted below.
A description of interfaces presented in the diagram, divided by component, follows.

INSERIRE IMMAGINE

***CLup* business logic**

- **NetworkManager**
  - ***manageHTTPRequest***: manages an input HTTP request
  - ***sendHTTPRequest***: sends an HTTP request
- **AuthenticationManager**
  - ***login***: manages a login event
  - ***signUp***: registers a user to *CLup*
  - ***logout***: handles a logout request
- **ReservationManager**
  - ***bookVisit***: manages a “book a visit” request
  - ***getTicket***: manages a “get a ticket” request
  - ***deleteReservation***: manages a “delete a reservation” request
  - ***confirmVisit***: books a visit that was proposed to the user (and it was temporarily “blocked”, so not reservable by any other customer)
  - ***confirmTicket***: gets a ticket that was proposed to the user (and it was temporarily “blocked”, so not reservable by any other customer)
  - ***discardVisit***: “frees” a “blocked” visit which had been proposed to a user (the visit was temporarily “blocked”, so not reservable by any other customer)
  - ***discardTicket***: “frees” a “blocked” ticket which had been proposed to a user (the ticket was temporarily “blocked”, so not reservable by any other customer)
  - ***save***: saves the phone number associated to a reservation (referring to call center service)
  - ***getTurn***: returns the current turn
- **NotificationSchedulerManager**
  - ***handle***: manages all activities connected to notifications for a reservation (notifications in predefinite times, notifications based on actual position, …)

- ○ ***handleOnlyPredefinite***: manages activities connected only to predefinite notifications (not related to user's position), with reference to a reservation (useful when more "complex" notifications are handled client-side, but the server still sends predefinite notifications)

- **NotificationManager**
  - ***notifySuggestion***: notifies a mobile app user about an "appealing" slot
  - ***notifyTurn***: notifies a user that his turn has come (with relation to a reservation)
  - ***notifyPredefinite***: notifies a user about in predefinite times, or also about position
  - ***sendSMSNotification***: notifies a user that his turn has come via SMS (with relation to a reservation)
- **DataManager**
  - ***getGroceryShopList***: returns the list of all grocery shops managed by *CLup*
  - ***checkForAvailability***: checks availability for given parameters connected to a reservation, and in case returns the available reservation
  - ***addReservation***: confirms to the DB the fact that a reservation (given as parameter) has been booked
  - ***deleteReservation***: confirms to the DB the fact that a booked reservation (given as parameter) has been "released" by the user
  - ***save***: saves user's preferences connected to a reservation (booked or only looked for)
  - ***getVisitSuggestion***: returns suggestions related to an unavailable visit (after the latter's request)
  - ***authenticate***: authenticates a user
  - ***register***: registers a user
  - ***setUserToken***: saves in the DB the token assigned to a client
  - ***deauthenticate***: deauthenticates a user
  - ***getFirstAvailableTicket***: returns the first available ticket for current day and given grocery shop
  - ***confirmVisit***: saves in the DB the fact that a visit that was proposed to a user (and it was temporarily "blocked", so not reservable by any other customer) is booked
  - ***confirmTicket***: saves in the DB the fact that a ticket that was proposed to a user (and it was temporarily "blocked", so not reservable by any other customer) is booked
  - ***discardVisit***: "frees" a "blocked" visit which had been proposed to a user (the visit was temporarily "blocked", so not reservable by any other customer); after the call, the visit can be reserved by any user that asks for it
  - ***discardTicket***: "frees" a "blocked" ticket which had been proposed to a user (the ticket was temporarily "blocked", so not reservable by any

other customer); after the call, the ticket can be reserved by any user that asks for it
    - ***getFirstAvailableVisit***: returns the first available visit for the given grocery shop
    - ***discard***: “frees” a collection of “blocked” visits which had been proposed to a user (the visits were temporarily “blocked”, so not reservable by any other customer); after the call, the visits can be reserved by any user that asks for them
    - ***checkVisit***: checks availability for a given visit request and returns a collection of possible “appealing” visits (given user’s search)
    - ***checkTicket***: checks availability for a given ticket request and returns the first available ticket
    - ***savePhone***: stores in the DB the phone number associated to a reservation (referring to call center service)
    - ***getDataForMining***: returns all data that can be useful for mining activity
    - ***writeStatistics***: writes statistics in the DB
    - ***getDataForStatistics***: returns all data that can be useful for building statistics activity
    - ***addBoughtItems***: adds to the DB information about items bought by a given user in a certain date and time

**Database**

- **DBMS**
    - ***manageSQLRequest***: handles and executes a SQL request

***CLup* Client mobile application**

- **NetworkManager**
    - ***manageHTTPRequest***: manages an input HTTP request
    - ***sendHTTPRequest***: sends an HTTP request
- **AuthenticationManager**
    - ***login***: manages a login request
    - ***signUp***: manages a registration request
    - ***logout***: manages a logout request
- **ReservationManager**
    - ***bookReservation***: manages a reservation request
    - ***deleteReservation***: manages a request for deleting a reservation
- **PositionManager**
    - ***monitorPosition***: monitors user’s position in order to be able to send proper notifications (with relation to a reservation)
- **NotificationManager**

    - ***notify***: notifies a user about an event connected to a reservation

***CLup*** **ticket machine application**

- **NetworkManager**
    - ***manageHTTPRequest***: manages an input HTTP request
    - ***sendHTTPRequest***: sends an HTTP request
- **ReservationManager**
    - ***bookReservation***: manages a reservation request
    - ***deleteReservation***: manages a request for deleting a reservation

***CLup*** **call center operator web application**

- **NetworkManager**
    - ***manageHTTPRequest***: manages an input HTTP request
    - ***sendHTTPRequest***: sends an HTTP request
- **AuthenticationManager**
    - ***login***: manages a login request
    - ***logout***: manages a logout request
- **ReservationManager**
    - ***bookReservation***: manages a reservation request
    - ***deleteReservation***: manages a request for deleting a reservation

**WebServer**

- **WebServer**
    - ***manageHTTPRequest***: manages an input HTTP request
    - ***sendHTTPRequest***: sends an HTTP request

**QR Code Reader Machine**

- **NetworkManager**
    - ***manageHTTPRequest***: manages an input HTTP request
    - ***sendHTTPRequest***: sends an HTTP request
- **ReservationManager**
    - ***checkValidity***: checks entrance code’s validity

**Entrance Turn Display**

- **NetworkManager**
    - ***manageHTTPRequest***: manages an input HTTP request
    - ***sendHTTPRequest***: sends an HTTP request

- **TurnManager**
    - ***getTurn***: get current turn “queue” number for a given grocery store

**IVR Server**

- **IVRServer**
    - ***manageHTTPRequest***: manages an input HTTP request
    - ***sendHTTPRequest***: sends an HTTP request

# 1 Introduction

## 1.A Purpose

*Customers Line-up* is an application thought for booking slots and visiting grocery stores. Its necessity arises in an epidemic situation, in which people's primary need is that of being as safe as possible (mainly with respect to the risk of crowding and consequent higher probability of disease transmission) in all their everyday activities.
In *Requirements Analysis and Specification Document*, *CLup*'s main identified goals are those of avoiding crowds inside and outside the supermarkets, managing entrances in them and guaranteeing a "transparent" customer experience (even in a pandemic context). In order to ensure the goals, several features of the software system have been listed and discussed.
Of course, all system's properties must reflect in some physical components (hardware and software), which are able to guarantee *CLup*'s appropriate functioning.
In the following sections, a complete description of the architecture chosen for the application is presented.

## 1.B Scope

In order to satisfy the goals and respecting the needs of all classes of customers, three channels between users and *CLup* have been identified in RASD: mobile application (available for smartphones and tablets), call center (with IVR or the possibility to talk to an operator) and ticket machine (placed outside a grocery store).
The main functionalities that the software system offers to the customers are those of getting a ticket for a supermarket in the current day, booking a visit* (for current day or any day in the following month) and deleting a reservation.
*CLup* mobile application, thanks to the intrinsic nature of nowadays apps, can offer more functionalities to its users: suggestions for "appealing" slots based on customized data analysis and statistics, alternatives in case of unavailability while booking, notifications for approaching turns (also basing on GPS, in order to sensibly decrease the probability of being late for a visit). Call centers, instead, offer basic functions, with the addition of some notifications via SMS; they represent a fallback option for customers who do not have a smartphone or tablet. Ticket machines functionalities, instead, are the most basic ones: in fact, an adopted domain assumption is that "only a very little part of customers uses ticket machines for reservations, because mobile application and call center guarantee a more comfortable service accessible to almost everyone" (DA6 discussed in RASD).
Besides the user application, the system is composed of the logic that manages all the functions (from getting tickets to sending notifications) and the part about data. Another aspect is that connected with entrances: shops' doors must be equipped with devices that allow customers to get in only if they have a reservation in a proper state.

** ticket machines have a limitation for what concerns the function of booking a visit: it is only possible when no tickets are available for current day and the only available visit is the next one*

## 1.C Definitions, Acronyms, Abbreviations

### 1.C.1 Definitions

- **Customer:** a person who does/is going to do the grocery shopping.
- **Ticket machine:** a machine equipped with a touchscreen display, a printer system, a QR code reader and an ad-hoc version of *Customers Line-up* application.
- **User:** a person who has downloaded *Customers Line-up* mobile application on his smartphone/tablet **OR** a person who uses *Customers Line-up* services through a ticket machine **OR** a person who uses *Customers Line-up* services by calling the call center.

### 1.C.2 Acronyms

- **CLup:** *Customers Line-up*
- **DB:** *Database*
- **DBMS:** *Database Management System*
- **DD:** *Design Document*
- **GPS:** *Global Positioning System*
- **HTTP:** *Hypertext Transfer Protocol*
- **HTTPS:** *Hypertext Transfer Protocol over Secure Socket Layer*
- **IVR:** *Interactive Voice Response*
- **QR:** *Quick Response*
- **RASD:** *Requirements Analysis and Specification Document*
- **SIM:** *Subscriber Identity Module*
- **SMS:** *Short Message Service*
- **UI:** *User Interface*
- **UML:** *Unified Modeling Language*

### 1.C.3 Abbreviations

| | | |
|---|---|---|
| **DAn** | Domain assumption number n | *Defined in RASD, section 2.D.1* |
| **UCn** | Use case number n | *Defined in RASD, section 3.B.2* |
| **Rn** | Requirement number n | *Defined in RASD, section 3.B.4* |

## 1.D Revision history

| Version | Date | Authors | Summary |
|---|---|---|---|
| 1.0 | 10/01/2021 | Cosimo Sguanci, Roberto Spatafora, Andrea Mario Vergani | First release |

## 1.E Reference Documents

- Software Engineering 2 slides (available on the Beep page of the course)
- Project assignment document ("R&DD Assignment A.Y. 2020-2021.pdf" available on the Beep page of the course)
- DDs developed by colleagues of past years (available on the Beep page of the course or on GitHub)

## 1.F Document structure

➢ **Section 1** is a summary of *Customers Line-up*'s goals and functionalities, coming from RASD. In addition, all released versions of this document are listed in an appropriate paragraph.

➢ **Section 2** provides a detailed description of the architecture of the system, with the aid of proper UML diagrams. The analysis is both hardware and software, with components, design patterns and most important algorithms specifications.

➢ **Section 3** presents *CLup* mockups with the purpose of showing the user interface of the application.

➢ **Section 4** describes the mapping between requirements (identified in RASD) and architectural components, thus enforcing the relationship between functionalities and design choices.

➢ **Section 5** identifies the implementation, integration and testing strategy for the system, with a particular focus on the order of components to follow while developing and testing.

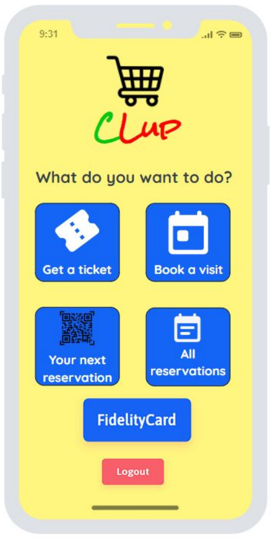
➢ **Section 6** summarizes the total effort spent for realizing the *Design Document* by each group member.

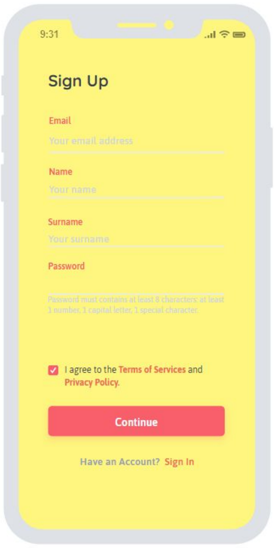
➢ **Section 7** lists all references that helped the team during analysis and document writing.

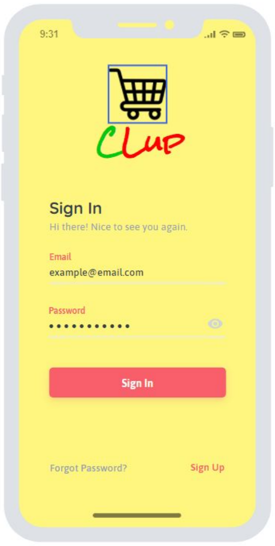
# 3 User interface design

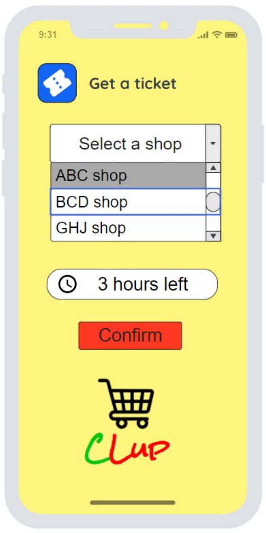
This section aims at providing an overview of how *CLup* user interface looks like, for the various clients (mobile app, ticket machine and call center, under call center's operator point of view).
Some of the presented mockups have already been shown in RASD (section 3.A.1); for this reason, they are simply referred to, without any further explanation. All extensions, instead, are described in the following subsections.

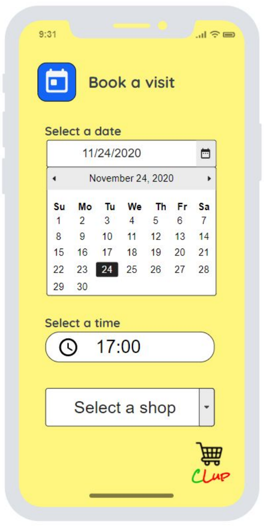
## 3.A *CLup* mobile application interface

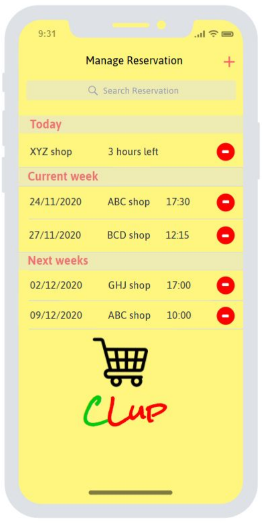



9:31
Sign Up
Email
Your email address
Name
Your name
Surname
Your surname
Password
Password must contains at least 8 characters: at least 1 number, 1 capital letter, 1 special character.
I agree to the Terms of Services and Privacy Policy.
Continue
Have an Account? Sign In









Today

| | | |
|---|---|---|
| XYZ shop | 3 hours left | |

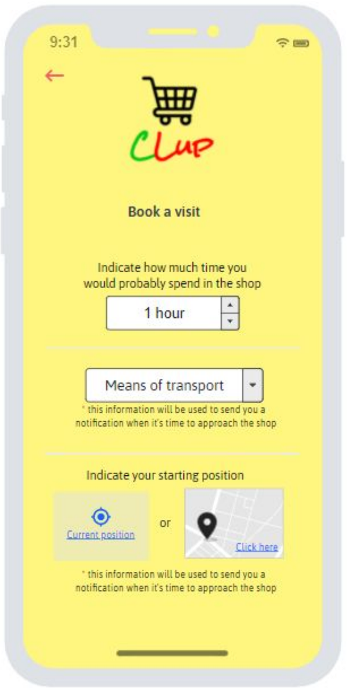
Current week

| | | |
|---|---|---|
| 24/11/2020 | ABC shop | 17:30 |
| 27/11/2020 | BCD shop | 12:15 |

Next weeks

| | | |
|---|---|---|
| 02/12/2020 | GHJ shop | 17:00 |
| 09/12/2020 | ABC shop | 10:00 |

These six mockups have already been described in RASD (section 3.A.1); the suggestion is to directly refer to the description contained in *Requirements Analysis and Specification Document*.



Book a visit

Indicate how much time you would probably spend in the shop

1 hour

Means of transport

* this information will be used to send you a notification when it's time to approach the shop

Indicate your starting position

Current position

or

Click here

* this information will be used to send you a notification when it's time to approach the shop

- Optional additional information (book a visit)

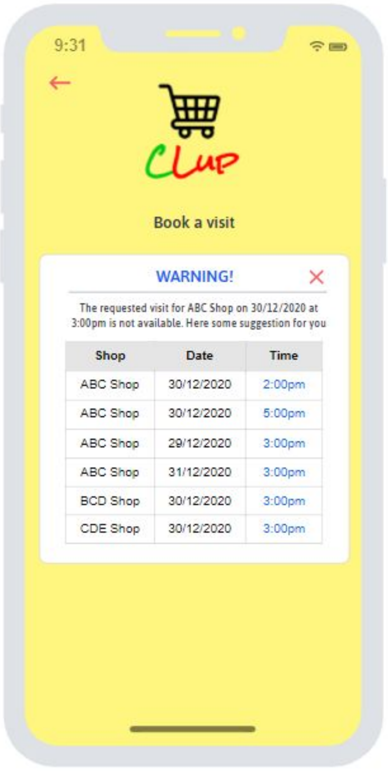
This page is shown after a mobile app user decides to book a visit. It is known that *CLup* allows him to insert additional information, in order for notifications (user-side) and other reservations management (system-side) to be more effective.

The page offers the possibility to insert the expected duration of the visit, the customer's starting position and the means of transport. This information is optional, so the user is free to fill the fields completely or partially; in fact, his reservation has already been confirmed.

Another additional information is the categories of products the customer is going to buy; the page for it is shown immediately after the presented one (it is not shown here for the sake of brevity, but it is similar to this one).

- Alternative slots suggestion after failure in booking a visit

When a "book a visit" request fails because of shop unavailability, the system proposes to mobile app users some alternatives: the idea is to suggest similar slots (similar day, similar time or close shop, always fixing two variables and varying the third).
This mockup shows the situation: a warning message tells the user about unavailability and a list of proposals is presented.

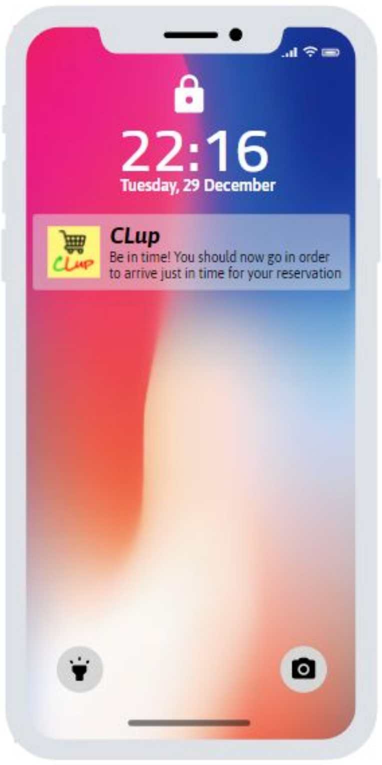


Book a visit

WARNING!

The requested visit for ABC Shop on 30/12/2020 at 3:00pm is not available. Here some suggestion for you

| Shop | Date | Time |
|---|---|---|
| ABC Shop | 30/12/2020 | 2:00pm |
| ABC Shop | 30/12/2020 | 5:00pm |
| ABC Shop | 29/12/2020 | 3:00pm |
| ABC Shop | 31/12/2020 | 3:00pm |
| BCD Shop | 30/12/2020 | 3:00pm |
| CDE Shop | 30/12/2020 | 3:00pm |



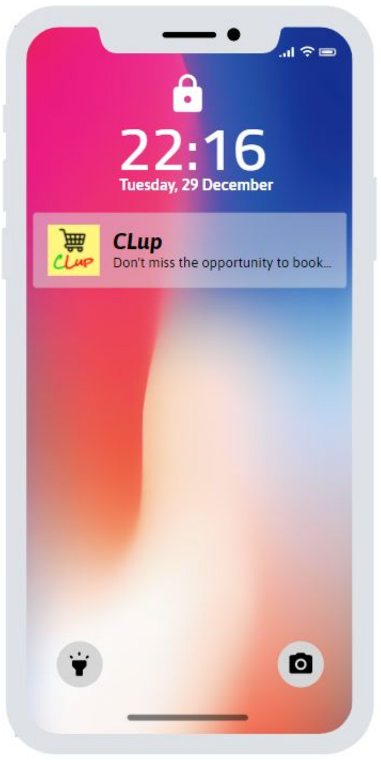
CLup: Be in time! You should now go in order to arrive just in time for your reservation

- Notification for approaching the grocery shop

*CLup* sends proper notifications to mobile app users, in order to help them not to arrive late/miss a reservation. The notification system takes advantage of the user's declared position and means of transport; alternatively, GPS is used for tracking the customer.
The mockup shows the notification that reminds to approach the grocery shops, because the turn is coming.
Similar notifications are also sent in predefinite times (just as reminders) and in order to enter the supermarket (when the turn comes). They are not shown here, because they are completely analogous to the presented one.

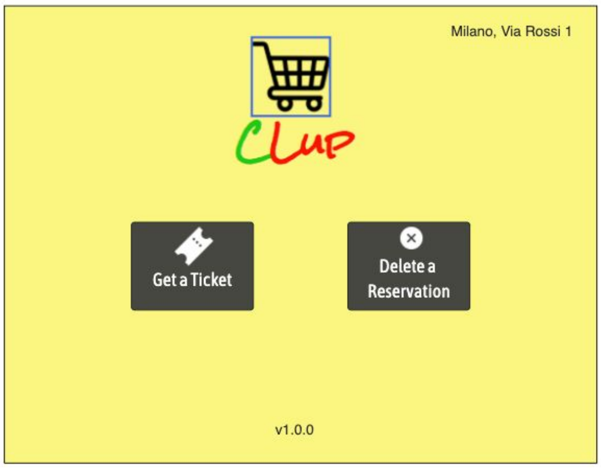
- Suggestion: an “appealing” slot is available

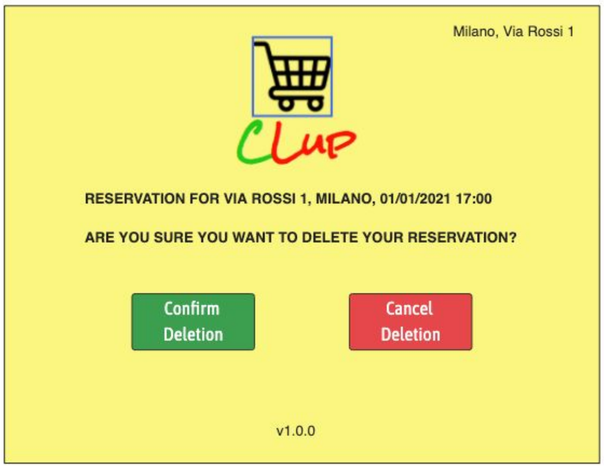
*CLup* infers mobile app users' habits and proposes them some “appealing” slots (in advance) when they are available.
The mockup shows the notification about this suggestion.



CLup: Don't miss the opportunity to book...

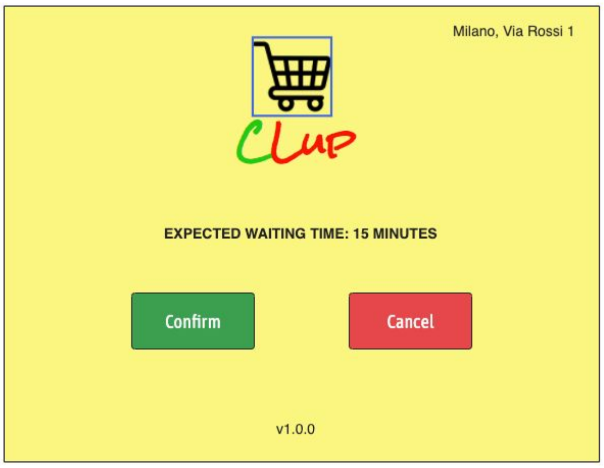
## 3.B *CLup* ticket machine interface



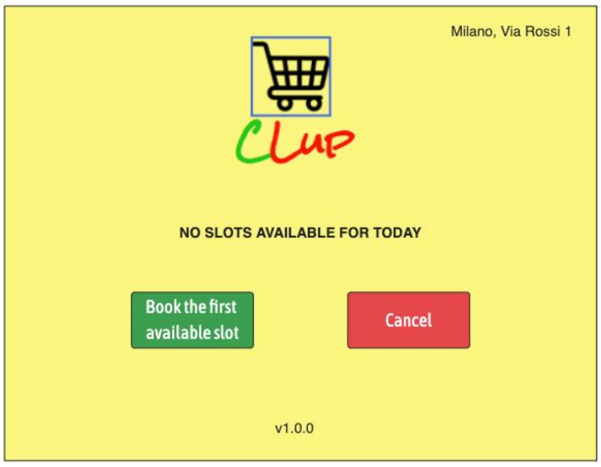


RESERVATION FOR VIA ROSSI 1, MILANO, 01/01/2021 17:00

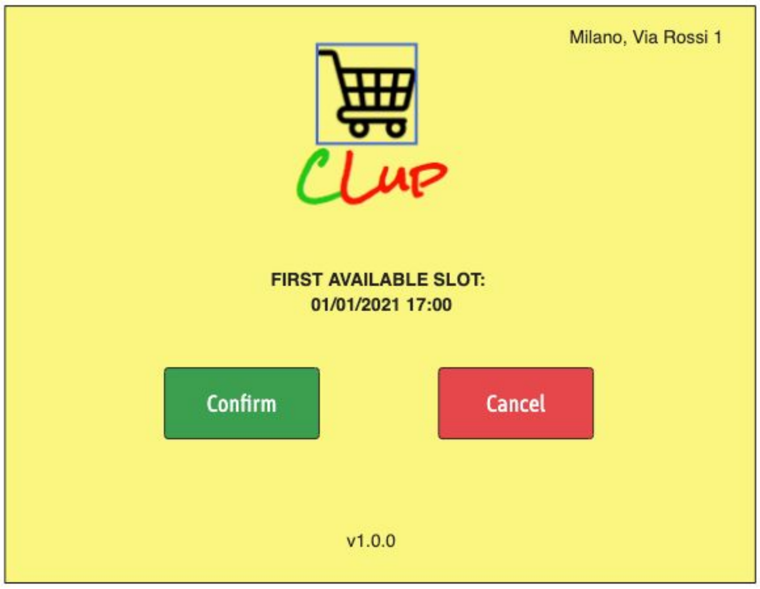
ARE YOU SURE YOU WANT TO DELETE YOUR RESERVATION?



EXPECTED WAITING TIME: 15 MINUTES



NO SLOTS AVAILABLE FOR TODAY

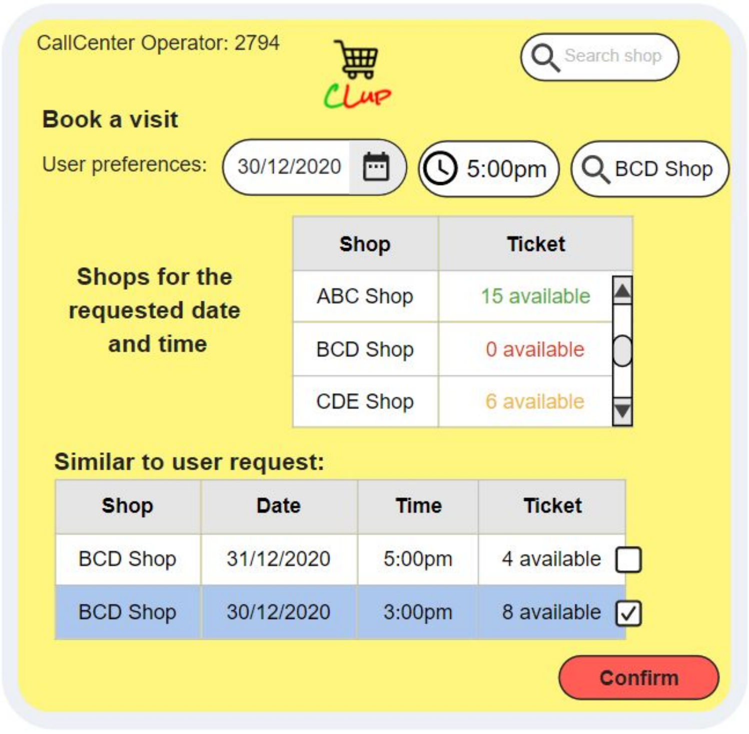


FIRST AVAILABLE SLOT:
01/01/2021 17:00

The presented mockups have already been presented in RASD (section 3.A.1.2).

## 3.C *CLup* call center interface

This section presents *CLup* WebApp interface used by call center operators. In fact, for what regards customers, they simply interact with the system through phone calls.



| Shop | Ticket |
|---|---|
| ABC Shop | 15 available |
| BCD Shop | 0 available |
| CDE Shop | 6 available |

| Shop | Date | Time | Ticket |
|---|---|---|---|
| BCD Shop | 31/12/2020 | 5:00pm | 4 available ☐ |
| BCD Shop | 30/12/2020 | 3:00pm | 8 available ☑ |

- Book a visit

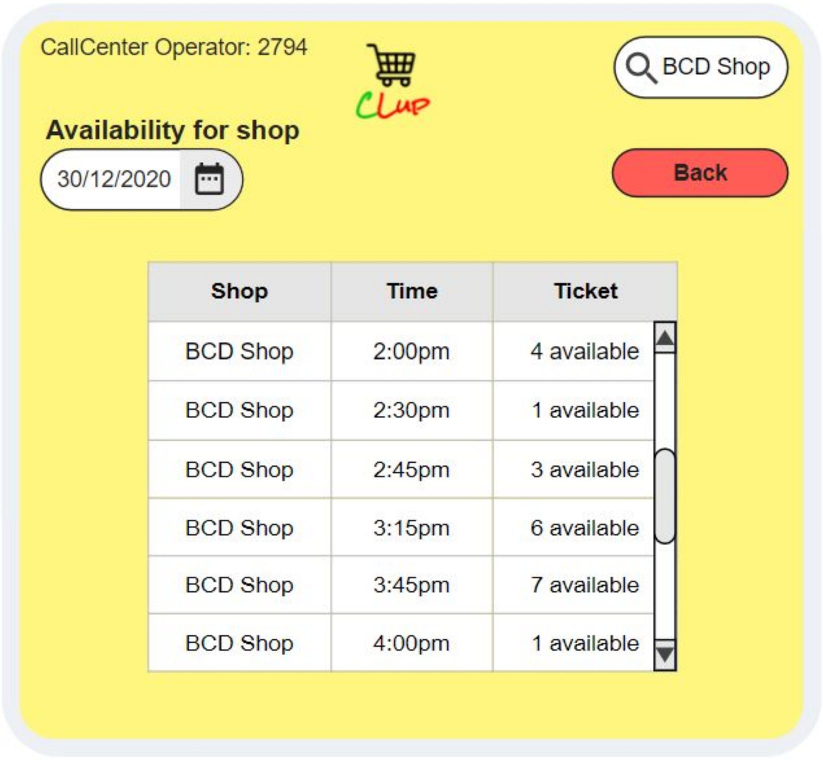
A logged call center operator can book visits, following customers' requests during phone calls.

The operator can set customer preferences (day and time for the visit, grocery shop). In a proper web page, details for the requested reservation appear: availability for requested visit, options in case of unavailability, …

An important remark is that a call center operator has at his disposition more information than a "normal" user: for example, he can see the number of available slots for every grocery store.

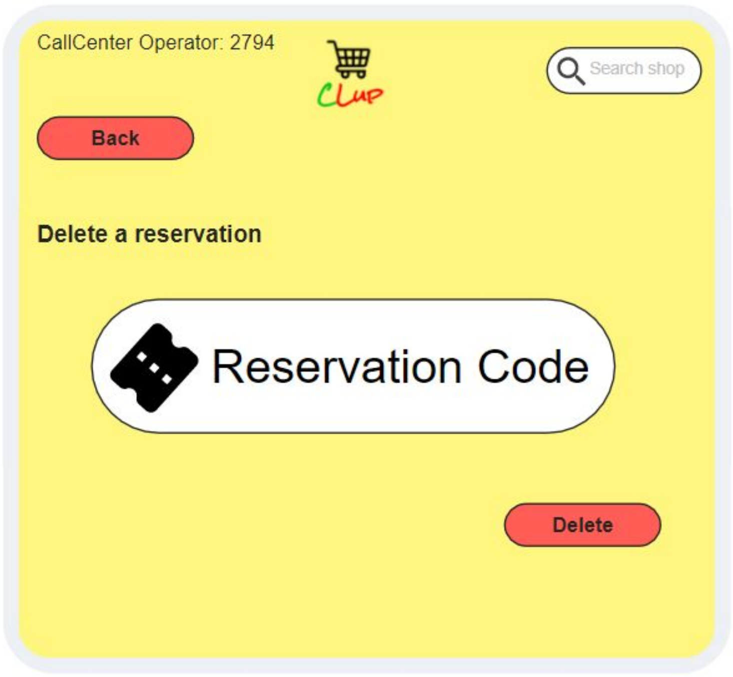
- Shop availabilities

Going back to the fact that a call center operator has more options than a "normal" *CLup* user, here another example is presented.
The web page shows all the availabilities for the selected shop and date. This feature is important for the call center operator in order to help as much as possible a call center user: in fact, users calling the call center primarily interact with the IVR system; if they switch to the operator, they are



| Shop | Time | Ticket |
|---|---|---|
| BCD Shop | 2:00pm | 4 available |
| BCD Shop | 2:30pm | 1 available |
| BCD Shop | 2:45pm | 3 available |
| BCD Shop | 3:15pm | 6 available |
| BCD Shop | 3:45pm | 7 available |
| BCD Shop | 4:00pm | 1 available |

probably in difficulty and need more precise help for booking the reservation they really want.



- Delete a reservation

This web page simply allows a call center operator to delete a reservation.
By inserting the reservation code into a proper field and confirming the deletion process, the reservation is cancelled.

***Remark***

In the first two call center mockups, the word “Ticket” refers to available slots: of course, saying that there are 3 available “tickets” at 2:45 pm can only have this meaning.

In many parts of the document (and in RASD, too), the word “Ticket” assumes the technical meaning of a reservation at the end of the waiting “queue”, in contrast to a “Visit” (which, instead, has a fixed date and time). However, availabilities for specific dates and times can be checked only for what regards reservation slots; so, in the mockups there is an abuse of the technical word “Ticket” meaning the slots, which is however evidently self-explanatory considering *CLup* features: call center operators do not get confused by this situation, but on the contrary the word “Ticket” seems to be more “familiar” (of everyday life context) to mean exactly available slots.

# 5 Implementation, Integration and Test plan

## Implementation plan

The implementation plan of the *CLup* system will follow a ***bottom-up*** approach, in order to take into account the dependences among the components of the same subsystem. Thus, the implementation process will follow a “step-by-step” approach. “Step-by-step” approach is based on a gradual approach, consisting of different steps: it starts by implementing, in any system’s machine, the *core* components (those that do not depend on other components’ interfaces) going, in the successive steps, towards the implementation of the others dependent component (those that need the interfaces offered by other components, already developed).

The implementation of these other components will also follow the approach described: from the more independent to the less independent ones.

The reported approach allows to start the development process in parallel on all the different machines involved in the system. Thus, it is important to note that even in the case of two dependent components belonging to different machines, the implementation with this approach would not be affected thanks to the possibility of using mocked data (useful to test the interaction that two different machines’ components would have even if one of the two has not been implemented yet; this type of test would be better explained in the next section).

***Assumption***: the external components included in the system do not need to be neither implemented nor tested since it is assumed that they are reliable.

The first step of the implementation plan, for all the client-side applications (client mobile application, ticket machine application, call center operator web application, QR code reader machine, entrance turn display), starts by developing the UI and the NetworkManager components. The NetworkManager component should have methods for requests and responses forwarding. At the same time, it is possible to develop the WebServer component that simply provides the web pages content to the call center operator.

Moreover, the first step in the client mobile application would also include the implementation of the NotificationManager (it includes methods for notification sending).

On the business logic’s side (Application Server), the first step includes the implementation of the NetworkManager, NotificationManager (same reasons as the client side) and DataManager components. This last component includes the methods to manage data.

The second step, for all the client-side application excluding the client mobile application, includes the implementation of the ReservationManager, that locally saves every reservation the user has booked (useful for offline use of the app by the user) and AuthenticationManager, that locally saves the access token (it will be used for every interaction with the server in order to authenticate the user) components (w.r.t. their presence in that machine, as reported in section 2.B. Component View).
The client machine application, in the second step, includes the AuthenticationManager and the PositionManager components. It is useful to delay the ReservationManager implementation after that of the PositionManager, since the ReservationManager will use interfaces provided by the PositionManager component.
For the Entrance Turn Display application, the second step (final step) includes the implementation of the only remaining component: TurnManager.
The second step in the business logic includes the implementation of the StatisticsManager, AuthenticationManager, NotificationSchedulerManager (developed at this step in order to be available for the successive implementation of the ReservationManager component), BoughtItemFetcher. The StatisticsManager only interacts with the DataManager and would be useful to implement it at this step because it can be algorithmically difficult.

The third step only involves the client mobile application and the business logic machine. Both of them include the implementation of the ReservationManager component. Due to the great importance of this component, which is the core of the *CLup* system, this choice allows this module to be developed once all the other components it depends on are already implemented.

Finally, the last phase for the business logic includes the implementation of the RealTimeTurnManager, SuggestionManager and the external IVR Server component. All of them need the implementation of the ReservationManager interfaces and therefore it would be better to implement them in this phase, once that ReservationManager component has been completed. SuggestionManager, even if it does not directly use any ReservationManager’s interface, it strongly depends on the reservations. IVR Server indirectly depends on the ReservationManager component since it depends on the entire reservation flow.

## Integration and Test plan

The integration and test plan will follow the bottom-up approach described for the implementation phase, starting from the more crucial components that do not depend on other components but instead offer interfaces used by other modules in the system.
A technique which is extensively used both in the implementation phase and in the testing phase is the one called *data mocking*. Once the interfaces between components and the data format (regarding parameters, return values…) are well defined, it is possible to use the data mocking method to simulate components (both in the same application and in different applications) in order to implement features but most of all to test behaviours of the single module (in *unit tests*) and also the inter-component interaction (in *integration tests*).

## Unit Tests Overview

Regarding unit tests, every component should have a test code coverage which is between 95% and 100%, especially the core components (**DataManager** and **ReservationManager**).
It's important to point out that the selected bottom-up approach let the developers follow a *test-oriented* development process, writing and performing unit tests while the specific component is being implemented.
This path has the important benefit of making *bugs discovery* easier and quicker (as development and testing are almost contextual) and this results in having more reliable and robust components.

## Integration Tests Overview

Given the bottom-up methodology chosen in every application included in the *CLup* architecture, every time a component is developed and unit-tested, all the other components it depends on are already completed, and an integration test for the specific module can be performed.
However, in the context of each single application, once all the components have been implemented a more thorough integration test is needed, in order to make sure that modules which are not directly connected don't interfere with each other.

## Unit/Integration Tests Plan

In the following section, an important focus will be given to the tests phases related to the business logic, even if a brief description of the client-side testing scheme is present, too.

### Business Logic (Application Server)

#### Step 1

**DataManager**, **NetworkManager** and **NotificationManager** don't have dependencies on other components, thus unit tests for these modules can be performed. No integration test at this stage is needed, since there is not a direct connection between these parts of the business logic.

### Step 2

**StatisticsManager**, **AuthenticationManager**, **NotificationSchedulerManager** and **BoughtItemsFetcher** can be unit-tested at this level. Regarding the integration, the following interfaces can be tested to make sure that inter-component interaction is correct:

- AuthenticationManager - DataManager
- NotificationSchedulerManager - NotificationManager
- StatisticsManager - DataManager
- BoughtItemsFetcher - DataManager

Regarding the last integration to be tested, it is necessary to also check the correctness of the integration between the BoughtItemsFetcher component and the external shop service, offered by the shop to get the items bought by each user with a fidelity card. In fact, BoughtItemsFetcher acts as an *adapter* between the two databases.

### Step 3

At this level only the **ReservationManager** implementation has been planned: as it is the core of the business logic, a particular attention to this component is certainly a good practice to follow. This approach is particularly suitable for the testing plan; the component will be unit tested (possibly with a code coverage of nearly 100%) and then the following integrations will be tested:

- ReservationManager - DataManager
- ReservationManager - NetworkManager
- ReservationManager - NotificationManager
- ReservationManager - NotificationSchedulerManager

The first two connections are fundamental since they provide the main functionality of *CLup*: reservation management in terms of storing the details on the database, and the ability to communicate to all the clients connected through the Internet.

### Step 4

**SuggestionManager** and **RealTimeTurnManager** are tested in this last step of the testing plan. The integration needed for these components it’s here described:

- SuggestionManager - DataManager
- SuggestionManager - NotificationManager
- RealTimeTurnManager - ReservationManager
- RealTimeTurnManager - NetworkManager'

The main focus at this step is testing a communication protocol which was not used in the other components. In fact, while all the other components interact with the NetworkManager to produce HTTP requests and responses, here the WebSocket protocol is used to have a reliable and efficient real time communication channel. Since it is a technology only used by RealTimeTurnManager, it is important to pay attention to the integration test of this component with the NetworkManager.

In addition, since this is the step designated for the implementation of the IVR Server, it's appropriate to unit test this component. Then, integration with both Application Server and the external IVR service is needed.

### Clients

The implementation plan sections stated that distinct applications can be implemented in parallel by making use of *mocked data.*
This technique allows us to decouple not only the implementation plan, but also the test and the integration process regarding different applications. For this reason, the steps that are described below are meant to be unrelated to the steps described for the business logic. Nevertheless, it is still fundamental to write dedicated integration tests once each client (and the business components) of the system is completed, in order to fully cover the architecture and test the clients with real data. After these tests, the client components will be ready to be deployed in a production environment.

In all clients, since it is a good practice to strictly separate graphics implementation and application logic, UI can be unit tested separately and integrated only once the other components are completed.

#### Step 1

The **NetworkManager** is the first component implemented at each client, so it will be the first to be tested. In parallel, also the unit test related to the Mobile App **NotificationManager** can be performed. Speaking about the NotificationManager, it is also important to test the integration between the component and the API offered by the Mobile Operating Systems, since its job is to exploit these functions to locally provide notifications to users that did not provide their starting position for a booked visit.

#### Step 2

All clients share an authentication layer (represented by the **AuthenticationManager**) that is tested here. While the **ReservationManager** component of the ticket machine and the web applications is tested at this step, the mobile app must follow a different path: since the

**PositionManager** is needed to handle the case in which user's GPS location tracking is necessary to provide notifications, that component has testing priority. The ReservationManager will be left to the next step of the testing plan.
The Entrance Turn client contains a TurnManager to receive real time turns from the Application Server, and this component is now ready to be tested and integrated.
At this level it is indeed possible to start integration tests for components interaction in the same client application:

- AuthenticationManager - NetworkManager
- ReservationManager - NetworkManager (not for the mobile app)
- PositionManager - NotificationManager (only for the mobile app)
- TurnManager - NetworkManager (only for entrance turn display)

#### Step 3

This phase is dedicated to the testing of the mobile app **ReservationManager** component, as well as its interaction with the NetworkManager and the PositionManager.
Regarding the other client-side applications, at this point all the components should have been integrated, so, once the UI is built, a complete integration test in the context of the single application can be performed.

## Non Functional requirements testing

In section 3.C of the RASD, *performance requirements* were described. Once an integration test of the whole system (also called *system testing*) has been performed, the next step is to test the system performances. To achieve this goal, it is necessary to set up scripts which will put stress on the system, measuring the response time of the architecture, in order to make sure that performance requirements are fully satisfied.

In section 3.E.2 of the RASD, *reliability and availability constraints* have been defined, with a minimum up-time of 99%. A method similar to the one described before can be exploited for this purpose: scripts will simulate client requests for a predefined time frame (in the order of a few weeks), and the system will be monitored with logging tools, in order to ensure that the integrated system is reliable and satisfies the quality standard mentioned above.

Regarding security requirements, as DDOS protective measures are taken, it would be useful to also ensure that this protection is effective with a dedicated test. A penetration test could instead be delegated to an external security company, to test the behaviour of the Firewall components, whose job is to filter only allowed packets to the internal system network.

Ultimately, it is crucial to point out that the back-end services, depending on the technology stack chosen for its realization, will be probably based on a multithreaded environment. It is indeed important to plan intensive stress tests that will ensure that synchronization does not represent a problem in the context of the developed system. This is important in order to avoid problems that would likely show up in the production stage, that are typically difficult to find and to solve once the system is up and running.

## Additional testing practices

In addition to unit tests and integration tests, at the end of each component development process, it’s important that a member of the team (who was not involved in the making of the module) performs an informal code review. The code review is useful in order to increase the chances that each component is (at least) at a minimum quality level, before being accepted at a *production-ready* stage.