AY 2020/2021

Politecnico di Milano

DD: Design Document

Alice Piemonti Luca Pirovano Nicolò Sonnino

Professor
Matteo Rossi

Version 1.1
February 4, 2021

Contents

1 Introduction

1.1 Purpose

The purpose of this document is to provide an exhausting explanation about the S2B, focusing in particular on the architecture that will be adopted, the modules of the system and their interfaces.

Furthermore, a runtime view of the core functionalities of the S2B is provided, accompanied by some detailed interactions diagrams that show the message exchanging between the components.

Finally, there are mentions about the implementation, testing and integration processes.

1.2 Scope

The application provides different actors that will use the application, which are users, store administrators and store attendants.

Users can access to the application in order to plan their visit to the grocery shop, which can be booked in two different ways. In fact, they can grab the first available ticket (which is called *ASAP* mode) or they can choose a date and time slot in which they plan to visit the store, and also a duration or some articles they are intended to buy, in order to optimize schedules and reduce queues.

Store Administrators can register their shop on the platform, in order to become visible and bookable by users. They can also edit store information (such as opening hours, capacity of slots, etc.) and add or remove the store attendants.

Finally, **Store Attendants** can sign up with their personal store code in order to monitor entrances through a specific section of the application.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

- **Client-side scripting:** it is performed to generate a code that can run on the client end (browser) without needing the server side processing.
- **Code On Demand:** in distributed computing, it is any technology

that sends executable software code from a server computer to a client computer upon request from the client's software.

- **Middleware:** in distributed applications, it represents the software that enables communication and management of data.
- **RESTful:** it s a software architectural style that defines a set of constraints to be used for creating Web services.
- **Slot:** it is a day/time range. It can be reserved by a limit number of users in order to guarantee a maximum number of people who are inside a store in every time of the day.
- **Tier:** it is a row or layer in a series of similarly arranged objects. In computer programming, the parts of a program can be distributed among several tiers, each located in a different computer in a network.
- **Visit:** it refers to the customers entering the shop, and also to their staying time. It is associated to both a certain store and a visit slot.
- **Web Interface:** it permits to use a service only through the web browser.

1.3.2 Acronyms

- **API:** Application Programming Interface, it indicates on demand procedure which supply a specific task.
- **ASAP:** As Soon As Possible. It refers to the possibility of getting an appointment on the first available slot.
- **DBMS:** DataBase Management System.
- **DD:** Design Document
- **ER:** Entity-Relationship model, it describes interrelated things of interest in a specific domain of knowledge.
- **HTTPS:** Hypertext Transfer Protocol Secure (HTTPS) is an extension of the Hypertext Transfer Protocol (HTTP). It is used for secure communication over a computer network, and is widely adopted on the Internet.

- **MVP:** Minimum Viable Product, it is a version of a product with just enough features to be usable by early customers who can then provide feedback for future product development.
- **RASD:** Requirements Analysis and Specification Document
- **S2B:** Software to Be, it is the one designed in this document and not yet implemented.
- **TLS:** Transport Layer Security, it is a protocol which aims primarily to provide privacy and data integrity between two or more communicating computer applications

1.3.3 Abbreviations

- **Gn**: goal number n.
- **Rn**: requirement number n.
- **ID**: identifier.

1.4 Revision History

- January 9, 2021: version 1.0 (first release)
- February 4, 2021: version 1.1:
  1. Fixed typos;
  2. Updated Sequence Diagram and related descriptions;
  3. Updated Interface Diagram;
  4. Updated Component Diagram;
  5. Updated ER Diagram;
  6. Detailed description ORM, document's scope, components.

1.5 Reference Documents

- Requirements Analysis Specification Document (RASD)
- UML official specification: https://www.omg.org/spec/UML/

1.6 Document Structure

- **Section 1: Introduction**
  This section offers a brief description of the document that will be presented, with all the definitions, acronyms and abbreviations that will be found reading it.

- **Section 2: Architectural Design**
  This section is addressed to the developer team and offers a more detailed description of the architecture of the system. The first part describes the chosen paradigm and the overall split of the system into several layers. Furthermore, an high-level description of the system is provided, together with a presentation of the modules composing its nodes. Finally, there is a concrete description of the tiers forming the S2B.

- **Section 3: User Interface Design**
  This section is useful for graphical designers of the S2B and contains several mockups of the application, together with some charts useful to understand the correct flow of execution of it. The presented mockups refers to the client-side experience.

- **Section 4: Requirements Traceability**
  This section acts as a bridge between the RASD and DD document, providing a complete mapping of the requirements and goals described in the RASD to the logical modules presented in this document.

- **Section 5: Implementation, Integration and Test Plan**
  The last section is again addressed to the developer team and describes the procedures followed for implementing, testing and integrating the components of our S2B. There will be a detailed description of the core functionalities of it, together with a complete report about how to implement and test them.

2 Architectural Design

2.1 Overview

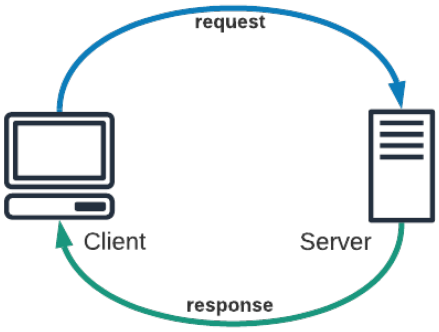


Figure 1: Client-Server paradigm

As figure 1 represents, the system is a distributed application which follows the common known client-server paradigm.

In particular, there are two different types of client, which makes it either thin and fat at the same time.

The first one is a RIA *Web Application*, which is by definition a thin client, because of its total dependency from the server. This type of client does not contain the application business logic, but only the presentation layer.

The second one, instead, is a mobile application, which contains an internal database in order to make it less dependent from the server. This aspect makes it a more thick client.

In both cases the server is *fat* and contains all the data management and business logic.

In this section the architecture will be described in an easy way, justifying all the choices for adopted patterns.

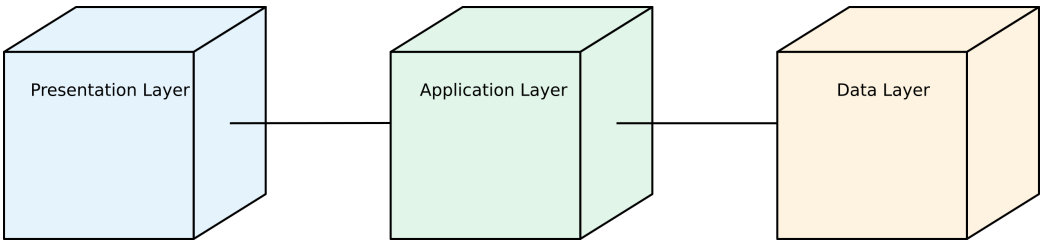


Figure 2: Three layers application

In figure2 the three S2B layers are shown, which respectively are:

- **Presentation Layer:** it manages the presentation logic and, consequently, all the interactions with the end user. This is also called *rendering layer*.
- **Application (Logic) Layer:** it manages the business functions that the S2B must provide.
- **Data Layer:** it manages the safe storage and the relative access to data.

As shown in the high level representation of figure 3 the S2B is divided into three layers that are physically separated by installing them on different tiers. A tier is a physical (or a set of) machine, each of them with its own computational power.

The application described in this document is composed by four tiers.

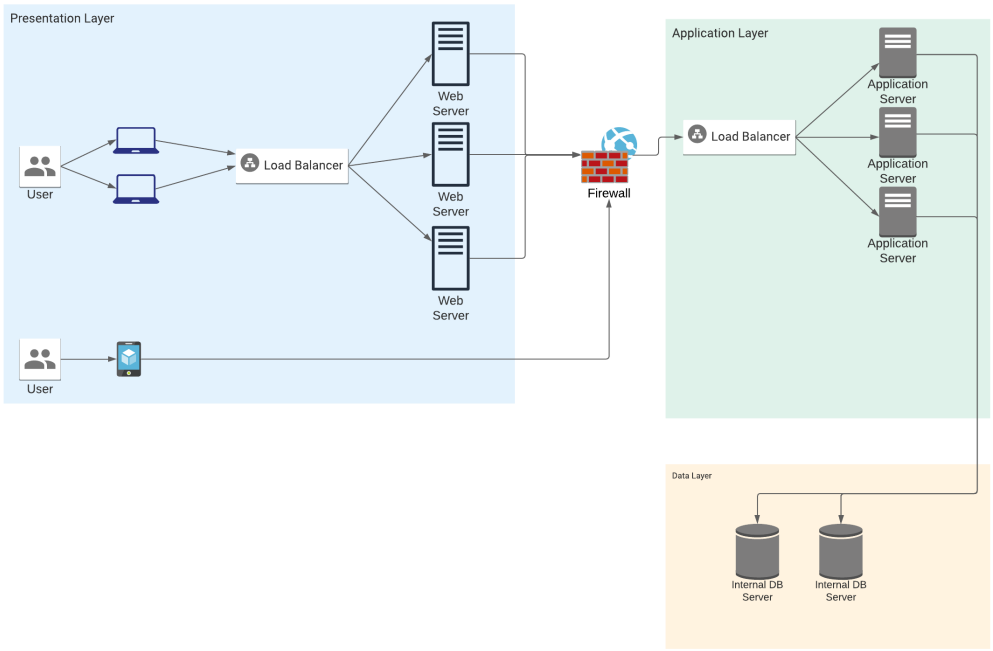


Figure 3: Architecture of the application

The service is supposed to be accessed through both a web interface and a mobile application, and this is valid for all kinds of users.

To make possible the construction of the web application, a client-side scripting paradigm will be adopted; this one will be described in detail in the final section of this document.

The architectural figure divides the application in the layers described above, and contains some replicated Web Servers, which act as a middleware between the user's browser and the application servers. In case of using the mobile application, instead, the core of the software installed on user's device

will interfaces with the business layer's APIs, which send and receive all the information in order to work properly directly to the application servers.

Finally, the application servers interfaces with the DBMS APIs, in order to retrieve and store the data required for the considered computation.

The applications servers are expected to be stateless, according to the REST standard definitions (more details in section 2.7.2). For accessing the data, they will use an ORM programming technique in order to interface with the DBMS exploiting the advantages of the object-oriented paradigm.

The nodes are separated by firewalls to guarantee a higher level of security of the whole system.

All the component will be described in depth in the following sections.

In case of third module development, which permits a custom deploy on buyer organization's servers, the entire architecture will be deployed and then configure on that servers, without interfacing with CLup ones.

2.2 Component View

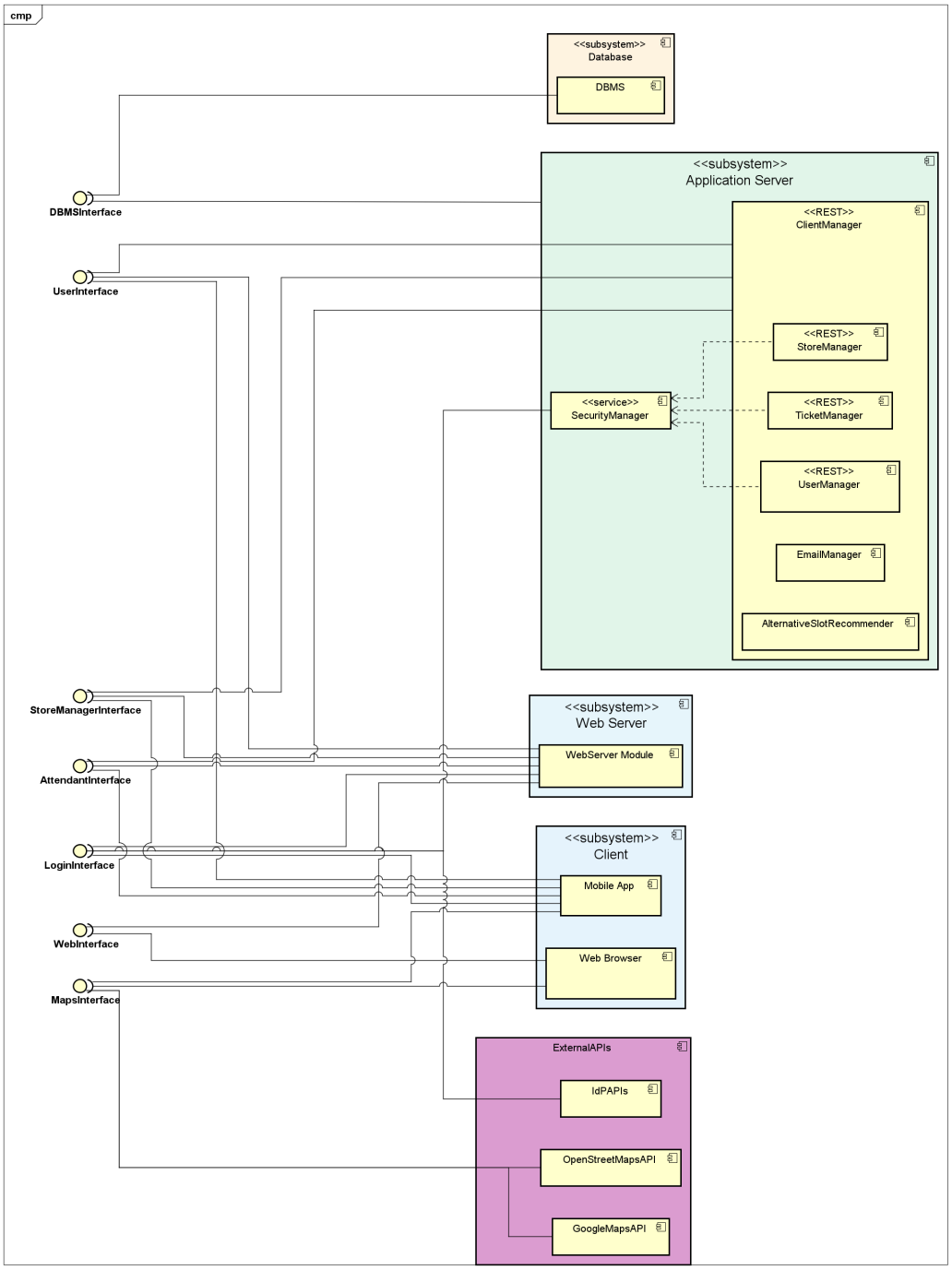


Figure 4: Component Diagram

In figure 4 we can see a more detailed diagram representing the layers described before.

The web server has the function to route the browser requests to the application server and send back its responses.

- **Client Manager**
  This module handles all the requests made by the client. At the beginning, when the client is not logged in, the module offers (through the user manager) a loginInterface which permits to execute a sign up or sign in operation. Once the client has logged in, the module shows only the other types of interfaces relying on the type of log in: a client logged as a user will exploit the UserInterface, a store attendant an AttendantInterface and a store administrator an Administrator Interface.

- **User Manager**
  This module contains all the features in order to manage the user side. In fact, it includes the log in and sign up manager, together with the other user services (i.e. information retrieving, role checker, etc.).

- **Store Manager**
  This module contains the API reference of the store side. In fact, it includes store creation, store update, stores retrieving and so on. Many components of it are used by the store administrator.

- **Ticket Manager**
  This module's aim is to handle the ticket functionalities, such as the line up feature (ASAP) and the "Book a visit" once. Furthermore, it provides several ways to manage the ticket (e.g. void it, change its status, validate it through the QRCode content and so on)

- **Email Manager**
  This component handles the email notifications, such as the sign up and forgot password ones. It also contains several scheduled tasks in order to accomplish to the periodic notifications functionality.

- **Alternative Slot Recommender**
  This component's aim is to suggest alternative slots to the end user, relying on people flows estimation which derives from an analysis of all bookings' time slots and inserted duration time.

- **Security Manager**
  Finally, this components handles all the security issues of the S2B. In fact, its aim is to authenticate and authorize requests, relying on the token provided from the client (since it should be a REST application). If a request is not authenticated, it takes the user to a login page; otherwise it simply replies with an unauthorized state message.

2.3 Deployment View

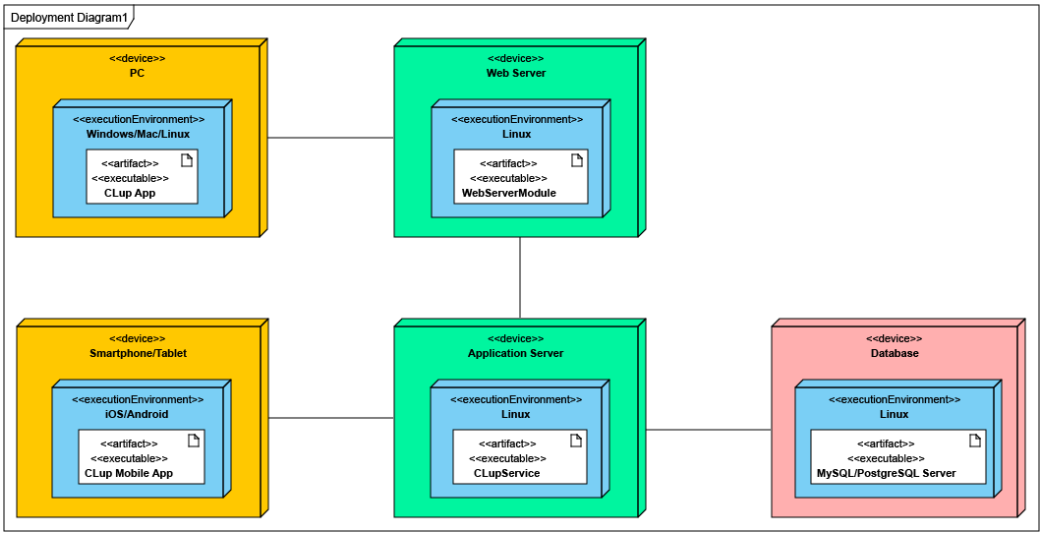


Figure 5: Deployment Diagram

The deployment diagram in figure5 shows the needed components for a correct system behavior, except the Maps APIs and the Authentication APIs ones.

Each device has its own Operating System where the software runs. The tiers in the image are the following:

- **Tier 1:** it is the client machine, which can be a computer with a web browser (running, for example, on Windows 10 OS) or the downloadable mobile application (available on both Apple's store and Google's Store).

- **Tier 2:** it includes the replicated web servers, which do not execute any business logic, but simply receive requests from the client, route them to the application servers and serve an HTML file fo the client, which will build the page thanks to client-side scripting. They also append the styling logic of the page (CSS sheets, JS sheets, etc.).

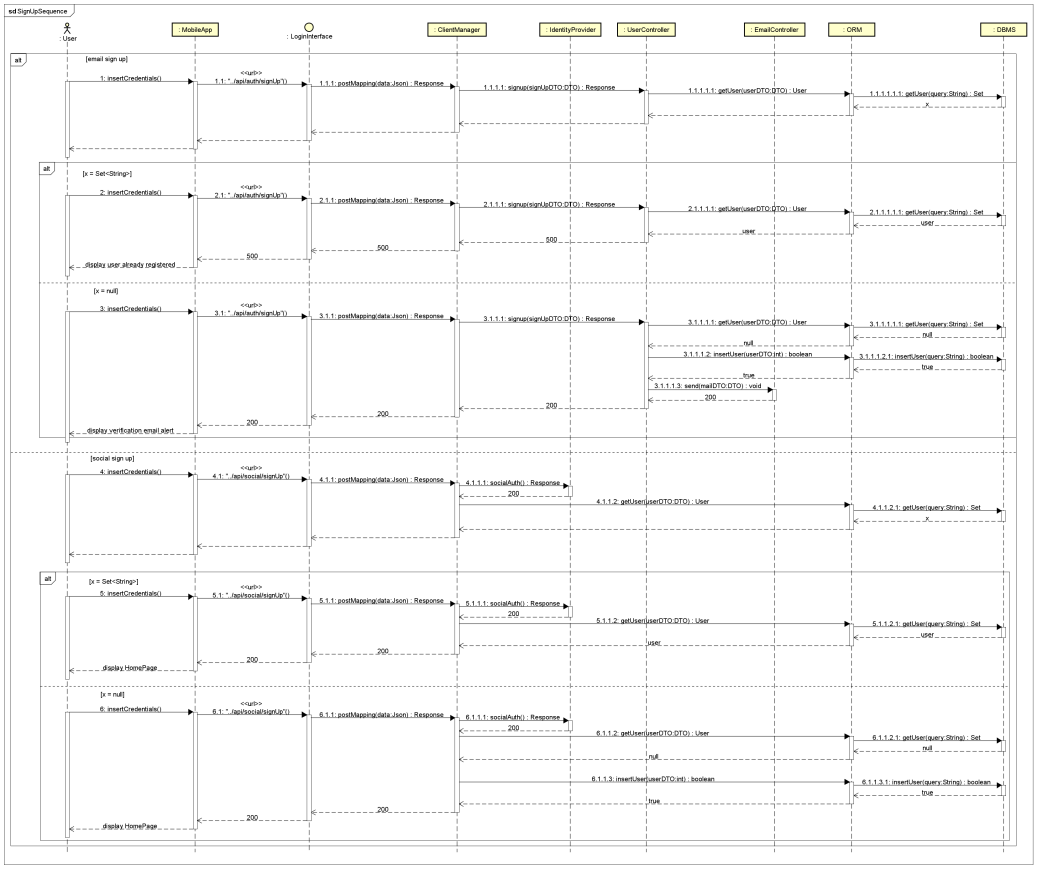
- **Tier 3:** it contains the application servers, which run the core functionalities of the S2B. The whole application layer is mapped into this tier, which communicates to the client tier through APIs, which will be used from the web servers (in case of webapp) and the native application (in case of mobile app download). Furthermore, it communicates to the data tier through the DBMS gateway.

- **Tier 4:** it is composed by the DBMS servers. They store the data and execute actions on it, according to the instruction given by the application servers.

2.4 Runtime View

All the following diagrams represent the runtime view from the mobile app's perspective, in order to increase readability, the web app's perspective isn't shown as it only differs from the other one by using the web server's module before calling the Application Server.
Every interface use REST API through HTTP GET or/and POST calls using urls, where the " **..** " before every path it's to be intended as the IP address + port of the server or the domain.
The login and registration on the mobile app is only permitted for store attendants or customers, managers need to use the web app interface.

2.4.1 Sign Up



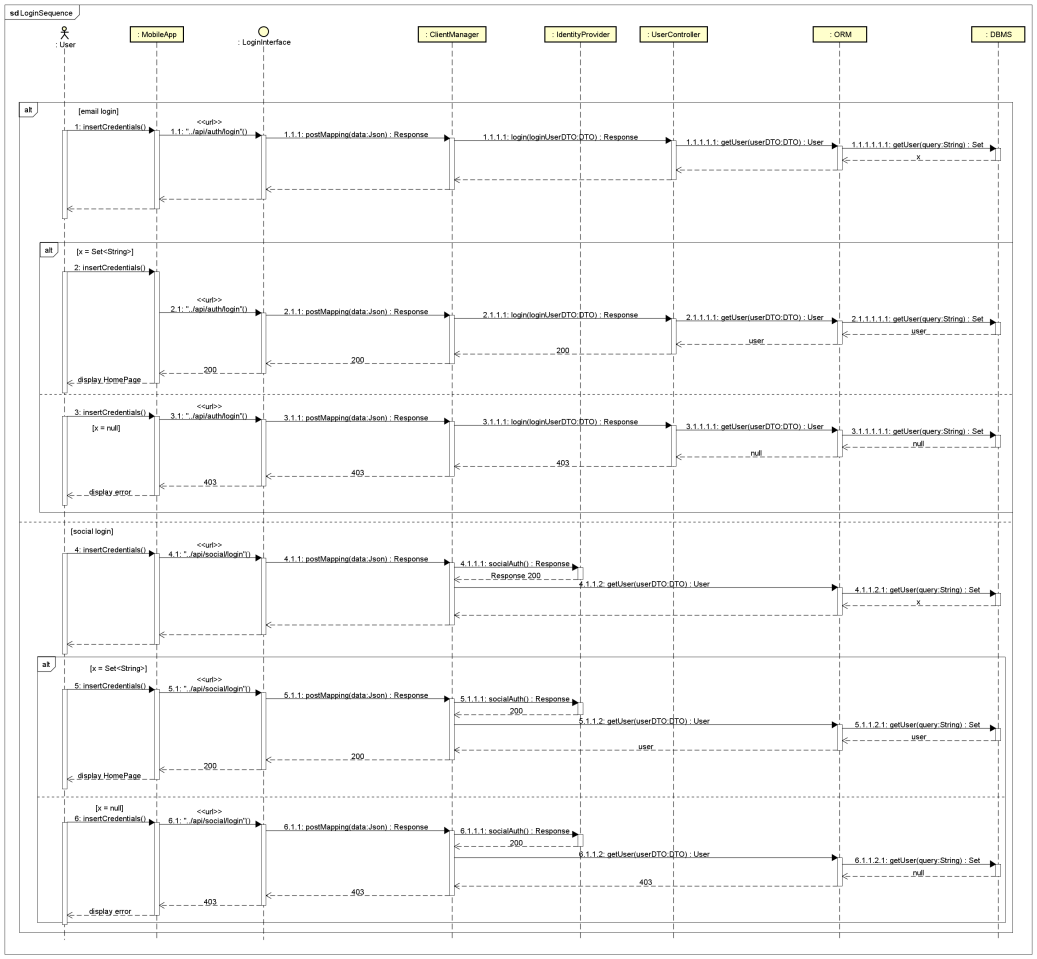
The diagram above represents the process of signing up a user (store attendant and administrator registration are omitted in order to avoid unnecessary repetitions).
There two possible situations:

- User is already registered;
- User isn't already registered.

The two situations differ only after submitting the user's data to the database and are analyzed below.
The user is presented with two choices: registering via social network or with

an email and password; if the user chooses the first one, they click on their preferred social network displayed in the Mobile App.
Afterwards the app sends the request to the ClientManager through the LoginInterface (url "../api/social/signUp"), accessing later the IdentityProvider.
The IdentityProvider returns user's info to the ClientManager which generates a userDTO, passing to the ORM and ultimately to the DBMS.
Based on the returned value of the DBMS the ClientManager inserts the User into the database or returns 200 response's status code (as shown above).
If the user decides the login via email, the path from the Mobile App to the ClientManager remains unchanged but this last one component delegates the database's query to the UserController (via ORM and DBMS).
If the returned value isn't null the UserController sends a 500 response's status code, warning the User that a previous registration exists.
If, instead, the DBMS returned null then the Uses is inserted to the database and lately the UserController via an EmailController sends an email to the provided one and displays to the user the instructions for the verification.

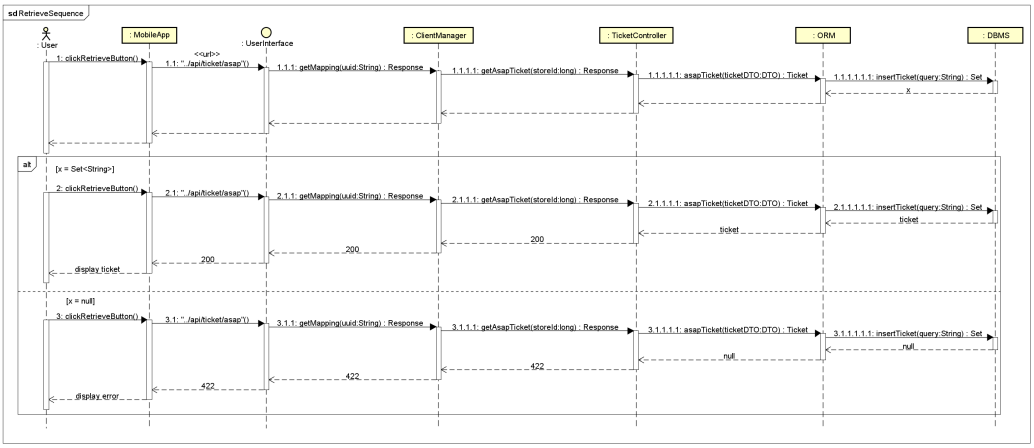
2.4.2 Login



The user chooses their preferred login method, whether it's social network or email and password.
If the first one is chosen, then the MobileApp sends the request, through the LoginInterface ("../api/social/login") to the ClientManager.
After requesting user credentials to the IdentityProvider, the ClientManager interrogates the DBMS (via ORM) and searches for the credentials retrieved. If the returned value is null the ClientManager proceeds with the creation of an User (as explained in sign up phase), otherwise a 200 response status

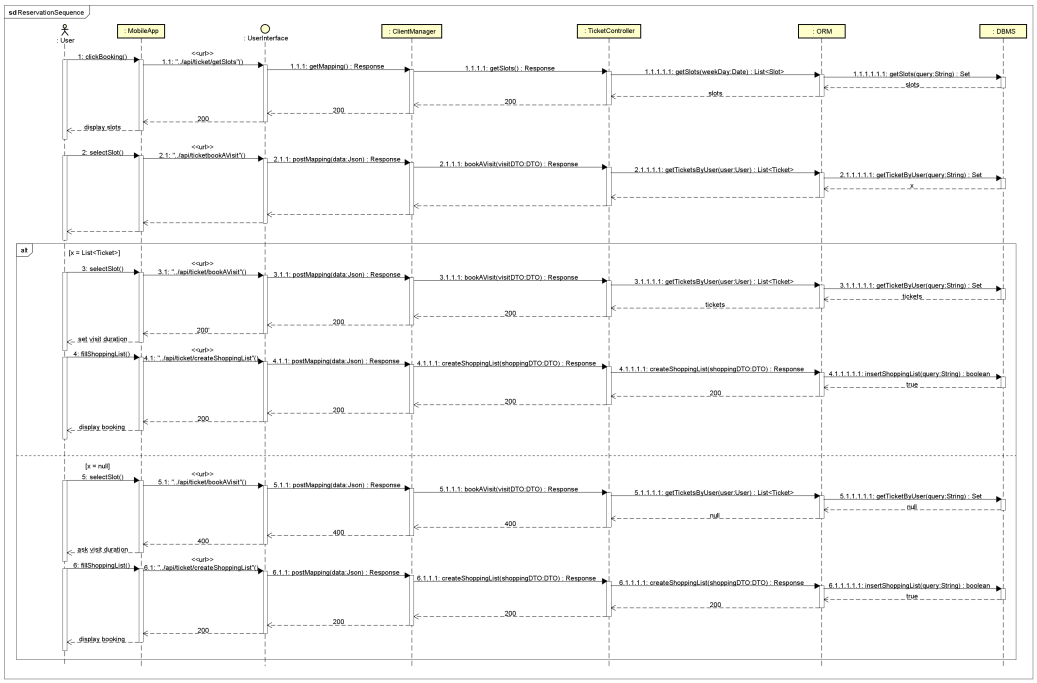
code is sent to the MobileApp.

2.4.3 Retrieve a ticket



The user clicks on the 'LINE-UP' button and the MobileApp contacts the ClientManager via the UserInterface ("../api/ticket/asap").
The ClientManager then calls the TicketController which creates a ticket-DTO and, using the ORM, queries the DBMS.
Afterwards, if a ticket is available, it is sent via a HTTP Response to the MobileApp using the previous path, otherwise an error is propagated up to the MobileApp.

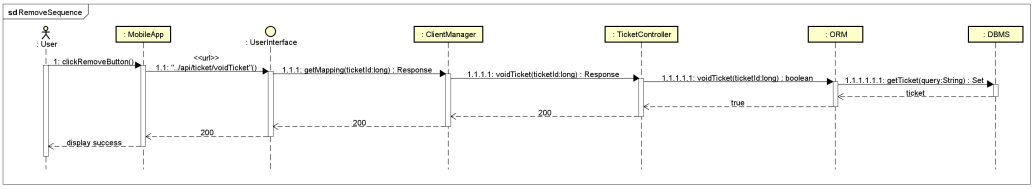
2.4.4 Make a reservation



In addition to the 'Retrieve a Ticket' function, the user can book tickets.
After pressing the "BOOK" button, the MobileApp requests all possible slots for the next weeks (T.B.D), to the ClientManager via the UserInterface ("../api/ticket/getSlots").
The ClientManager retrieves them by calling the TicketController, which queries the DBMS via ORM.
The data is sent to the MobileApp where it is displayed; the user then chooses a slot and confirms it by sending its data to the ClientManager, via UserInterface ("../api/ticket/bookAVisit").
The TicketController inserts the visit into the database while checking for previous bookings; if the DBMS finds any entries, it returns it to the ORM and than to the TicketController, which generates the estimated visit duration.
Based on the returned value the TicketController sends a 200 response's status code with the previous estimated visit duration (without it otherwise) and the MobileApp displays the shopping list's form.

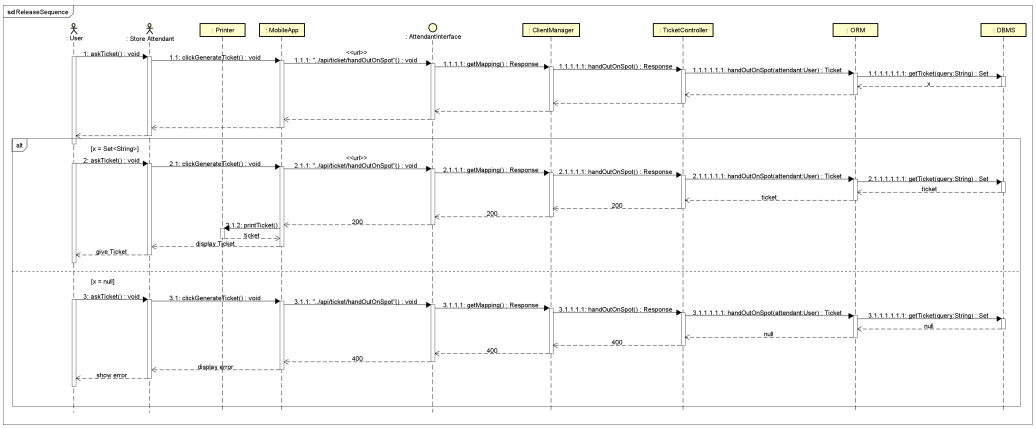
The User fills it with the products and sends it to the Database.

2.4.5 Remove a reservation



The user selects a slot and clicks on the "Remove" button.
The MobileApp calls the ClientManager, via the UserInterface ("../api/ticket/voidTicket"); then the ClientManager requests the removal to the TicketController (calling the ORM and the DBMS).

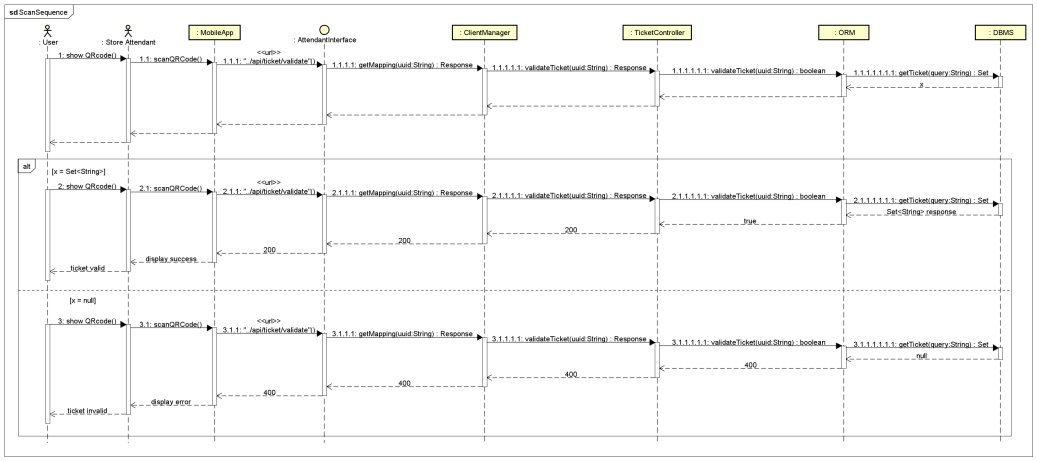
2.4.6 Release a ticket



This functionality is only available to the Store Attendant, if a customer can't install the app, they can still ask the staff to generate a ticket directly on the spot as shown above.

The Store Attendant navigates to the right page and requests a new ticket from the MobileApp to the ClientManager through the AttendantInterface ("../api/ticket/handOutOnSpot").
The remaining path is the same as the **Retrieve a ticket** functionality: the ClientManager requests the ticket to the TicketController, then the ORM and finally the DBMS.
The response's status codes are analogous to the mentioned sequence, the only difference is that the attendant, once received the ticket, can use the printer in order to give the ticket to the customer.

2.4.7 Scan QR code



The diagram above describes the process of scanning a customer's QR code.
The Store Attendant greets the customer, asking for the ticket's QR code, and opens the specific page.
After that they press the "SCAN" button and, using the webcam on the phone, scan the code; at this point the MobileApp sends the data to the ClientManager through the AttendantInterface ("../api/ticket/validate").
The ClientManager interfaces with the TicketController which sends the data to the ORM and ultimately to the DBMS.
If the ticket scanned is either INVALID, USED or VOID the returned response's status code is 400, 400 otherwise.

2.4.8 Modify time slots

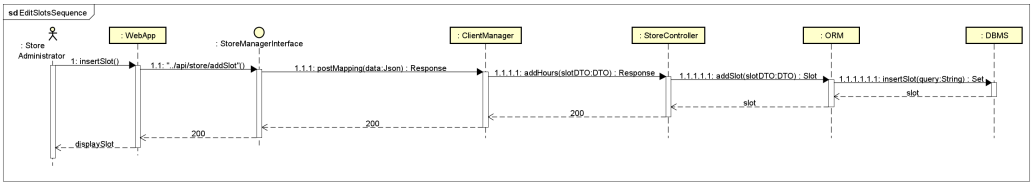


Figure 6: Modify time slots sequence diagram

This sequence is the only one available in the WebApp and not in the MobileApp.
The Store Administrator change a selected slot and clicks on the button in order to confirm.
The WebApp calls the ClientManager via the StoreManagerInterface ("../api/-store/addSlot").
This component sends the request to the StoreController which modify the database using the ORM; later a 200 response's status code is sent back to the WebApp.

2.5 Component Interfaces

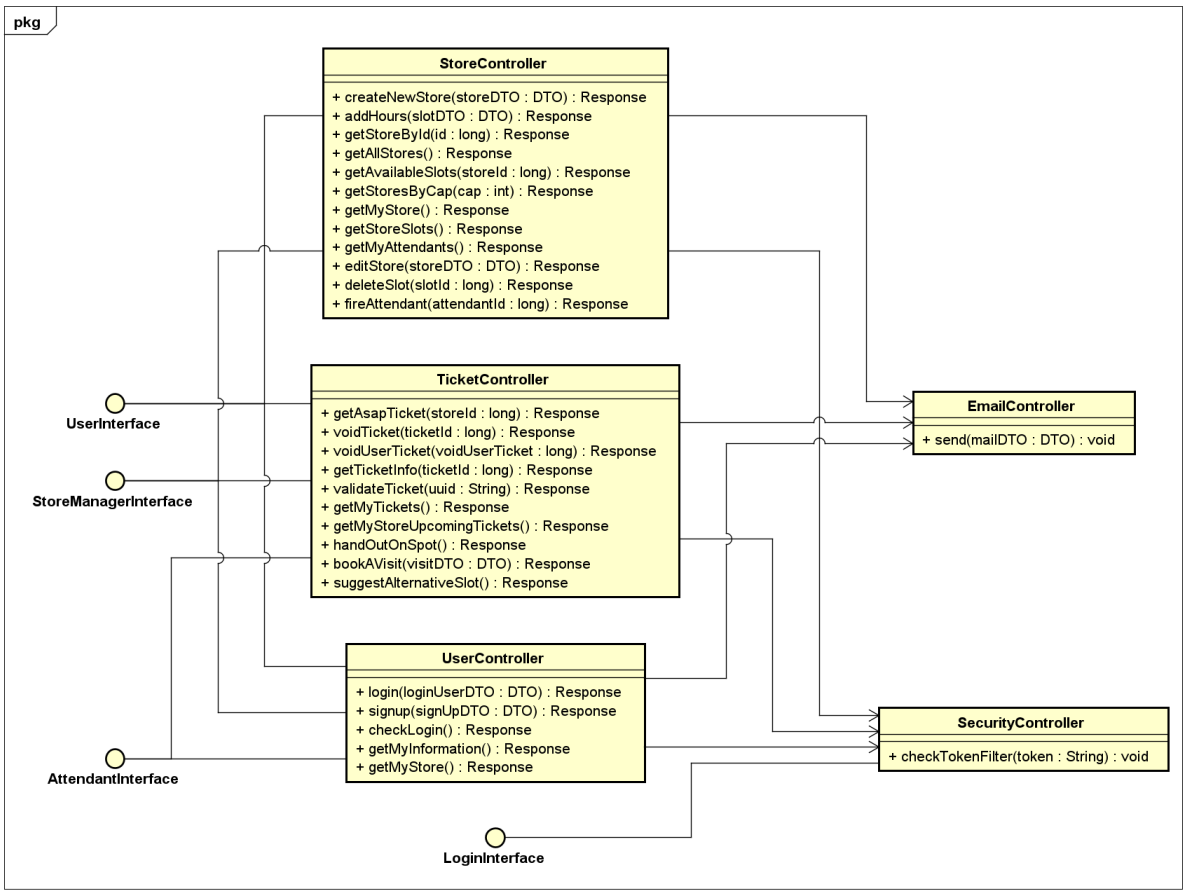


Figure 7: Interfaces Diagram

The interfaces shown in figure7 are the principal ones needed to make CLup run. The arrows represent the implementation classes of the interfaces.

Starting from the beginning, the **AccessInterface** is needed in order to hold the customer, store administrator and store attendant sign up and login processes. It is realized through two different controllers, called SignUpHandler and LoginHandler.

Then we can see the **UserInterface**, which main goal is to achieve user functionalities, such as the ASAP or the Reservation ones. It is realized by

five different controllers, one for each functionality.

- **UserController**: it manages the operations w.r.t. the user interface. In fact, it contains the login and signup methods, toghether with all the useful other methods in order to handle the interaction with him. For example, getMyInformation() returns the serialized user object with some information useful for a client computation (e.g. username, role, etc.).

- **StoreController**: in the case of the user, it simply contains methods in order to retrieve a list of stores relying on city or cap. It can also retrieve a complete list of stores. Finally, it also contains methods for retrieving the first available slot (given a store id) or other information about a specific store.

- **TicketController**: it manages the process of asap or book a visit ticket retrieving, which will be used by the customers in order to plan their visit to the shop. It also contains a method to void a ticket given its id.

- **SecurityController**: it manages the correct authorization of the user operations. In fact, through a correct tagging of them, methods can be defined only for users who have a specific role.

Then we have a store attendant interface, which main features are realized through different controllers.

- **UserController**: it manages the operations w.r.t. the attendant interface. In fact, it contains the login and signup methods, toghether with all the useful other methods in order to handle the interaction with him. For example, getMyInformation() returns the serialized user object with some information useful for a client computation (e.g. username, role, etc.).

- **TicketController**: in the attendant case, this controller manages the validation process of a ticket and gives the possibility of voiding an existent ticket. The validation method expects a ticket unique id (on the client it would be represented under a QR Code form) and check if it is valid. In this case, a positive status code is returned.

- **SecurityController**: it manages the correct authorization of the attendant operations. In fact, through a correct tagging of them, methods can be defined only for users who have a specific role.

Finally, we have the **StoreManagerInterface**, which permits to each store manager to edit all the information about the shop. This job is done through different methods:

- **UserController**: it manages the operations w.r.t. the manager interface.
- **StoreController**: it contains methods in order to manage the store. It permits creating a new store, editing an existent one, inserting new time slots (or deleting the existent ones) and to retrieve the list of active bookings. Finally, it permits to manage the store attendants (i.e. firing them from the application).
- **SecurityController**: it manages the correct authorization of the attendant operations. In fact, through a correct tagging of them, methods can be defined only for users who have a specific role.

Out of every previous context, there is the email controller, which manages the feature of periodic notification, sending messages relying on active bookings. It also sends the confirmation and recovery emails for the accounts.

2.6 Logical Description of Data

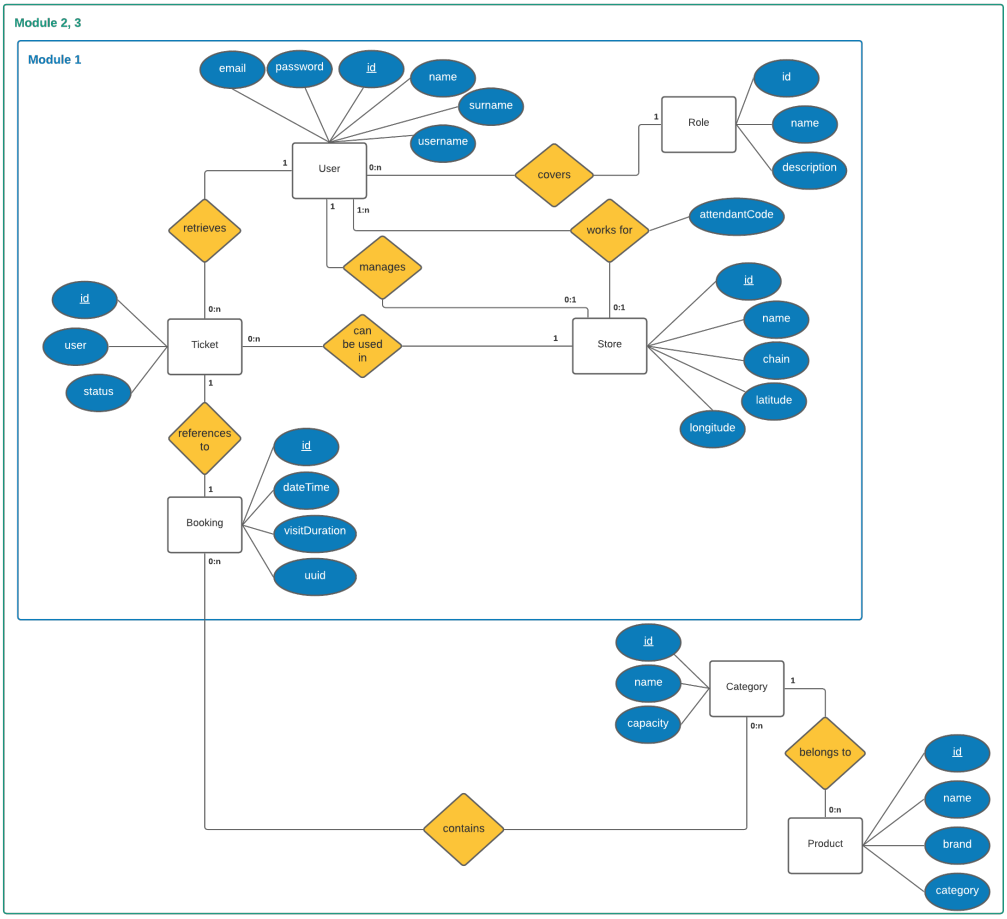


Figure 8: ER Diagram

The diagram in figure8 shows a logic representation of what kind of data is stored in the internal database of the S2B.

First of all, we can see the distinction in different modules (as s. 1.1 of RASD explains) and their respectively database structure in case of deployment on organization's servers. Of course, if the buyer chooses to remain on CLup official servers, the structure will remain the entire one, limiting the functionalities store by store.

Another important fact is that there is a distinction of responsibility in the user table, which provides several roles.

Then, we can see a table containing the list of all tickets, each of them associated to a specific store and (in case of *Reservation* functionality) to a specific booking.

Finally, a booking can contain a list of categories, each one with a set of product inside it.

Looking at the relations, we can see that there is a *bridge table* between Booking and Category, which associates a set of categories to one or more bookings. This association is used if a customer decides to insert a list of product in a reservation request.

2.7 Architectural Style and Patterns

2.7.1 Four-tiered architecture

We chose this architecture for many reasons:

- **Flexibility:** Once the interfaces of the S2B are defined, then the interior logic is dependent from outside. This fact implicates that each module can be improved without changing all the others.
- **Scalability:** an application divided on several tiers guarantees that the approach of scaling the architecture is adopted only for the most critical components. The result obtained maximizes the performance but also minimizes the costs.
- **Load Distribution:** the presence of several application servers, preceded by a load balancer, guarantees an acceptable division of requests. Otherwise, the presence of a single node means that node can become over-requested, sending the entire system down.

2.7.2 RESTful Architecture

The restful application will be adopted both on web and mobile side. This architecture is based on the stateless principle, in which the server does not contain any information about the state of client, that is managed directly on client side.

An useful property of this architecture is the *code on demand* one, which permits sending some code snippets from the server to the client, and then

make the client executing them locally (usually in the web browser). This behavior guarantees less computational load on the server and also a dynamic attitude of the service.

The application is then intended to be developed through *client side scripting*, which means that all requests and update of the page are made on client side. This behavior also improves the user experience, and prevent refreshing the page each time an action is made.

2.7.3 Model View Controller (MVC)

Model–view–controller (usually known as MVC) is a software design pattern commonly used for developing user interfaces that divides the related program logic into three interconnected elements. This is done to separate internal representations of information from the ways information is presented to and accepted from the user.

These three components are:

- **Model:** the central component of the pattern. It is the application's dynamic data structure, independent of the user interface. It directly manages the data, logic and rules of the application.
- **View:** any representation of information such as a chart, diagram or table. Multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants.
- **Controller:** accepts input and converts it to commands for the model or view.

2.8 Other Design Decision

2.8.1 Scale-Out

This method consists of cloning the nodes in which we expect to have a bottleneck in order to increase the general system scalability.

This choice leads to a higher deployment effort but also to a lower hardware upgrade cost when the limits are reached. In conclusion, the scale-out is a preferable road.

Once split, the system requires a load balancer in order to correctly redirects the incoming requests to the node with the lowest workload.

2.8.2 Thin and thick client and fat server

The thin client will be the web application.
This architecture consists of keeping as low information as possible on client side. It means that the business logic resides only on server side.
The minimum requirement of this choice is a stable connection between the parts; otherwise the application would not work as expected.
Of course, the main advantage of choosing this implementation style is that the client machine is not required to have an high computational power.

Instead, in the case of mobile application, the best choice is to save useful information on a local database, in order to avoid continuos requests to the server (less computational load) and also to keep information even when the Internet connection is not available. In this second case it is said to be a thick client.

2.8.3 Adoption of IdP Providers

We decided to adopt some external IdP providers (such as Facebook, Google, etc.) in order to simplify the process of user registration, without asking him any additional information.

This service is based on the providers API, which will communicate with our service in order to provide the necessary information (such as an email address).

3 User Interface Design

The aim of this section is to show the design of the main screens of the User app, describing the flow of the main functionalities for which the application was intended. The flow is created according to specific and illustrated input of the final user
Please note that we used some mockups presented in the RASD but more mocks have been created and added to this part in order to better clarify the user experience. In addition, we chose to show only screenshot of the mobile application because in our opinion Users would interact mostly with it. We want to remember that the application will work also on web browsers, but the design and the interfaces of the web app will derive from these ones, and the accurate design choices of them are out of the aim of this project.

3.1 User Mobile Interface

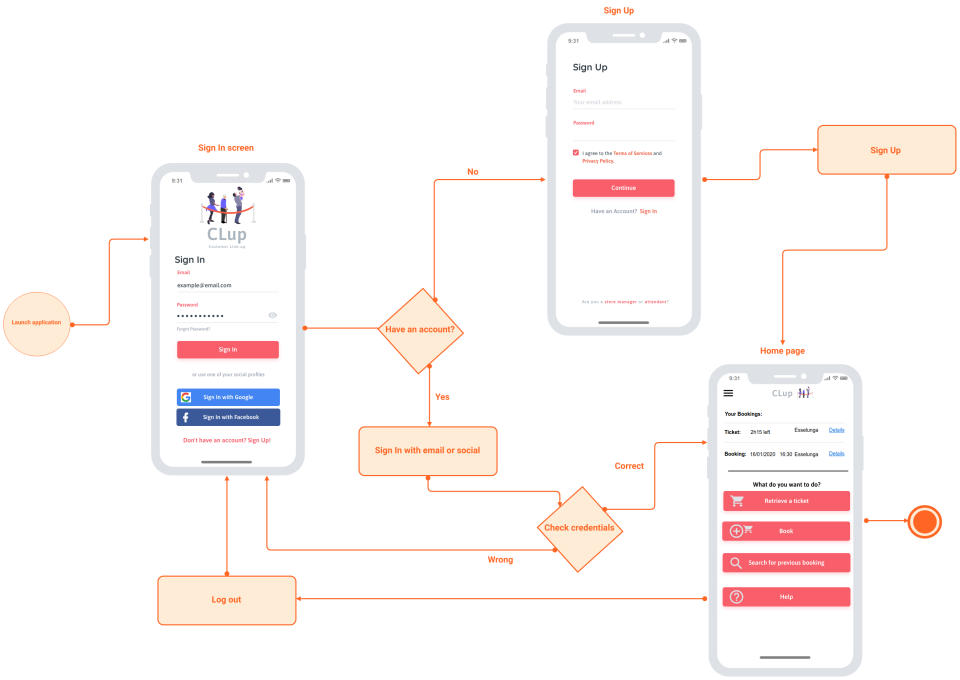


Figure 9: Sign Up & Sign In

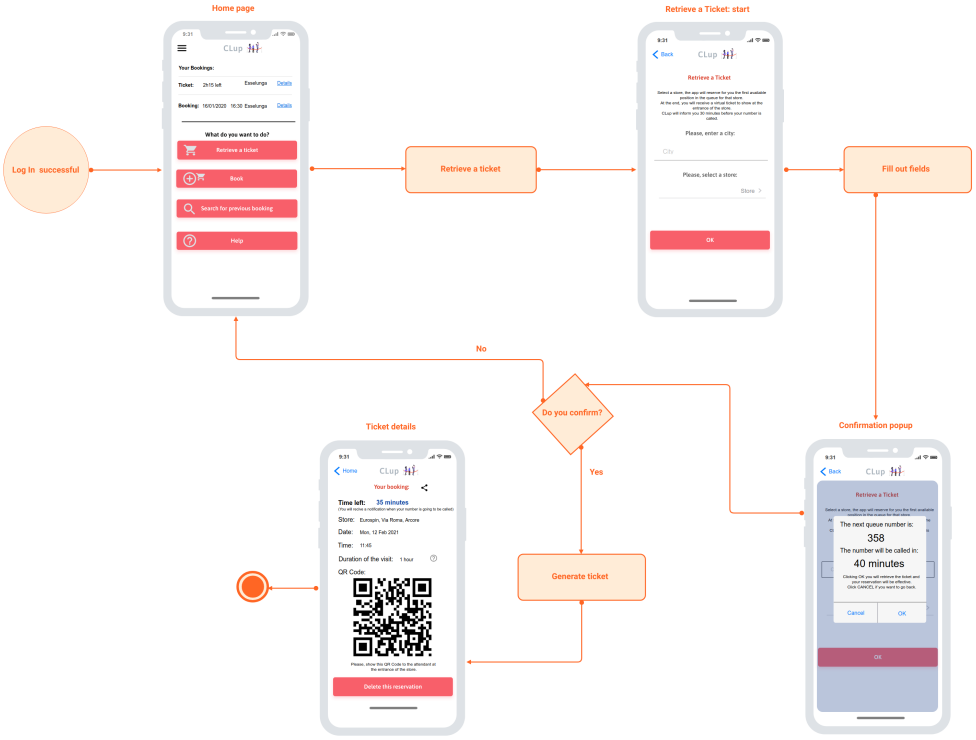


Figure 10: ASAP functionality

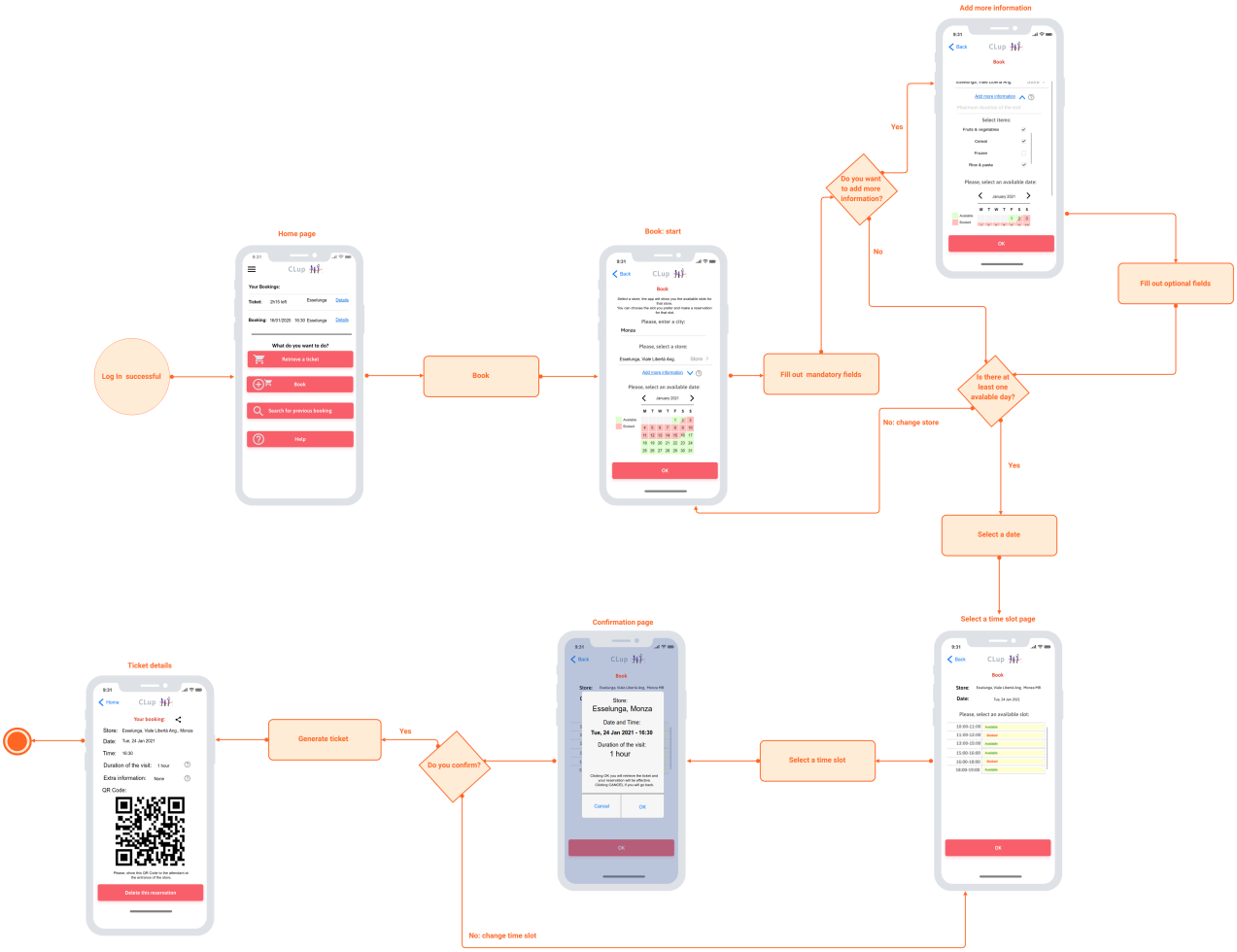


Figure 11: Reservation functionality

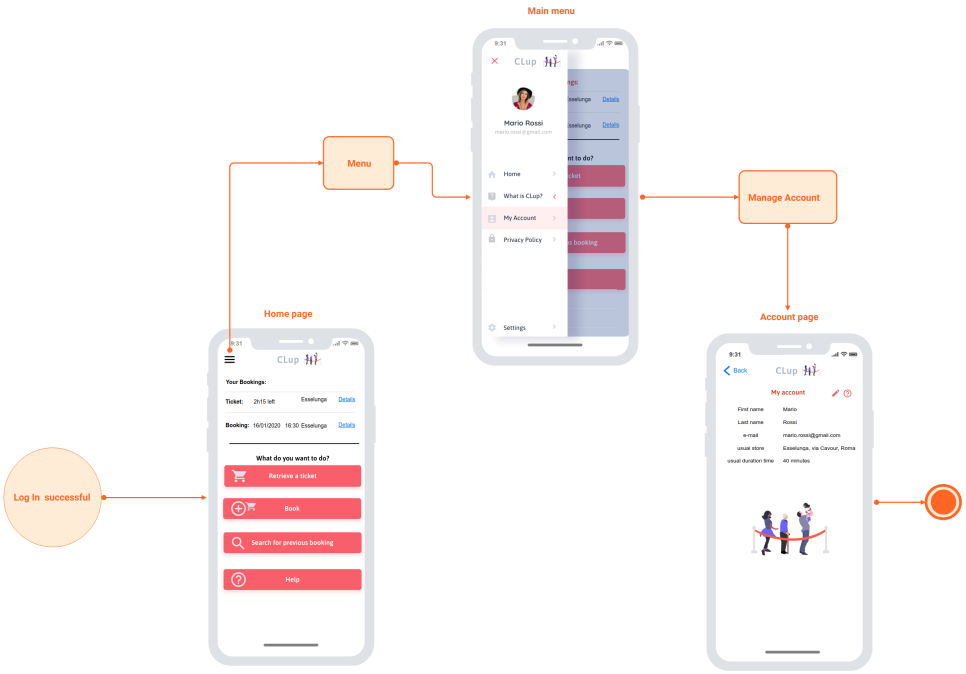


Figure 12: Manage account

3.2 Attendant Mobile Interface

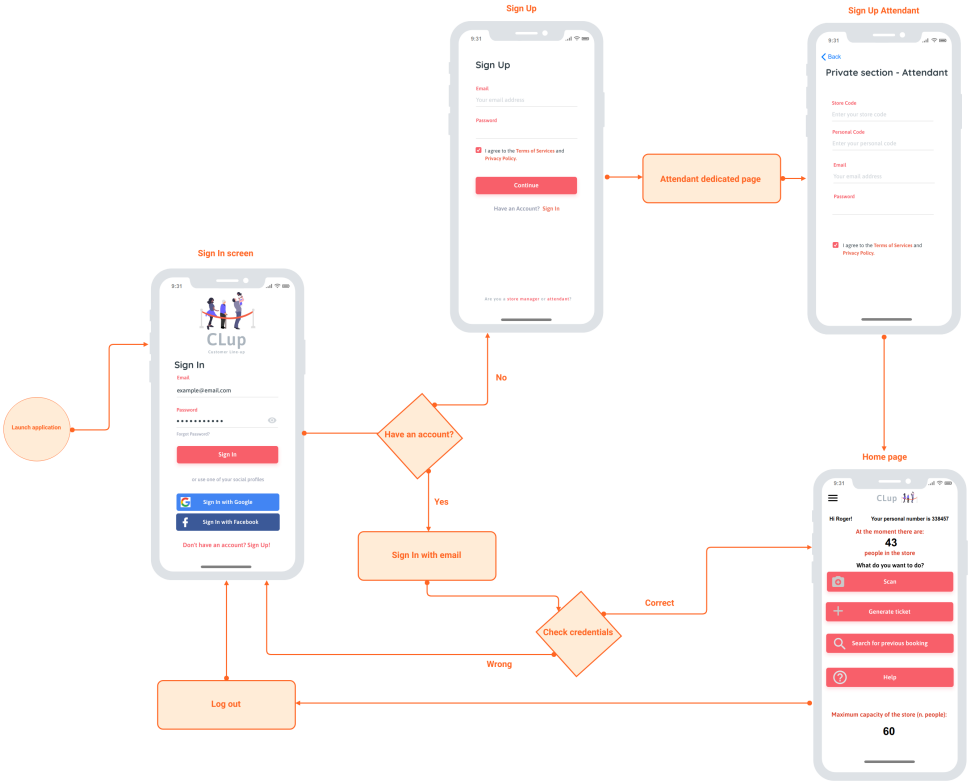


Figure 13: Sign Up & Sign In

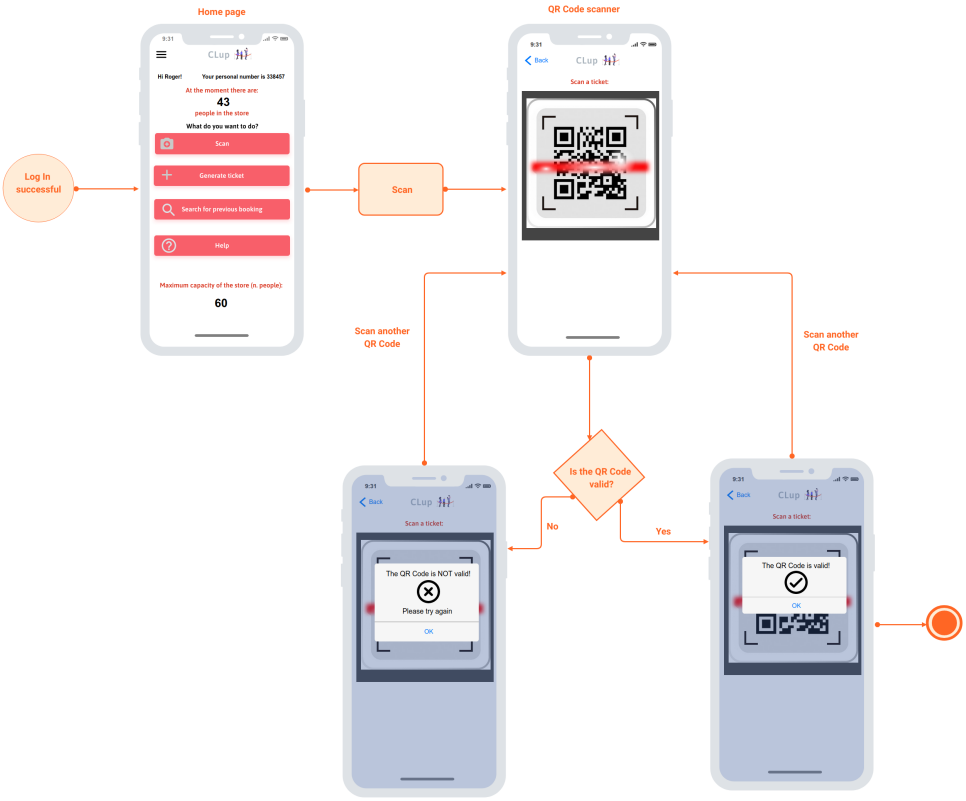


Figure 14: Scan QR Code

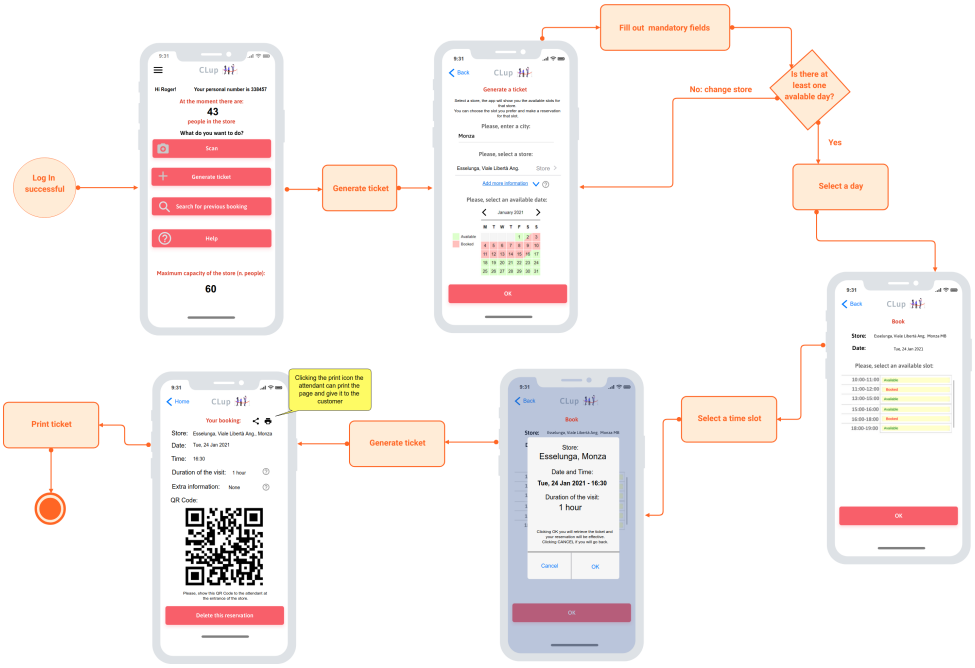


Figure 15: Release ticket on spot

3.3 Store Administrator Web Interface

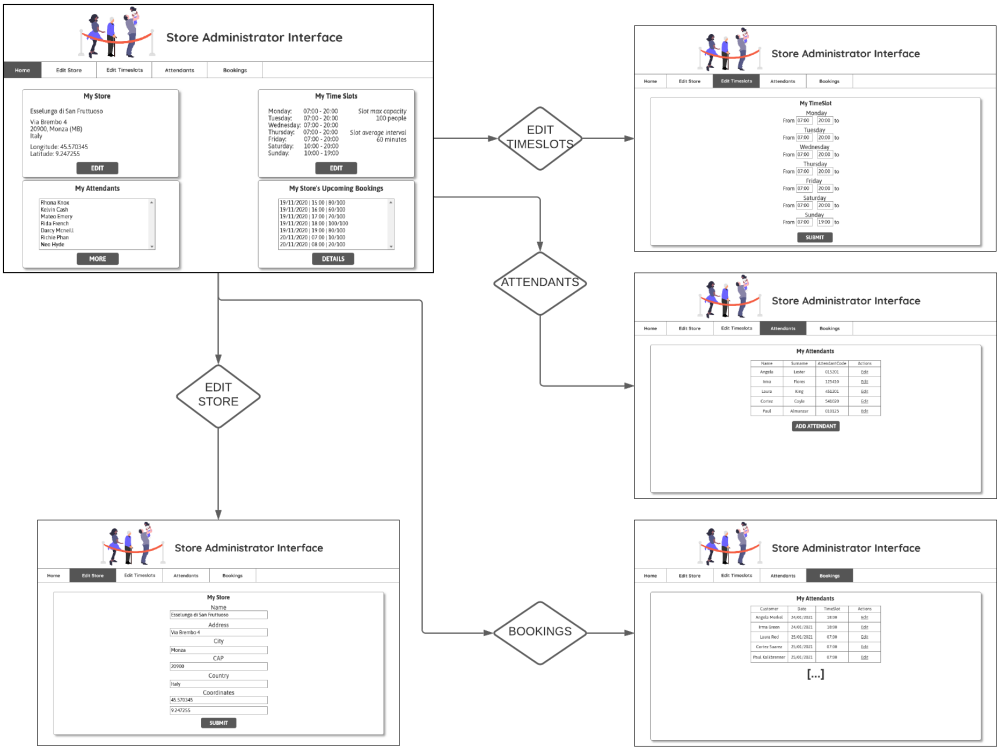


Figure 16: Store Administrator Interface

4 Requirements Traceability

This section will shown the traceability between requirements and modules described in component diagram.

- **R1:** The system allows registered users to select the store where they want to shop
  - **User Manager**: in order to login or sign up.

  - **Store Manager**: in order to choose a store.
  - **Ticket Manager**: in order to retrieve or book a ticket.

- **R2:** The system allows users to retrieve a queue number
  - **Store Manager**: in order to choose a store.
  - **Ticket Manager**: in order to retrieve or book a ticket or to use the handle on spot functionality.

- **R3:** The system generates a QR Code associated to the ticket
  - **Ticket Controller**: it simply returns a unique id to the client, which has then to show the information under form of QR Code.

- **R4:** The system allows Store Administrators to add GPS Position and opening hours of the store
  - **Store Manager**: this module allows store administrator to edit all the information about their stores.

- **R5:** The system allows Store Attendants to register with their attendant code
  - **User Manager**: this module allows the three types of users to register to the platform of CLup.

- **R6:** The system allows users to select a day/time slot from the available ones
  - **Store Manager**: in order to choose a store.
  - **Ticket Manager** this module allows to accomplish to book a visit functionality given a store id.

- **R7:** The system allows users to add a list of products (categories) to purchase and the duration time of the visit
  - **Ticket Manager**

- **R8:** The system computes a prediction of expected time duration of a customer's visit

  - **Alternative Slot Recommender** this component lets the system suggests alternative choices relying on people past bookings.

- **R9:** The system recommends alternative day/time slots or store/chains to a user
  - **Alternative slot recommender**
- **R10:** The system takes note about the actual in-queue tickets
  - **Ticket Manager**
- **R11:** The system notifies the user when its queue number is going to be called
  - **Email Manager**: this module triggers an action when a number is going to be called and informs the user through an email notification. On client side, it would be done by an integrated mobile app behavior.

Also the system attributes are guaranteed by the design choices explained in this document; more precisely:

- **Easy usability**: guaranteed through a very simple, minimal and intuitive user interface. Since the main target of the application is the customer-side, the experience is designed to be very simple. In fact, there are only a few buttons with clear and precise functionalities. The aim is to complete the *ASAP* ticket retrieval in no more than five taps (or clicks).
- **Reliability and Availability**: accomplished through a replication of the running application in different clones, following the *scale-out* method. The physical nodes of the system would then work in parallel, avoiding system downtimes due to a failure. In fact, when a clone breaks down, there are others in parallel which can supply the requested service, but with some performance issues. The scope is to obtain at least 97% of availability for each tier (the total system would then have 97% of availability).

- **Security**: guaranteed through an encrypted communication between client and server. If the client is connecting through the web interface, connection moves on HTTPS protocol. Otherwise, connection goes on TLS protocol.
- **Cross Platform**: firstly obtained through a development of a web interface, which can be accessed from any type of connected device with a browser installed. The native mobile application, instead, will be available only for iOS and Android users and will be downloadable from App Store and Google Play.
- **Maintainability and Modularity:** the first is accomplished through the second, because the application will be divided in some small parts (called modules) which interacts each other to provide the requested service.

5 Implementation, Integration and Test Plan

The S2B will be divided as follows:

- Client: it includes either web browser or mobile application;
- Web Server;
- Application Server;
- Internal Database;
- External services (such as OpenStreetMaps, IdP providers, etc).

These elements will be implemented following a bottom up logic, in order to avoid stub structures that would be more difficult to implement and test.

The main focus is on the module of the application server, because of its importance in the S2B development and also due to its testing difficulty.

The following table presents the functionalities described in the RASD document and highlight for each of them the importance for the customer and the difficulty of their implementation.

Functionality	Module	Importance for customer	Difficulty
Sign Up and Login	1 (MVP)	High	Low
ASAP	1 (MVP)	High	Medium
Make a reservation	2	Medium	High
Hand out on spot	1 (MVP)	High	Medium
Periodic notifications	1 (MVP)	Low	High

Table 1: Implementation and Testing precedences

All the modules described in this document rely on DBMS of the internal database, which must be implemented as the first component.

According to the table1 we decided to implement the functionalities relying on their importance.

- **SignUp and Login:** the two functionalities are strictly related, since the Login result is somehow affected by the SignUp function. So it would be a good idea to implement the sign up method first and then the login one and then execute a unit and integration test. The test of the User Manager requires a demo of the DBMS containing some accounts. In fact it's necessary to test the correct call to the DBMS functions in order to retrieve the account already registered and then it is important to check the module recognize when a user is already registered or if the sent data presents some inconsistencies. Relying on the DBMS functions, an integration test is required to check the correct communication between the module and DBMS.

- **ASAP:** for this functionality it is necessary to implement and execute tests on the Store Manager and the Ticket Manager components. They will be both tested through an automated test class. In fact, it is necessary to create a store, add a time slot and, finally, retrieve a ticket through different controller calls.

- **Hand out on spot:** this functionality is managed by the Ticket component, on which the unit tests will be made. They will check that the module manages in a correct way all the requests of ticket that are made on spot, by an attendant of the considered store. Finally, an integration test with the DBMS is made.

- **Make a reservation:** this module is an extension of the MVP one. It exploits some components, which are the Store Manager, Ticket Manager and Alternative Slot Recommender ones. They will need some unit tests and, in particular, the Ticket Manager one also needs an integration test with the DBMS.

- **Periodic notifications:** this functionality is accomplished through the Email Manager, on which some unit tests are needed to be executed. To properly make this tests, a demo scenario is needed, in which a customer retrieves a ticket (through, for example, the ASAP functionality) and the system triggers because the ticket is going to be called.

5.1 Clarification on component integration

In this section there is be a description about how the components are integrated and communicate, in order to build the entire S2B.

First of all, in figure17 it is clear that the first component to build is the DBMS, followed by the main application components that exploit it.

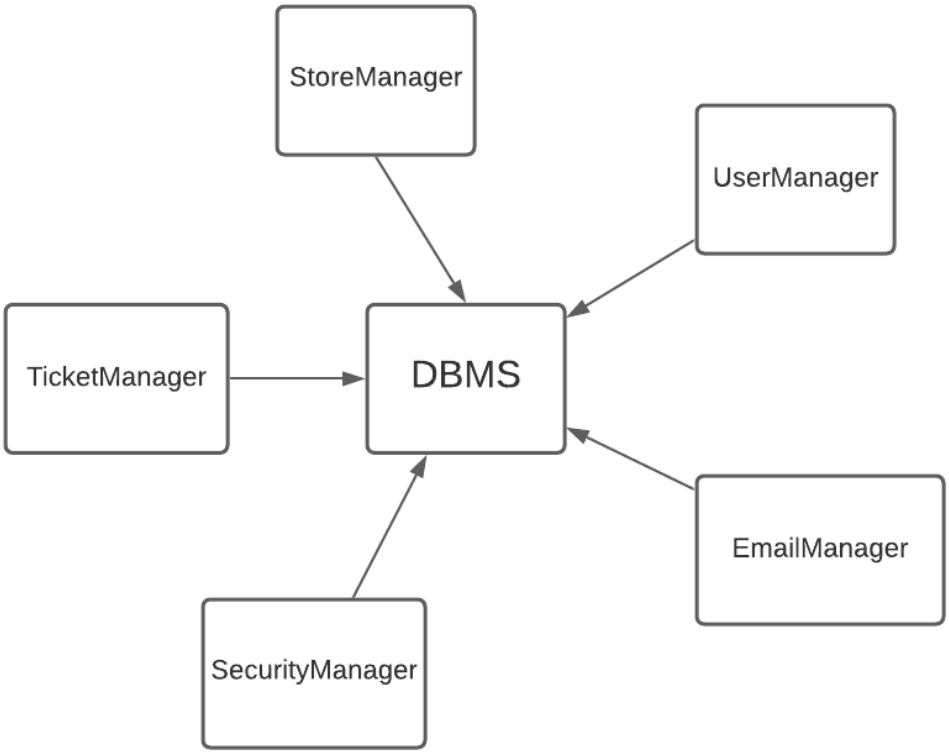


Figure 17: Components Integration

After this, we have to integrate the API communication between the S2B and the external services that will be used, which are some *IdP* providers, as explained in figure18.

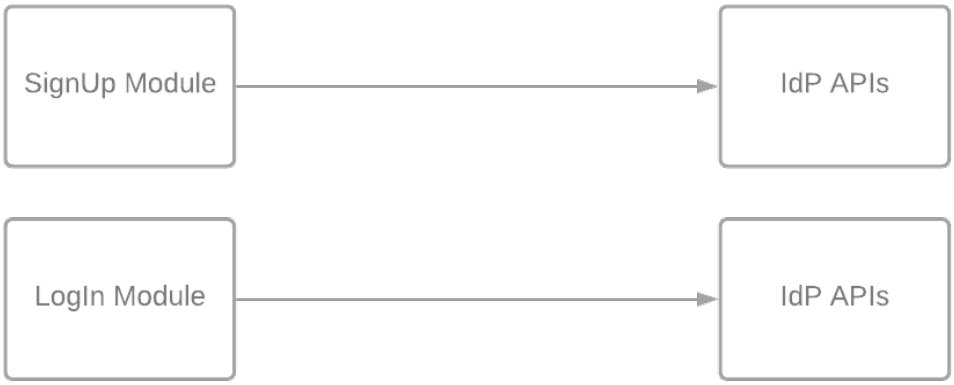


Figure 18: External APIs integrations

At this point it is possible to integrate the Client Manager, which permits to the end user to exploit all the functionalities, as shown in figure 19.

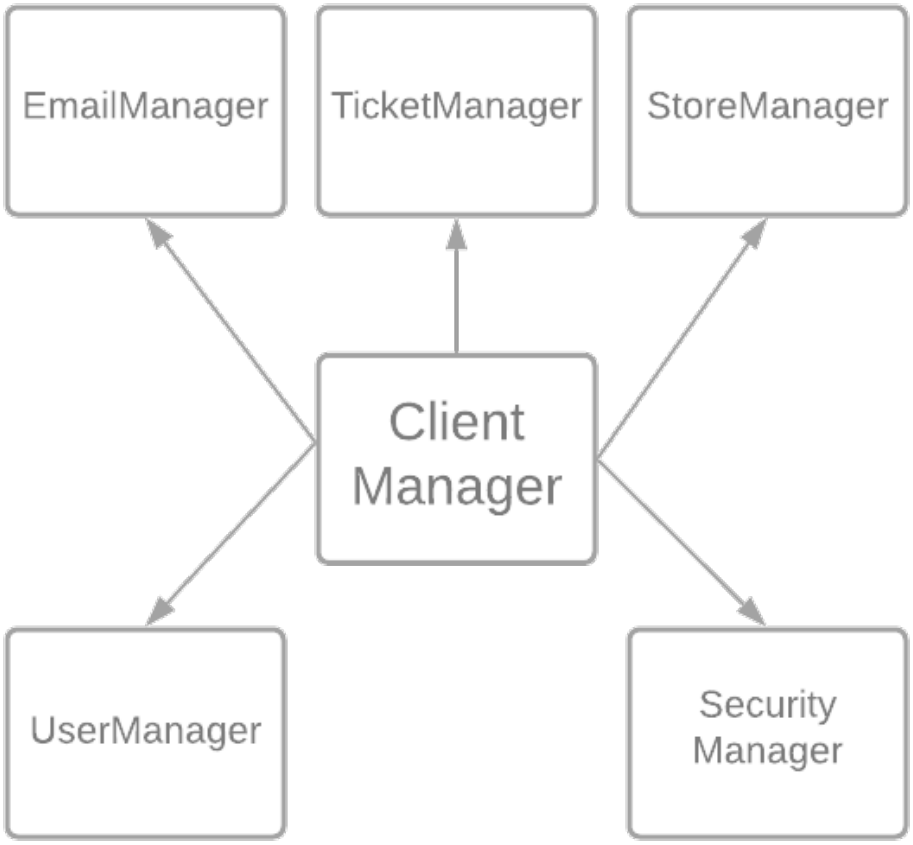


Figure 19: Client Manager Integration

Finally, it is possible to integrate the web server module and the mobile application module and the browser with the web server module. This work in necessary in order to make possible the client-server communication.

This last part is shown in figure20.

Together with this, as shown in figure21, the client will be integrated with the maps API, which are provided by *OpenStreetMaps*.

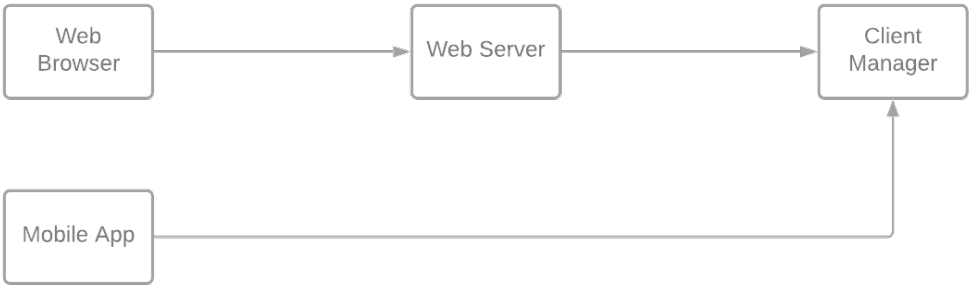


Figure 20: Client-Server Integration

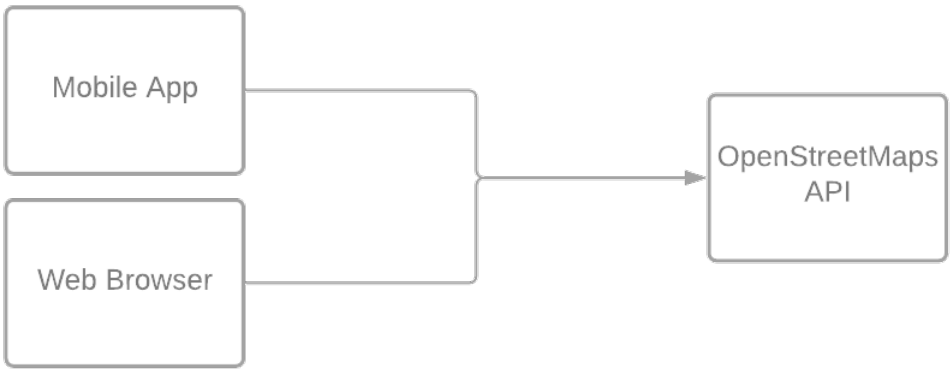


Figure 21: Client-Side Maps Integration

6 Effort spent

Student	Time for S.1	S.2	Time for S.3	Time for S.4	Time for S.5
Alice Piemonti	2h	3h	6h	1h	1h
Luca Pirovano	2h	7h	2h	2h	2h
Nicolò Sonnino	3h	7h	2h	2h	2h