POLITECNICO
MILANO 1863

SOFTWARE ENGINEERING 2 PROJECT

Design Document

CLup - Customer Line-up

Version 1

Authors
Galzerano Arianna
Lampis Andrea
Leone Monica

Supervisor
Dr Di Nitto Elisabetta

Sommario

1.Introduction

1.1 Purpose

This document is the *Design Document (DD)* of the *CLup – Customers line-up* application. It contains a functional description of the system, and an accurate overview of all parts of the S2B through the component view, the deployment view and the run-time view.

1.1.1 Description of the given problem

CLup is an application with two main purposes: the first one is to allow store managers to regulate the flow of people in the building, and the second one is to save people from having to line up and stand outside of stores for a huge amount of time.

More details can be found in the RASD document.

1.1.2 Goals

In this section the goals we aim to accomplish through the different functionalities we plan to implement are listed out.

NAME	BRIEF DESCRIPTION
G1.1	Allows OnlineUser to line up
G1.2	Allows OnlineUser to book a slot of time
G2	Allows OnlineUser to visualize the map of the city with the stores
G3	Allows OnlineUser to see statistics about the stores
G4	The system gives suggestions about alternative stores when the chosen one is not available
G5	The system notifies the onlineUser when he should start coming closer to the supermarket
G6	The system notifies the onlineUser to remember him about his slot reservation and asks for a confirmation
G7	The system interacts with the Manager Service

1.1.3 Manager service

The S2B has to interact with the Manager Service. It is a software developed separately and which is not covered in this document, in order to give priority to the aspects linked to the other actors mentioned. It has two different functions which are connected to the S2B: the first one is to register a store to the service, inserting all the mandatory details such as the store capacity, the departments, the opening hours; the second one is to monitor the store. The way the two systems interacts is explained in the following sections: notice that the manager service will be represented like a black box as its structure is not the subject of the document.

1.2 Scope

In this section, the phenomena related to the machine – which is the software to be developed – and to the world – which is the real environment in which CLup will be used – are enumerated. A phenomenon can be shared if it is controlled by the world and observed by the machine or vice versa.

1.2.1 World phenomena

NAME	BRIEF DESCRIPTION
WP1	User needs to go grocery shopping
WP2	OnlineUser takes a smartphone with himself
WP3.1	OnlineUser goes to the store with a booking
WP3.2	PhysicalUser goes to the store without a booking
WP4	Each store has a certain capacity to contain people
WP5	Costumer line up out of the store
WP6	OnlineUser is interested about statistics on the stores
WP7	Manager is interested in complying with the Covid rules in his store

1.2.2 Machine Phenomena

NAME	BRIEF DESCRIPTION
MP1	Generation of QR Code
MP2	Calculation of the estimated time to arrive to the chosen supermarket
MP3	Computation of the open and currently available stores
MP4.1	Esteem of the residence time inside the store for a usual OnlineUser
MP4.2	Esteem of the residence time inside the store through the analysis of the type of items that the OnlineUser needs to buy
MP5	Encryption of sensitive data
MP6	The system stores and reads data
MP7	The system retrieves data from the database

1.2.3 Shared phenomena

NAME	BRIEF DESCRIPTION
SP1	User shows the QR Code entering the supermarket
SP2	User shows the QR Code leaving the supermarket
SP3	The system sends a notification that invite him to come closer to the store
SP4	The system sends a notification that remind him about his booking
SP5	OnlineUser make a booking
SP6	OnlineUser delete a booking
SP7	The system tracks the position of OnlineUsers
SP8	OnlineUser is able to look at the map and find the stores near to him

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

NAME	BRIEF DESCRIPTION
QR Code	It is a type of matrix barcode which contains information about a reservation
User	He is a generical costumer of the store
OnlineUser	He is a customer of the store who goes to the supermarket using CLup
PhysicalUser	He is a customer of the store who goes to the supermarket not using CLup
Ongoing Reservation	Reservation during its creation process

1.3.2 Acronyms

NAME	BRIEF DESCRIPTION
API	Application Programming Interface
GPS	Global Positioning System
FCM	Firebase Cloud Messaging
DMZ	Demilitarized Zone
DB	Database
URI	Uniform Resource Identifiers
DW	Data Warehouse

1.3.3 Abbreviations

NAME	BRIEF DESCRIPTION
S2B	Software to be
Gn	Goal number n
WPn	World phenomena number n
SPn	Shared phenomena number n
MPn	Machine phenomena number n
Dn	Domain assumption number n
Rn	Functional requirement number n

1.4 Revision history

Date	Version	Changelog
29/12/2020	**1.0**	First Release

1.5 Reference documents

[1] Slide of the lectures
[2] Specification document "R&DD Assignment A.Y. 2020-2021"
[3] High availability in Azure Database for PostgreSQL – Single Server: https://docs.microsoft.com/en-us/azure/postgresql/concepts-high-availability
[4] Integration Testing strategies: https://www.guru99.com/integration-testing.html
[5] Hans van Vliet (2008), *Software Engineering: Principles and Practice*

1.6 Document structure

The document is divided in six parts.

- **INTRODUCTION**: It gives an overlooking on the purpose and scope of the document, defining the main goals and phenomena and the definitions, acronyms, and abbreviations of the most used terms. It also contains the revision history and the reference documents to better underline how it has been developed.
- **ARCHITECTURAL DESIGN**: It describes the high-level architecture, highlighting the main components, the interfaces, their interaction with runtime views and other design decisions.
- **USER INTERFACE DESIGN**: It provides an overview on how the user interfaces of the S2B will look like.
- **REQUIREMENTS TRACEABILITY**: Explain how the requirements you have defined in the RASD map to the design elements that you have defined in this document, describing the connection between the RASD and the DD.
- **IMPLEMENTATION, INTEGRATION AND TEST PLAN**: It Identifies the order in which it has been planned to implement the subcomponents of the S2B and the order in which it has been planned to integrate such subcomponents and test the integration.
- **EFFORT SPENT**: It includes information about the number of hours each group member has worked for the development of the document

2. Architectural Design

2.1 Overview

The S2B is developed with a three-tier architectural style, which allows the distribution of application functionalities across three independent systems: a presentation layer, an application layer, and a data layer. They will be discussed in detail in the section 2.2.
Moreover, the S2B will interact with some external services/systems.

The figure below represents a high-level description of the main components which constitute the S2B, or that interact with it.

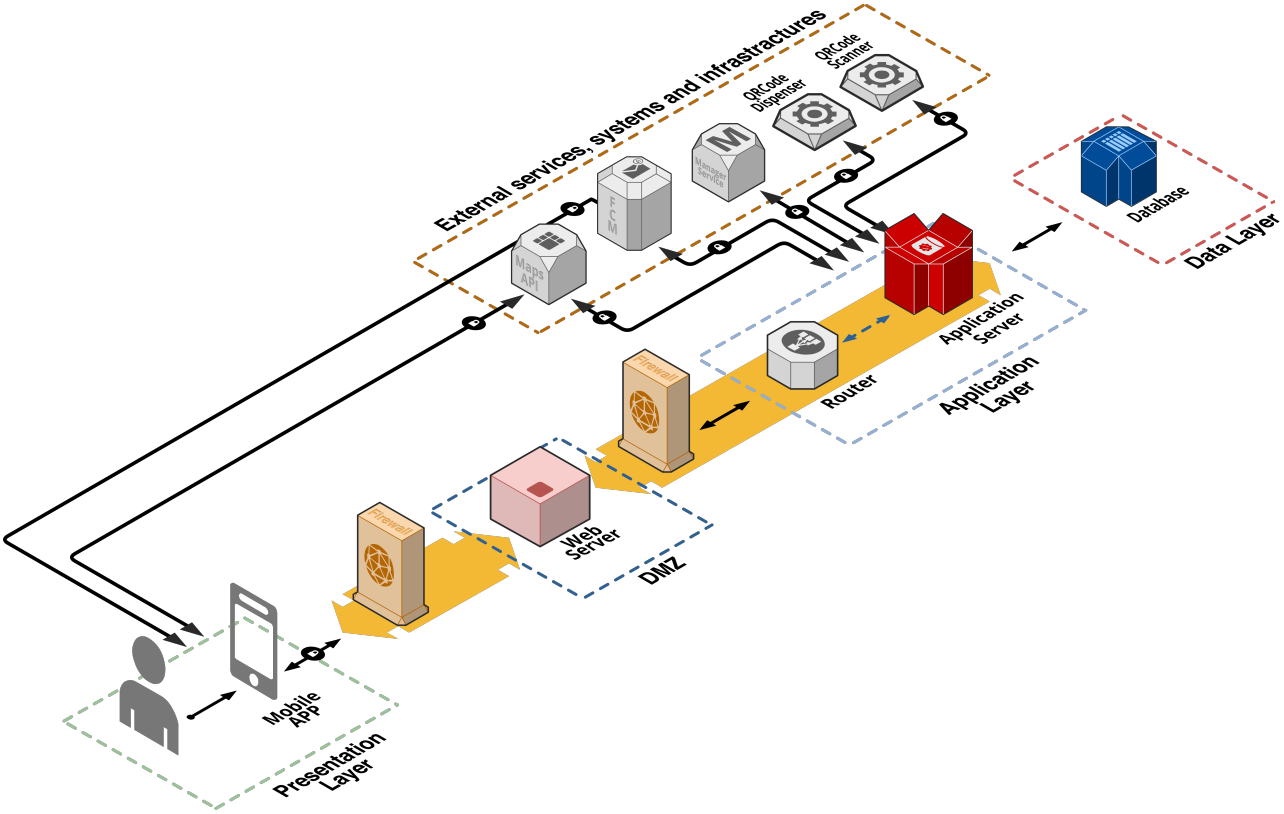

Figure 1 - System Architecture

As we can see, a demilitarized zone (DMZ) is created thanks to the usage of two firewalls: the first one must be configured to allow traffic destined to the DMZ only, the second one only allows traffic to the DMZ from the internal network. In this way we can isolate the public services of the system (i.e.: the web server) from the local, private LAN machines in our network.

The main components are the following:

- Mobile App: Application installed on the OnlineUsers' smartphone that communicates with the system. Its purpose is to show data to the OnlineUser and to forward his/her requests to the Application Server. The application will be available for both Android and iOS systems.
- Firewall: It is a network security system that monitors and controls incoming and outgoing network traffic based on predetermined security rules. It typically establishes a barrier between a trusted network and an untrusted network, such as the Internet.
- Web Server: Processes incoming network requests over HTTP(s), such as the REST API calls. It is the only entry point for the Mobile App to access the system functionalities.
- Router: It dispatches any incoming request to the specific component designed to interpret and process it.
- Application Server: Main back-end component on which the logic of the application takes place: it elaborates the requests coming from the App, interacts with the data layer and communicates with the external systems.
- Database: Component responsible for data storage.
- External services, systems and infrastructures: Systems that interact with CLup; they provide functionalities not internally developed but needed in order to provide CLup main functionalities.

2.2 Component view

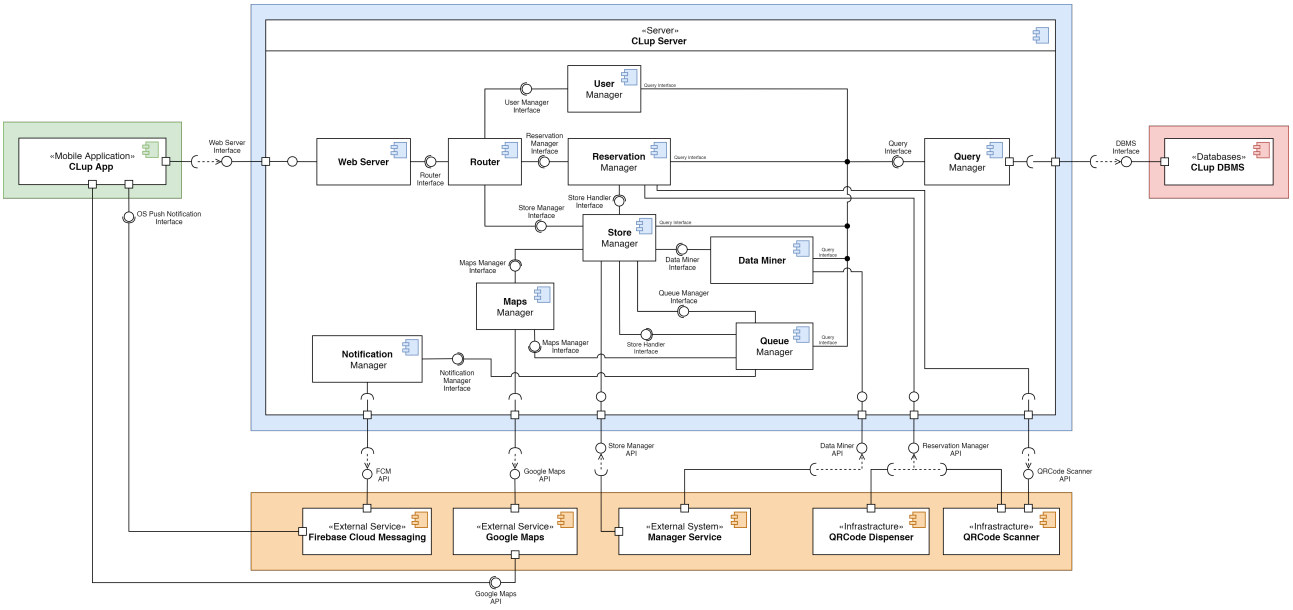

Figure 2 - Component diagram

The component diagram gives a specific view on the system focusing on the internal structure and showing how the components interact. Colors have been used to highlight the different tiers: green for the presentation layer, blue for the application layer, red for the data access layer and orange for the external services, systems, and infrastructures.

2.2.1 Presentation layer

The presentation layer represents the front-end of the system through which the user interacts with the software. It consists of just one component which is the **CLup App**, which means the mobile application. It is the software provided to the onlineUsers through which they can exploit all the product functions. The mobile application, to access all the functionalities offered by the backend, can only interacts with the Web Server, and it can make requests which are sent to three different subcomponents thanks to the Router:

- A request, redirected to the User Manager, which is about getting information on his personal account (change the password, delete the account, check the active and the older reservations).
- A request, redirected to the Store Manager, which is about the loading of the map with all the stores placemarks or the statistics on a certain store, or the selection of a store to make a reservation.
- A request, redirected to the Reservation Manager, which is about handling a reservation or completing a new one.

2.2.2 Application layer

The application layer implements the business logic of the S2B. It is the tier that connects the presentation layer with the data layer, coordinating the flow of information between them and keeping a persistent connection with the DBMS. All the components are now described with their main functionalities. Further details are given in section 2.5 and 2.6, describing the methods related to the interface and internal to the components.

- **Router** — As already mentioned in section 2.1, it communicates with the Web Server through TCP/IP protocol; its function is to forward requests and data coming from the Web Server to the right component in the Application Server.
- **Queue Manager** — This component manages the queue of each store. It also controls the notification process.
- **Notification Manager** — This component sends a notification to a certain OnlineUser in two different cases: the first one is when he has a pending line-up reservation and the waiting time is almost equal to the time he takes to reach the store; the second one is when he has a pending booking reservation which will start in two hours.
- **Query Manager:** — This is the only one component that interfaces with the data layer. It is linked to all the main components of the application server except for the Notification Manager. They send requests to the Query Manager about reading, inserting or deleting data from the DBMS. It is important to underline that the Store Manager and the Data Miner can make queries coming also from the Manager Service, that has a different visibility privilege.
- **User Manager** — It handles all the operations linked with users' account.
  Its sub-components and their respective functionalities are:
  - Login Handler: handles account authentication by checking credentials, querying the database through the Query Manager.
  - SignUp Handler: allows guests registration, checking if the chosen credentials (i.e.: email) are unique or if an account linked to them already exists.
  - Edit Data Handler: allows OnlineUsers to change their credentials.
  - Reservation History: retrieves the active, if any, and past reservations of the indicated OnlineUser.

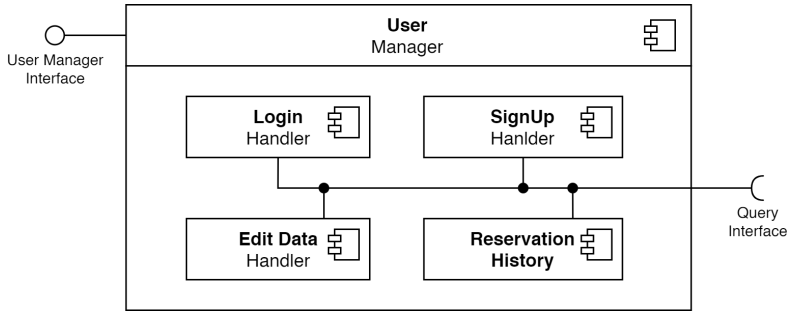

Figure 3 - User Manager details

- **Data Miner** It is used to mine statistics about stores and users. Its main components are:
  - ETL: it runs periodically the ETL (Extract, Transform, Load) process. In short, it extracts the data from the database, transforms it applying a series of rules and functions, and then loads the generated data into the Data Warehouse.
  - Miner Algorithm: provides to the ETL all the functions needed to transform the data coming from the database into new information (i.e.: statistics).
  - Manager Service Handler: provides a set of APIs that can be used from the external system "Manager Service" in order to retrieve statistics from the Data Warehouse. These statistics are on the store owned by the Manager that is using the Manager Service, and on the users that do shopping in that store.

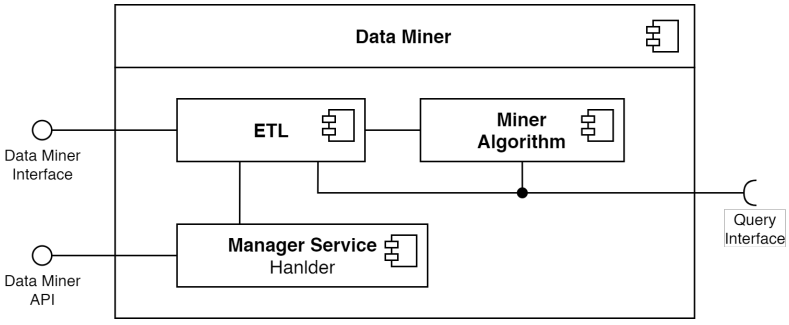

Figure 4 - Data Miner details

- **Reservation Manager** It receives and manages new reservation submissions done by users. Its subcomponents are:
  - Booking Handler: it handles the booking process.
  - LineUp Handler: it handles the lineup process. Also, the PhysicalUsers interacting with the QRcode Dispenser can request and create a reservation thanks to the API offered by this component.
  - QRCode Handler: it allows the generation of a QRCode containing all the data relative to a reservation. Also, to handle the scan of the QRCode, it exposes some API used by the QRCode Scanner and interfaces with the API offered by the scanner itself.
  - Delete Reservations: it is the component responsible for the cancellation of a reservation.

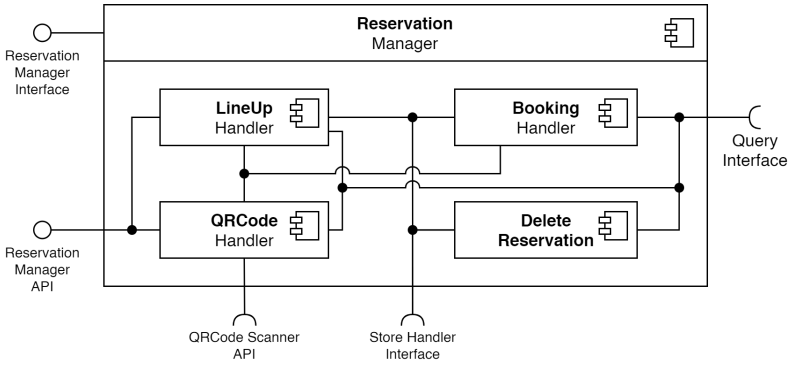

Figure 5 - Reservation Manager details

- **Store Manager** It has the aim of managing all what concerns the stores, and it is made of 6 subcomponents:
  - Suggested Stores Retriever: it allows to retrieve which group of stores can be considered as alternative suggestion of a given store.
  - Store Info: provides information concerning general aspects of the store, such as its capacity or its departments.
  - Queue Handler: queries the database through the *Query Manager* to handle the queue of the specific store. It also computes the waiting time.
  - Statistics Retriever: it is able to retrieve the statistics regarding a store interacting with the Data Miner Manager.
  - Map Handler: it is responsible to interface with the *Maps Manager Interface* to correctly position the stores on a map
  - Store Manager Handler: it deals with all what concerns the management of a store. In particular, it acts as link between the S2B and the Manager Service.

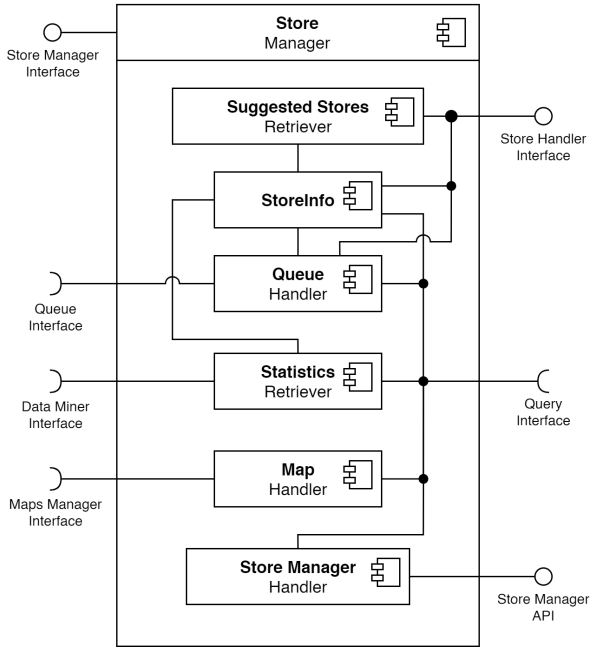

Figure 6 - Store Manager details

- **Maps Manager** This is the only one component which interface with Google Maps. In this way it is possible to replace this external service with another maps provider without the needing of modify all the components of the system, since the interface of Maps Manager will be the same.

2.2.3 Data access layer

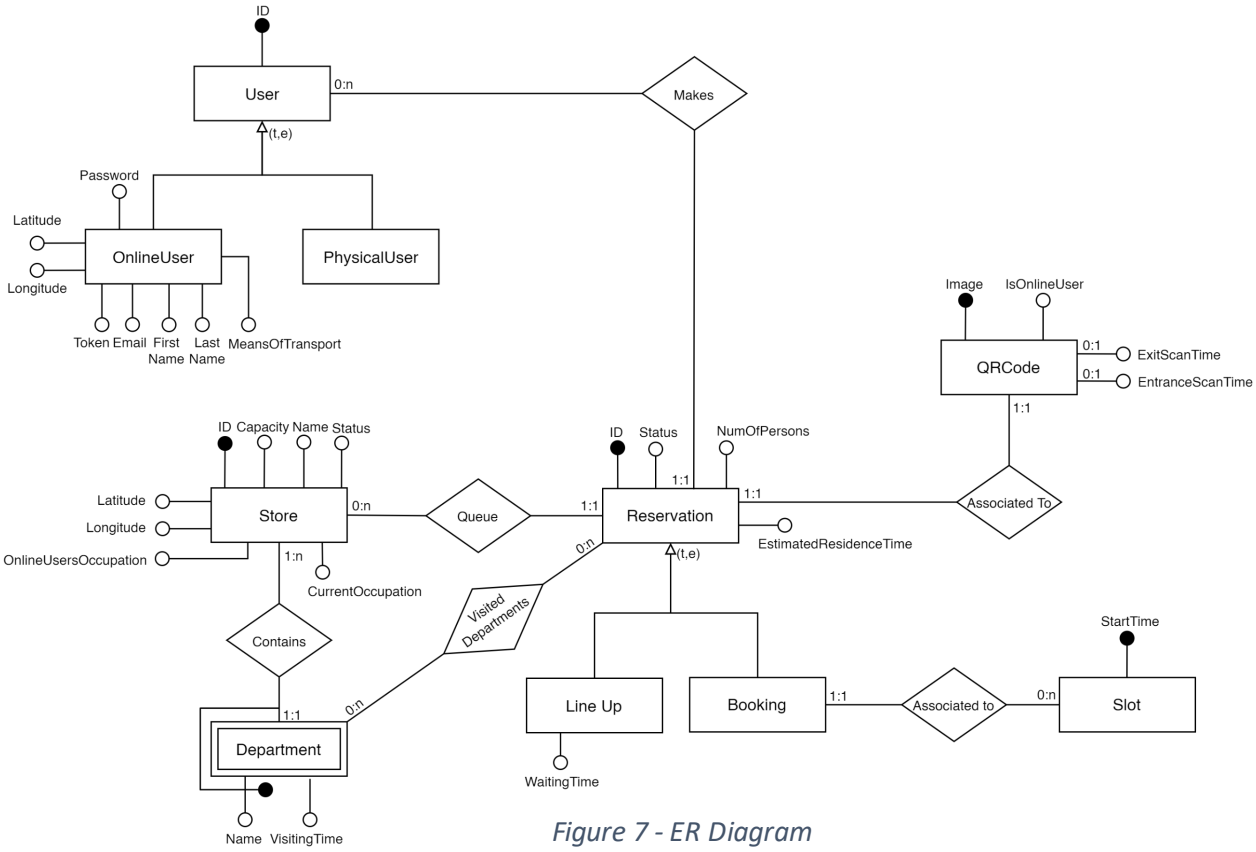

Figure 7 - ER Diagram

The data layer is composed by a relational database that contains all the information and by a data warehouse that stores and mines data. An entity-relationship diagram has been designed in order to provide a better overall view of the system. Now some details about the diagram are explained:

- The *User* is in relationship with the *Reservation* and the latter's cardinality is one. In this way it is possible to obtain the list of reservations made by a specific user.
- The relationship between *Store* and *Reservation* is called *Queue* because through it is possible to find all the reservations associated to a certain store. Moreover, by putting the condition Reservation.Status = 'Pending' it is possible to find the queue related to a given store (see section 2.8.2).
- The *Store* contains the attributes currentOccupation and onlineUserOccupation, which respectively represent the current number of users in the store and the current number of OnlineUsers in the store. These attributes are kept updated through two triggers (described in section 2.8.1) which are activated when a change of a reservation's status occurs.
- In the *QRCode* there are two attributes - EntranceScanTime and ExitScanTime – which are optional as when the QRCode is inserted into the DB these values do not exist yet but will be updated later on.
- The *Department* is a weak entity due to the fact that a department exits only inside a specific store. Hence, a department will be uniquely identified by its name and the associated store.
- The *VisitedDepartments* relationship associates to a reservation the visited departments. A reservation will have this information only if during the reservation process the OnlineUser has indicated the departments in which he intends to do shopping.

There is also a *TemporaryReservation* entity with the attributes and the relationships shown below.
It will be used to store the information of a reservation during its creation process. When a reservation will be completed (i.e.: the user sent all the required fields), it will be moved into the *Reservation* entity, using the *Type* attribute to determine the correct table between *Line Up* and *Booking*.
For the sake of clarity, this entity and its relationships are showed in a dedicated diagram. However, where the name is the same, there is a one-to-one correspondence between the entities of this diagram with the ones of the diagram in Figure 7.

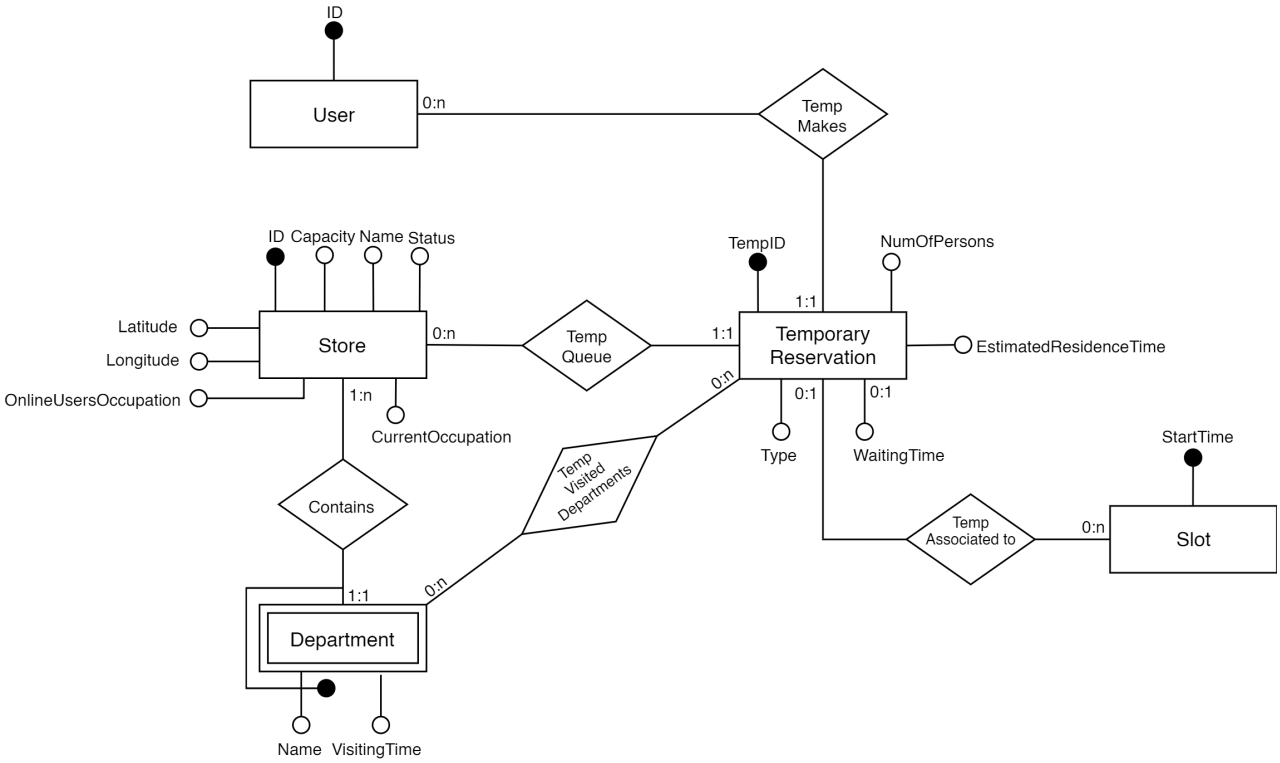

Figure 8 - ER Diagram: Temporary Reservation

In this way, we can avoid storing the reservation, during its creation process, in the main memory of the server. This allows to exploit all the advantages of a stateless interaction, which will be discussed in section 2.7.1.

Below two Dimensional Fact Models (DFM) are provided in order to better illustrate the measures for the analysis of the data and the dimensions through which data are mined for the two types of Reservations. LineUp and Booking have similar models, which are realized separately in order to have the statistics of each one of them. In the illustrations, the most significative measures for the realization of the statistics and other main queries are shown, while some of the support measures, omitted for simplicity, are: Entrance and Exit ScanTime of the QRcode and estimatedResidenceTime for the computation of the average time spent in a store by a usual user, and storeCapacity.

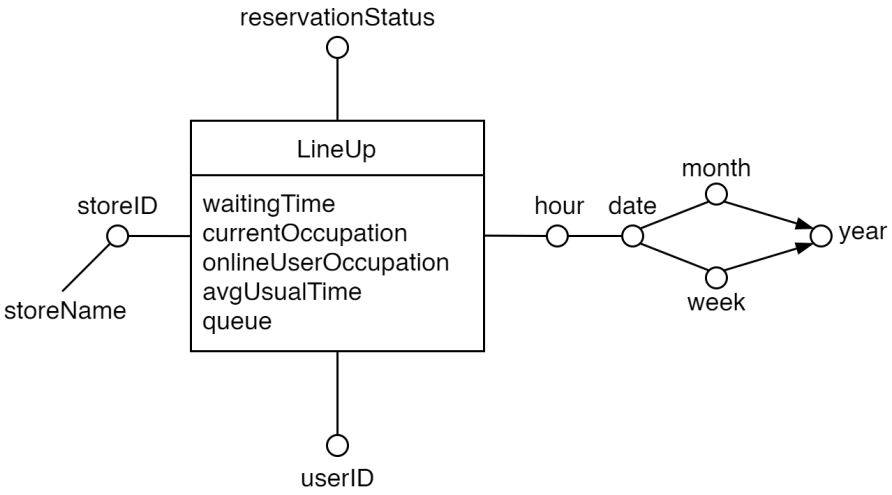

Figure 9 - DFM: LineUp

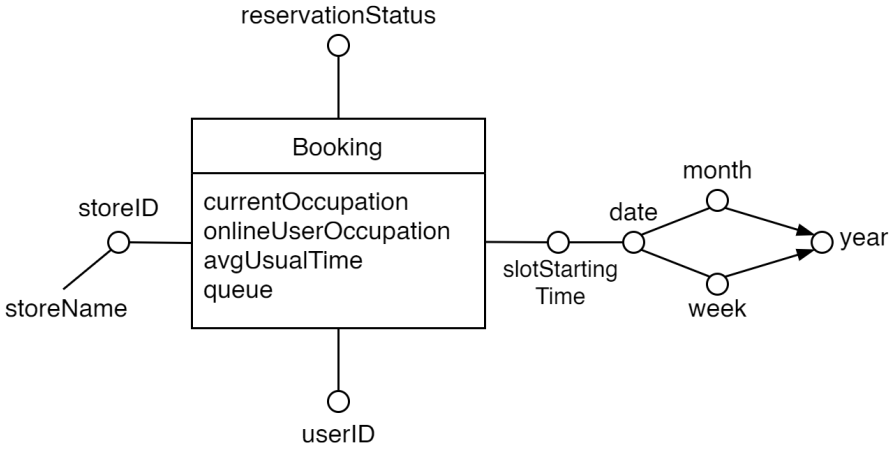

Figure 10 - DFM: Booking

2.2.4 External services, systems, and infrastructures

2.2.4.1 External services

- **Google Maps** It is a web mapping service used to render maps directly on the client side, to calculate distances from the onlineUser, and to calculate the time needed to reach a store from a given position.

- **Firebase Cloud Messaging** It is a is a cross-platform cloud solution for messages and notifications. It is used to send notifications to OnlineUsers' mobile application.

2.2.4.2 External systems

- **Manager Service** The S2B will have to interact with the **Manager Service,** that is an external system developed separately. They are linked through two subcomponents of the Application Server: the first one is the Store Manager, used to add a store to the CLup service, inserting it to the database with all the mandatory information such as the capacity or the position; the second one is the Data Miner, used to get statistics about the stores. Moreover, a manager is able to get further information about the stores that are associated to him.

  Summing up, the Manager Service is able to access the CLup DBMS through the Data Miner and to make several kinds of queries, such as insert store in the database. Nevertheless, the document will not release further details about this process as it does not concern to the developers of the S2B.

2.2.4.3 Infrastructures

- **QRCode Dispenser** This infrastructure is used to hand out tickets at the entrance of a store for the PhysicalUsers. It is able to generate a ticket that contains a unique QRCode, which is associated to a PhysicalUser's reservation, and the correspondent estimated waiting time.

- **QRCode Scanner**: This infrastructure is used to scan a QRCode at the entrance or exit of a store, since each user must scan his QRCode both entering and leaving the store. The scanner will analyze the QRCode and will send it to the system to allow it to register the entrance and exit time of each user and to monitor the influx of people in the store.

2.3 Deployment view

The deployment of the S2B, for the server part, will be done exploiting the services provided by Microsoft Azure. Microsoft Azure, if properly configured, handles server calls in a distributive way, and offers reliability, security, and scalability.

The benefits may be summarized as follows:

- **Scalability on Demand** Azure services are able to auto-scale according to the demands of the application usage. In particular, since the S2B is developed as a set of microservices, it is very easy to scale independently each microservice depending on its demand.
- **Flexibility**: Azure is simple to adapt and offers a host of application building blocks and services that will allow you to customize the cloud as needed.
- **Load Balancing** Azure offers load balancing of the incoming traffic in order to distribute it equally among all the virtual instances of the application server.
- **Virtual Machines** Azure provides VMs to host out apps and services. In this way there is no need to maintain or choose the physical hardware, leaving this job to the Microsoft experts.

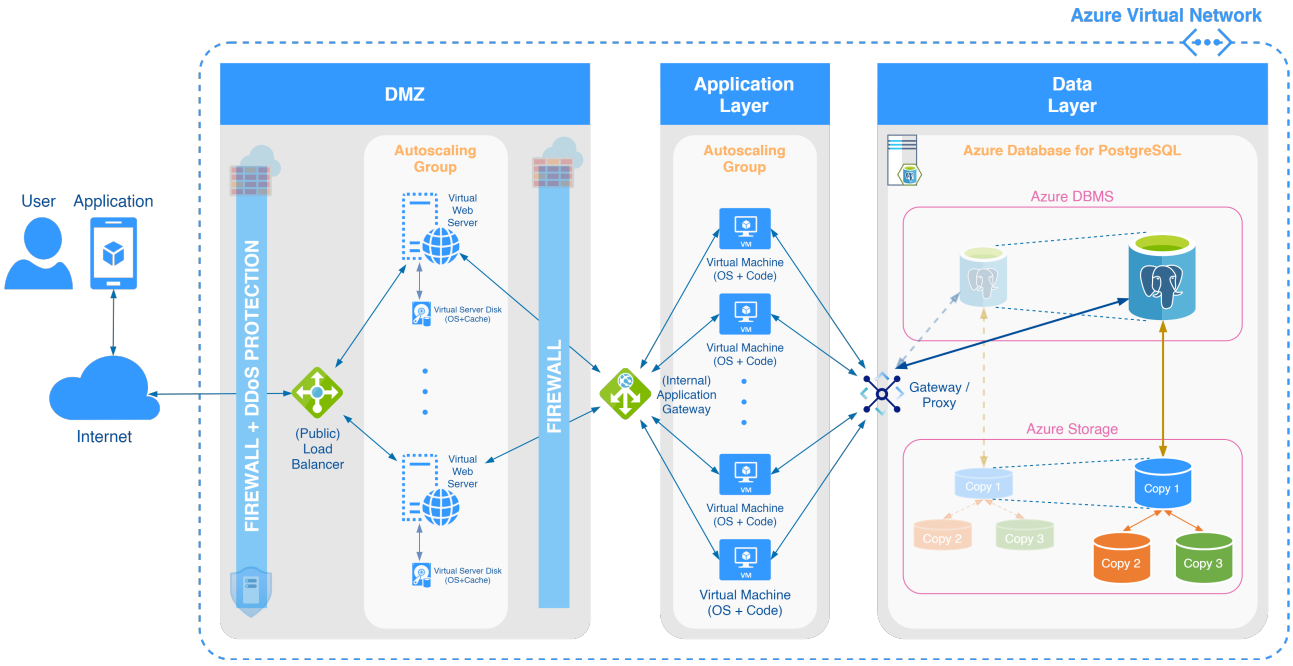

Figure 11 - Microsoft Azure Overview

In Figure 11, an overview of the exploited features offered by Microsoft Azure is shown.

The Azure Virtual Network is organized as follows:

- **DMZ** — The DMZ is realized thanks to the firewall capabilities offered by Azure. Also, a DDoS protection is added to mitigate possible DDoS attacks.
  The *Web Server* is implemented in an autoscaling group through *Virtual Web Servers*.
  The (Public) *Load Balancer* is the only entry point of the system and is used to distribute the incoming requests to the different virtual instances of the *Web Server*.

- **Application Gateway** — It acts as an advanced load balancer. Indeed, *Azure Load Balancer* works with traffic at Layer 4, while *Application Gateway* works with Layer 7 traffic, and specifically with HTTP (including HTTPS and WebSockets). In this way, the *Application Gateway* can make routing decisions based on URI path or host headers, distributing the traffic only among a specific set of virtual instances configured to handle the incoming request.

- **Application Layer** — The microservices that implements the business logic are allocated on different virtual machines inside an autoscaling group.

- **Data Layer** — For what concerns the data management, we rely on the *Azure Database for PostgreSQL*, which is architected to provide scalability and high availability.
  All the PostegreSQL physical data files are stored on *Azure Storage*, which is designed to store three copies of the data to ensure data redundancy, availability, and reliability.
  The Gateway acts as a database proxy, routing all client connections to the database server. [3]

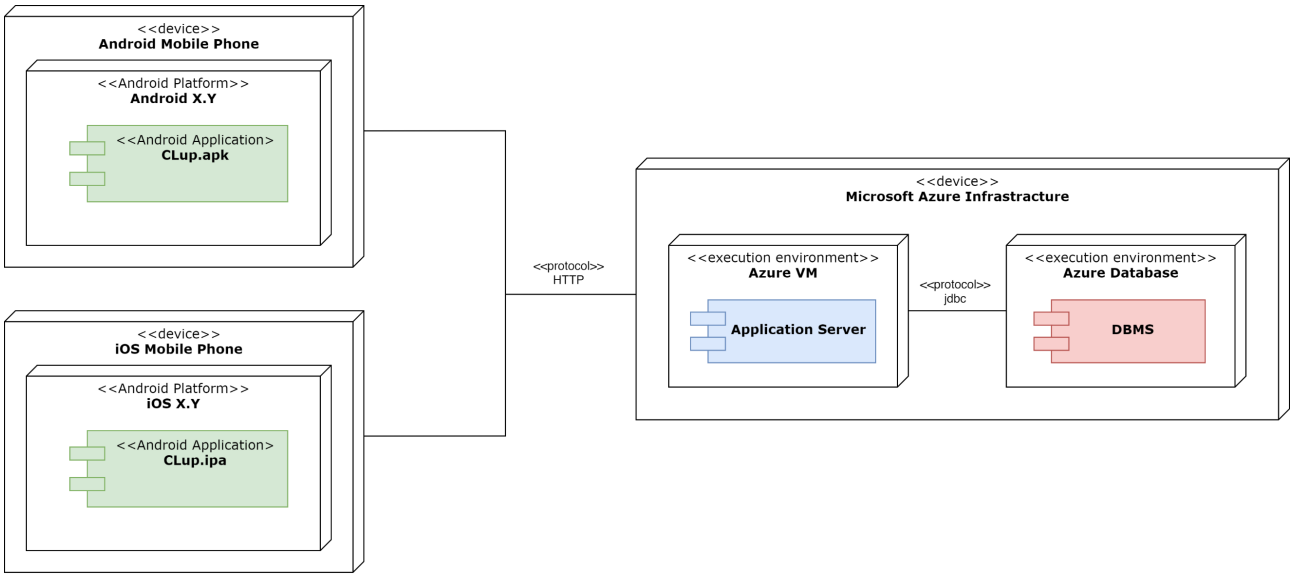

Figure 12 - Deployment View

Regarding the client side, the S2B is deployed for the two main mobile OS (Android and iOS), which interact through the HTTP protocol with the Microsoft Azure Infrastructure illustrated in Figure 11, here shown in a simplified way.

2.4 Runtime view

In this section, the sequence diagrams that represent the main interactions between the components of the system are presented. The methods are described in section 2.5 and 2.6. All the components have been colored following the same scheme used in section 2.2 to highlight different tiers.

General considerations: since we adopted a REST architectural style, there is not the concept of session. Due to this fact, each request will contain the URI to specify the addressed resource. Moreover, every request will contain an authentication header that will be used to identify uniquely the onlineUser and to check his permissions.

2.4.1 Signup

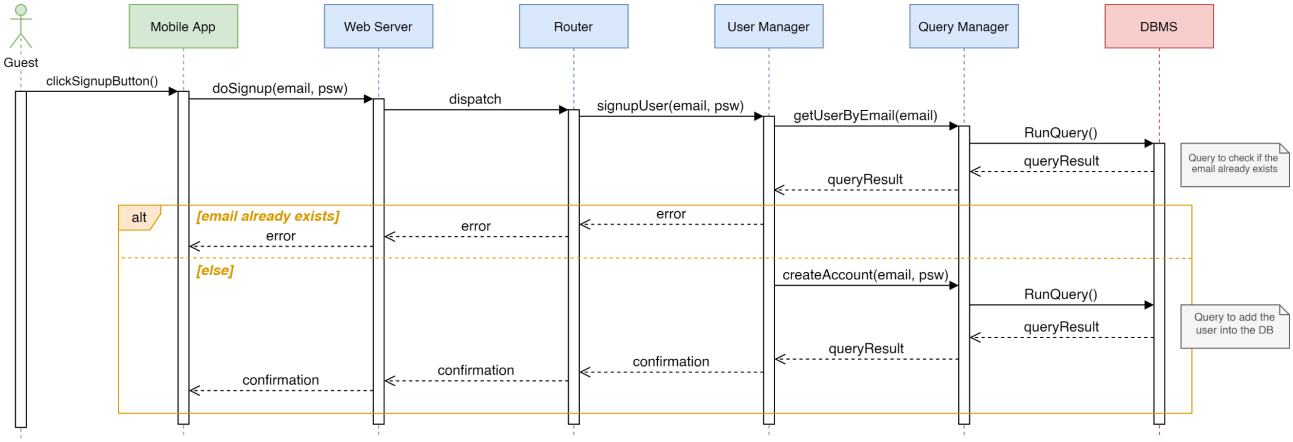

Figure 13 - Runtime View: Signup

The diagram in Figure 13 represents the sign-up operation of a new onlineUser. When the *User Manager* receives the data associated to the new account (email and password), it checks if the email is already associated to another account. If it is, the system returns to the client an error, otherwise the registration completes successfully, and a confirmation is sent.

2.4.2 Login

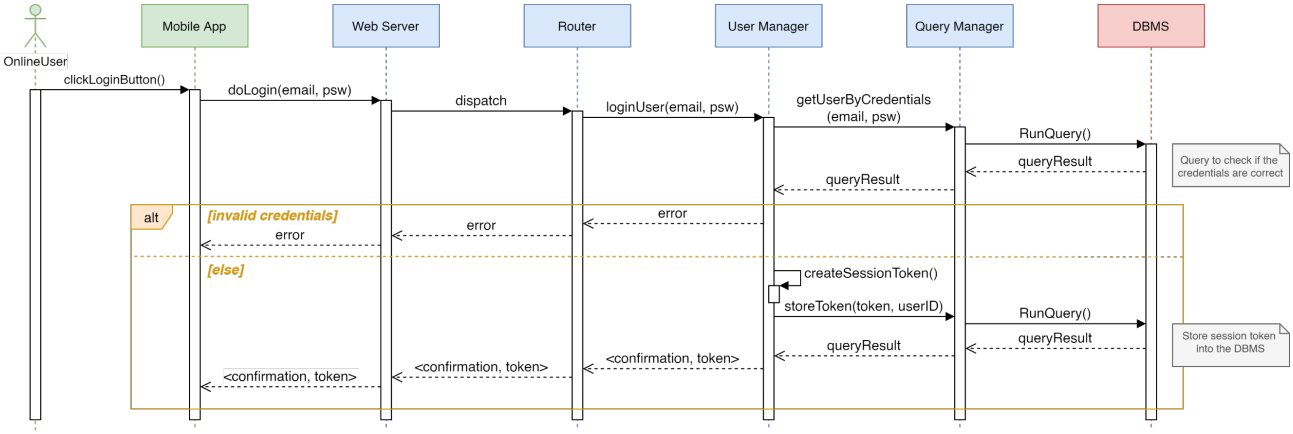

Figure 14 - Runtime View: Login

The diagram in Figure 14 represents the login of an onlineUser. When the *User Manager* receives the data, it checks if there is an account associated to the given email and password by doing a query to the DBMS. If it is not, the system returns to the client an error, otherwise the login completes successfully and a session token is stored into the DBMS.

2.4.3 Make Reservation

The diagrams in Figure 15, 16 and 17 represent the creation and completion of a new reservation, which is the core functionality of the S2B. For the sake of clarity, it has been split into three parts. The first one is referred to the initial part of the process, that is in common between the two types of reservation. The second one is referred to the final part of the process, which is divided between the two alternatives. Finally, the last one is referred to the retrieval of suggested stores.

As mentioned in section 2.2.3, there exists a temporary *Reservation* table in which the server can store all the information about a reservation during its creation process.
In the following three diagrams, each time on the *ReservationManager* is invoked a method to add an information about the ongoing reservation, a query will be performed to add this information into the *TemporaryReservation* table of the DB. To not add more complexity in the diagrams, these queries are omitted.

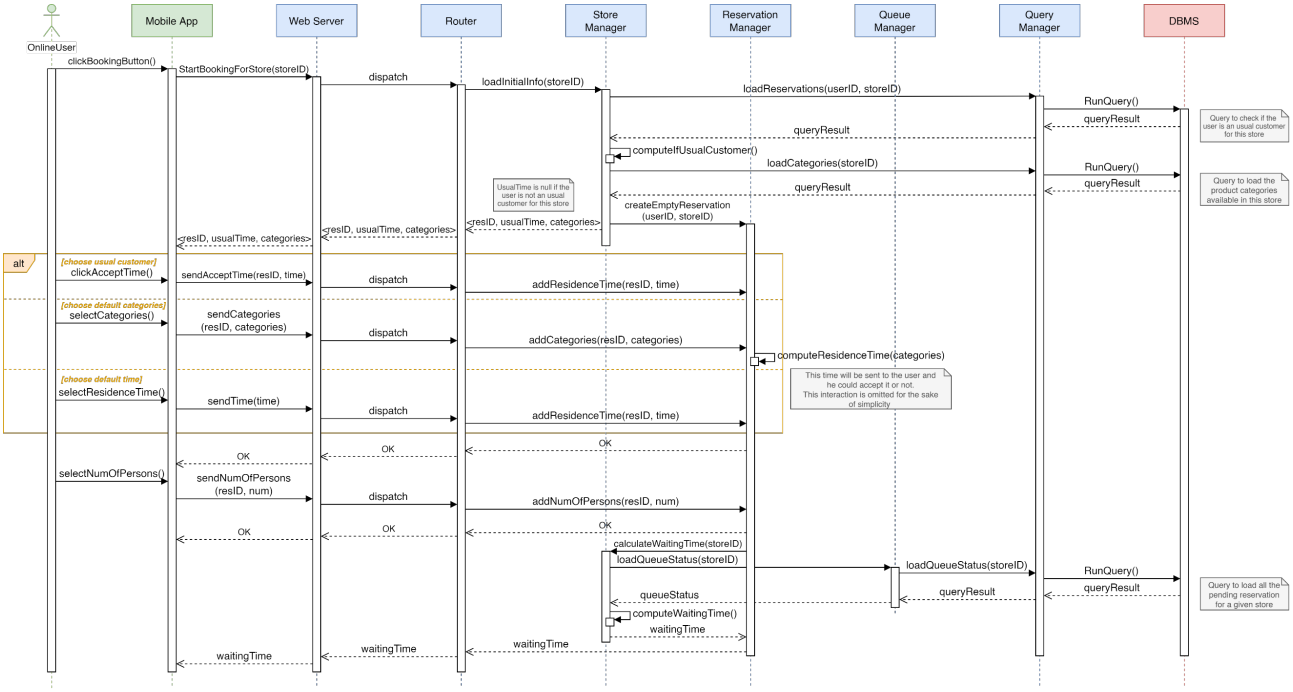

Figure 15 - Runtime View: Reservation Part 1

When the onlineUser selects a store, the system sends to the mobile application the related available categories and checks if the onlineUser has already been in the store multiple times. In this case, it computes the usual time by computing the average of the durations of the past visits. After that, the system creates a new empty temporary reservation and sends to the *Mobile App*, other than the information collected before, the ID of this temporary reservation. Each subsequent request will contain this ID in order to identify the reservation.
Then, the onlineUser is able to choose between three alternatives to estimate the residence time of his visit, which can be determined as the average of the previous visits, depending on the categories chosen or by choosing between predefined intervals. Moreover, he must choose the number of persons that will take part to the reservation. At this point, the *Store Manager* retrieves the pending reservations for the given store through a query and computes the waiting time, which is sent to the client.

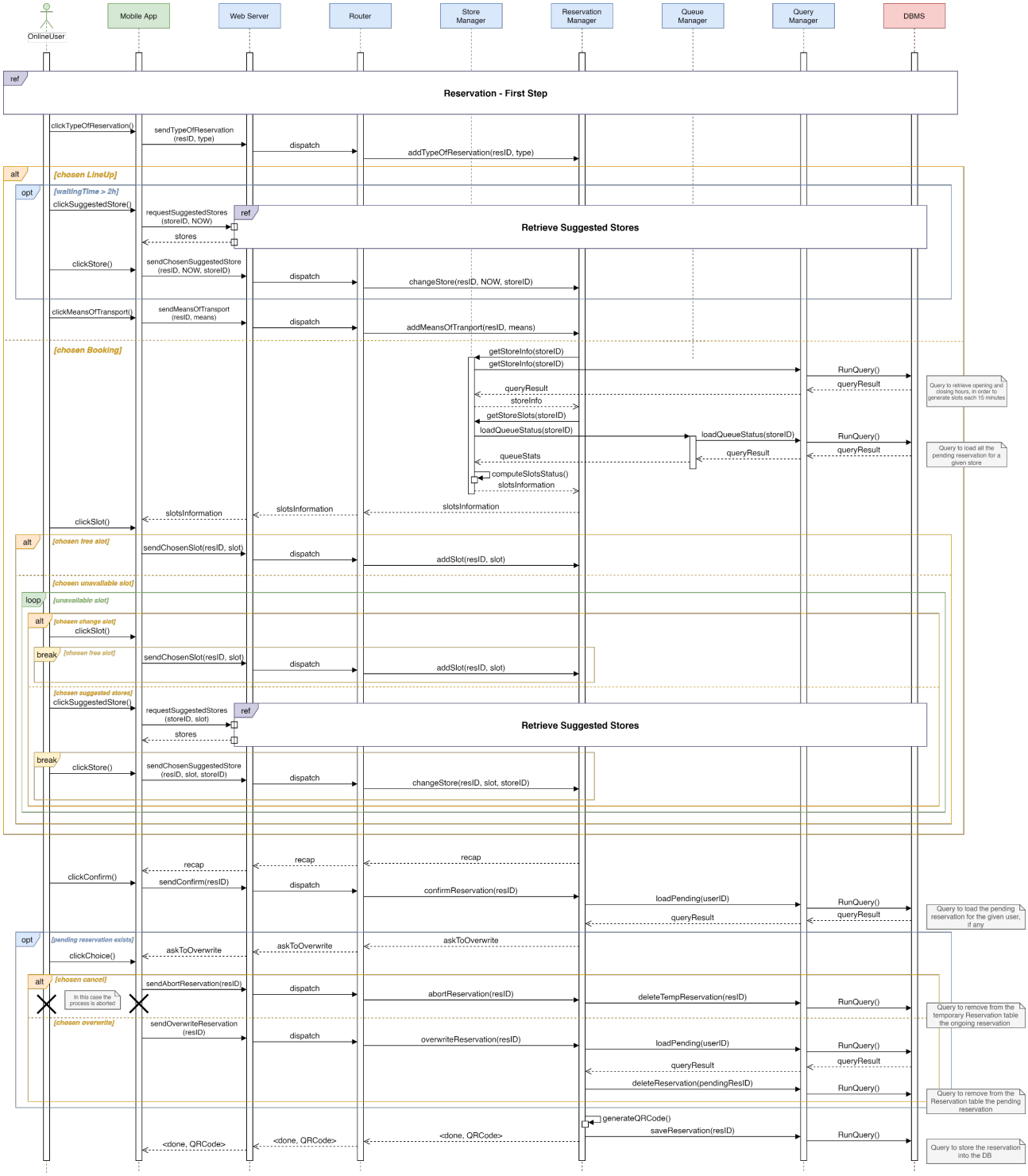

Figure 16 - Runtime View: Reservation Part 2

At this point, the onlineUser is able to choose the type of reservation he wants to make. If he chooses the line-up alternative, the waiting time will be displayed and the only thing he has to do is confirm the means of transport that he will use to reach the store. On the other hand, if he chooses the booking alternative, the *Store Manager* makes a query to generate the slots depending on the opening hours of the store. Afterwards, it retrieves the pending reservations in order to obtain information about the slots and to distinguish the available ones from the busy ones. Once the analysis is completed, all the information is sent to the mobile application and the onlineUser can choose one of the available slots or change the store picking one between the suggested ones. This process is repeated until the onlineUser selects and confirms a free slot or

an alternative store. Then, the reservation is confirmed by the user and at this point the system checks if it already exists a pending reservation by this user. In this case, the user must choose if cancel the ongoing reservation, and abort the reservation process, or overwrite the pending one.
The reservation is then saved into the database by the *Reservation Manager*, which also generates the corresponding QRCode.

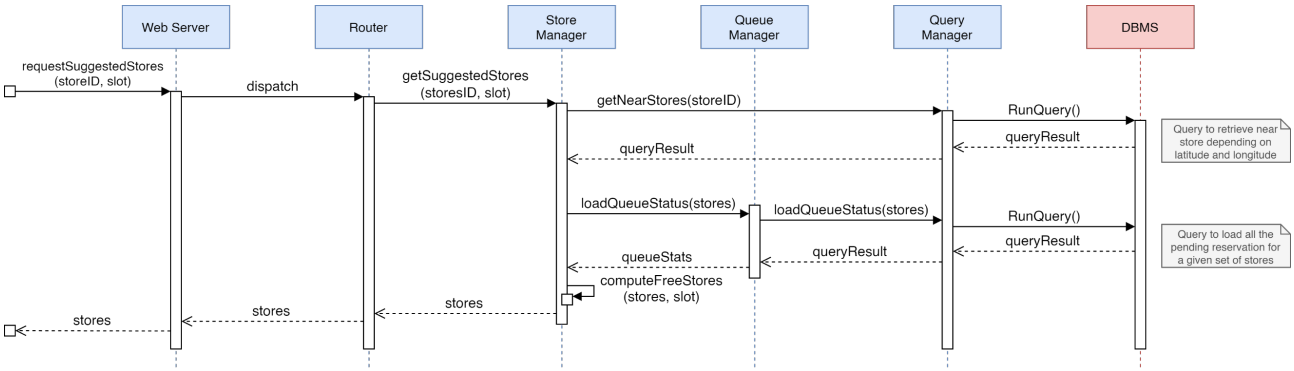

Figure 17 - Runtime View: Retrieve suggested stores

During a booking, if the onlineUser does not want to change the chosen slot when unavailable, he is able to select an alternative store between the suggested ones. In this case, the *Store Manager* retrieves the list of stores which are near to the one associated to the current reservation and retrieves the current queue for these stores to compute which ones are available during the slot of time that the onlineUser has chosen. This process can be also invoked during a lineup, and in that case the list of near stores where the waitingTime is less than 2 hours is retrieved. Lastly, the *Store Manager* forwards the suggested stores to the mobile application.

2.4.4 Delete Reservation

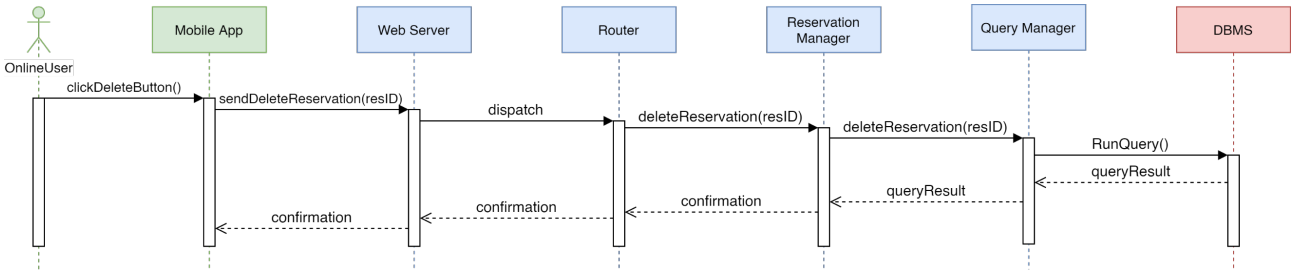

Figure 18 - Runtime View: Delete Reservation

The diagram above illustrates the procedure of deletion of a reservation done by an onlineUser. When the delete button is clicked, a request is sent to the Mobile App with the ID of the reservation to cancel. The DBMS is queried to find and delete the reservation. If the procedure ends successfully, a confirmation is sent.

2.4.5 Notification – LineUp

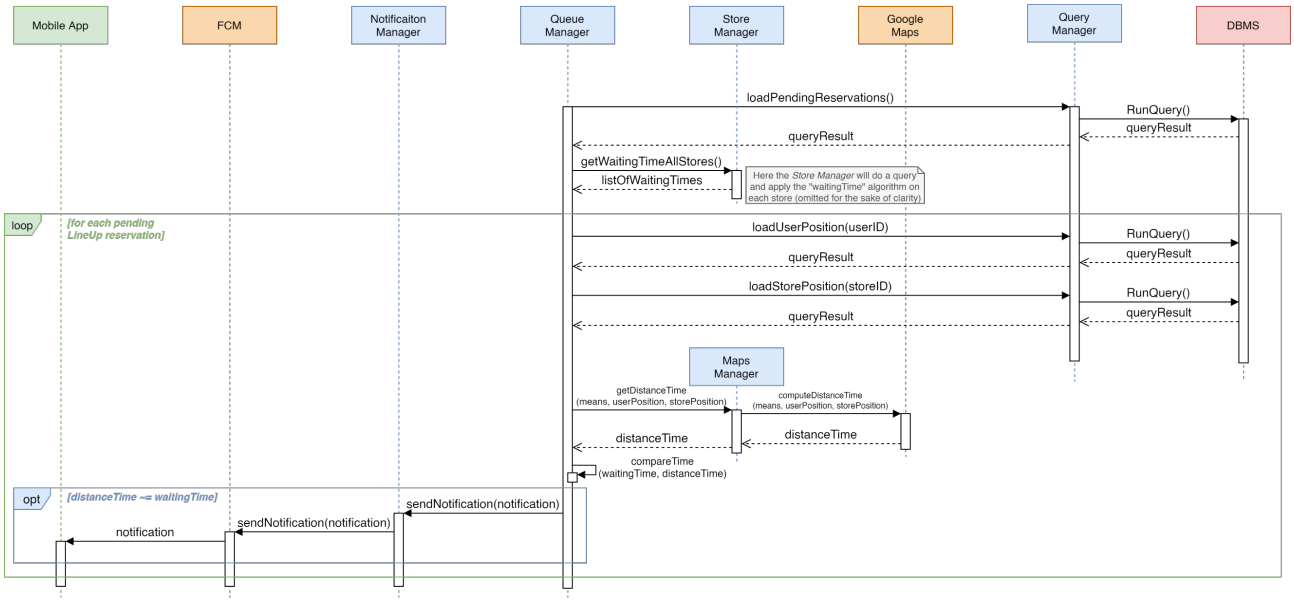

Figure 19 - Runtime View: Notification - LineUp

The sequence of actions behind the sending of a notification to an onlineUser with a pending reservation of type LineUp, shown in the diagram, consists of: a periodic request from the Queue Manager to retrieve the pending reservations, then the calculation of the waiting time for the reservations, done by the Store Manager, after a query to the DBMS. Afterwards, the position of the onlineUser and of the chosen store is loaded. At this point, the Map Manager computes the time it will take the onlineUser to reach the store of the pending reservation (DistanceTime) and the comparison between the waiting time of the queue and the distance time is done. If the first one is about the same of the second, (since the periodic computation might not find the exact moment the two values are the same) the notification is sent to the specific onlineUser.

2.4.6 Notification – Booking

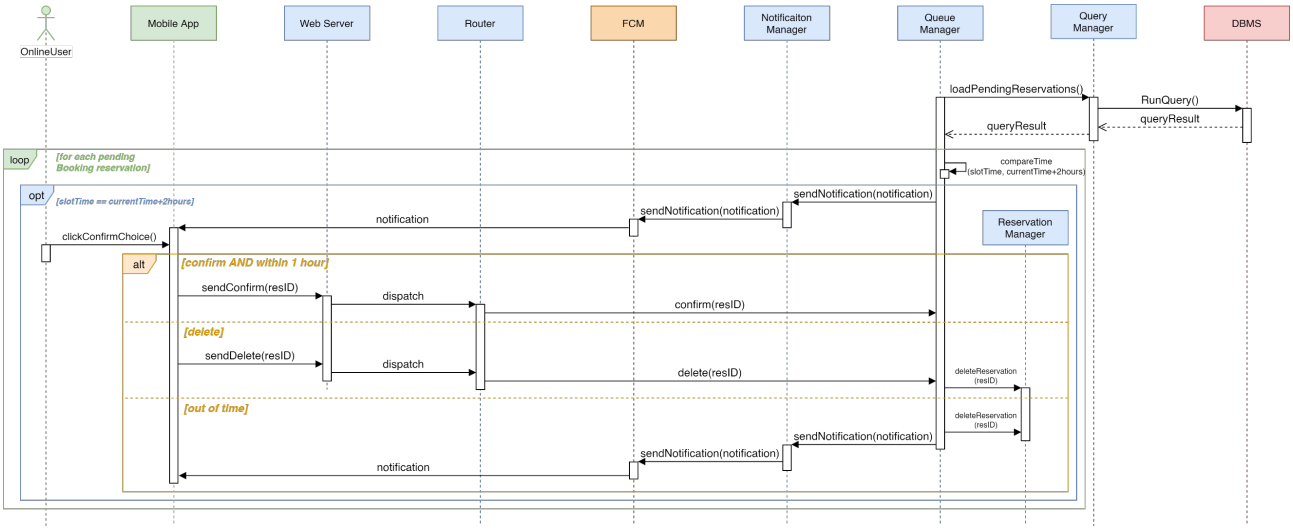

Figure 20 - Runtime View: Notification - Booking

The diagram in Figure 20 illustrates the process behind the sending of a notification to an onlineUser with a pending reservation of type Booking. The Queue Manager periodically loads the pending reservations and for each of them of type Booking it compares the time of the booked slot with the current time incremented by 2 hours. If the two values are about the same, a notification is sent to the onlineUser which has to confirm the reservation. If the onlineUser chooses to not confirm the reservation or the timer of one hour expires without a confirmation, the reservation is deleted.

2.4.7 QRCode Scan

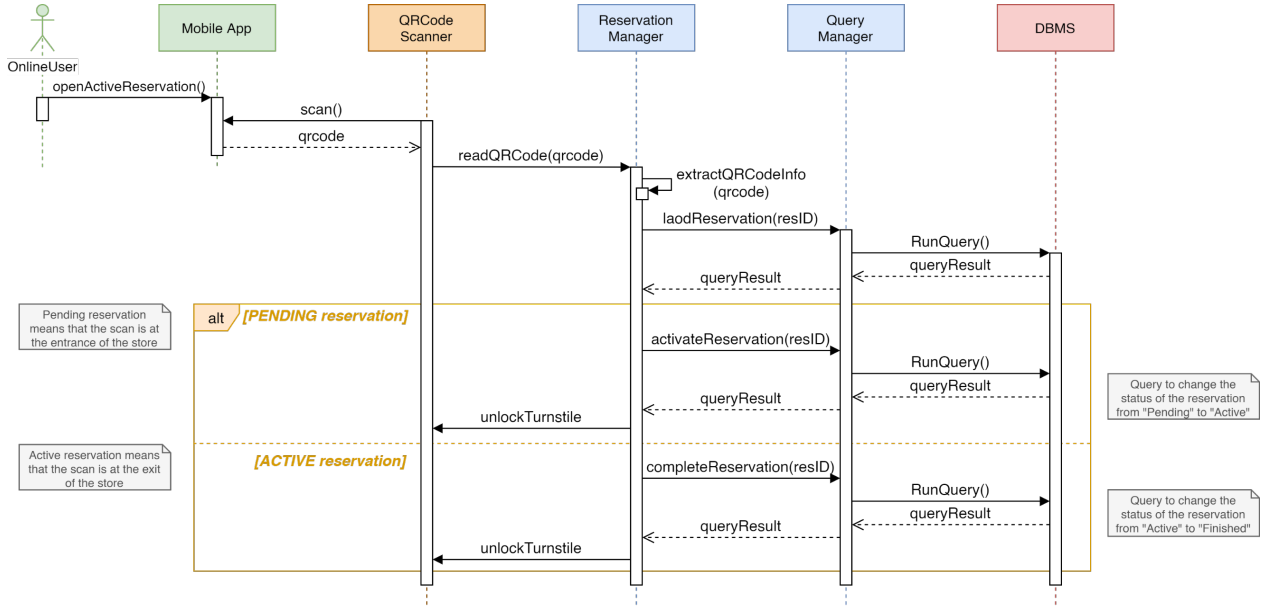

Figure 21 - Runtime View: QRCode Scan

The sequence diagram above illustrates the QRCode Scanning operation. When the onlineUser opens the active reservation, the QRCode scanner reads the QRCode and sends it to the Reservation Manager which extracts, loads and sends the request to find the reservation in the DBMS. If the reservation is 'pending', it means that the onlineUser is entering the store, so it is activated, its status is changed with a query and the turnstile is unlocked. Otherwise, if the reservation is 'active', it means that the onlineUser has finished the visit, so the status of the reservation is changed to 'finished' and the turnstile is unlocked.

2.4.8 Retrieve Statistics

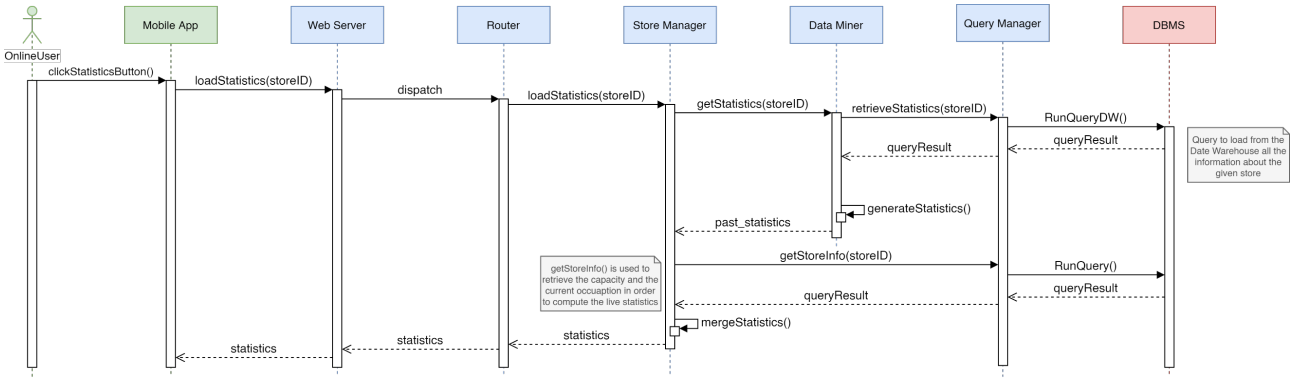

Figure 22 - Runtime View: Retrieve Statistics

The MobileApp has the functionality of displaying statistics about a store when requested by the onlineUser. The DBMS is queried to load all useful information from the Data Warehouse. The query result is then used by the Data Miner to generate the statistics, such as the occupation of the chosen store on different days of the week. Another query is necessary to display in the MobileApp also the live statistics of a store such as the current occupation.

2.4.9 Mine Data

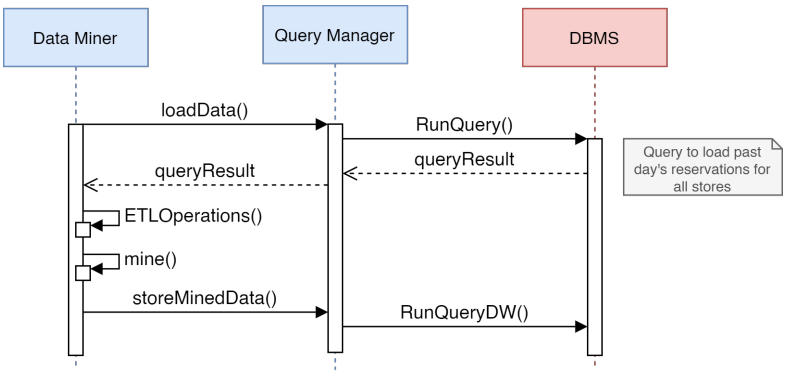

Figure 23 - Runtime View: Mine Data

In order to maintain data constantly updated, every day the system automatically executes a mining process on the new data, loading all the reservation which has been completed in the last 24 hours. This process is made through the *Data Miner*.

2.5 Component interfaces

In this section, details about the method used in the sequence diagrams presented in section 2.4 are provided. Moreover, some other methods have been added as they will be part of the S2B even if they are not part of the runtime views.

2.5.1 User Manager Interface

- *signUpUser(email, psw)*: creates a new account, throwing an exception if the given email is already associated to another onlineUser.
- *loginUser(email, psw)*: starts the process of login of an onlineUser.
- *deleteUser(email)*: delete an account, throwing an exception if it does not exist.
- *updatePosition(userID, position)*: periodically invoked by the Server to update the current onlineUser position.

2.5.2 Reservation Manager Interface

- *createEmptyReservation(userID, storeID)*: starts the process of creation of a new reservation associating it with a specific user and store. This reservation record will be stored into a dedicated table (*TemporaryReservation*) of the database and will updated during the booking process. When the reservation will be confirmed by the user, this record can be moved into the table of the completed reservations.
- *addTypeOfReservation(resID, type)*: defines the type (line-up or booking) of the reservation identified by resID.
- *addMeansOfTransport(resID, means)*: adds the chosen means of transport to a line-up.
- *addSlot(resID, slot)*: adds the chosen slot to a booking.
- *changeStore(resID, slot, storeID)*: modifies the store of a booking and add the corresponding chosen slot.
- *confirmReservation(resID)*: confirms the creation of a new reservation checking if there is already an existing one.
- *abortReservation(resID)*: deletes the ongoing reservation from the *TemporaryReservation* table.
- *overwriteReservation(resID)*: overwrites the pending reservation of a user with the one identified by resID.
- *addResidenceTime(resID, time)*: adds to the reservation the residence time, which can be generated by the system based on the time of the previous visits done by the onlineUser or chosen between the default available intervals.
- *addCategories(resID, categories)*: adds the chosen categories to a reservation.
- *addNumOfPersons(resID, num)*: adds to the reservation the number of persons that will visit the store.
- *deleteReservation(resID)*: deletes a certain reservation from the *Reservation* table.
- *readQRCode(qrcode)*: reads the given QRCode (as base64 string) extracting all the information from it (e.g.: resID, storeID, etc.).
- *createPhysicalReservation(data)*: creates a new reservation made by a physicalUser. Data contains all the information related to the reservation encoded as JSON.

2.5.3 Store Manager Interface

- *loadInitialInfo(storeID)*: sends to the mobile application the available categories of a store and the usual time that the onlineUser spend inside of it.
- *getSuggestedStores(storeID, slot)*: starts the process which suggests alternative stores to the onlineUser.
- *loadStatistics(storeID)*: sends to the mobile application the statistics about a specific store.
- *addNewStore(data)*: this method is only invoked by the Manager Service to add a new store to the system.
- *modifyStore(storeID, data)*: this method is only invoked by the Manager Service to modify some parameters about a store.
- *loadMap(position)*: loads the map related to a specific position with all the nearest stores.

2.5.4 Store Handler Interface

- *getStoreInfo(storeID)*: gets general info of a store (ex: opening hours).
- *getStoreSlots(storeID)*: gets all the information about the slots of a store.
- *calculateWaitingTime(storeID)*: starts the process of retrieving the waiting time of a store.
- *getWaitingTimeAllStores()*: returns the waiting time of all the stores.

2.5.5 Notification Manager Interface

- *sendNotification(notification)*: sends a notification to the mobile application.

2.5.6 Queue Manager Interface

- *loadQueueStatus(stores)*: gets the current queue of a set of stores.
- *loadQueueStatus(storeID)*: gets the current queue of a store.
- *confirm(resID)*: keeps the confirmed booking in the queue.
- *delete(resID)*: removes the deleted booking from the database and consequently from the queue.

2.5.7 Map Manager Interface

- *getDistanceTime(means, userPosition, storePosition)*: returns the distance of an onlineUser from a specific store taking into account also the means of transport.

2.5.8 Data Miner Interface

- *getStatistics(storeID)*: returns the past statistics related to a store.

2.5.9 Query interface

- *getUserByEmail(email)*: retrieves from the DB the onlineUser associated to an email, returning an empty result if it does not exist.
- *createAccount(email, psw)*: inserts into the DB a new account with all the information given during the registration.
- *deleteAccount(email)*: removes from the DB the account associated to the given email.
- *getUserByCredentials(email, psw)*: retrieves from the DB the onlineUser related to a specific email and password, returning an empty result if it does not exist or if the password is incorrect.
- *storeToken(token, userID)*: stores into the DB the authentication token associated to a given user.
- *getStoreInfo(storeID)*: retrieves from the DB general info about a store, as the opening hours or the capacity.
- *loadQueueStatus(stores)*: retrieves from the DB the pending reservations (which represents the current queue) of a set of stores.
- *loadQueueStatus(storeID)*: retrieves from the DB the pending reservations (which represents the current queue) of a store.
- *loadPending(userID)*: retrieves from the DB the pending reservation related to an onlineUser, returning an empty result if it does not exist.
- *deleteTempReservation(resID)*: removes from the DB a temporary reservation.
- *deleteReservation(prendingResID)*: removes from the DB a reservation.
- *saveReservation(resID)*: moves the ongoing reservation from the TemporaryReservation table of the DB to the Reservation table.
- *loadReservations(userID, storeID)*: retrieves from the DB the past reservations of an onlineUser related to a store.
- *loadReservation(resID)*: retrieves from the DB the reservation identified by resID.
- *loadCategories(storeID)*: retrieves from the DB the available categories of a store.
- *getNearStores(storeID)*: retrieves from the DB the stores which are near to a specific store.
- *retrieveStatistics(store)*: retrieves from the DW the statistics related to a store.
- *loadData()*: retrieve from the DB all the reservations related to the previous day.
- *storeMinedData()*: inserts into the DW the new statistics.
- *loadPendingReservations()*: retrieves from the DB all the pending reservations.
- *loadUserPosition(userID)*: retrieves from the DB the current position of an onlineUser.
- *activateReservation(resID)*: sets the status of the given reservation to "Active".
- *completeReservation(resID)*: sets the status of the given reservation to "Complete".

2.5.10 External interfaces

- *computeDistanceTime(means, userPosition, storePosition)*: related to the **Google Maps** interface. It returns an integer which represents the time required to reach a store by a certain position using a specific means of transport.
- *sendNotification(notification)*: related to the **FCM** interface. It sends a notification to the client.
- *unlockTurnstile()*: related to the **QRCode Scanner** interface. It unlocks the turnstile at the entrance of the store to allow a user to enter.

2.6 Components Methods

There are several methods which are internal to a specific component and not publicly exposed through the interfaces.

- *createSessionToken():* internal to the **User Manager**. It creates a unique authentication token that will be sent to the client, which then will attach it to every request as proof of his authentication into the system. In this way the server can check whether the user is authorized to perform the actions or not.
- *computeResidenceTime(categories)*: internal to the **Reservation Manager**. It computes the estimated residence time through the selected categories.
- *generateQRCode()*: internal to the **Reservation Manager**. It generates the QRCode associated to a certain reservation.
- *computeIfUsualCostumer()*: internal to the **Store Manager**. It computes the usual time spent by an onlineUser inside a store (it returns null if the onlineUser is not a usual customer).
- *computeFreeStores(stores, slot)*: internal to the **Store Manager**. It computes which are the stores, inside the provided set, that are available on a specific slot of time (if the reservation is of type lineup, 'NOW' is passed as the *slot* parameter).
- *computeSlotsStatus()*: internal to the **Store Manager**. It computes all the information about the slots of a store such as which ones are still available and which ones are not.
- *computeWaitingTime()*: internal to the **Store Manager**. It computes the waiting time of a store analyzing the current queue. As this represents the main computation made by the system, more details about the algorithm are provided in section 2.8.3.
- *mergeStatistics()*: internal to the **Store Manager**. It merges the statistics retrieved by the Data Warehouse with the current occupation.
- *generateStatistics()*: internal to the **Data Miner**. It generates statistics.
- *ETLOperations()*: internal to the **Data Miner**. It makes the operation related to the ETL component.
- *mine()*: internal to the **Data Miner**. It mines the data.
- *compareTime(time1, time2):* internal to the **Queue Manager**. It compares two times and return true if they are almost equal.
- *extractQRCodeInfo(qrcode)*: internal to the **Reservation Manager**. It extracts from the given base64 string all the contained information.

2.7 Selected architectural styles and patterns

2.7.1 Architectural styles

As already mentioned, the mobile application shall use RESTful API to communicate with the server. REST is designed to build systems that are lightweight, maintainable, and scalable (see "Stateless" advantage in the table below).
Azure offers different features to exploit at its best the REST principle, such as the load balancer and the application gateway components.

The key advantages of the REST style are:

- **Stateless** Every HTTP(S) request encapsulates all the info needed for its execution and the server does not have to maintain any session information, allowing different requests to be handled from different server nodes.

- **Cache** Since the requests are stateless, it is possible to define some of the responses to specific HTTP requests as cacheable.

- **Standardization** It provides a standardized way of communicating between client and server. In other words, it does not matter how the server is put together or how the client is coded up, as long as they both structure their communications according to REST architecture guidelines, using HTTP.

It should be noted that in the Runtime Diagrams (Section 2.4) the methods between the Mobile App and the Web Server are only to represent the interaction. However, in practice, these methods are mapped to the corresponding REST API, to be compliant with the standardized HTTP method semantics (GET, PUT, POST, DELETE).

When needed, the body of a request/response will contain data formatted in JSON.

2.7.2 Patterns

- **Observer Pattern** This pattern has been used in order to implement in a correct way the Notification Manager. The S2B should do a periodical verification and notify the interested users to approach the store when their reservation time is near. More details are provided in section 2.1.2 of the RASD, which also contains a UML diagram of the pattern.

- **Adapter Pattern** This pattern should be used in order to be able to interface with the different external API which might be subject to changes.

- **Facade Pattern** In order to improve the readability and usability of interfaces provided by some complex components, the facade pattern has to be used. For instance, it is used in the Router to allow the client to access the system through a simple interface and to hide the complexity of internal deployment.

2.8 Algorithms

In this section the algorithms behind some of the most important computations and processes of the S2B are proposed using a pseudocode format.

2.8.1 Triggers

For the computation of real-time statistics, two triggers are executed in order to automatically update the currentOccupation and onlineUsersOccupation attributes. These values are often displayed and used during the main operations done by the system, so it is important to guarantee the validity and integrity of these metrics. For the implementation of the triggers, two tables are considered: Reservation and Store; the activation of the triggers happens when there is an update on the first table while the latter table is the one in which the interested values are increased or decreased depending on the modification that has occurred in the reservation status.

```
CREATE TRIGGER UpdateCurrentOccupation after UPDATE
ON Reservation
FOR EACH row
begin
  IF ( new.Status = 'Active' AND old.Status = 'Pending' ) THEN
    UPDATE Store
    SET    CurrentOccupation = CurrentOccupation + 1
    WHERE  ID = new.StoreID;
  ELSEIF ( new.Status = 'Finished' AND old.Status = 'Active' ) THEN
    UPDATE Store
    SET    CurrentOccupation = CurrentOccupation - 1
    WHERE  ID = new.StoreID;
  END IF;
end


CREATE TRIGGER UpdateOnlineUsersOccupation after UPDATE
ON Reservation
FOR EACH row
begin
  IF ( new.Status = 'Active' AND old.Status = 'Pending' AND
       new.IsOnlineUser = 1) THEN
    UPDATE Store
    SET    OnlineUsersOccupation = OnlineUsersOccupation + 1
    WHERE  ID = new.StoreID;
  ELSEIF ( new.Status = 'Finished' AND old.Status = 'Active' AND
           new.IsOnlineUser = 1) THEN
    UPDATE Store
    SET    OnlineUsersOccupation = OnlineUsersOccupation - 1
    WHERE  ID = new.StoreID;
  END IF;
end
```

2.8.2 Queries

In this section, the main queries computed by the system are listed out:

- Retrieve the pending reservation of a user 'X'

```
SELECT R.ID
FFOM User AS U, Reservation AS R
WHERE U.ID = R.UserID AND U.ID = 'X' AND R.Status = 'Pending'
```

- Retrieve the queue related to a store 'X'

```
SELECT R.ID
FROM User AS U, Reservation AS R
WHERE S.ID = R.StoreID AND S.ID = 'X' AND R.Status = 'Pending'
```

2.8.3 Computation of the waiting time

In the following Java-like pseudocode, the procedure of the computation of the waiting time is shown.
The code is divided in 4 sections to highlight the main steps.

```
(1)     List<List> pendingReservations = QueueManager.loadQueueStatus(storeID);
        List<Booking> bookings = pendingReservations[0];
        List<LineUp> lineups = pendingReservations[1];

        bookings = Utils.sortByDate(bookings);
        bookings = bookings.stream().filter(b -> b.startTime < CurrentTime + 2h)
                           .collect(Collectors.toCollection(ArrayList::new));
        lineups = Utils.sortByID(lineups);

        StoreInfo storeInfo = StoreManager.getStoreInfo(storeID);
        int capacity = storeInfo.capacity;
        int onlineCapacity = capacity * 0.8;
        int physicalCapacity = capacity - onlineCapacity;

        List<Time> slotList = new ArrayList<Time>();
        List<Integer> onlineOccupationForSlot = new ArrayList<Integer>();
        List<Integer> physicalOccupationForSlot = new ArrayList<Integer>();

        /* Omitted for the sake of brevity: here we init the lists above to have 24 slots
           (each 5 minutes for 2 hours) */

(2)     for (Booking b: bookings) {
            numOfSlotsOccupied = b.estimatedResidcenceTime / 5; // 5 minutes
            for(i=0; i < numOfSlotsOccupied; i++) {
                int index = slotList.indexOf(startTime +  i*5);
                onlineOccupationForSlot.set(index, onlineOccupationForSlot.get(index) +
                                                   b.numOfPersons);
            }
        }
```

```
(3)     for(LineUp l : lineups) {
            numOfSlotsOccupied = l.estimatedResidcenceTime / 5; // 5 minutes
            int indexFound = -1;
            int waitingTime = -1;

            int correctCapacity = l.isOnlineUser ? onlineCapacity : physicalCapacity;

            outerloop:
            for(i=0; i < 24; i++) {
                for(j=i; j < numOfSlotsOccupied; j++) {
                    int correctOccupation = l.isOnlineUser ?
                    onlineOccupationForSlot.get(j) : physicalOccupationForSlot.get(j);
                    if(correctOccupation > correctCapacity) {
                        break;
                    }
                    else if(j == numOfSlotsOccupied - 1) {
                        indexFound = i;
                        break outerloop;
                    }
                }
            }

(4)         if(indexFound != -1) {
                for(j=0; j < numOfSlotsOccupied; j++) {
                    if(l.isOnlineUser)
                        onlineOccupationForSlot.set(j+indexFound,
onlineOccupationForSlot.get(j+indexFound) + 1);
                    else
                        physicalOccupationForSlot.set(j+indexFound,
physicalOccupationForSlot.get(j+indexFound) + 1);
                }

                waitingTime = CurrentTime - slotList.get(indexFound);
            }
        }
```

Below the 4 steps are explained in detail:

(1) First of all, the pending reservations are retrieved from the DB through the QueueManager. At this point they are distinguished between Booking and LineUp reservations, and then ordered and filtered to keep only the Bookings of the next two hours. Moreover, all the parameters and structures needed for the computation are initialized. In particular:

- slotList: a list of 24 elements, each one representing a period of time of 5 minutes, starting from the current time
- onlineOccupationForSlot: a list which keeps the occupation of each period of time, given by the OnlineUsers
- physicalOccupationForSlot: a list which keeps the occupation of each period of time, given by the PhysicalUsers

  N.B: the three lists are built in a way such that each index indicates the same element in all of them

(2) In this step the algorithm iterates over the bookings, computing and updating the occupation of each slot covered by the booking lifespan.

(3) Here the focus are the lineups, which are iterated to find at which time each of them could start, depending on its estimated residence time and the current occupation of the store (distinguishing between onlineOccuaption, if the LineUp is done by an OnlineUser, and physicalOccuaption otherwise). In particular, for each LineUp the algorithm checks which is the time when the store has available space for all the duration of the reservation.

(4) This step, done inside the iteration explained at point (3), is executed when an appropriate time for the lineup has been found. Here, the occupation in the slots which will be covered by the reservation is updated, and the waiting time is calculated through the difference between the current time and the estimated starting time.

3. User interface design

In the RASD we have shown a series of mockups that show the application screens. Here we will extend the UI by providing the navigation flow between the screens.

The graphs should be read as follow: nodes in red for screens; arcs or nodes in light red for actions; arcs connected to diamond symbol, used to represent a decision point in the process, for the response of the decision.

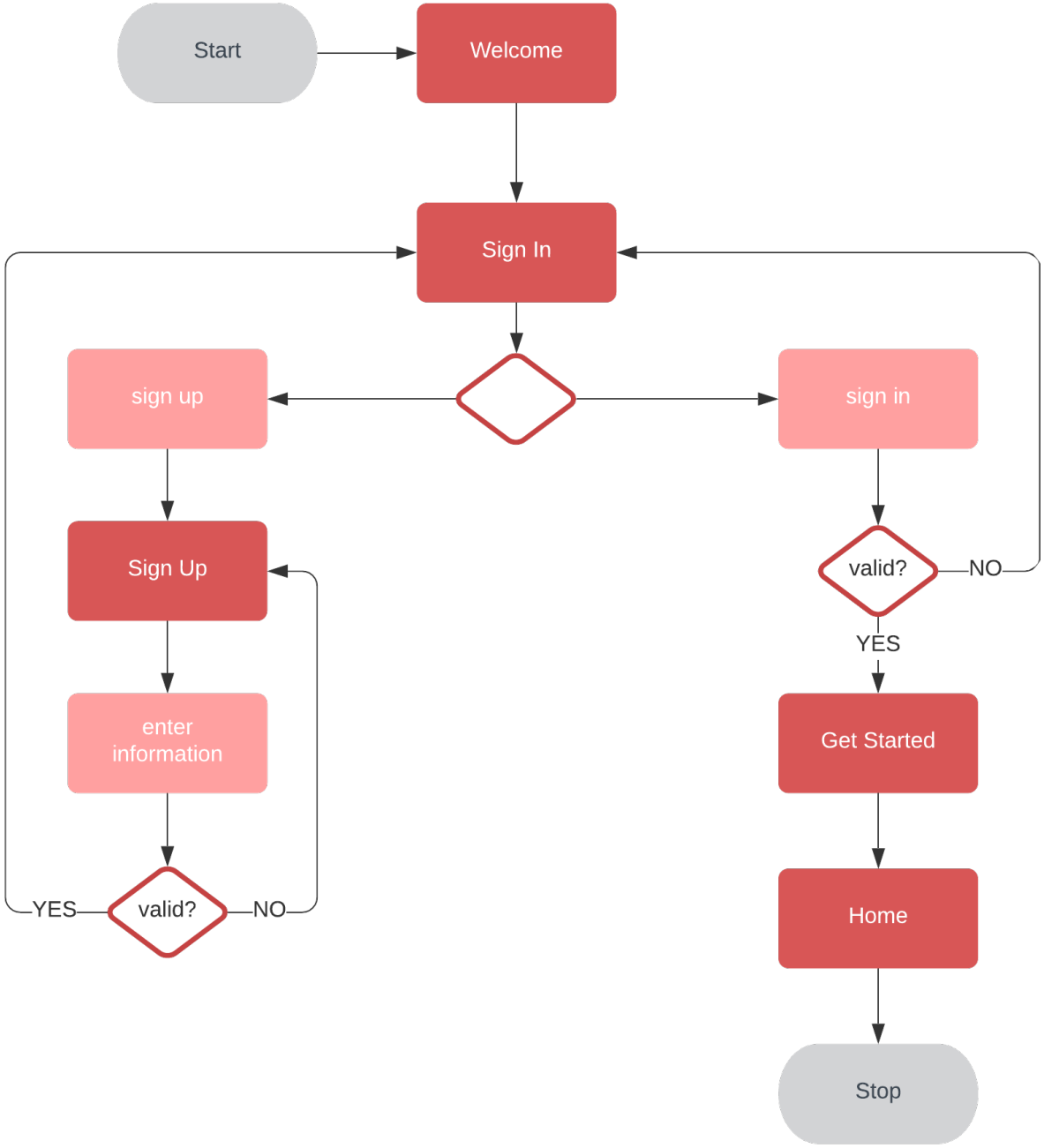

Figure 24 - UX Diagram: Login/SignUp

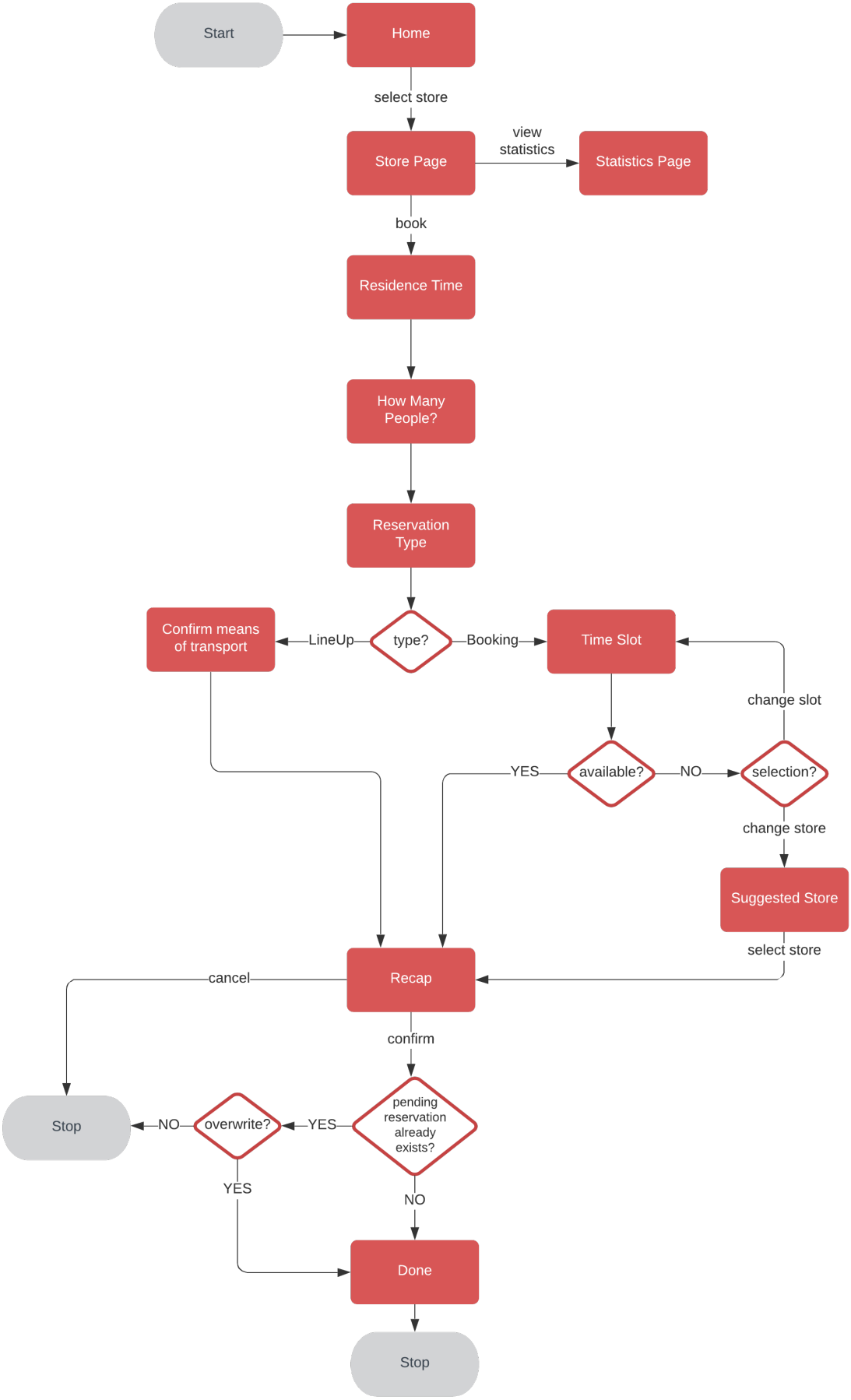

Figure 25 - UX Diagram: Reservation

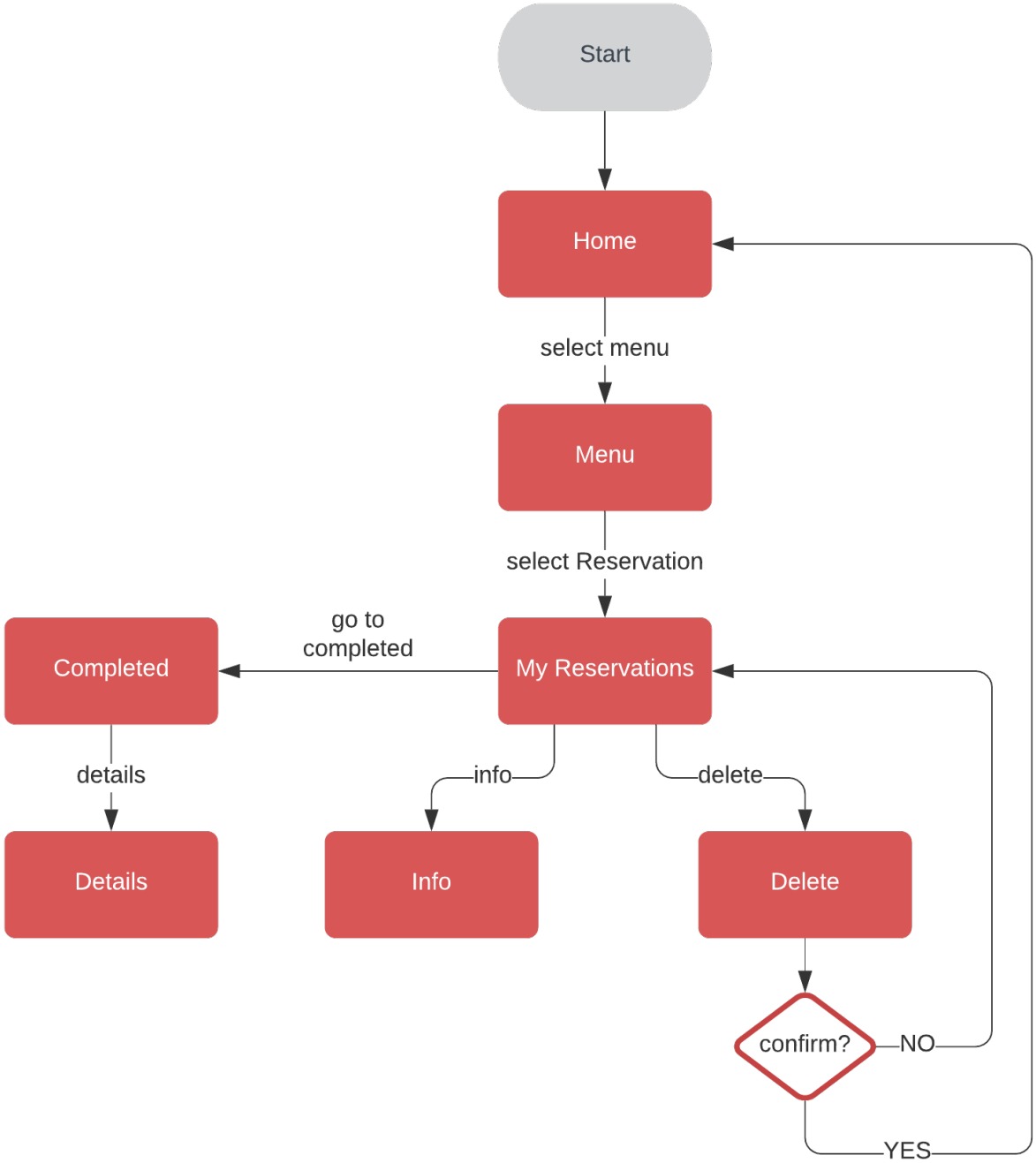

Figure 26 - UX Diagram: My Reservations

4. Requirements traceability

In the following table the requirements described in the RASD (reported in the following page) are matched with the components described in Section 2.2.

		Components																
		Web Server	**Router**	**User Manager**	**Reservation Manager**	**Store Manager**	**Maps Manager**	**Queue Manager**	**Notification Manager**	**Query Manager**	**Date Miner**	**Google Maps**	**Firebase Cloud Messaging**	**Manager Service**	**QRCode Dispenser**	**QRCode Scanner**	**DBMS**	**Mobile App**
Requirements	**1.1**	✓	✓		✓					✓							✓	✓
	1.2	✓	✓		✓	✓				✓							✓	✓
	1.3	✓	✓		✓	✓				✓							✓	✓
	2	✓	✓			✓		✓		✓							✓	✓
	3	✓	✓	✓		✓		✓		✓							✓	✓
	4	✓	✓	✓	✓					✓							✓	✓
	5.1	✓	✓	✓	✓					✓							✓	✓
	5.2	✓	✓	✓	✓					✓							✓	✓
	6	✓	✓			✓				✓							✓	✓
	7									✓	✓						✓	
	8				✓					✓						✓	✓	✓
	9									✓	✓					✓	✓	
	10																	✓
	11																	✓
	12						✓					✓						✓
	13				✓					✓							✓	✓
	14	✓	✓		✓					✓							✓	✓
	15									✓							✓	
	16	✓	✓		✓	✓		✓		✓							✓	✓
	17	✓	✓					✓										✓
	18	✓	✓		✓	✓				✓							✓	✓
	19	✓	✓		✓	✓				✓							✓	✓
	20								✓				✓					
	21												✓					✓
	22				✓					✓					✓		✓	
	23					✓				✓				✓			✓	

List of the requirements identified in the RASD:

R1.1	The onlineUser must be able to point out the duration of his visit by choosing a defined period of time
R1.2	The onlineUser must be able to point out the duration of his visit by selecting which categories of items he will buy
R1.3	The system must be able to calculate the duration of an onlineUser's visit by statistics on his previous visits
R2	The onlineUser must be able to see in real-time an estimated waiting time to enter for each store
R3	The onlineUser must be able to see in real-time the waiting time for his booking
R4	The onlineUser must be able to delete a booking within an hour from its start
R5.1	The onlineUser must be able to select the means of transportation
R5.2	The onlineUser must be able to change the means of transformation at any moment
R6	The onlineUser is not allowed to line up to a store where the current queue has a waiting time longer than two hours
R7	Information must be used to build statistics about waiting time, duration of visits and most busy times of the day
R8	The system must be able to count the number of people in each store in real-time
R9	The system must be able to compute the amount of time of each visit inside the store
R10	The internet connection of the OnlineUser must work properly
R11	The smartphone of the OnlineUser must have an integrated GPS sensor
R12	The system must be able to correctly compute the amount of time required to reach the store from the current position of the onlineUser
R13	The onlineUser is not allowed to book a new reservation if he already has an active one
R14	The onlineUser must be able to choose the number of people who will come to the store (1 or 2)
R15	The system must be able to retrieve all the information about each past reservation
R16	The onlineUser is not allowed to book a slot of time which is already full
R17	The onlineUser must be able to confirm his reservation when he receives the related notification
R18	The onlineUser is not allowed to book a slot for a time which is closer than two hours from the moment he makes the reservation
R19	The onlineUser is not allowed to book a slot for a time which is farther than seven days from the moment he makes the reservation
R20	The system must be able to send notifications to onlineUsers
R21	The smartphone of the OnlineUser must be able to receive notifications
R22	The system must be able to hand out physical tickets
R23	The system must be able to register a new store or modify an existing one

The non-functional requirements, instead, are guaranteed by the design choices:

Requirement	Design Choice
Reliability and Availability	Microsoft Azure Infrastructure (e.g.: replicated components, multiple copies of the persistent storage, DDoS Protection)
Security	DMZ, DBMS privileges, HTTPS protocol, encryption of sensitive data, token-based authentication
Maintainability	Microsoft Azure Infrastructure, REST architectural style (i.e.: microservices)
Portability	REST architectural style, choice of software independent or multi-platform services (i.e.: Google Maps SDK and Firebase Cloud Messaging)
Scalability	Microsoft Azure Infrastructure (i.e.: auto-scaling groups, load balancing)

5. Implementation, integration and test plan

Here it is provided a table which shows the relationship between each component and its difficulty of implementation. Moreover, it provides the importance for the customer for each component, in order to highlight which components represent the core of the S2B.

Components	Difficulty of implementation	Importance for the customer
User Manager	Low	Low
Reservation Manager	High	High
Store Manager	High	High
Maps Manager	Medium	Low
Queue Manager	Low	High
Notification Manager	Medium	Low
Query Manager	Medium	Low
Data miner	High	Medium

For the implementation and integration procedure, we adopt a bottom-up approach, which is a strategy in which the lower-level modules are tested first. These tested modules are then further used to facilitate the testing of higher-level modules. The process continues until all modules at top level are tested. Once the lower-level modules are tested and integrated, then the next level of modules is formed. Even if an early prototype is not possible, we chose this approach because it offers different advantages during the integration and testing phases.

5.1 Implementation Plan

The implementation has to be done from the lower components up to the top because in this approach the implementation is incremental. Nevertheless, the DBMS is developed at the beginning and independently since all the modules interact with it. For this reason, also the Query Manager is one of the first components to be realized, with an incremental implementation of its methods, since each of them corresponds to a query needed by a just-developed component.

The components on the same level and with the same priority can be developed simultaneously. Below, the implementation order of the components is given:

- **User Manager**, **Notification Manager**, **Maps Manager** and **Data Miner** are the four components which constitute the first level since they require only external services and/or queries that are previously implemented in the Query Manager.
- **Reservation Manager**, **Queue Manager** and **Store Manager** realize the second level. These components rely on the components of the first level (and also on the Query Manager) and have a high degree of dependency between each other because they are all fundamental for the main functionalities of the S2B (all of them take part in the realization of a reservation).
- **Router** has to be implemented at this point since it provides the redirection of the requests to the other components.
- **Web Server** is the last component to be developed since it is the only entry point for the client's requests.

5.2 Integration Plan

In Figure 27 the flow of dependencies between the components of the S2B is shown. In the flow, the component pointed by the arrow is the one required by the component from which the arrow starts.

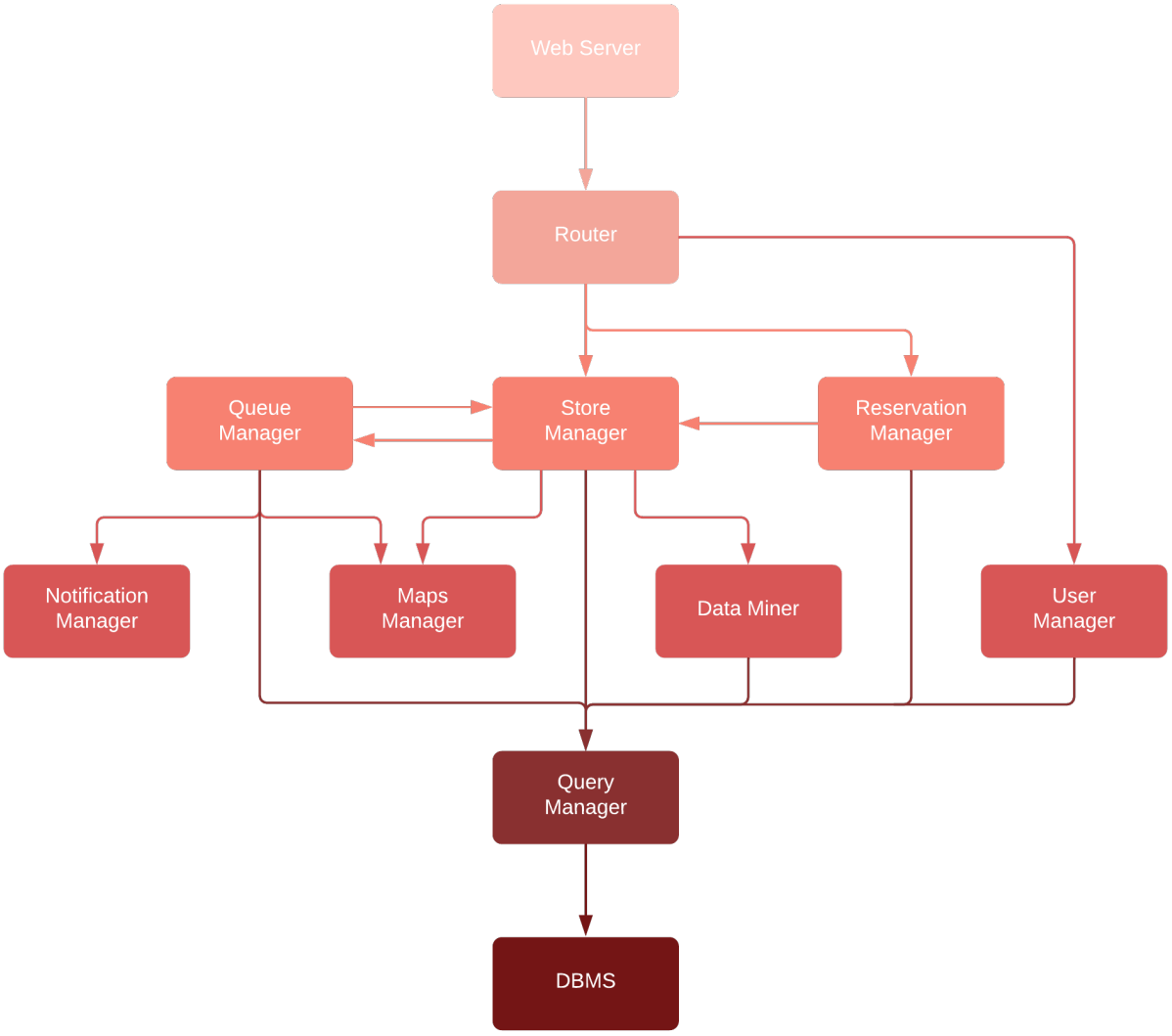

Figure 27 - Integration Plan: dependencies

Since the bottom-up approach has been adopted, the diagram must be read from the bottom to the top. That means, during the integration process, the components above the level in development are simulated through drivers, which are dummy programs that acts as a substitute for the missing components.

5.3 Test plan

During software development, errors are made. To locate and fix those errors, different testing phases are chained together.

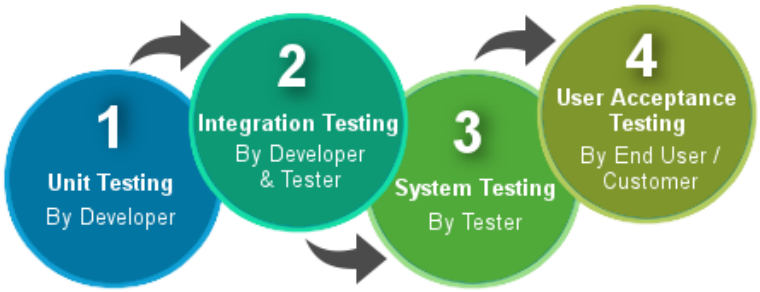

Figure 28 - Levels of Testing

(1) Firstly, concurrently with the implementation phase, white-box **Unit Testing** is done in order to test the individual components using tools such as JUnit. Through the white-box technique, the tester has a high knowledge of the code, allowing therefore to obtain maximum coverage. Also, in this phase test cases can be easily automated.

(2) Next, while the components are incrementally integrated into a system, black-box **Integration Testing** is done. The chosen approach is bottom-up, which consists of starting by testing the low-level components which are then integrated and coupled with components at the next higher level. The subsystem thus obtained is tested next. Then gradually we move towards the highest-level components. In this way, it should be easier to localize any eventual fault and no time is wasted waiting for all modules to be developed. (The environment in which the component being tested has to be integrated is simulated through drivers.)

(3) Once the System is completely integrated, it must be tested in its entirety to verify that functional and non-functional requirements are satisfied. **System Testing** can be divided in:

- **Functional Testing** — Validates the software system against the functional requirements described in the RASD.
- **Performance Testing** — It determines how a system performs in terms of responsiveness and stability under a normal workload. Thanks to this it is possible to identify the presence of inefficient algorithms, query optimization possibilities or network issues.
  During this validation process, speed, scalability, stability, and reliability are tested.
- **Load Testing** — It is performed to determine the system's behaviour under peak workload
- **Recovery Testing** — It is done to demonstrate that the software is reliable, trustworthy and can successfully recoup from possible crashes.

(4) Eventually, **User Acceptance Testing** (UAT) is performed by the end users and the client to verify/accept the software system before moving the software application to the production environment (this phase is commonly known as beta testing).

6. Effort spent

6.1 Galzerano Arianna

HOURS	TASK
2	Initial discussion
3	Component View
6.30	Runtime View
2	Details about the component view
2.30	Dimensional Fact Schemas and description
2	Descriptions in Runtime View
3	Traceability matrix
1	Description and implementation of triggers
4	Algorithm
0.30	Design patterns
4	Implementation Plan
2	Test Plan
5	Final Review

6.2 Lampis Andrea

HOURS	TASK
2	Initial discussion
3	Component View
2	Overview
1	Details about the Component View
10	Runtime View
0.40	ER diagrams and description
4	Deployment View
3.30	Traceability matrix
0.40	Architectural styles
4	Algorithm
3	UX Diagrams
4	Implementation Plan
2	Integration Plan
5	Final Review

6.3 Leone Monica

HOURS	TASK
2	Initial discussion on first part
5	Component View and descriptions
1.30	Introduction and document layout
2	ER diagrams and description
8.30	Runtime View and descriptions
3	Component interfaces and internal methods
3	Traceability matrix
0.15	Queries
4	Algorithm
0.15	Design patterns
4	Implementation Plan
5	Final Review