POLITECNICO
MILANO 1863

CLup

Design Document

Simone Abelli
Stefano Azzone

Deliverable:	DD
Title:	Design Document
Authors:	Simone Abelli, Stefano Azzone
Version:	1.0
Date:	7 January 2021
Download page:	https://github.com/StefanoAzzone/AbelliAzzone
Copyright:	Copyright © 2020, Simone Abelli, Stefano Azzone – All rights reserved

Contents

1 Introduction

1.1 Purpose

The purpose of this document is to provide more technical and detailed information about the software discussed in the RASD document. The Design Document is a guide for the programmer that will develop the application in all its functions. The document will explain and motivate all the architectural choices by providing a description of the components and their interaction. We will also enforce the quality of the product through a set of design characteristics. Finally we describe the implementation, integration and test planning.
The topics touched by this document are:

- high level architecture
- main components, their interfaces and deployment
- runtime behavior
- design patterns
- more details on user interface
- mapping of the requirements on the components of the architecture
- implementation, integration and test planning

1.2 Scope

CLup is a system that allows customers to line up in a virtual first in first out queue, in order to avoid overcrowding outside of stores. Customers can queue up remotely or on premise (by using a device installed outside the store). When a customer queues up remotely he/she can choose to line up immediately or to book a future visit. The system alerts customers when it is time for them to depart to reach the store. The system builds statistics on customer entry and exit in order to provide a better estimation of waiting times. The system allows the store owners to control the occupation of each of their stores. This is just a summary of all the features of the system, for a more detailed description of the software functionalities read the RASD.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

Reservation	Virtual or physical artifact used to identify the position of a customer in a queue
Queue up	Customers are lined up in a FIFO queue
Enqueued	A customer is enqueued when he has provided the system with a means of identification and requested a reservation
Authorized	A customer is authorized when he has been enqueued and is allowed temporary access to the store.
Occupation	Number of customers currently present in the store
Printer	Device that can read a social security card and print tickets that contains a progressive number and an estimate of the waiting time.
User	Either a customer or a store owner.

1.3.2 Acronyms

RASD	Requirement Analysis and Specification Document
GPS	Global Positioning System
S2B	Software to be
UI	User Interface
FIFO	First in first out

1.3.3 Abbreviations

Gn	Goal number n
Rn	Requirement number n

1.4 Revision history

Ver.1.0 : 7 Jan 2021

1.5 Reference Documents

1. IEEE Std 830-1998 Recommended Practice for Software Requirements Specifications
2. Specification Document: R&DD Assignment A.Y. 2020/2021
3. uml-diagrams.org

1.6 Document Structure

- Chapter 1: gives an introduction about the Design Document, enumerating all the topics that will be covered. Moreover this section contains specifications such as the definitions, acronyms, abbreviation, revision history of the document and the references.
- Chapter 2: contains the architectural design choices, with an in-depth look at the high level components and their interactions. It include several views: the component view, the deployment view and the runtime view. Here are described the interfaces (both hardware and software) used for the development of the application, their functions and the processes in which they are utilized. Finally, there is the explanation of the architectural patterns chosen with the other design decisions.

- Chapter 3: this section provide an overview of how the user interfaces of the system will look like.
- Chapter 4: This chapter maps the requirements defined in the RASD to the design elements defined in this document.
- Chapter 5: here we define the plan for the implementation of the subcomponents of the system and the order in which the integration operations and their testing will be performed.
- Chapter 6: shows the effort which each member of the group spent working on the project.
- Chapter 7: includes the reference documents.

2 Architectural Design

2.1 Overview

The architecture of the S2B is a distributed client-server architectural design, structured according to three logic layers:

- **Presentation level P**: manages the user interaction with the system. This layer contains the interfaces able to provide the functions of the application to the users.
  To the presentation layer belong the web app, the phone application and the software on the ticket printer and on the QR reader.
- **Business logic or Application layer (A)**: handles the business logic of the application and its functionalities. This layer represent the core of the application logic.
- **Data access layer (D)**: manages information and data, by accessing the database.

Every logic layer can be mapped in an hardware layer.
The presentation layer is composed by the smartphone or the computer of the user, the ticket printer outside the stores, the QR reader and the turnstiles.
The application layer is composed by the application server. The data layer is composed by the database server.

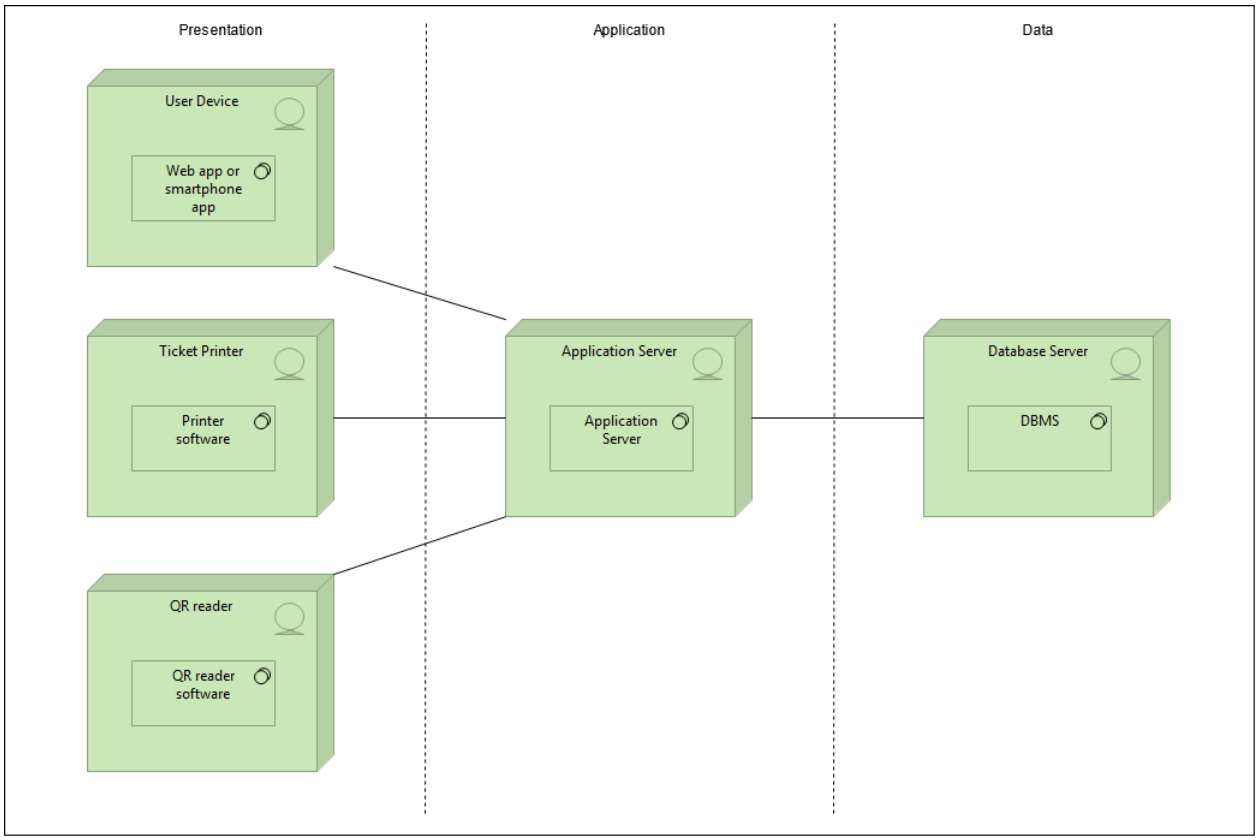


Figure 1: High level system architecture

This image shows the high level representation of the three-layer architecture, where we have all the devices used to interact with the user and their respective software, that have to connect to the application

server, which is able to communicate with the database server.
Although very simple, this high level view shows how a three tier architecture can provide more flexibility to the system, splitting the server side in two logical layers. It is also very useful from a security viewpoint: in fact the data are kept separated from the user by the application layer, so that all sensitive information are protected from undesired access.

Let us now have a more in-depth look at the system architecture.

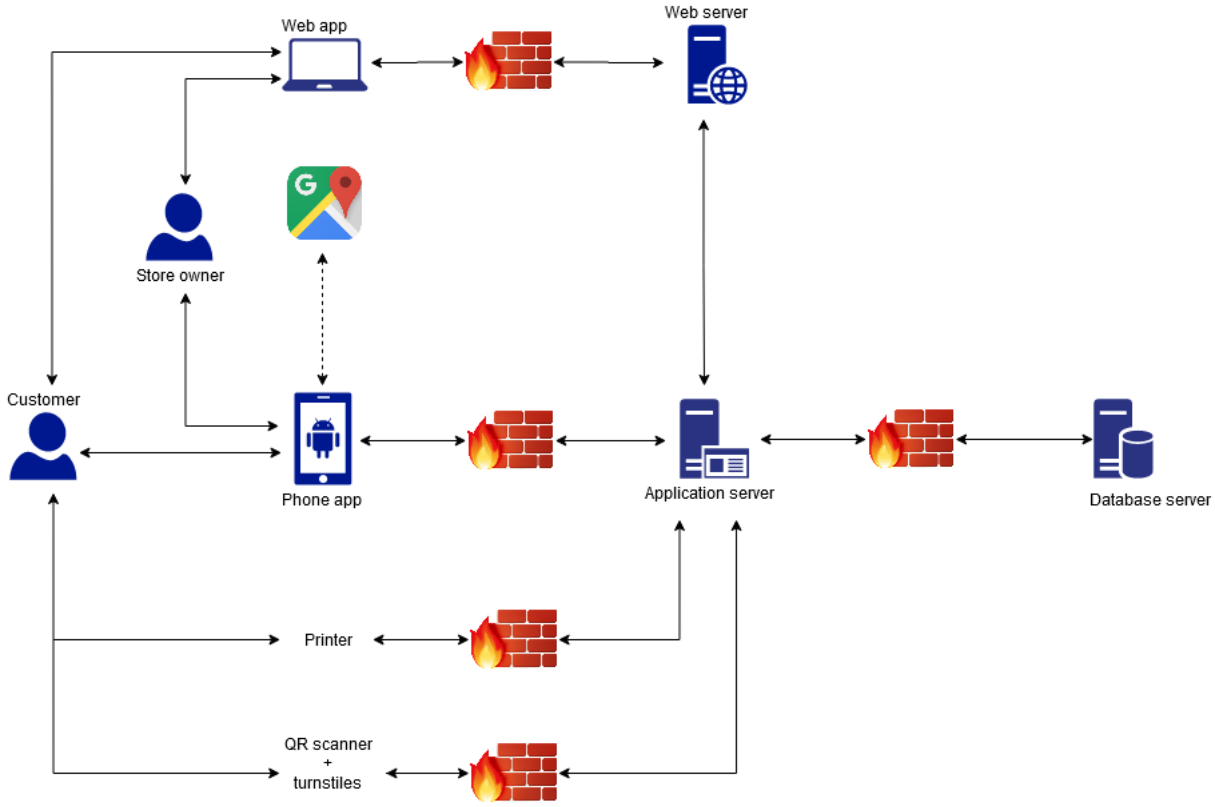


Figure 2: System Architecture

Here we can better see how each tier of the previously defined architecture is mapped in the system. The customer is capable to interact with the system in several ways: through the phone app on his smartphone, by the web app on his computer, using the printer outside the store (in order to create an on premise reservation) or via QR scanner, which unlocks the turnstiles and validate enter and exit from the store. In order to provide an estimation of the travel time from the location of the customer to the store, the system uses the GPS technology through the Google Maps API.
The store owner, from his/her side, can only access the system from the web app or the phone application; this is because in the role of owner he/she does not need to access the market, and use the system only to control occupation and statistics of the stores.
The web app does not communicate directly with the application server, but must traverse the web server, which provide the web pages to the browser.
The application and web servers ere protected from the outside by a first firewall, which creates a demilitarized zone for them. Then, the application server communicates with the database server, which contains the database management system, through a second and more restrictive firewall, that isolate the database from possible attacks. This double firewall ensures the security properties for the system.
In order to create reservations, the clients must communicate synchronously with the server, as they have no information on the queue. The application server, then communicates synchronously with the database server to retrieve information or asynchronously to store information when needed.
In the image above the server element must not be considered as a single machine: if so the architecture would be poorly scalable, and heavy traffic would make the system crash. To guarantee scalability we use a node replication approach, which involves the use of a node balancer which redirect the traffic

between multiple servers.

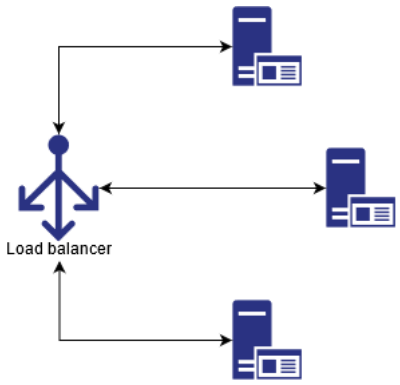


Figure 3: Load Balancer

This image shows an example of how this method works for the application tier: in this case the load balancer splits the work between three application servers, but of course there might be more. Moreover this approach is used also for the web server and the database server, according to the needs of the system. Until now we have used an informal view of the architecture; in the next sections we will keep deepening the architecture and its characteristics.

2.2 Component View

Here we display the main component architecture of our S2B. Since the ApplicationServer contains all the business logic, we will describe in detail the structure of its subcomponents.

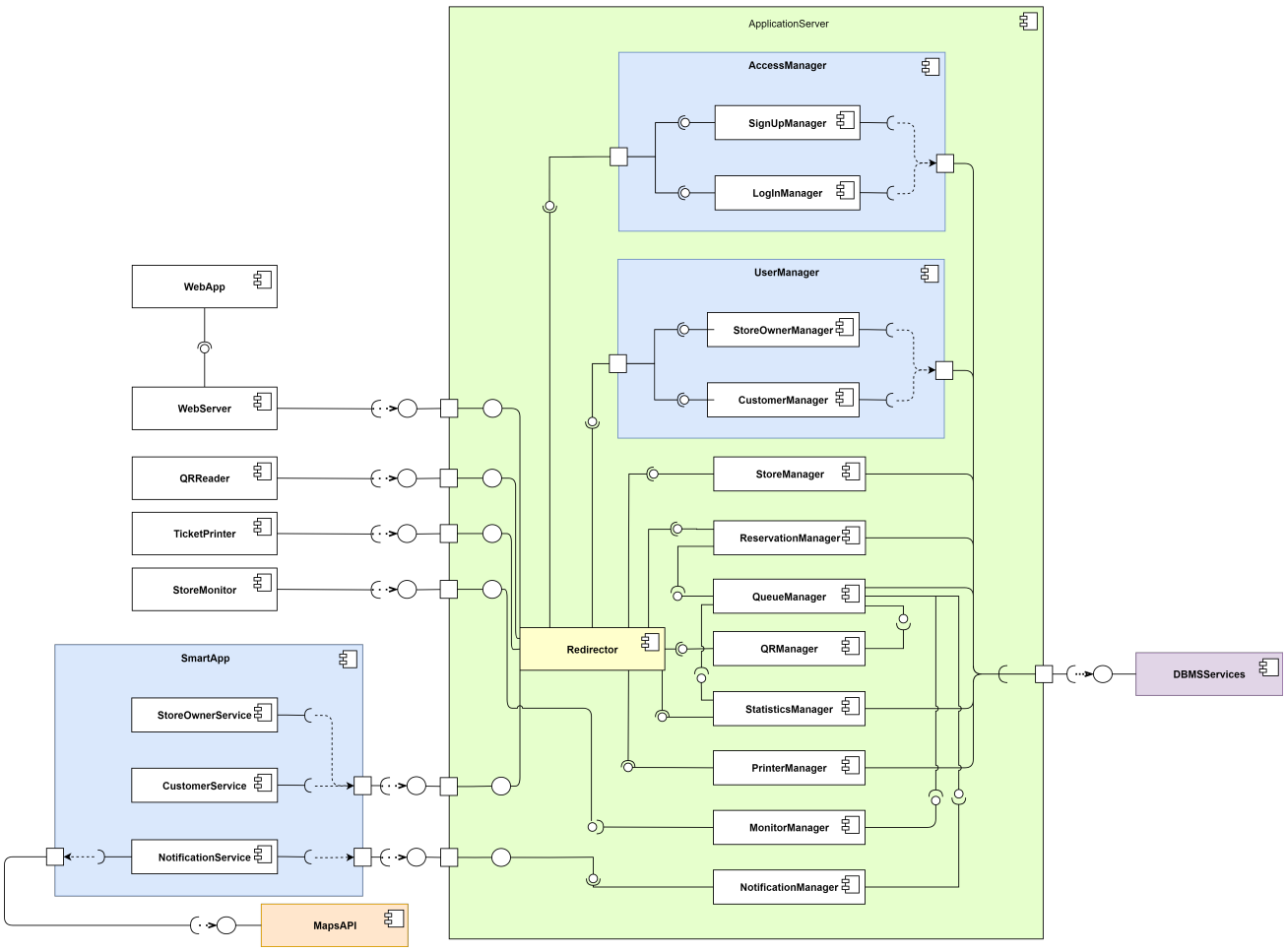


Figure 4: Component view

2.2.1 WebApp

This component allows customer and store owners to access their respective services on a computer. It requires a WebServer.

2.2.2 WebServer

This component communicates directly with the ApplicationServer and serves web pages to implement the WebApp component.

2.2.3 SmartApp

This component allows customers and store owners to access their respective services on a smartphone, by interfacing with the ApplicationServer. It is composed of the following subcomponents:

2.2.3.1 StoreOwnerService Allows the store owner to access the functionalities that the system reserved to him/her.

2.2.3.2 CustomerService Allows the customer to access the functionalities that the system reserved to him/her.

2.2.3.3 NotificationService Periodically receives the waiting time from the NotificationManager and calls MapsAPI to get the time to reach the store from the current position and if it is greater or equal to the waiting time (plus a constant, to give a little notice to the customer) it notifies the customer.

2.2.4 QRReader

This component reads user provided QRCodes and sends them to the ApplicationServer, thus enabling authentication for turnstiles.

2.2.5 TicketPrinter

This component accepts a user document and after having validated it through the ApplicationServer, it prints a reservation ticket with a QR.

2.2.6 StoreMonitor

This component updates periodically calling the ApplicationServer, which provides it with the number of the last authorized reservation, thus notifying customers in front of the store when they can enter.

2.2.7 Redirector

This component provides an external interface for the previously described components, and allows them to communicate with the components that are located within the ApplicationServer, that we describe below.

2.2.8 AccessManager

This component allows users to connect to the services, via sign up and login.

2.2.8.1 SignupManager This component allows customers and store owners that provide a valid identification document to register to the service, thus gaining access to its functionalities, provided that they log in. It is called through WebApp or SmartApp, and needs to access the DBMSServices to search for existing users with same identification document (to avoid duplication), and to create a new user.

2.2.8.2 LoginManager This component allows customers and store owners to log in the service, thus gaining access to its functionalities. It is called by WebApp and SmartApp components, and needs to access the DBMSServices to verify user credentials against the stored ones.

2.2.9 UserManager

This component allows users to change their profile.

2.2.9.1 StoreOwnerManager This component is accessed through WebApp or SmartApp and allows store owners to edit their credentials (obtained during sign up process), thus it needs access to DBMSServices.

2.2.9.2 CustomerManager This component is accessed through WebApp or SmartApp and allows customers to edit their credentials (obtained during sign up process), thus it needs access to DBMSServices.

2.2.10 ReservationManager

This component is accessed through WebApp or SmartApp by customers and allows them to send a reservation for a specific store, to delete an existing reservation, or to view status of non expired reservations (QR code, position in queue, status) by accessing QueueManager. It needs to access DBMSServices to retireve the list of the departments of the store that the reservation targets and, in case of an immediate reservation, to verify that the store is open at the current time. It needs access to DBMSServices also to fetch the list of user reservation along with their information, and to delete them if required. Whenever a customer creates a reservation it is saved into the database. This component is also accessed by the TicketPrinter whenever a customer creates an on premise reservation.

2.2.11 StoreManager

This component is accessed through WebApp or SmartApp, and allows store owners to view owned stores, add new stores, delete existing stores or update information of existing stores, such as the list of departments and their respective maximum occupation. It needs to access the DBMSServices to execute the previous functions.

2.2.12 QueueManager

This component contains the information about the people that are currently waiting to enter the store and those that are already inside it.
It is accessed by the ReservationManager to add or remove a reservation from a queue of a specific store. It is also accessed by the QRManager whenever an authorized customer wants to view the QR and when he/she enters or exits the store. Moreover it accesses MonitorManager every time the number of the last authorized reservation of a store changes. Finally it is accessed by NotificationManager in order to get the list of the customers currently waiting.
It needs access to DBMSServices to update the status of each reservation and entry and exit times. It checks periodically the number of customers currently present and the maximum occupation for every store and computes an estimated time that customers have to wait before gaining authorization to enter a specific store if such store has already reached maximum occupation.
This component accesses StatisticsManager to improve the computation of the time a customer needs to wait before being authorized to enter the store.

2.2.13 QRManager

This component accesses QueueManager whenever a customer needs to view a QR for a reservation: if a QR has already been generated for that reservation it is returned to QRManager, otherwise it is generated by QRManager and sent to QueueManager.
When QRReader reads a QR code from a customer it calls QRManager, that accesses QueueManager to check that the reservation is existing and authorized, and if so it send the entry/exit time to the QueueManager and returns the QRReader the permission to unlock turnstile.

2.2.14 StatisticsManager

This component is accessed by QueueManager to improve the estimate of the time that a customer needs to wait before being authorized to enter a specific store. This component is accessed by a store

owner through WebApp or SmartApp to visualize statistics. This component accesses the DBMSServices periodically to obtain the data with which to generate the statistics, and to save such statistics.

2.2.15 PrinterManager

This component allows the store owner with WebApp or SmartApp to register a TicketPrinter. This component also allows customer identification through TicketPrinter by validating a provided identification document. This component needs to access the DBMSServices to register or unregister instances of TicketPrinter.

2.2.16 MonitorManager

This component allows to update periodically a StoreMonitor. It is updated by the QueueManager every time the number of the last authorized reservation of a store changes, and then updates all the StoreMonitor registered for that store.

2.2.17 NotificationManager

This component accesses the NotificationService to provide the wait time. This component periodically accesses the QueueManager to get this information.

2.2.18 MapsAPI

This component accesses an external mapping service (i.e. GoogleMaps) to compute the estimated time to reach the store from the current position of the device with the selected means of transport.

2.2.19 DBMSServices

This component is the DataBase Management System accessed by the ApplicationServer to perform create/read/update/delete operations on the DB.

2.2.20 Additional specification

For what concerns the smartphone application, it is a thick client: in fact beside the presentation logic, it contains a small part of the application logic. Indeed, the phone app can check the validity of the information inserted by the user when creating a reservation and, more importantly, it checks when it is time to notify the customer about its turn to reach the store, based on the position of the device and the position of the store (received by the server). It is important that the phone app does this computations, because the position of the customer, necessary to know how long will it take to go to the market, can only be detected by the end device.
The web app, of course, is instead a thin client, because, being it a browser application, will only work on the presentation logic, to show the interface to the user.
In order to guarantee the reliability of the system, both the web servers and the application servers must be replicated, so that in case of one of them goes down the application can still work.
The load balancer, that in one of the previous diagram has been represented as a single component, may become overloaded by the requests. To prevent this it must be replicated too.
To optimize the routing process we can divide the redirector in several subcomponents, one for each of the possible devices that will use it.

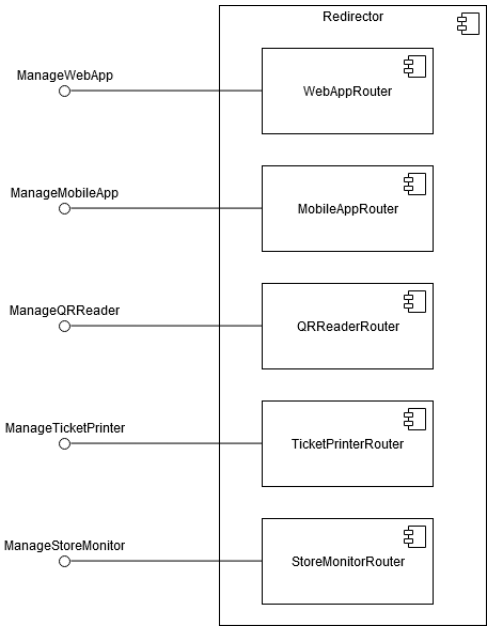
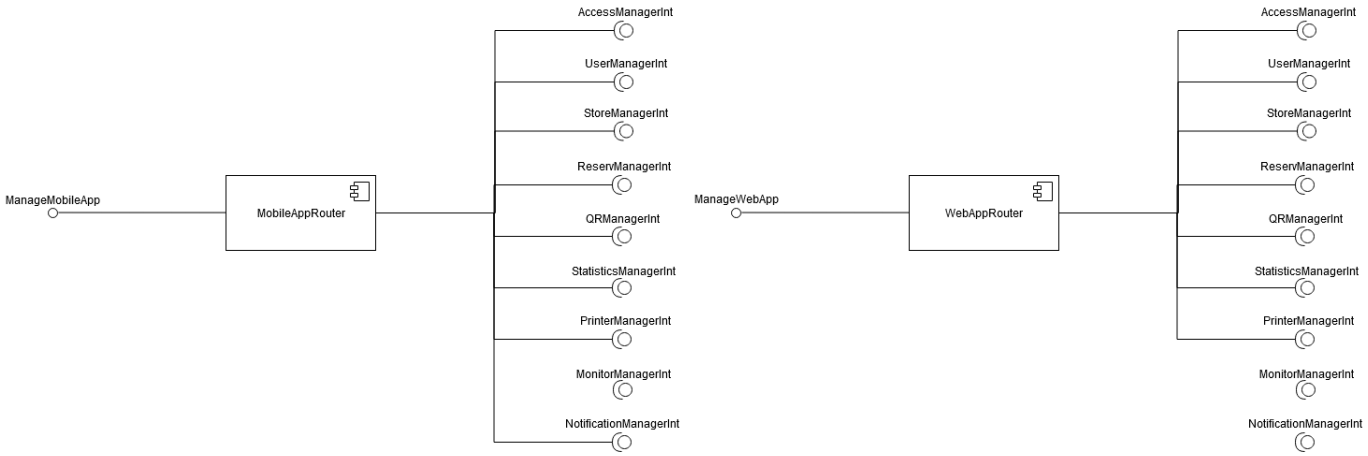


Figure 5: Redirector

This image shows how each interface connects with the redirector's subcomponents. Of course, each of them will provide a different routing on the possible interfaces of the system, as shown in the following diagrams.



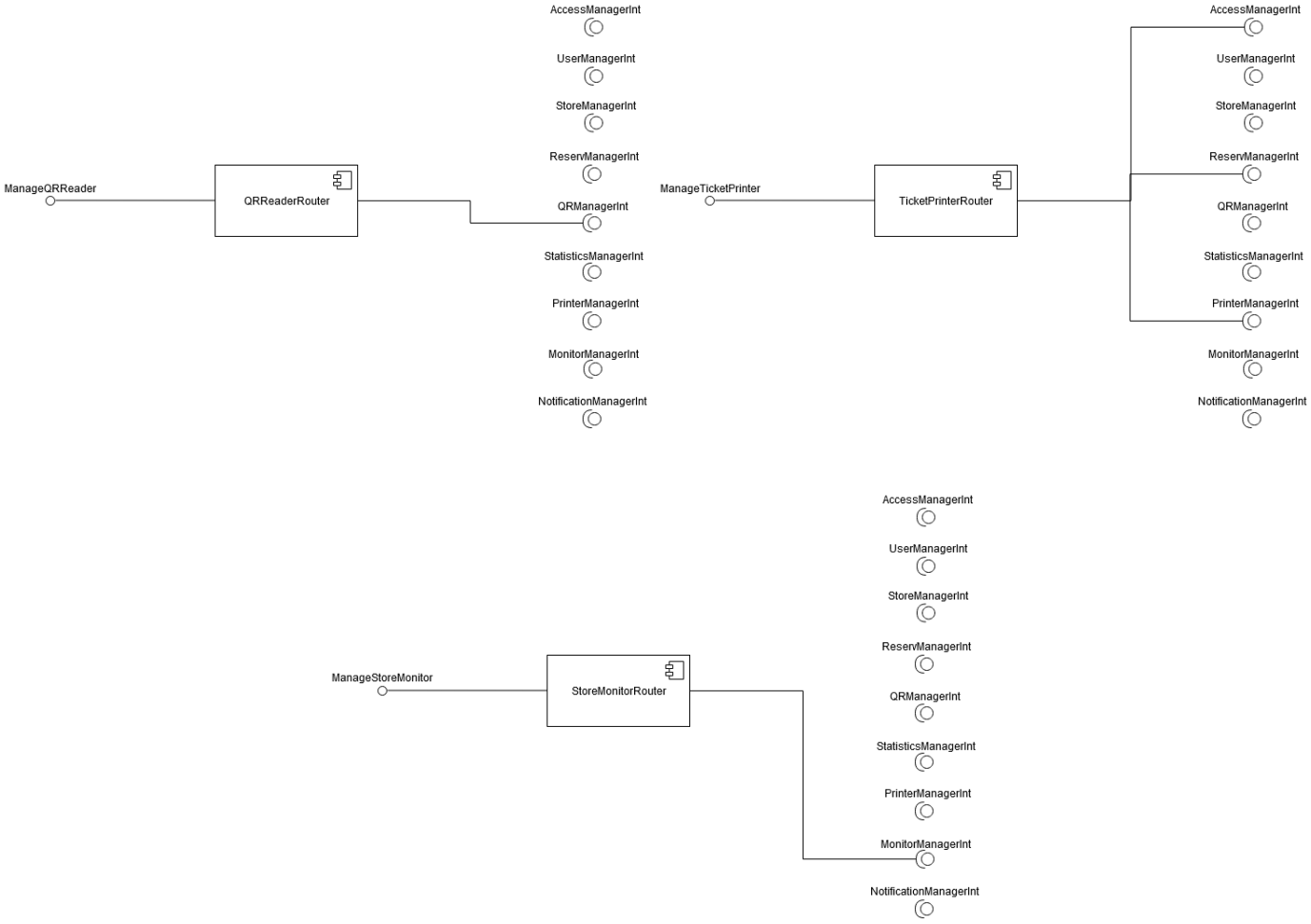


Figure 6: Routing

2.3 Deployment View

The following image shows the deployment diagram of the system by presenting the distribution (deployment) of software artifacts to the nodes, i.e. the deployment targets.

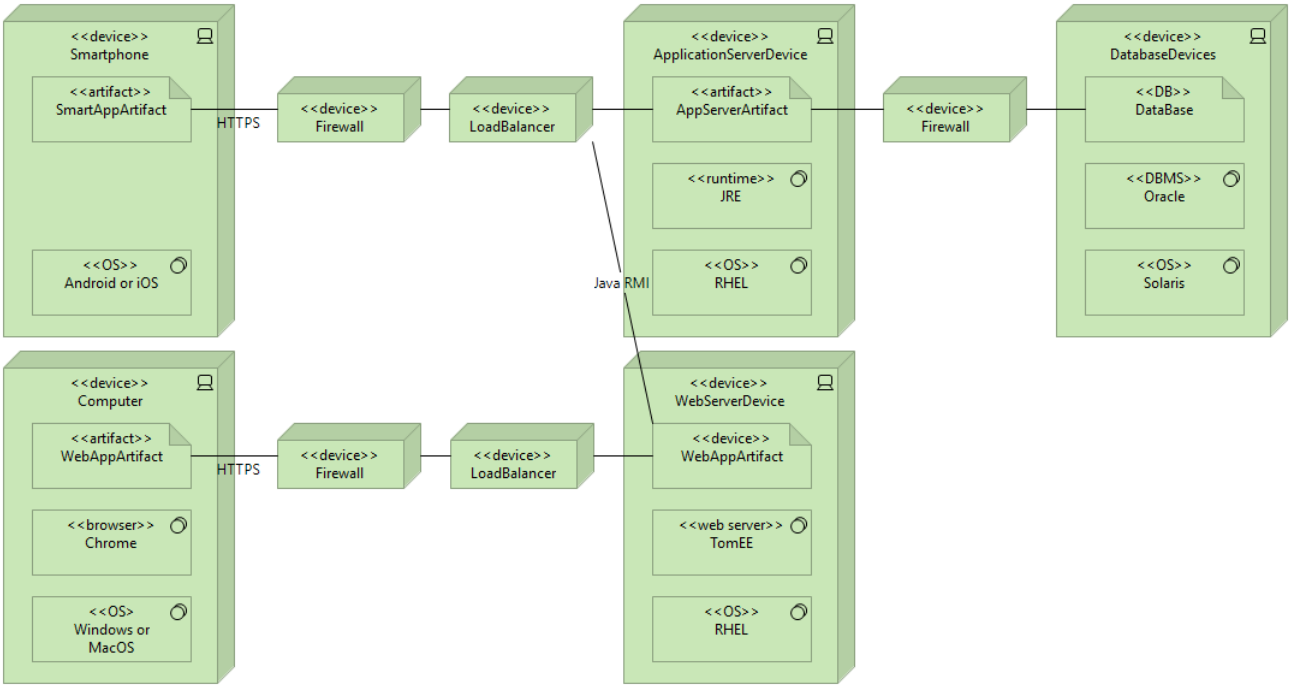


Figure 7: Deployment view

Here we describe the components of the deployment view:

2.3.1 Smartphone

The customers and the store owners can access the functionalities of the system through a smartphone, that communicates through a firewall and a load balancer with the ApplicationServerDevice.
The SmartAppArtifact allows communication between the Smartphone and the ApplicationServerDevice, and its operation is supported by the operative system of the smartphone, either Android or iOS.

2.3.2 Computer

The customers and the store owners can access the functionalities of the system through a computer that communicates through a firewall and a load balancer with the WebServerDevice which in turn is connected through a firewall and a load balancer with an ApplicationServerDevice.
The WebAppArtifact will be downloaded from the WebServerDevice and it will be shown by the browser installed on the device which we assume for simplicity to be Chrome, whose operation is supported by the operating system of the Computer, either Windows or MacOs.

2.3.3 Firewall

The firewalls allow to separate the different parts of the system: the DatabaseDevice is isolated from the ApplicationServerDevice, which in turn is isolated from the Smartphone, and the WebServerdevice, which in turn is isolated from the Computer.

2.3.4 LoadBalancer

The LoadBalancer allows to balance the load of the system on the replicated ApplicationServerDevices and WebServerDevices.

2.3.5 ApplicationServerDevice

This device implements the main business logic of our system. It is accessed by Smartphone and Computer devices to make use of the functionalities offered by the system. It is supported by a DatabaseDevice that allows the ApplicationDevice to create, read, update and delete all the information necessary for the correct functioning of the system.
The AppServerArtifact is run by the JRE which in turn is supported by the RHEL operating system.

2.3.6 WebServerDevice

This device allows customers and store owners to access their respective services by using Java RMI to communicate with the ApplicationServerDevice.
The web server TomEE will server the browser the artifact it needs to show to the users. This web server's operation will be supported by Red Hat Linux Enterprise operating system.

2.3.7 DatabaseDevice

This device allows the ApplicationServerDevice to create, read, update, delete information that supports the system operation.
The DataBase is managed by the Oracle DBMSServices which in turn is supported by Oracle Solaris.

2.4 Runtime View

2.4.1 Login via smartphone app

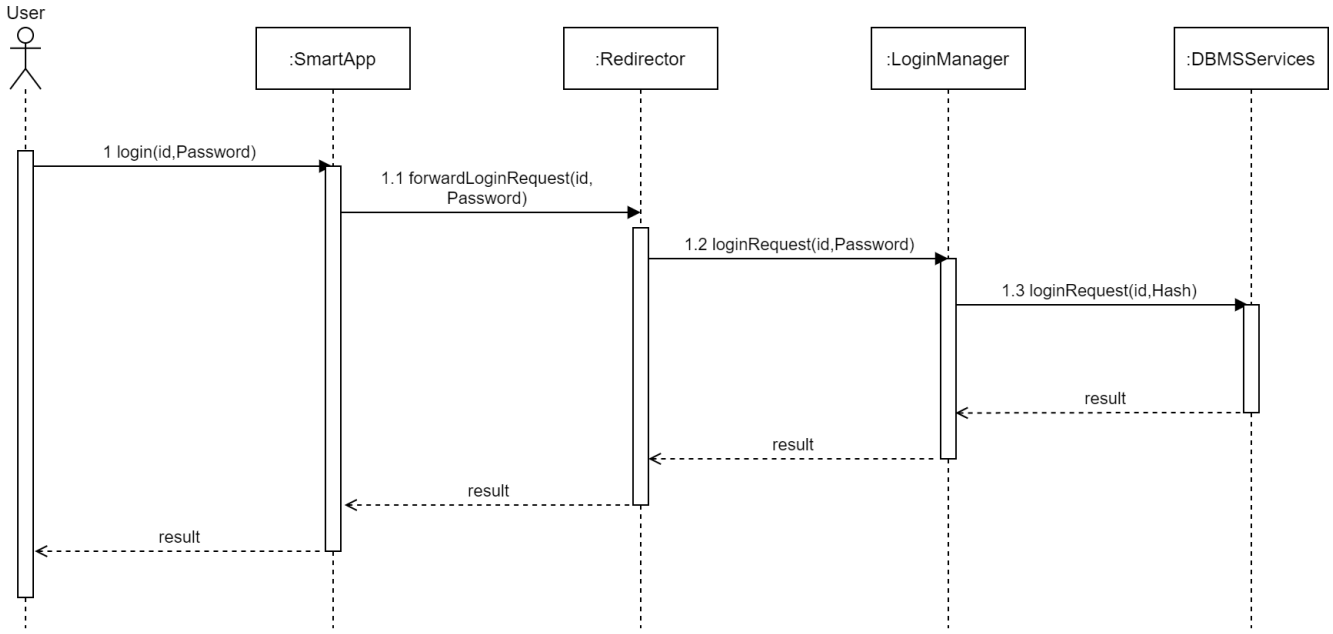


Figure 8: Login

By opening the application, user calls the SmartApp component. SmartApp calls Redirector, that calls LoginManager, that retrieves from the DBMSServices the password hash of the specified user (if he/she exists), and compares it with the provided password. The result of this operation is then sent back to the user.

2.4.2 Immediate reservation via smartphone app

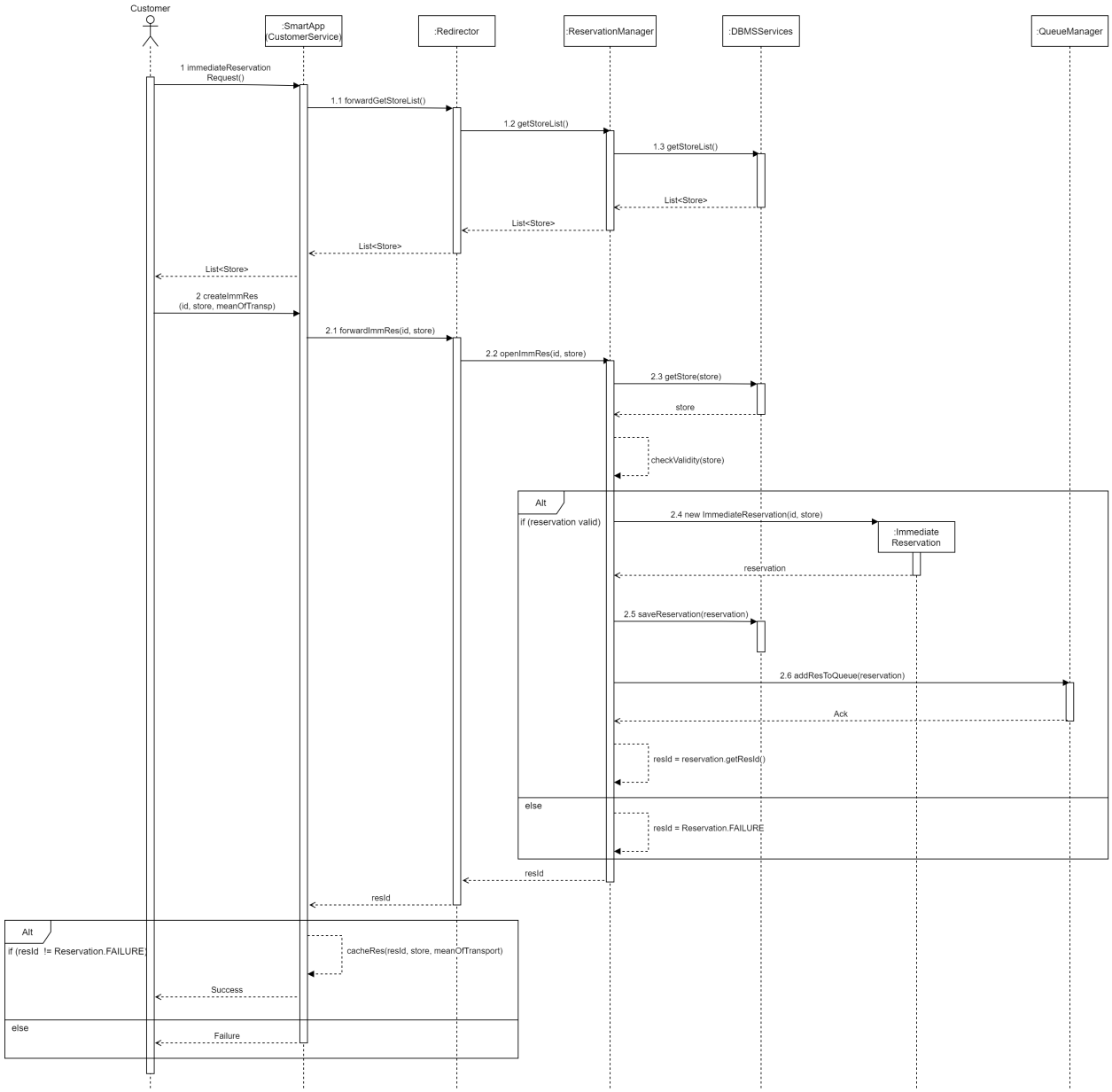


Figure 9: Immediate reservation

The customer selects the option to create an immediate reservation. The SmartApp requests a list of stores from the ApplicationServer and this request is then forwarded by the Redirector to the ReservationManager that retrieves the list of stores form the DMBS and sends it back to the customer. The customer then selects a store form the list provided and a means of transport. This information is then sent to the ApplicationServer where the Redirector routes it to the ReservationManager that checks that the selected store is open at the time, loading it from the DBMSServices, and if it is, the reservation is confirmed, added to the queue of that store by calling QueueManager, and saved in the database. It is rejected otherwise.

2.4.3 Future reservation via web app

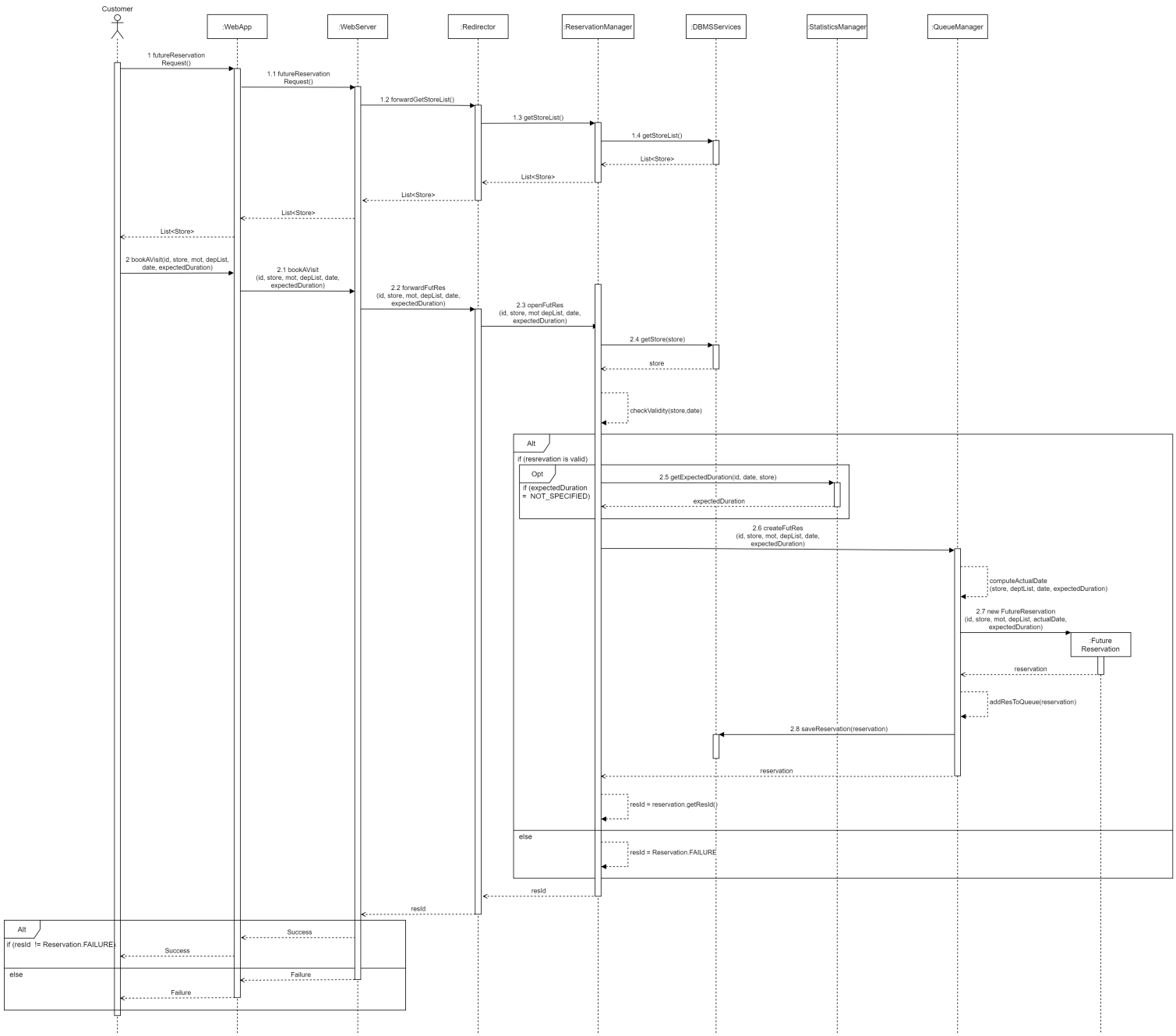


Figure 10: Future reservation

When the customer selects the option to book a visit, the WebApp, through the WebServer, requests a list of stores from the ApplicationServer and this request is then forwarded by the Redirector to the ReservationManager that retrieves the list of stores form the DMBS and sends it back to the customer.
The customer can then select a store form the list provided, the date he would like to book and, optionally, a list of departments he is interested in and how long he thinks the visit will be. This information is then sent to the ApplicationServer where the Redirector routes it to the ReservationManager that checks that the resrvation is valid (the selected store is open at the time, the maximum number of reservations has not yet been reached), and if it is, the reservation can be generated: first, if the expected duration has not been specified, it is inferred by the StatisticsManager, based on the past reservations created by the customer,

and returned to the ReservationManager; then it ask the queue manager to create the reservation and insert it into the queue in an appropriate time slot, considering the other reservations already present. The reservation is then saved into the database.
If the reservation was not valid, then it is rejected.

2.4.4 On premise reservation

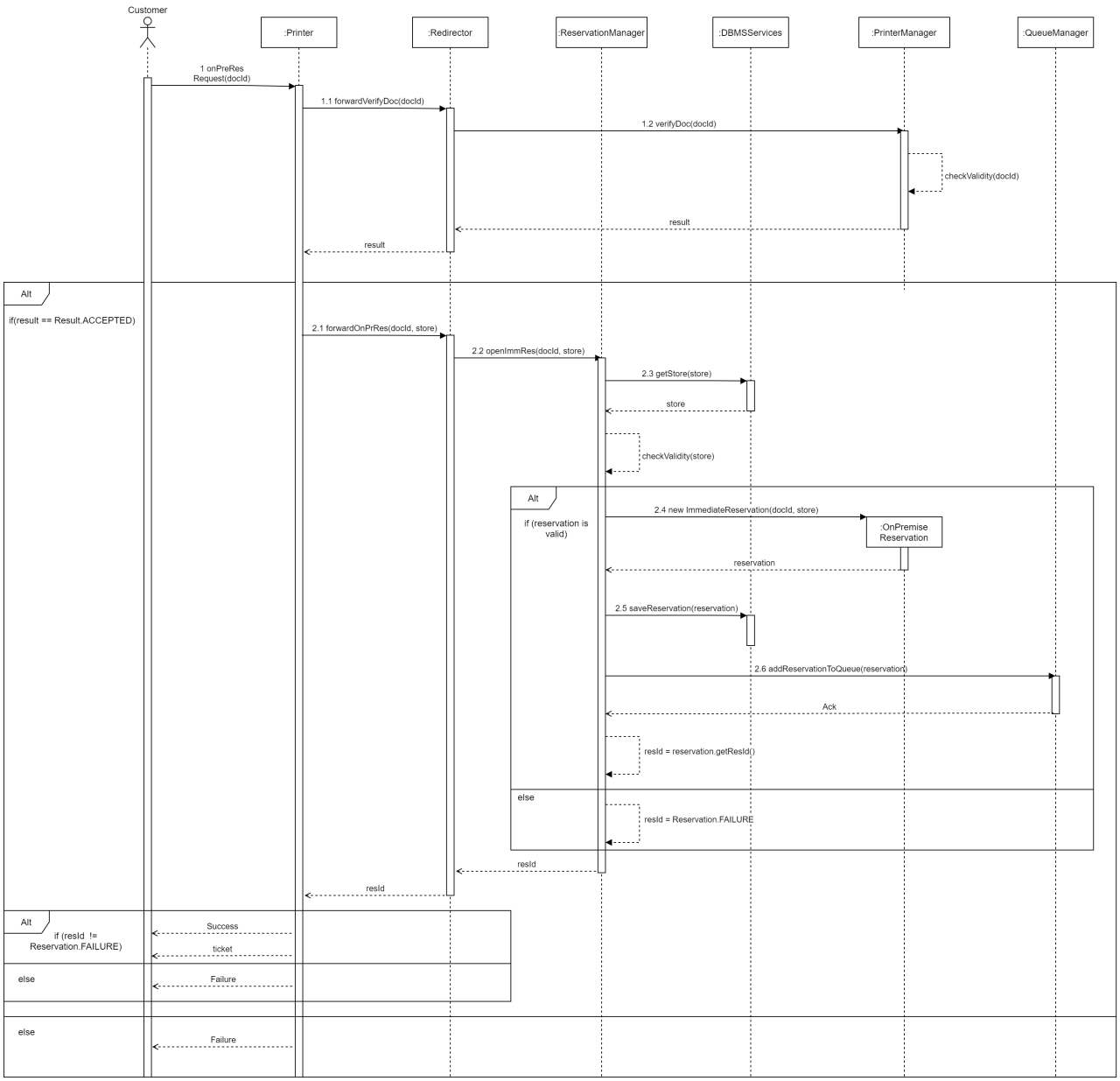


Figure 11: On premise reservation

The customer selects the option to create an on premise reservation, and provides the TicketPrinter with an identification document. This request is then forwarded by the Redirector to the PrinterManager that checks that the document is valid. If it is, an on premise reservation request is then sent to the ApplicationServer where the Redirector routes it to the ReservationManager that checks that that the reservation is valid for the selected store at the time of creation (e.g. the store is open), loading the store info from the DBMSServices, and if it is, the reservation is confirmed, added to the queue of that store

by calling QueueManager, and saved in the database. It is rejected otherwise.

2.4.5 Statistics generation

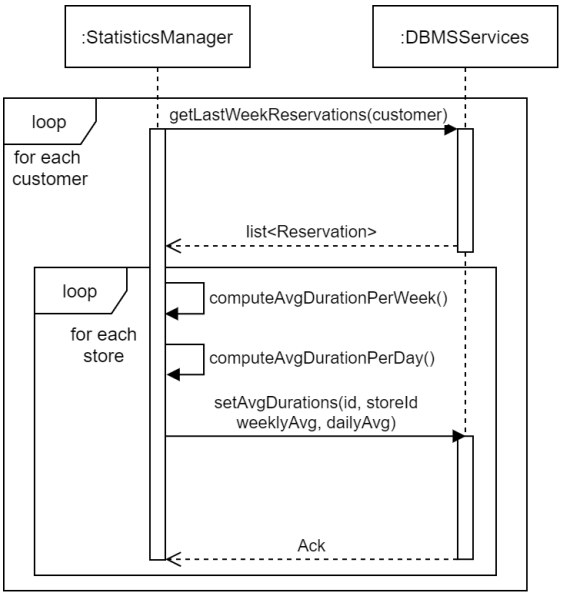


Figure 12: Statistics

Every week the StatisticsManager retrieves a list of the reservations of the past week, and for each customer and each store computes the average daily and weekly duration of the visit, and stores them in the database.

2.4.6 Customer enters store

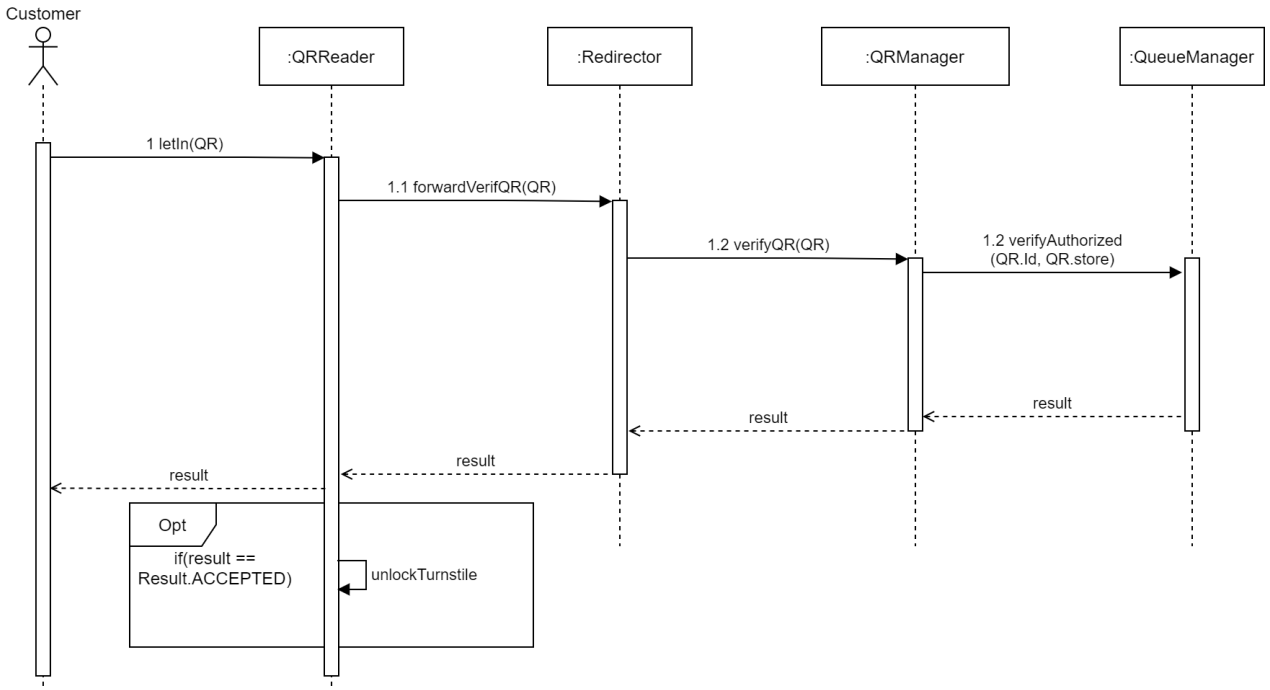


Figure 13: Store entry

The customer provides the QR reader with a QR code (be it on the screen of his/her smartphone or printed on paper). That code is then forwarded by the redirector to the QRManager that decodes it and calls the QueueManager to check that the QR code belongs to a customer that is now authorized to enter the store. If this is the case, the turnstiles unlock. Whenever the customer enters or exits a store, QueueManager saves the time of arrival/departure in the reservation, and then saves them in the database.

2.4.7 Store owner registers store via web app

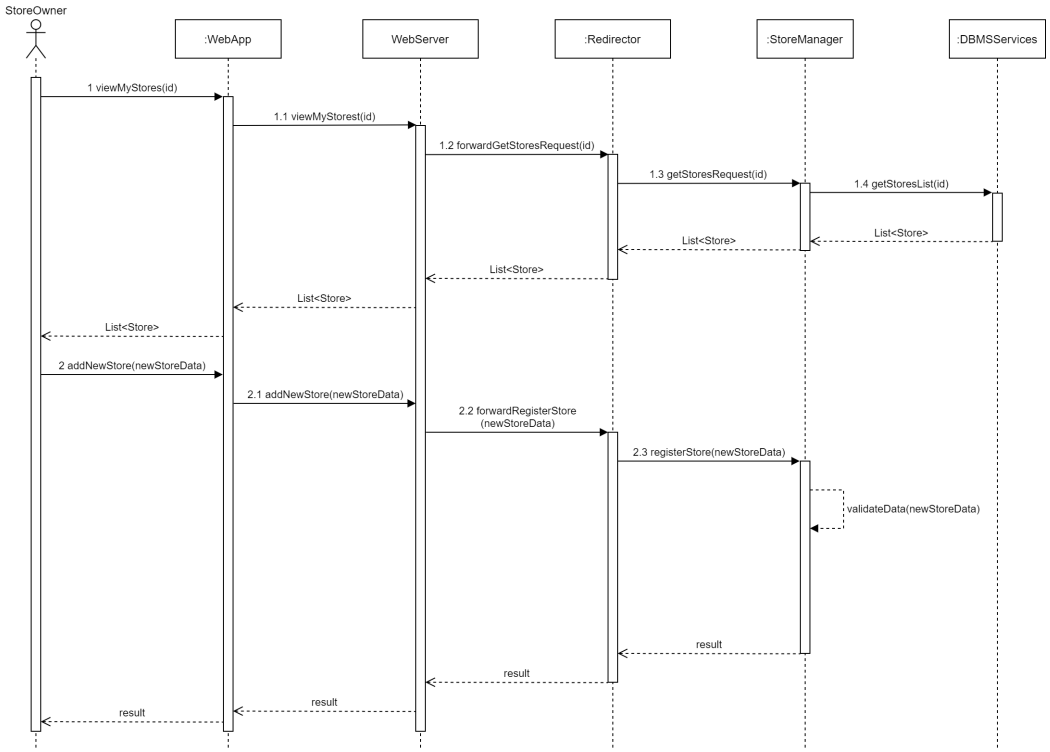


Figure 14: Store registration

The store owner opens the app and look at all his/her stores; he/she decides to add a new store, so he/she insert all required data, that are sent, through the redirector, to the StoreManager. This component validates the data and, if everything is correct, asks the DBMSServices to save the data of the new store. Of course, in this process, only the store has been added, so the store owner will then also need to register the TicketPrinter, the QRReader and the StoreMonitor to the system.

2.4.8 Customer is notified by smartphone app

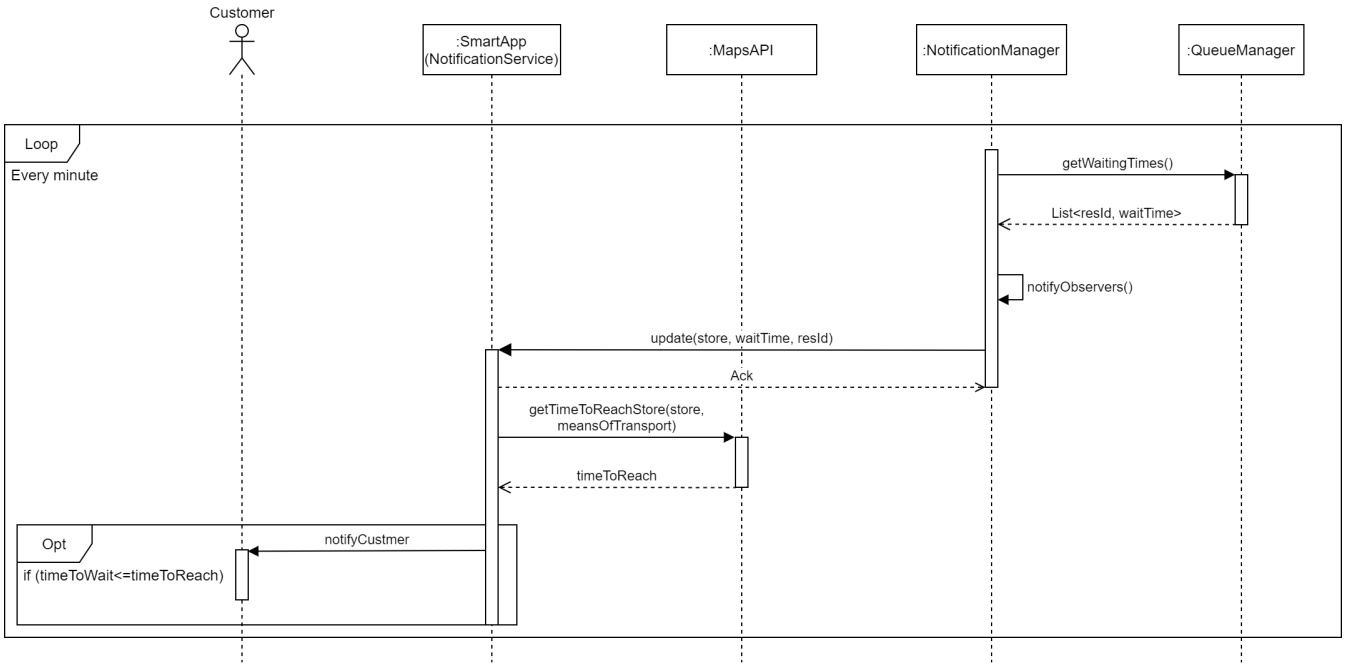


Figure 15: Notification

Periodically the system must check if it is time to notify the customer about his/her reservations.
Once every minute, the NotificationManager will ask the QueueManager to send, for each of the relatively closest reservations(those that hava as target time less than an hour with respact to the current time) the time remaining before they become authorized. Each of this times is then sent to its respective customer, if the reservation has been generated by a phone app.
Then the phone app will use the MapAPI to get an estimation of the time necessary to reach the store from the location of the device, by the means of transport specified at the creation of the reservation. Now, by comparing this two numbers, it can decide whether notify the customer or not. Of course there can actually be an additional value n such that the customer is notified if the time to wait is less or equal to the time to reach plus n. In this way the customer has a little notice before departing.

2.5 Component Interfaces

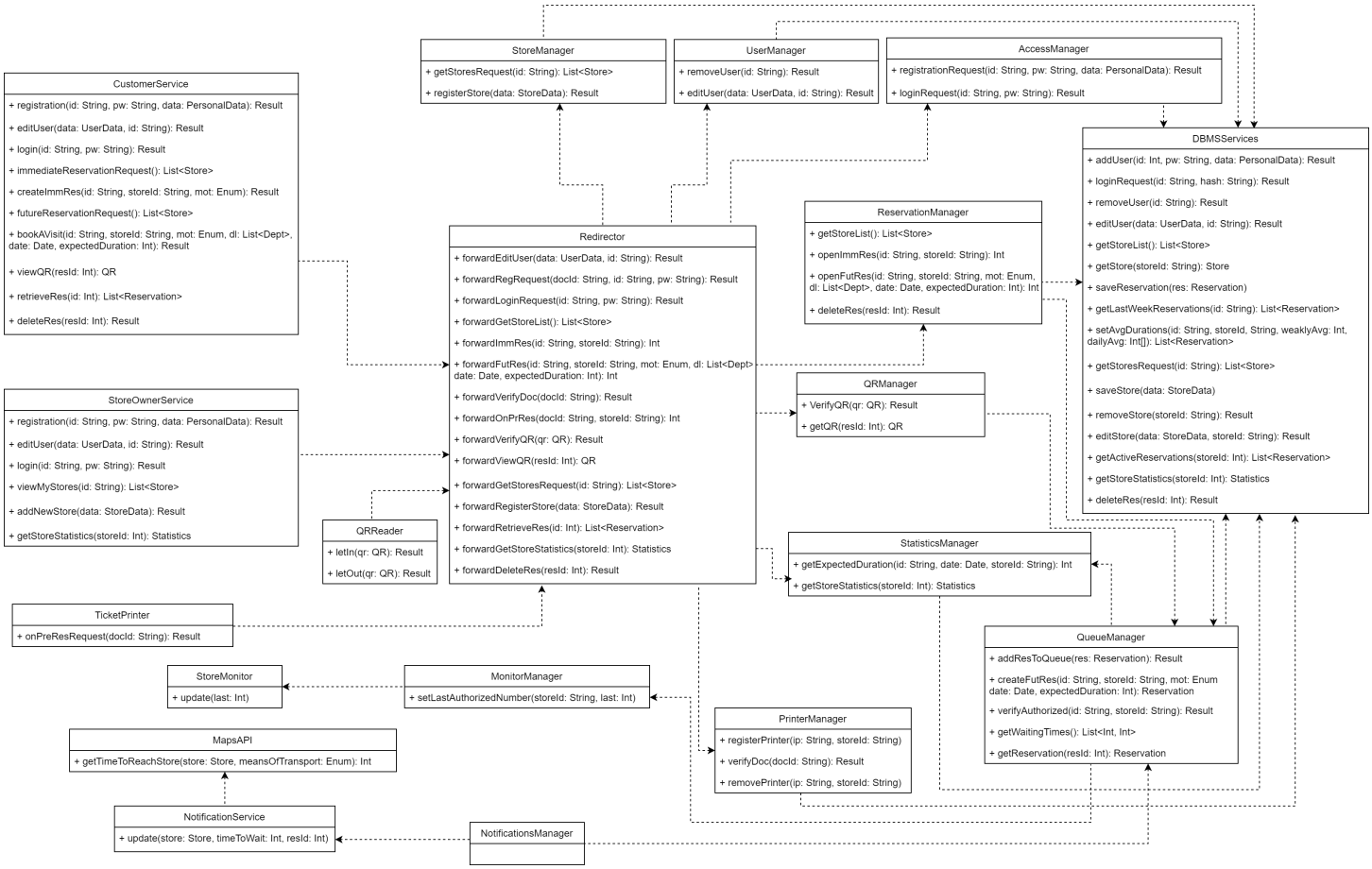


Figure 16: Component Interfaces

In this diagram we represented the interfaces of the system components with their methods, and such interfaces interact with each other.
For ease of representation we split WebApp and SmartApp component functionalities into three interfaces: CustomerService, StoreOwnerService and NotificationService. The WebApp component, that allows customers and store owners to interact with the system with a computer, implements the CustomerService and the StoreOwnerService interfaces. The SmartApp component, that allows customers and store owners to interact with the system with a smartphone, implements the CustomerService, StoreOwnerService and NotificationService interfaces.

All communication from the outside world (except DBMSServices) to the ApplicationServer is mediated by the Redirector, while the ApplicationServer sends messages directly to the interested components: NotificationService communicates directly with the NotificationService, and MonitorManager communicates directly with the StoreMonitor component.
Methods written here provide a logical representation of what component interfaces have to offer. They will be a soft guideline for the developers, that will be able to adapt them to create working components, facing the various problems that emerge during the implementation.
At time of creation a user is assigned a string called id that uniquely represents that user. At time of creation a reservation is assigned a string called resId that uniquely represents it.

2.6 Selected Architectural Style and Patterns

The proposed architecture of the system make use of several architectural patterns, some of which has already been breafly described in the previous sections. We now present a more expanded specification of those architectural choices.

2.6.1 Three layer Client-Server

The overall architecture is a Three layer (presentation, application, data) client-server structure. This pattern has already been previously detailed, so check Section 2.1: Overview for more information about this style and why it has been chosen.

2.6.2 RESTful architecture

To manage the distributed system we choose a Representational state transfer (REST) architectural style. This style provides a data transmission system based on HTTP, with no other layers, which creates a stateless communication, i.e. it doesn't support the existence of sessions. In the RESTful Web Service requests are made to the URL of the resource or of the set of resources, which must be well defined and unambiguous. All operations will be mapped to the HTTP methods, e.g. GET, POST, PUT, DELETE.

This architectural style is used for fast performance, due to the absence of coupling between client and server components and upper layers to the HTTP protocol, and to ensure the ability to grow, by reusing components that can be managed and updated without affecting the system as a whole. In fact, due to the pandemic, the S2B has a strict time to market that we need to hit, in order to provide our functionalities when they are more needed; for this reason every extra function or modification we may want to add will need to be implemented after the release of the application, so an easy to update architecture is highly advised.

As a REST architecture the system will support cacheability, eliminating some client–server interactions, further improving scalability and performance.
Moreover it must have a single uniform interface, that simplifies and decouples the architecture, which enables each part to evolve independently. This constraint is checked by its subconstraints:

- in our system resources identified in requests are conceptually separate from the representations that are returned to the client
- resources given to the client can be manipulated through the client representation
- each message includes enough information to describe how to process the message
- the client is able from the initial URL to use server-provided links dynamically to discover all the available resources it needs

The other constraint of the REST style, i.e. the layered system with client-server architecture, has already been discussed in the previous sections of this document.

2.6.3 Model View Controller (MVC)

In order to separate the internal representation of the information from how it is presented to the user, we decide to apply the well-known Model View Controller pattern.
This style is based on the division of the whole sysetm in three parts:

- the model: the component responsible for managing the data of the application and for receiving user input from the controller; in our architecture it is represented by the database, and the DBMSServices which manages all the data of the system.

- the view: the component responsible for the presentation of the information to the users and their interaction with the system; in the proposed architecture it is represented by the components of the presentation layer, namely the SmartApp, the WebApp, the StoreMonitor, the TicketPrinter and the QRReader.

- the controller: the component responsible for receiving input from the user and, based on them, performs interactions with the model. In our system the controller is represented by the various components of the application server, which manage the input and perform changes to the database.

This pattern has been chosen to guarantee the maintainability and the reusability of the code, as well as to promote a parallel development.

2.6.4 Facade pattern

In the system architecture we can notice that the client accesses the application server through the redirector component; this element constitutes the main part of the facade pattern.
The redirector is a "facade", an exterior face of the application server, that hides the internal complexity of the system, providing a simple and unique interface that the client can access to use the various other components of the system.
Using this pattern the client does not need to know which component of the architecture he need to contact in order to request a service, because all the routing is internal to the redirector.
Another major benefit of this pattern is that any change in the server-side of the architecture will stay undetectable from the client side, provided that the facade stays the same.

2.6.5 Observer pattern

To implement the notification and store monitor functionalities, we have chosen to use the observer pattern.
The notification service is, in fact, provided as follows: when a reservation is created from the phone app, the application itself subscribes to the NotificationManager, and becomes an observer of it. When the manager update the waiting time for the next reservation (which is done periodically by asking the QueueManager), it will also call a "notifyObservers" method, that will fetch all the observer who has a reservation relative to one of the new waiting times, and updates them.
A analogous approach is used for the StoreMonitor: each monitor "observes" the MonitorManager, which receives an update by the QueueManager each time a reservation becomes current; each time this happens, the manager calls "notifyObservers" and looks for the monitors of the store for which the reservation has changed state, and updates them. In this way all the screens are updated only when they should change the displayed number, with no need of polling the server, improving efficiency.

2.7 Other Design Decisions

2.7.1 Thick/thin client

As already discussed in Section 2.2.20: Additional specification the smartphone app is an example of thick client, being it able to partially store the reservations and use the MapsAPI to compute an estimation of the time needed to reach a specific store.
The web app is, instead, a thin client, which mainly displays the pages provided by the WebServer.

2.7.2 Realational DataBase

A relational DB is very fitting for this system, because the data are bounded to each other, e.g. users with reservations with stores, and those relations are well captured by join operations.
Of course a non-relational DB may have some advantages, like the capability to handle large volumes of data at high speed with a scale-out architecture, but for our case the relational one has been preferred, due to the expressiveness of the SQL language and to the necessity to create data dependent structures.
Other advantages of our DB choice are the accuracy (data are stored once, no data duplication), the possibility for multiple users to access the same database and the security (data access can be limited to only particular users).
However an hybrid (relational + non relational) DB may be taken into account in future releases or updates.

3 User Interface Design

In the Requirements And Specification Document, Section 3.1.1 User interfaces, we already provide many mockups of the user interface for the smartphone app. Here we present an additional one to exemplify the interface of the web app for the creation of an immediate reservation.

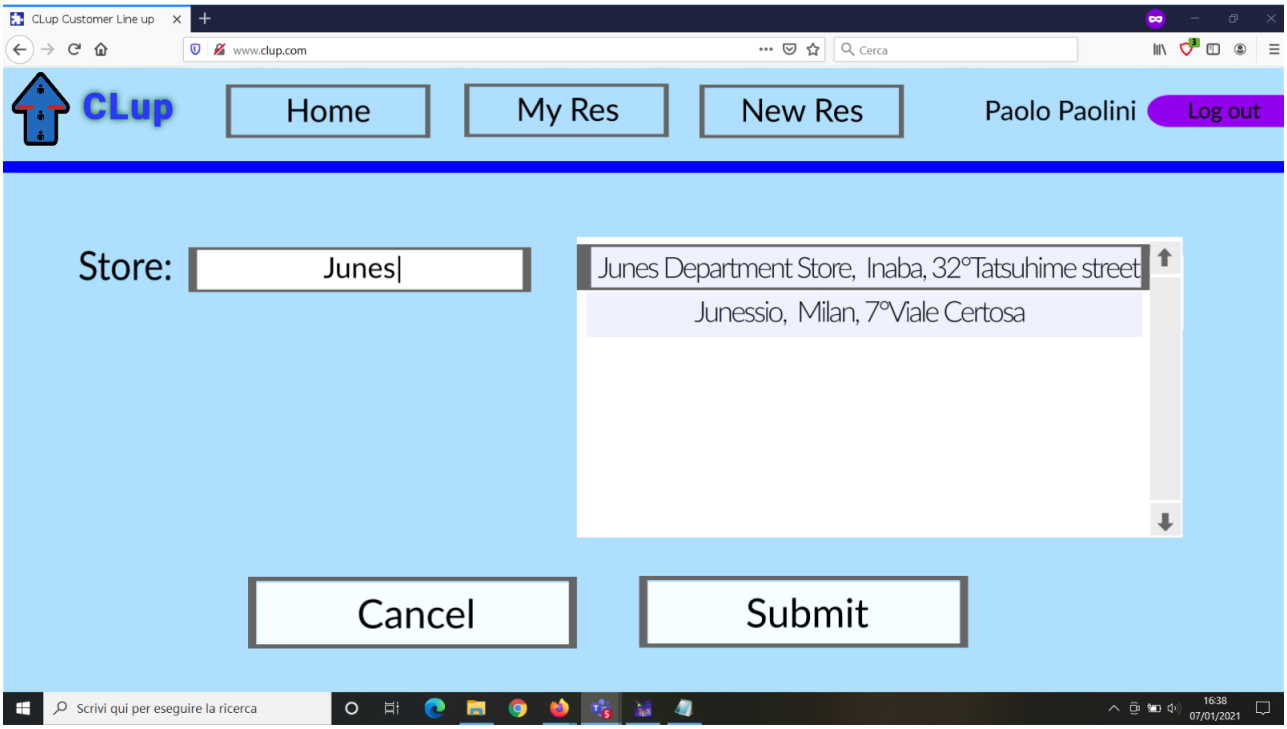


Figure 17: WebApp mockup

4 Requirements Traceability

4.0.1 Requirements Traceability

R1 Turnstiles unlock if and only if activated by authorized customers.

- **QueueManager**: this component manages the states of the reservations. A customer is considered authorized to enter a particular store if one of his/her reservations for that store is authorized.
- **QRManager**: this component calls QueueManager when a customer attempts to enter by providing QRReader with a QR code. The turnstile will be unlocked only if the customer is authorized.

R2 The number of customers in each department of the store never exceeds the occupation set by the owner.

- **QueueManager**: when the occupation of a department of a store has reached its maximum, this component stops authorizing reservations for that department until the occupation has decreased.
- **StoreManager**: this component allows store owners to set and change the occupation of one of the departments of their stores.

R3 The monitor outside the store displays the number of the last authorized customer.

- **QueueManager**: whenever a reservation of a store is authorized, this component notifies the MonitorManager of the number of that reservation.
- **MonitorManager**: when this component is notified that a new reservation is authorized for a store, it updates the value on every StoreMonitor of that store.

R4 The system allows customers and store owners to register and log in.

- **AccessManager**: this component allows customers and store owners to register and log in
- **UserManager**: this component allows customers to edit their personal information if desired.

R5 The system validates the authenticity of the identifying information provided.

- **AccessManager**: this component verifies the authenticity of the identifying documents provided by customers that register to the service.
- **PrinterManager**: this component verifies the authenticity of the identifying documents provided by customers that try to create an on premise reservation.
- **StoreManager**: this component verifies that documents regarding a store are valid.

R6 The system allows customers to search for a store among those registered by their owners.

- **ReservationManager**: this component allows customers to retrieve and view a list of all existing stores.
- **StoreManager**: this component allows store owners to create, view, update and delete their stores.

R7 Registered customers can send a reservation request to the system.

- **ReservationManager**: this component allows customers to send a reservation request, be it immediate or future via web app or smartphone app

R8 Non registered customers with an identifying document can request reservations through the printer.

- **ReservationManager**: this component recieves from the ticket printer on premise reservation requests.
- **PrinterManager**: this component identifies on premise customers.

R9 Registered customers that book a visit can specify estimated visit duration.

- **ReservationManager**: this component allows customers that book a visit to specify estimated visit duration.

R10 Registered customers that book a visit can specify desired product categories.

- **ReservationManager**: this component allows customers that book a visit to specify desired product categories.

R11 The system provides customers with a QR code to enter the store once authorized.

- **QueueManager**: this component manages the states of the reservations. A customer is considered authorized to enter a particular store if one of his/her reservations for that store is authorized.
- **QRManager**: this component provides a QR code if the customer requires it.

R12 The system uses gathered data to build statistics.

- **StatisticsManager**: this component uses the data of entry time and exit time of customers to create statistics about the estimated duration of customer visits.

R13 Registered customers with a smartphone are alerted when their turn is near.

- **QueueManager**: this component manages the states of the reservations.
- **NotificationsManager**: this component retrieves the time to wait for non expired reservations of customers from the QueueManager and sends this information to the respective customers. The SmartApp will notify the customer if time to wait is lesser or equal to the time to reach the store plus a slack.

R14 Registered customers can delete a pending or authorized reservation.

- **ReservationManager**: this component allows customers to delete pending or authorized reservations.

R15 Authorized reservations expire if they do not become current in a certain time window (specified by store owner)

- **QueueManager**: if an authorized reservation does not become current in a time windowm, this component changes its state to expired.

R16 Registered customers must specify desired means of transport while requesting a reservation in order to receive notifications.

- **ReservationManager**: this component allows customers to specify the means of transport while creating a reservation.

R17 Reservations are authorized according to a FIFO policy

- **QueueManager**: this component manages the queue of each store in a first come first served manner.

5 Implementation, Integration and Test Plan

5.1 Overview

In this last section we provide all the necessary specifications about the plan for the implementation and the testing of the system.
This phase is, of course, critical for the development of a reliable software system. It is important to observe that, while testing can show the presence of bugs in the code, passing the tests does not imply the absence of errors in the final application. Still, with our tests, we will try to find the majority of the bugs before the product hits the market (and also after, maintenance is important).

5.2 Implementation Plan

For all the implementation processes we have chosen a **bottom-up** approach, which consists in implementing first the base components, and then, in a iterative way, those that require them. We topologically sort the dependencies between the components such that every time we implement one of them we have already realized all the others on which it depends.
The unit testing will be carried out in parallel with the implementation. Using the bottom-up approach we will use several drivers, during all testing process, which will mimic the behavior of the elements that will use the component under test. On the other hand, the selected approach will never require the creation of stubs, elements that simulate the behavior of not yet implemented modules, required by the component under test.
With this incremental approach we have in each moment fully functional components, being sure that the work already done is complete and functional. Moreover, beginning from the bottom, the core components will be the first to be implemented, and so the ones that will be more tested, improving the robustness of the overall system.

The first element that we will implement is the **DBMSServices** component, which manages the model, i.e. the data, of our application, and so it is required by most of the components of the system.
After the data layer has been implemented we can proceed to the components that utilize it. The main logic of the application is implemented by the QueueManager component, so we would to implent it as soon as possible. In order to do this, we need to first realize the compoents on which the QueueManager depends. For this reason we will start by implementing **StatisticsManager**, and, in parallel, the **StoreMonitor**, then the **MonitorManager**.

At this point we have all the components needed for **QueueManager** to operate, so we implement it. Since almost all other components of the ApplicationServer rely on QueueManager and/or on DBMSServices to operate, we can now split our team to work in parallel on the following components:

- **AccessManager** composed of **LoginManager** and **SignUpManager**
- **UserManager** composed of **StoreOwnerManager** and **CustomerManager**
- **ReservationManager**
- **StoreManager**
- **QRManager**
- **PrinterManager**
- **NotificationManager**

After the creation of all these components, we can now create the element that routes the external queries to the various internal services: the **Redirector**. Once this is done, all the modules required for the operation of the ApplicationServer have been implemented, so the application layer is completed. We can now proceed to realize the front end of the system.

First we work on the components that will run on the hardware interfaces of the stores, i.e. the **QRReader** and **TicketPrinter** (the StoreMonitor has already been implemented at the beginning). Then we realize the services that allow user interaction: **CustomerManager**, and **StoreOwnerService** in their respective versions for the **WebServer** and the SmartApp. The WebServer is complete, so we can instantiate the **WebApp**. For the SmartApp we also need the NotificationService which depends on **MapsAPI** to communicate with external mapping service (e.g. Google Maps). Therefore first we implement the latter, then the **NotificationService**, with which the **SmartApp** is complete.

5.2.1 Integration Strategy

For the implementation and testing plan we decided to use the **bottom-up** approach. According to this approach, we carry on the procedure as described in the following diagrams.

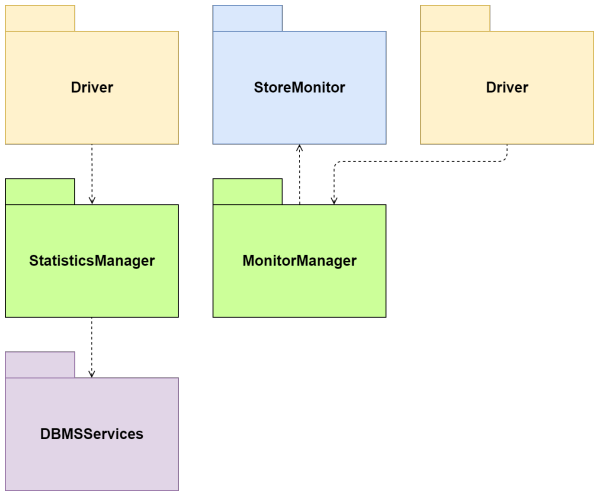


Figure 18: Integration 1

The first element implemented and unit tested is the DBMSServices, followed by the components that will be necessary for the QueueManager, the core of the application. Of course the DBMSServices is the first to be implemented and tested because all other components rely on it in one way or another.

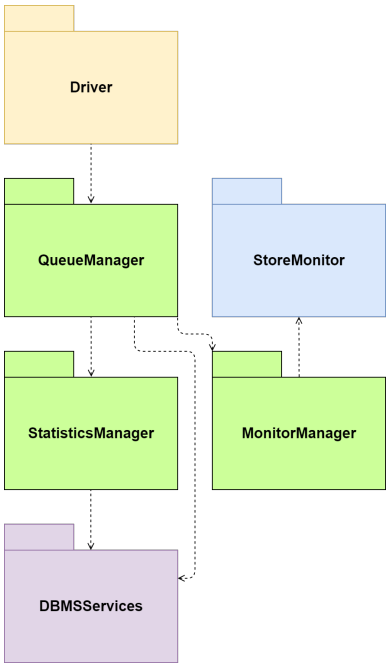


Figure 19: Integration 2

The core of the system is now implemented and tested; it will be critical for most of the functionalities offered by the system: management of the reservation, QR, to notify customers with a smartphone app. The QueueManager relies on the components previously realized.

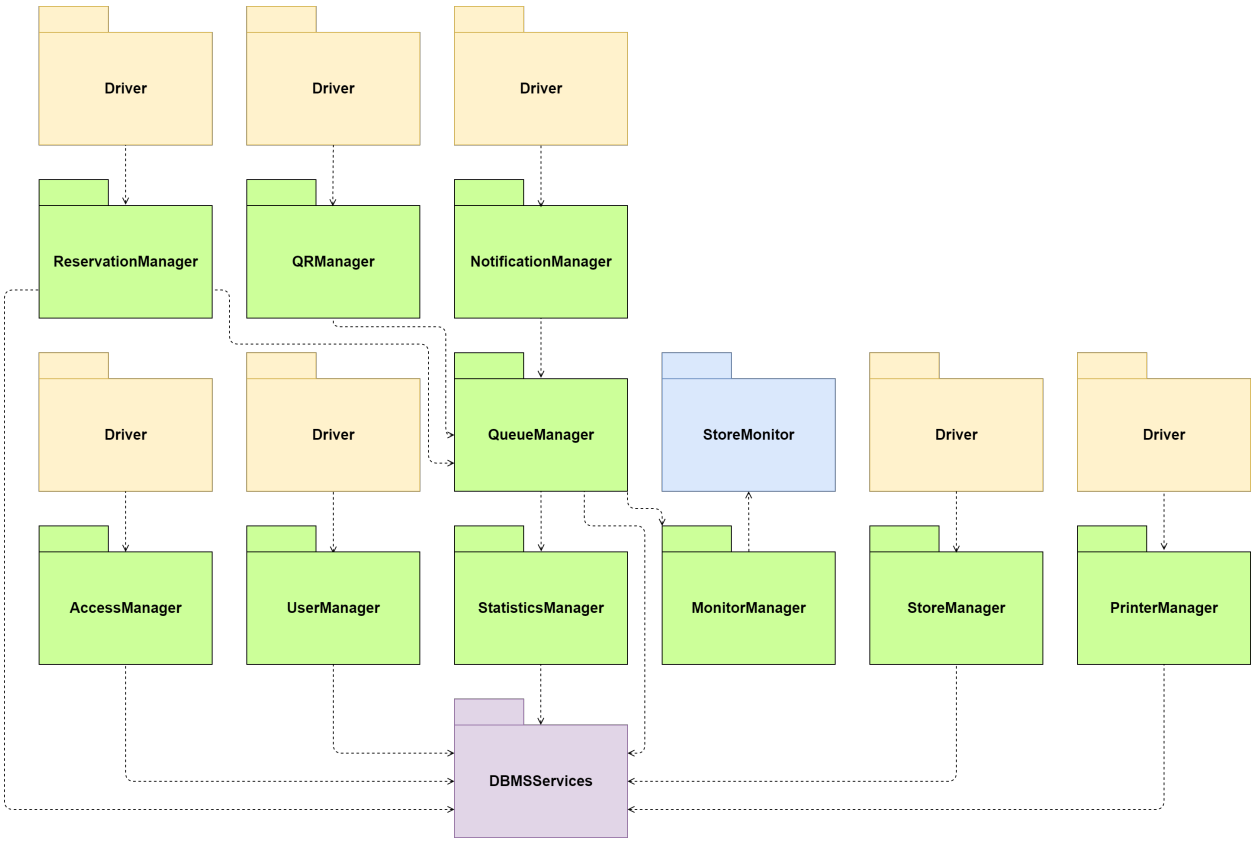


Figure 20: Integration 3

Many components are implemented and unit tested in parallel and now they are integrated with the core of the system. The integration testing can now focus on most of the functionalities of the final system: log in, registration, management of user and store data, creation and deletion of reservations of different kind, QR generation and recognition, aggregation of data to build statistics, management of printers and monitors of each store. Of course each of these functionalities will need a driver in order to be tested. Moreover the parallelism can speed up the general development process.

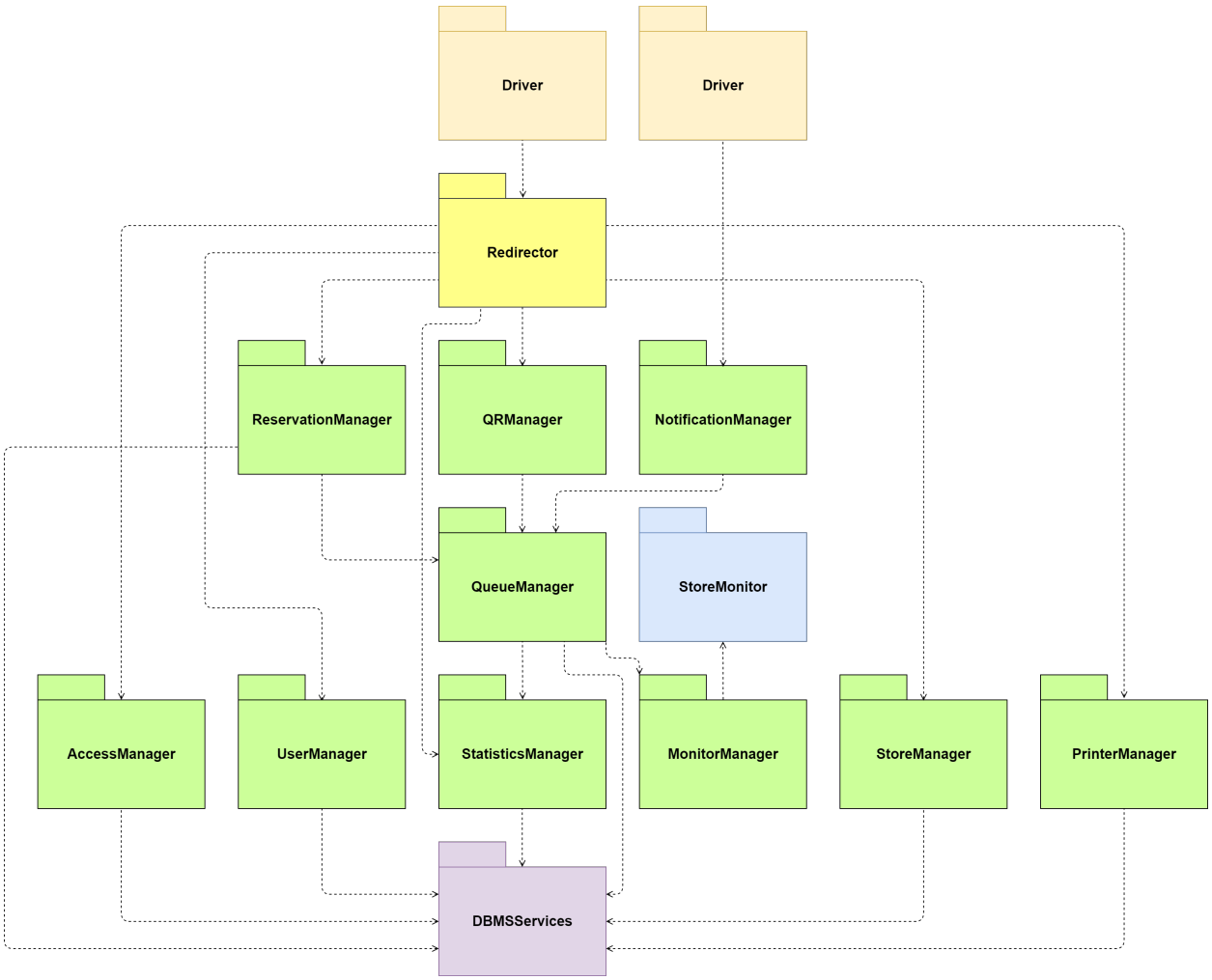


Figure 21: Integration 4

The component that realizes the facade pattern, the redirector, is implemented, unit tested and integrated with the rest of the system. Obviously according to the bottom-up approach this component must be developed after all the other components on which it relies have been implemented. The NotificationManager still uses a Driver because it is not accessed directly from the Redirector.

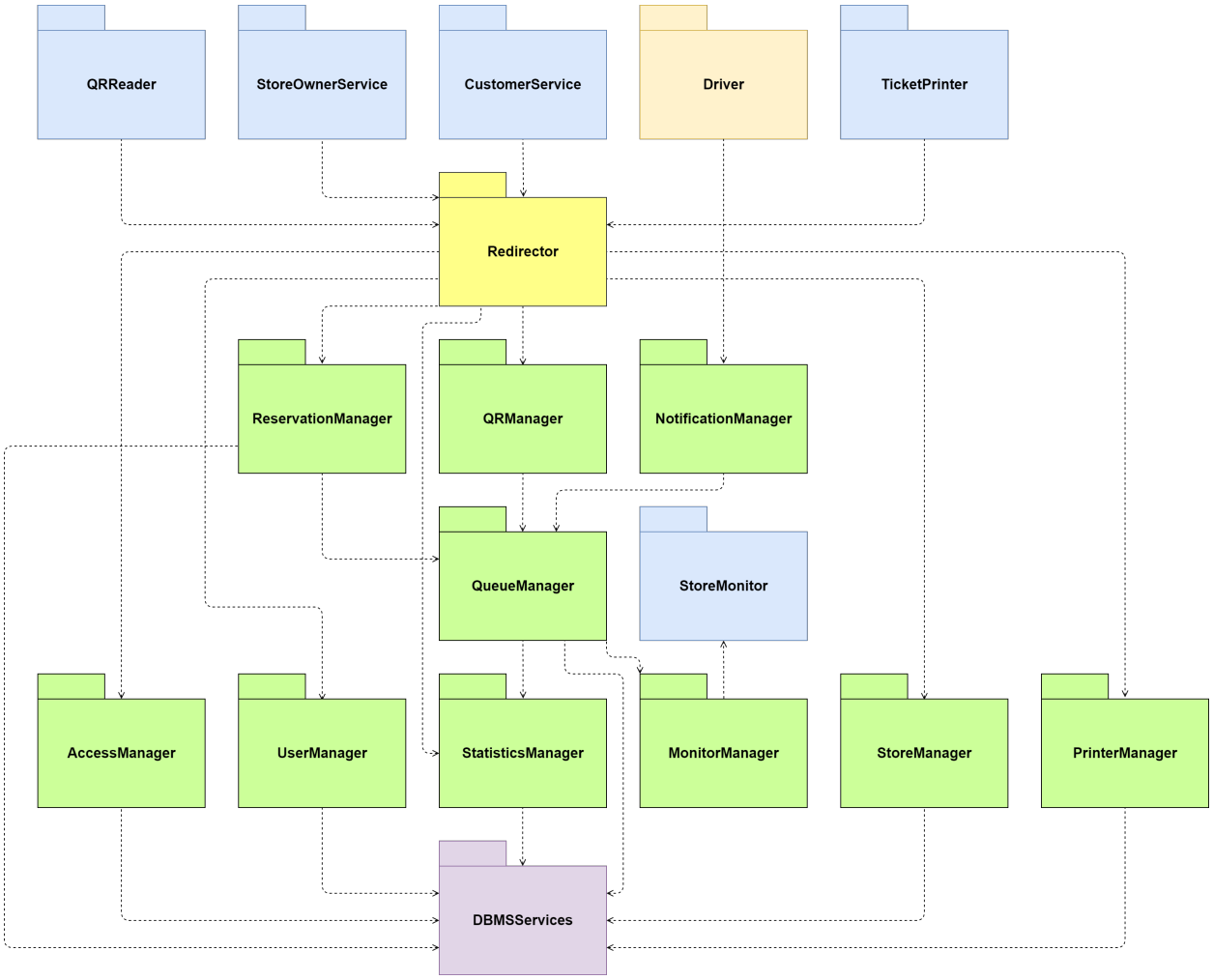


Figure 22: Integration 5

At this step we implement and unit test in parallel the components that operate as interface between the system and the users: QRReader, StoreOwnerService, CustomerService and TicketPrinter have been added and integrated with the redirector.

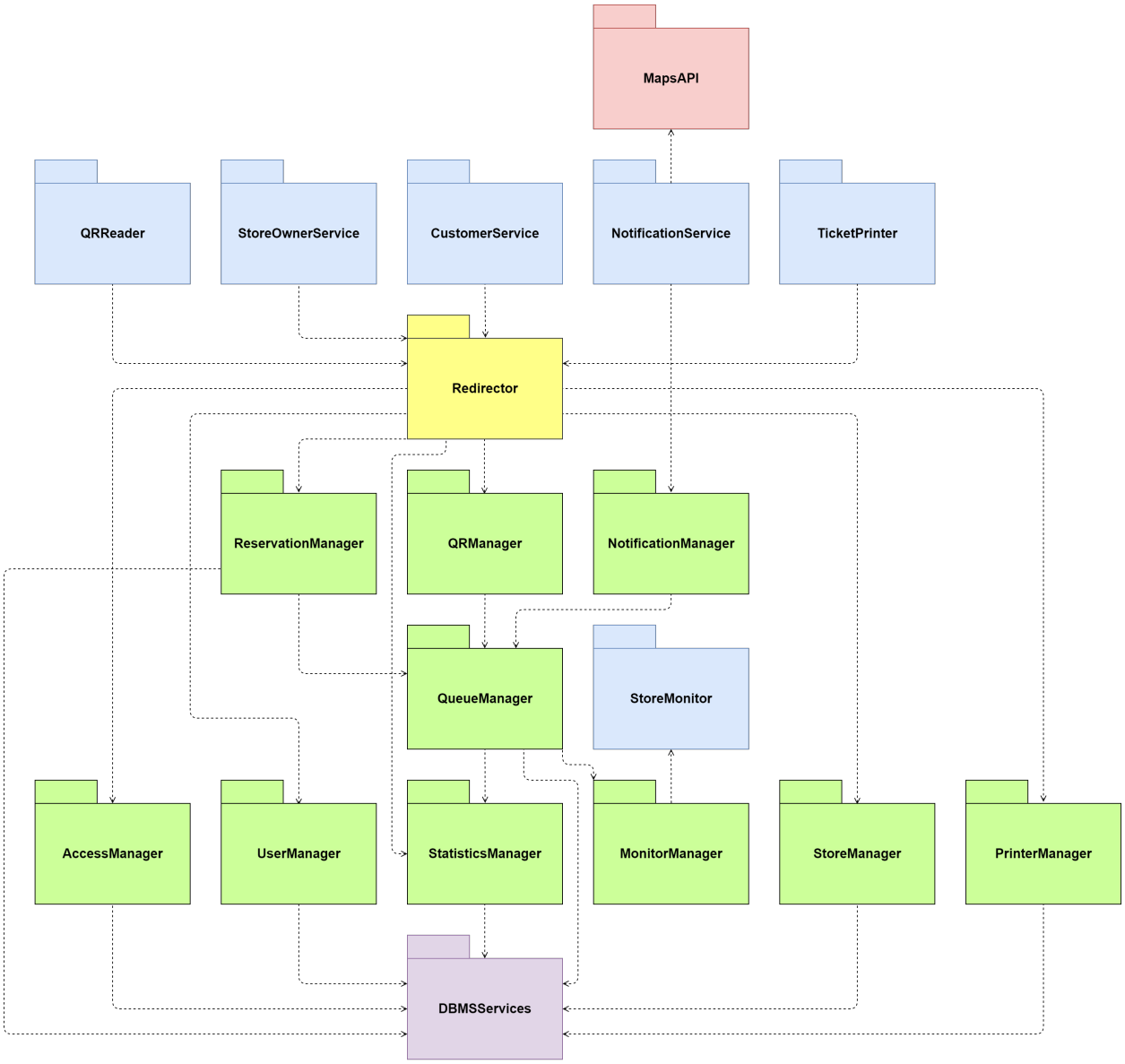


Figure 23: Integration 6

Finally the last step is to develop the components that will enable the notification functionality for the customers. The NotificationService makes use of MapsAPI, which connects it with the external mapping service.

Now that every component of the system has been implemented and integrated we can proceed to the system testing phase.

5.3 System Testing

Onece the whole system has been developed we have to check that all functional and non functional requirements have been met. This is the purpose of the system testing. Note that testing environment should be as close as possible to deployment environment so as to give credibility to the results obtained. These tests will be carried out by an independent team with no knowledge of the internals of the system (black box testing).
System testing focuses not only on guaranteeing that all functionalities work as expected, but also on

demonstrating the qualities of the application, through performance testing, stress testing and load testing.

5.3.1 Performance testing

Here we verify that the system matches expected performance requirements. This test will allow us to identify inefficient algorithms, bottlenecks, possibilities for query optimization as well as to produce a benchmark as a measure of the performance of the product, and to compare subsequent versions of the software.

5.3.2 Load testing

This phase aims to identify system behavior under heavy load conditions. Here we try to load the system until we reach its theoretical threshold and observe how and how long it manages to work without failing. This test's purpose is to find memory leaks, excessive use of memory, and buffer overflows.

5.3.3 Stress testing

In this phase we overload the system and observe how the system recovers from failure. We would like that the application could come back on line gracefully.

5.4 Additional testing specifications

In addition to unit, integration and system testing, the development team will need to carry out static analysis activities to verify the correctness of the artifacts produced.
There are two kinds of static analysis procedures: the walkthroughs and the inspections. They are both an in-depth examination of some work product from a team of reviewers with a limited time duration (no more than two hours). The focus of static analysis activities is to find errors and bugs, not to correct them (task assigned to the developers).
Walkthroughs are informal meetings held between peers (sw developers) aimed to verify product correctness in the scope of the developer team. The discussion is lead by the producer of the artifact.
Inspections are instead formal meetings, held by professional and qualified inspectors, which are the reviewers, aimed to check product correctness by examining a checklist of items. The discussion is lead by the official moderator of the reviewers team.

All developers are strongly advised to thoroughly comment the code they produce, in order to improve readability, to ease the review process and facilitate the process of updating the system. For instance every major method, class and attribute could be documented with Javadoc or Doxygen.

The developers working on the application logic must also provide, along with the code, enough test cases to cover at least 90% of it.

6 Effort Spent

6.1 Simone Abelli

Introduction	2.5
Overview	5
Component view	4.5
Deployment view	0.5
Runtime view	5
Component Interfaces	5.5
Selected Architectural Style and Patterns	4.5
User Interface Design	1.5
Implementation, Integration and Testing	6.5

6.2 Stefano Azzone

Introduction	2.5
Overview	3.5
Component view	6
Deployment view	2
Runtime view	6
Component Interfaces	6
Requirements Traceability	3.5
Implementation, Integration and Testing	6

References

- **drawio.org** was used to draw diagrams
- **alloy.mit.edu** was the reference for alloy model
- **uml-diagrams.org** was the reference for uml diagrams

