POLITECNICO
MILANO 1863

DD

CLup – Customers Line-up

Yasmin Awad, Lorenzo Carpaneto, Giovanni Dispoto

Contents

1 Introduction

1.1 Purpose

After showing a general description of the CL-up application within the RASD, this document focuses on analyzing the system's architecture and design, in order to satisfy the various requirements stated in the previous document. The purpose of the document is to provide a functional description of the main architectural components, showing their runtime behaviour and interactions and their interfaces. This document is mainly intended to be used by the developers and testers.

1.2 Scope

Customer Line-up (CL-up) is an application that aims to allow accesses to Stores in a safe way. The objective is to avoid as much as possible queue formation outside the Stores and limits the number of people inside it in an efficient way. To pursue this goal, the application allows Users to register either as Managers or Customers, providing services to both of them. Managers can create their Stores within the app and allow other Managers to help them in the organization of the Store rules. Tools are provided to allow Customers to book Visits or Tickets to the Stores remotely, allowing a greater and more efficient way for maintaining social distancing. This is done by the use of a virtual queue managed by the System, in which Customers' Reservations are stored and managed. In order to organize the various entrances in the Stores, an identifier for each Consumer will be used, that is a QRCode.

More information can be found in the Chapter 1 of the RASD.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

D.1 *Client*: Device or software that access a service made available by a Server

D.2 *Server*: Device or software that provides resources or functionalities to other programs or devices

D.3 *Firewall*: Network security system (hardware and/or software) that monitors incoming and outgoing traffic applying predefined security rules

D.4 *Store*: synonym of Grocery Store. A Store has one associated queue and Consumers can reserve Tickets or book Visits to shop there.

D.5 *Department*: macro area of the store. It can contain multiple categories of grocery.

D.6 *Guest*: anyone who downloads and opens the app but still has to sign up or log in. He/She cannot use any of the tools provided by CL-up.

D.7 *User*: a registered Guest.

D.8 *Customer*: someone who has logged-in with his/her credentials which is recognized by the System by his/her email, and is identified by his/her QR code. He/She can access their profile, request a Ticket or a Visit for a Store, and visualize them in the purchased section.

D.9 *Manager*: someone who has logged-in with his/her credentials which is recognized by the System by his/her email. He/She can access their profile, create and manage Stores.

D.10 *Owner*: a Manager who owns a specific Store.

D.11 *Capacity of the Store*: maximum number of people admitted inside the Store simultaneously.

D.12 *Virtual Ticket*: a Ticket reserved by a Costumer through the CLup application.

D.13 *Paper Ticket*: a Ticket reserved by a Consumer through the Totem

D.14 *Ticket*: the set of Virtual Tickets and Paper Tickets. Identifies both indistinctly.

D.15 *Visit*: a booked Visit reserved by a Customer through the CLup application.

D.16 *Ticket/Visit expiration*: A ticket or a Visit expires if the Customer does not delete them neither uses them, and the time to use them is past (the current time is greater then the entrance time on the ticket or visit).

D.17 *Temporal Quantum*: a time space of 5 minutes.

D.18 *Reservation*: any time of booking or requesting an entrance to the Store by a Consumer. It can be a Paper Ticket, a Visit or a Virtual Ticket.

D.19 *Duration of Shopping*: Duration of the visit booked.

D.20 *Queuing Mechanism*: the mechanism which handles the behaviour related to the queue.

D.21 *Totem*: a machine used to generate Paper Tickets, each one is related to a specific Store.

D.22 *Scanner*:a machine used to validate Tickets, each one is related to a specific Store.

D.23 *Queue Scaling Process*: process thanks to which Reservations are moved within the queue of the Store according to certain rules, so that the Capacity of the Store is respected and so that a Consumer who needs an additional amount of time to finish their shopping gets that time.

D.24 *Validated*: QR code associated to a Reservation that was correctly scanned at the entrance and at the exit.

1.3.2 Acronyms

A.1 *API*: Application Programming Interface

A.2 *RASD*: Requirement Analysis and Specification Document

A.3 *REST*: Representational State Transfer

A.4 *DBMS*: Database Management System

A.5 *GPS*: Global Positioning System

A.6 *HTTPS*: Hypertext Transfer Protocol Secure

A.7 *UML*: Uniform Modeling language

A.8 *MVC*:Model-View-Controller

A.9 *TSP*: Travel Salesman Problem

A.10 *MST*: Minimum Spanning Tree

A.11 *UX* : User Experience

A.12 *TDD*: Test Driven Development

1.3.3 Abbreviations

AB.1 [G$_i$]: i-th goal.

AB.2 [R$_i$]: i-th requirement.

AB.3 [D$_i$]: i-th definition.

1.4 Revision History

Date	Version	Comments
December 2020	1.0	first release

1.5 Reference Documents

- Specification document: R&DD Assignment AY 2020-2021
- UML documentation

1.6 Document Structure

The document is organized in this way:

- **Introduction** contains a general overview of the document.
- **Architectural Design** shows the main components of the system and their relationships, underlining the architectural choices of the design process.
- **User Interface Design** shows some user interfaces.
- **Requirement Traceability** shows the satisfaction of the requirements listed in the RASD by the design choices of the DD.
- **Implementation, Integration and Test Plan** shows the sequence in which the implementation and integration of subcomponents will occur and how the integration will be tested.
- **Effort Spent** shows the effort spent in developing the DD.

2 Architectural Design

2.1 Overview: High-level components and their interaction

In this chapter the architectural structure of the System will be described. In Figure 1 the high level overview of the System is shown. In the next sections we will describe in detail the various components and their interactions.

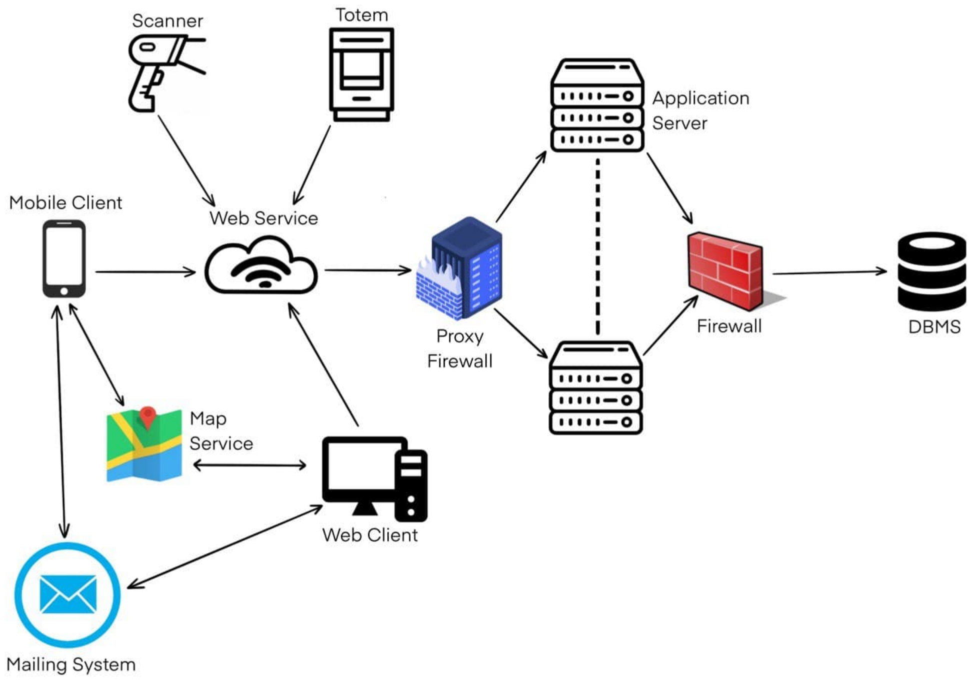


Figure 1: High-level overview of the System.

2.2 Component View

To describe the internal modular structure of the components, we show how they are connected together in the UML component diagram in Figure 2 Components are wired together using an *assembly connector* to connect the required interfave of one component with the provided interface of another component.

- **Mobile Client** and **Web Client**: represents the two machines that accesses the entire System and it's functionalities. We chose to implement them using a Thin Client approach.
- **Presentation**: it is used in order to display web pages of the application to a Web Client accessing through the browser. This layer only handles the presentation: it neither knows the logic of the application neither has access to the data base.
- **User Controller**: this component handles all the operations that affects the user data. It exposes method to change account credentials and stores them in the database. It also allows the client to access their data (for example their tickets or Visits).

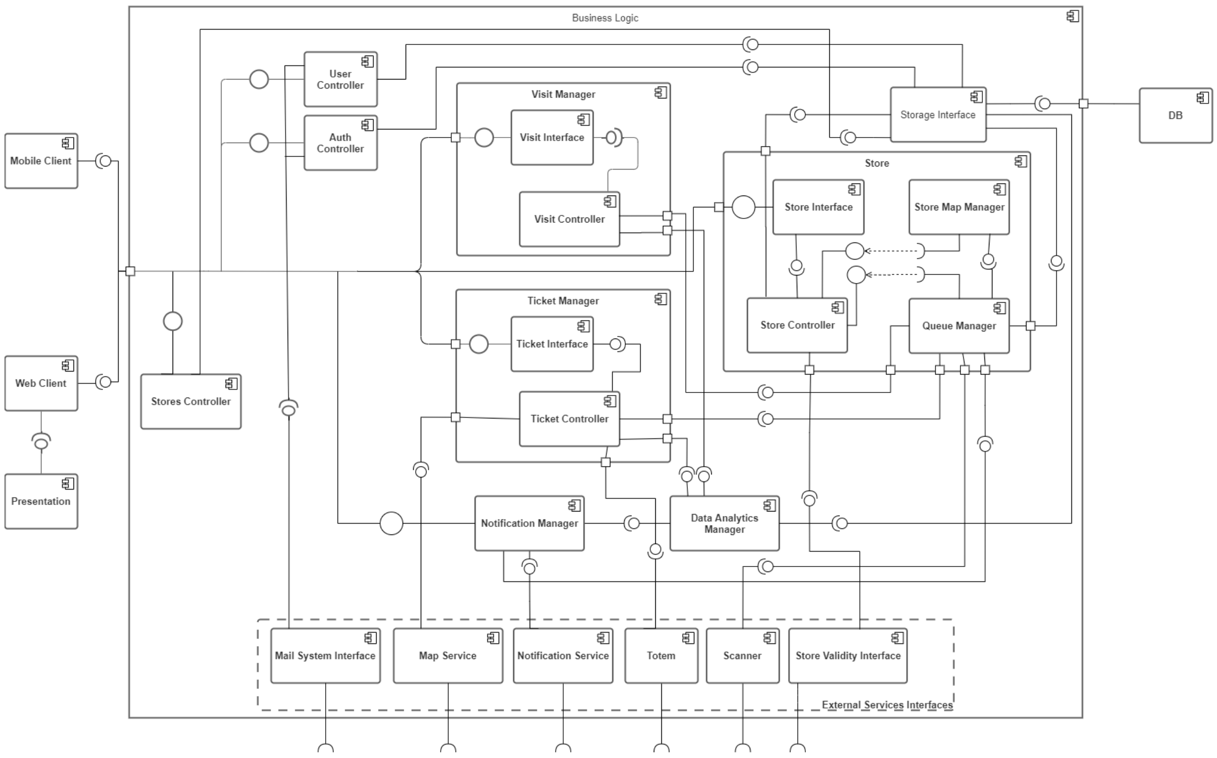


Figure 2: Component diagram of the system.

- **Auth Controller**: this component handles all the operations for the authentication such as registration process, login process and token checking. It communicates with the DBMS through Storage Interface in order to retrieve and insert user credentials. During the registration process or change password process this component use Mailing System Interface.
- **Data Analytics Manager**: this component analyzes Customers habits. Specifically it is a Recommender System that takes care of showing to a Customer the Stores most appealing to him and the best duration value to be included in a Reservation.
- **Notification Manger**: this component provides Notifications to the Customer, such as Virtual Tickets departures, appealing time slots of specific Stores and suggestions.
- **Ticket Manager**: this component is a subsystem in charge of managing Ticket requests:
  - **Ticket Interface**: it exposes to the Customer an Interface for compiling the requested Ticket attributes.
  - **Ticket Controller**: it checks the validity of the inserted Ticket's attributes requested. Moreover, it communicates with the Queue Manager in order to verify the possibility to accomplish the request.
- **Visit Manager**: this component is a subsystem in charge of managing Visit requests:
  - **Visit Interface**: it exposes to the Customer an Interface for compiling the requested Visit attributes.

  - **Visit Controller**: it checks the validity of the inserted Visit's attributes requested. Moreover it communicates with the Queue Manager in order to verify the possibility to accomplish the request.

- **DB**: it represents the DBMS, which provide an interface to read and store data. User credentials, Stores information and queues reservations are stored in the database.

- **Storage Interface**: it provide methods to access to database. This interface is needed in order to decouple all the components from the DBMS technology.

- **Store**: this component is a subsystem in charge of managing Store and its Queue:

  - **Store Interface**: it exposes to the Manager an Interface for creating and managing Stores.
  - **Store Controller**: it ensures that a Manager intending to become Owner of a Store within the application has the necessary authorizations, checking the validity of the information by acquiring specific credentials. The verification is done through an external service named Store Validity Interface.
  - **Queue Manager**: this component receives requests for the Reservations for the Store, checks that they can be placed in the queue and eventually inserts them. It communicates with the Ticket Controller, the Visit Controller, the Scanner and the Totem. It also communicates with the DB in order to retrieve and modify all the necessary information.
  - **Store Map Manager**: this component receives requests from the Customers who has already booked a Visit. It has to populate the map with the path that the Customer should follow during their Visit inside the Store. It communicates with the Queue Manager to look at the specified departments by other people during the time-slot of the Visit specified.

- **Stores Controller**: this component gives the list of the Stores to the client.

- **External Services Interfaces**: this set of components have as main objective to make API calls to the necessary third party services:

  - **Mailing System Interface**: it is an interface in charge of sending confirmation mails at the end of the registration process to the User. It also can send other type of notification mails.
  - **Map Service**: this service is used in order to retrieve the necessary information about a Customer position. This information is used by the Data Analytics System and by the Ticket Controller.
  - **Totem**: since the Totem of a Store has the task of issuing Paper Tickets, it needs to communicate with the Ticket Controller in order to make requests and allow the effective issue of the requested Ticket. This component is in charge of the communication.
  - **Scanner**: the Scanner application has the task of reading QR codes of Customers and Paper Tickets so that it can control the entries and exits from the Store. In order for it to carry out the correct checks, it needs to communicate with the Queue Manager to obtain the right information. This component is in charge of the communication.

- **Store Validity Interface**: it is used in order to check that the created store is a real store. To do this, it has to verify Special Credentials provided from the Owner.
- **Notification Service**: Is used from Notification manager in order to send push notifications to client application.

2.3 Deployment View

The following diagram shows how the whole system is distributed in various components. Is clearly possible to see that the system is structured in a 3 tier architecture and every components has their role as specified below.

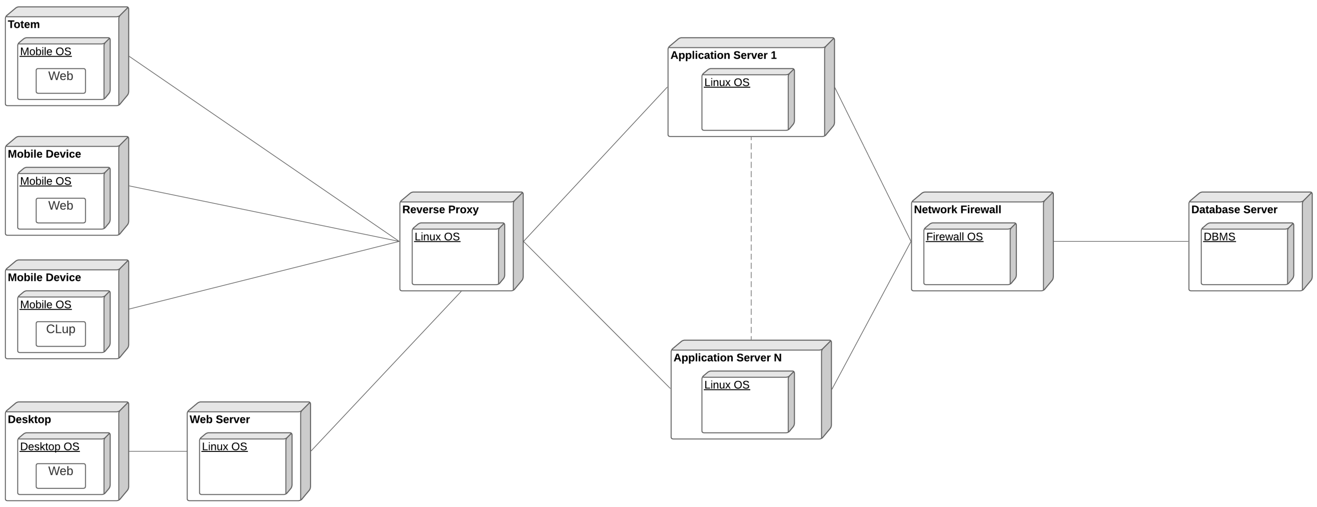


Figure 3: Deployment diagram

Clients
The first tier is composed by Clients machine. They could be Mobile with CLup App installed or Desktop that use the application through a Web Browser. There are also 2 Clients, Totem and Scanner App which are accessed through a web browser.

Web Server
Web Server is used to store, process and deliver web pages to desktop Clients. The communication between Clients and Server take place using HTTPS protocol.

Reverse Proxy
Reverse Proxy helps to achieve increased parallelism and scalability, load balancing requests distributing them between various Application Servers. This layer adds also a shield layer useful to protect the System from attackers.

Application Servers
Application Servers contains the Business logic of the System. They are many in order to achieve the Non Functional requirements described in the RASD.

Network Firewall
Network Firewall add another security layer before the Database Server, in order to avoid unauthorized access to data. This create a DMZ.

Database Server
Database Server contains all the data of the application.

2.4 Runtime View

Registration Runtime

When a client wants to register to access the functionalities of the app, they fill a form with the required credentials. These are sent through a post request HTTP to the Auth Controller. This component checks that everything sent by the client is in the apposite format (if it is not, it responds with HTTP error code). Then it communicates with the Storage interface which sends a query to the DB to insert the credentials if they have not been already used. If they are not available, the Auth Controller sends a with 402 HTTP error code. If they are available the Auth Controller asks the Mail Service to send a confirmation e-mail to the Client and the request is Accepted. The client has to then confirm the registration through the mail, and the confirmation is handled by the Auth Controller.

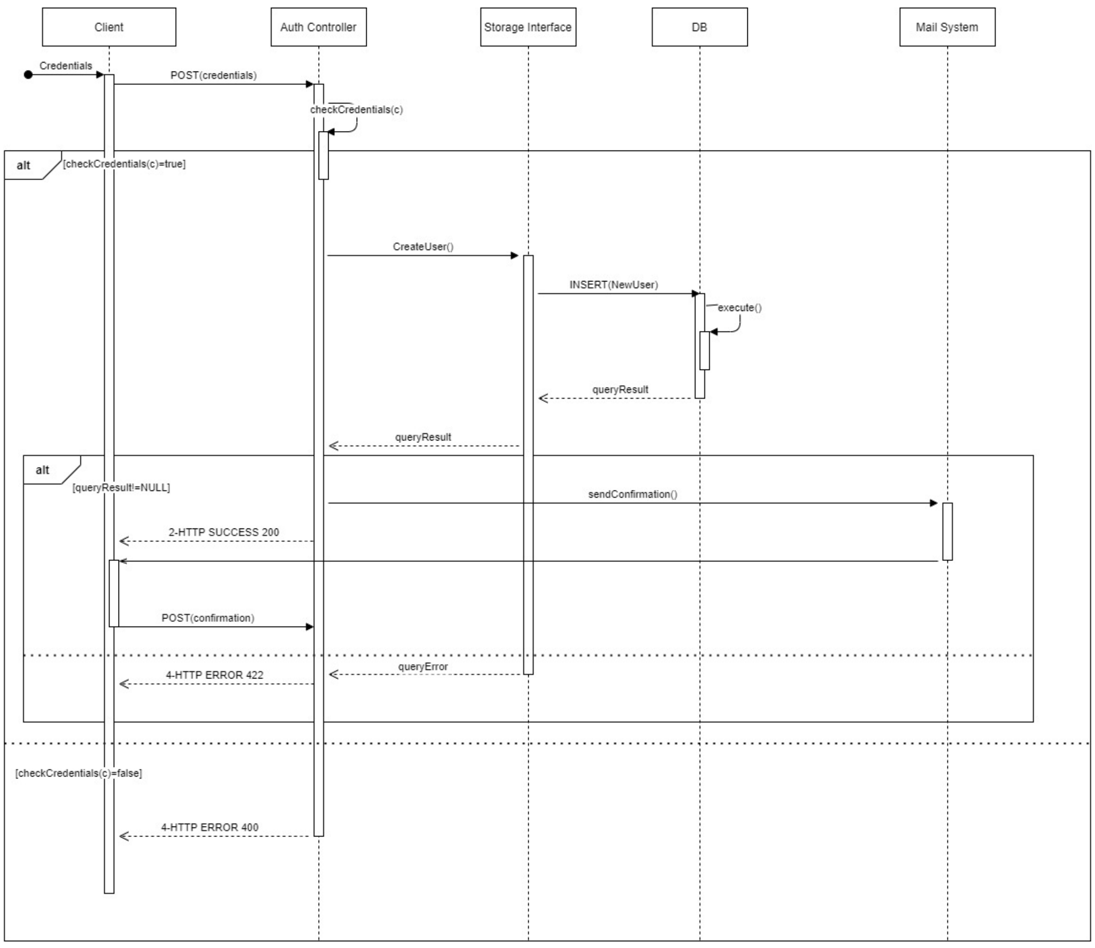


Figure 4: Registration runtime sequence diagram

Login Runtime

Once a client is registered, they can log into the app. The login process is handled by the Auth Controller. This receives a put request HTTP from the client and, after checking that the credentials have been sent in the right format, it delegates to the Storage interface the task of creating a query to check if the user is actually logged in and the credentials sent are correct. If they are not, the Auth Controller sends a response with an HTTP error code. Otherwise it checks whether the Client already has a valid token to access the app or not. If not it creates one and sends a request to the storage interface to query the Database for an update with the new information. It then grants the access to the app to the Client.

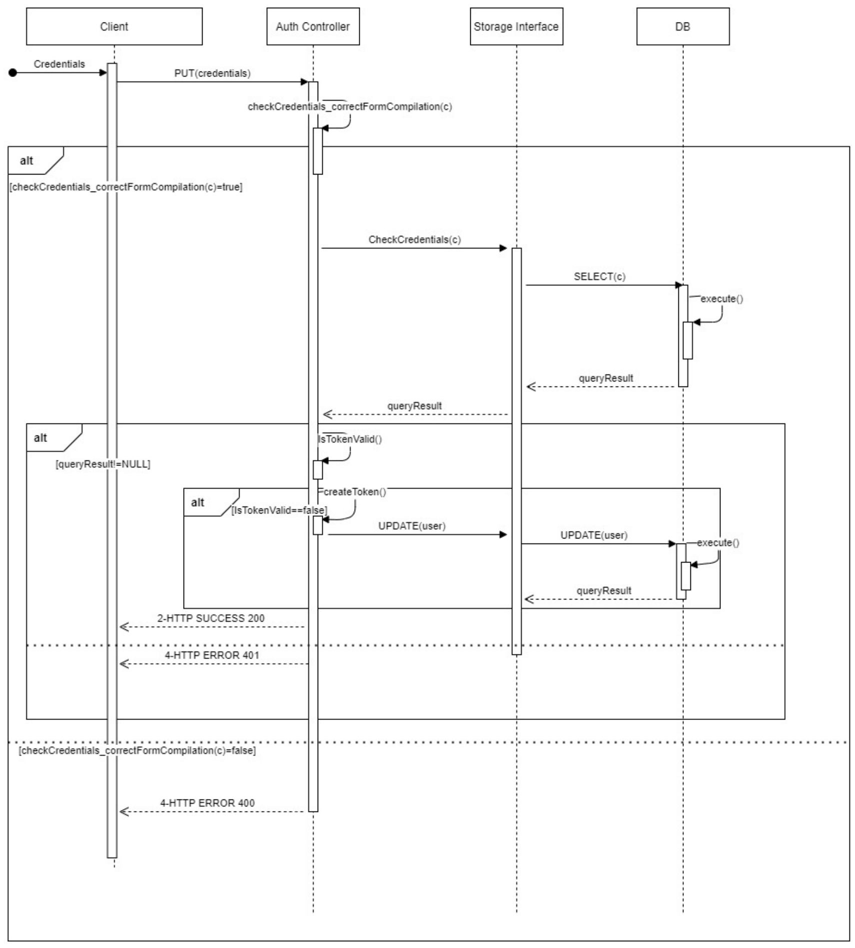


Figure 5: Login runtime sequence diagram

Get Map Runtime

When a client has booked a Visit they should request a map to avoid crowds and speed up their shopping. A Client requests the map through an HTTP request get. After having certified the credentials of the User, the Auth Controller sends the request to the Visit Controller, which sends it to the correct Queue Manager. At last the Store Controller sends the request to the Store Map Manager which, after having retrieved the necessary data from the database handles it. This component checks if the path has already been computed for the specified Visit. If not it computes one (and updates the data in the server through the Storage Interface). While having the path, the Store Map Manager computes the map and sends an HTTP response to the Client with accepted HTTP code and the requested map.

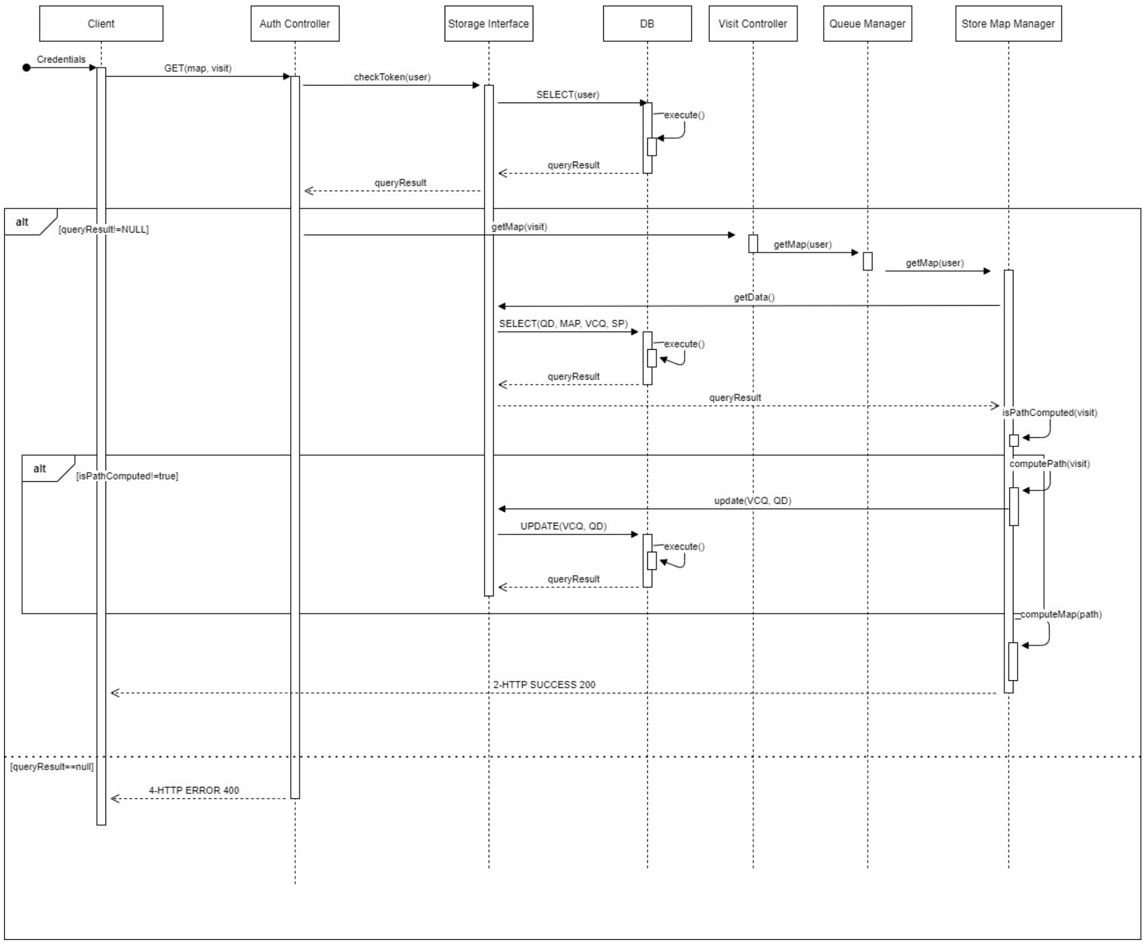


Figure 6: Get Map runtime sequence diagram

Access Store Runtime

When a Client accesses a Store, they scan their QR-code using the scanner provided by the Store. The Scanner communicates with the Queue Manager which checks if the User associated with the QR-code has a valid reservation for that time at that specific Store. If so, it updates the status of the reservation to see if the client is going into the Store or going outside the store. If they are going inside, the queue manager checks periodically if the user is late and it updates the queue status accordingly. If the client is going out, the queue manager updates the status of the reservation accordingly and the status of the queue if needed.

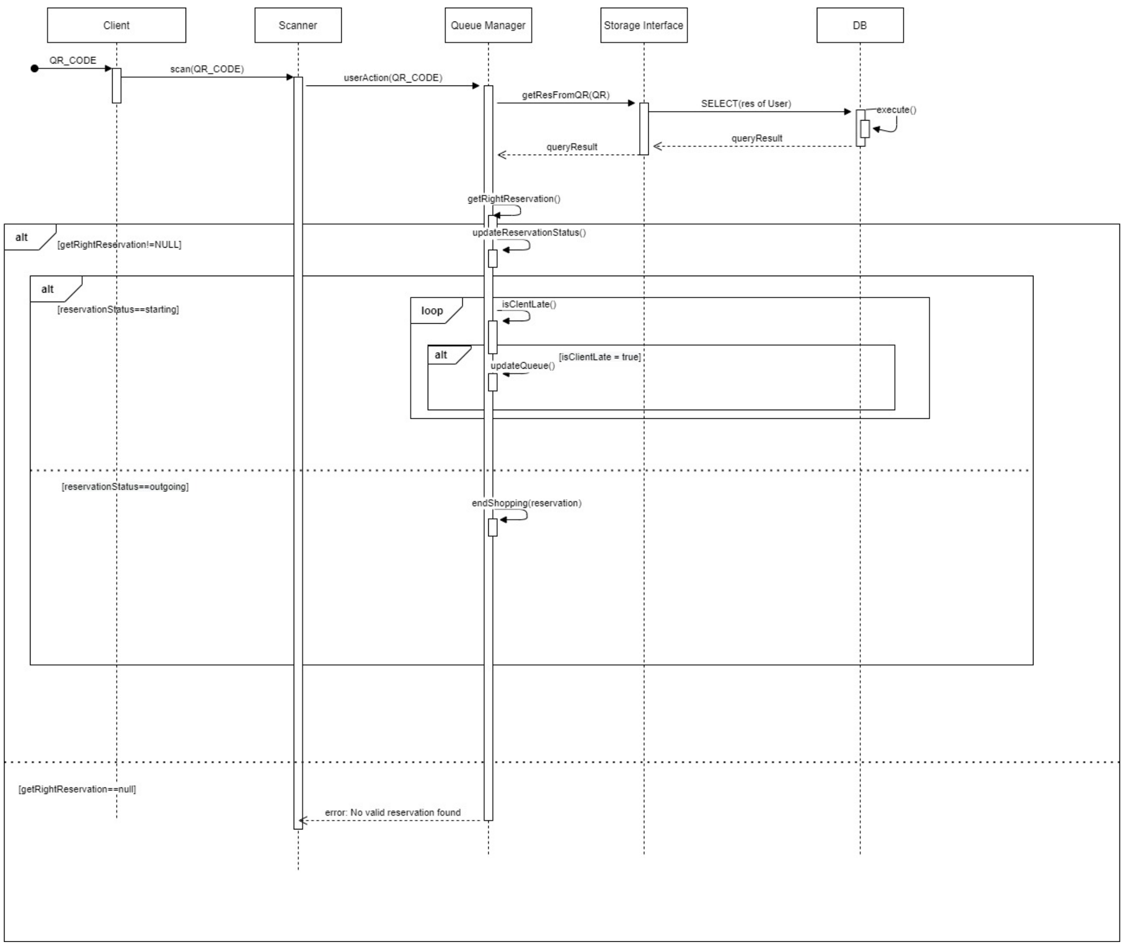


Figure 7: Access Store runtime sequence diagram

2.5 Component Interfaces

2.5.1 REST API

Authentication Controller

The Authentication Controller provides methods for login and registration. It provides the following methods:

Login

Endpoint	/login
Method	POST
Body	email: [string] password : [string]
Success Response	code: 200 OK Content: { userType : [string], authToken : [string] }

Error Response	code: 422 UNPROCESSABLE ENTRY Content: { error : ”Credentials are not correct” } code: 401 UNAUTHORIZED Content: { error : ”Wrong email or password” }
Notes	Allows User to login as Customer or Manager

Registration

Endpoint	/register
Method	POST
Body	email: [string] password : [string] userType : [integer]
Success Response	code: 200 OK Content: { message: ”Registration successful” }
Error Response	code: 422 UNPROCESSABLE ENTRY Content: { error : ”Registration Data are not correct” } code: 400 BAD REQUEST Content: { error : ”Registration Data are not correct” }
Notes	Allows User to register as Customer or Manager

Ticket Controller

The ticket controller provides methods for creating a ticket.

Get a Ticket

Endpoint	/tickets
Method	POST
Body	store_id: [string] shopping_duration : [string] means_of_transport : [integer] authToken : [string]

Success Response	code: 200 OK Content: { ticket_id }
Error Response	code: 422 UNPROCESSABLE ENTRY Content: { error : ”Data sent is not correct” } code: 401 UNAUTHORIZED Content: { error : ”wrong token” }
Notes	Allows User to get a ticket

Queue Manager

The store controller provides methods for getting information about reservations, deleting reservations, and data about the Store.

Delete a Ticket

Endpoint	/stores/:store_id/tickets
Method	DELETE
Body	ticket_id : [string] authToken : [string]
Success Response	code: 200 OK Content: { ticket_id deleted}
Error Response	code: 422 UNPROCESSABLE ENTRY Content: { error : ”Ticket id not correct” } code: 401 UNAUTHORIZED Content: { error : ”wrong token” }
Notes	Allows User to delete a ticket

Get information of a Ticket

Endpoint	/stores/:store_id/tickets
Method	GET
Body	ticket_id : [string] authToken : [string]

Success Response	code: 200 OK Content: { ticket_id deleted}
Error Response	code: 422 UNPROCESSABLE ENTRY Content: { error : ”Ticket id not correct” } code: 401 UNAUTHORIZED Content: { error : ”wrong token” }
Notes	Allows User to delete a ticket

Delete a Visit

Endpoint	/stores/:store_id/visits
Method	DELETE
Body	visit_id : [string] authToken : [string]
Success Response	code: 200 OK Content: { visit_id deleted}
Error Response	code: 422 UNPROCESSABLE ENTRY Content: { error : ”Visit id not correct” } code: 401 UNAUTHORIZED Content: { error : ”wrong token” }
Notes	Allows User to delete a Visit

Get information of a Visit

Endpoint	/stores/:store_id/visits
Method	GET
Body	visit_id : [string] authToken : [string]

Success Response	code: 200 OK Content: { visit_id deleted}
Error Response	code: 422 UNPROCESSABLE ENTRY Content: { error : ”Visit id not correct” } code: 401 UNAUTHORIZED Content: { error : ”wrong token” }
Notes	Allows User to delete a visit

Get store information

Endpoint	/stores/:store_id
Method	GET
Body	authToken : [string]
Success Response	code: 200 OK Content: { store1: id : [string], name : [string], address : [string], opening_hours : [string], categories_of_grocery : [ ] [string] time_slots: [ ] }
Error Response	code: 422 UNPROCESSABLE ENTRY Content: { error : ”Values are not corrected” } code: 401 UNAUTHORIZED Content: { error : ”Wrong email or password” }
Notes	Allows to retrieve information about a store

Get Map

Endpoint	/stores/:store_id/store_map_manager
Method	GET
Body	visit_id: [string] authToken : [string]

Success Response	code: 200 OK Content: { map: [list] }
Error Response	code: 401 UNAUTHORIZED Content: { error : ”wrong token” } code: 400 BAD REQUEST Content: { error : ”visit_id is not valid” }
Notes	Allows User to get the map for a Visit

Stores controller

The Stores controller provides methods to retrieve all the stores.

Get stores

Endpoint	/stores
Method	GET
Body	latitude : [float] longitude : [float] authToken : [string]
Success Response	code: 200 OK Content: { store1: id : [string], name : [string], address : [string] }
Error Response	code: 422 UNPROCESSABLE ENTRY Content: { error : ”Values are not corrected” } code: 401 UNAUTHORIZED Content: { error : ”Wrong email or password” }
Notes	Allows to retrieve all the stores near the User

Visit Controller

The Visit controller provides methods for creating a Visit.

Book a visit

Endpoint	/visits
Method	POST

Body	store_id: [string] shopping_duration : [string] day : [string] types_of_groceries : [ ] authToken : [string]
Success Response	code: 200 OK Content: { message: ”Registration successful” }
Error Response	code: 422 UNPROCESSABLE ENTRY Content: { error : ”Registration Data are not correct” } code: 401 UNAUTHORIZED Content: { error : ”wrong token” }
Notes	Allows User to get a ticket

2.6 Selected Architectural Styles and Patterns

Client-Server Architecture
The Client-Server Architecture is a computing model in which multiple components work in strictly defined roles to communicate. These roles can be divided into two main categories: the Server that hosts, delivers and manages most of the resources and services, and the Client that exploits those services. This architecture is also known as a networking computing model because all the requests and services are delivered over a network. When the Client needs a service, sends a resource or process request to the Server over the network connection, which is then processed and delivered to the Client. A Client can be connected to several Servers, each providing a different set of services, and a Server can manage several Clients simultaneously.

Although a network operating system is needed, the Client-Server Architecure is characterized by several advantages that make it a good choice. First of all, thanks to the presence of a network that integrates various components, the System can be defined as distributed. Therefore, it’s easy to replace, repair, upgrade and relocate a Server while Client remains unaffected. We can therefore say that this architecture is characterized by good scalability and maintanability. The other advantage is the security: indeed Servers control accesses and resources to ensure that only authorized Clients can access or manipulate data.

Three-Tiered Architecture
The Three-tiered Architecture is a specific type of Client-Server system. It is a type of

software architecture which is composed of three “tiers” or “layers” of logical computing:

- *Presentation Tier*: is the front end layer and consists of user interface which displays content and information useful to an end user. This is the top-most tier and the only one accessible from the Client.
- *Application Tier*: contains the functional business logic of the application, it controls the application’s core functionality.
- *Data Tier*: comprises of the database/data storage system. It includes the data persistence mechanisms and the data access layer. In practice it is the layer that exposes the data.

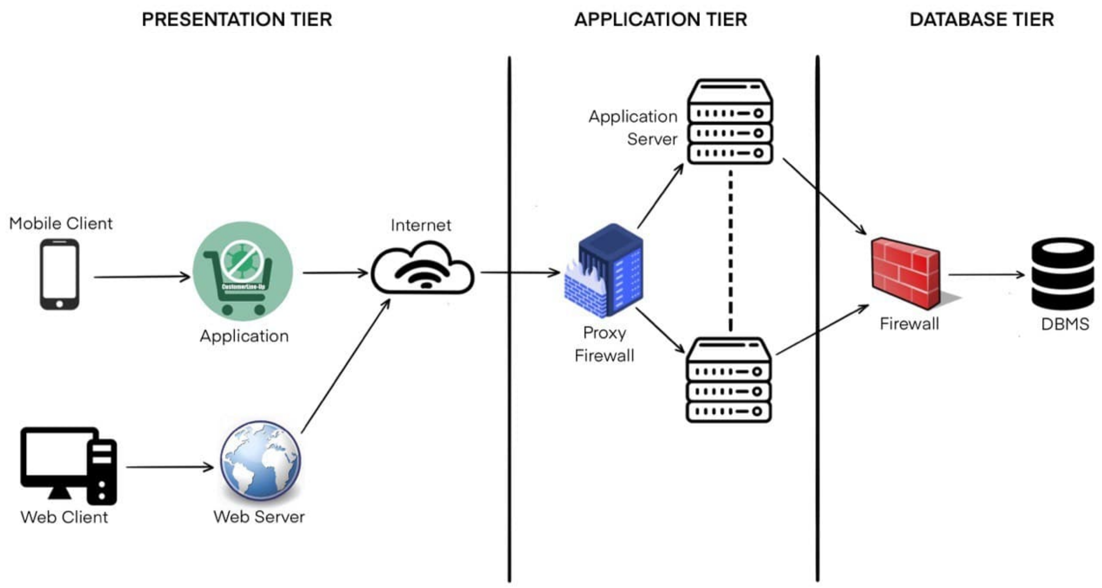


Figure 8: Three Tier Architecture schema.

The benefits of using a Three-Tier Architecture are many, including speed of development, scalability, performance and availability. Indees, modularizing different tiers of an application, gives the possibility to cìthe development team to develop the product with greater speed, because a specific layer can be upgraded with minimal impact on the other layers. In this way separate teams of the development team can focus on different components. Another other big advantage as we said is the scalability. By separating different layers, you can scale and load balance each independently, improving overall performance with minimal resources.

Thin Client
A Thin client is a computer that serves to remote into a server, which handles all the computational load. Since Thin Clients hand off the heavy lifting to a Server, they don’t need to have large memory or powerful computing capabilities to run the application.

Being CLup an application for mobile phones, characterized by low computational power and small memories, thanks to the choice of adopting Thin Client, an internet connection is sufficient to use the application itself. and this is because all the computation is moved to the Server side, also allowing a simplification of the font end implementation.

Reverse Proxy Design Pattern
a reverse proxy is a type of proxy server that retrieves resources on behalf of a client from one or more servers. Typically it sits behind a firewall and directs client requests to the appropriate Server. A reverse proxy provides an additional level of abstraction and control to ensure the smooth flow of network traffic between clients and servers. The avantages on using a Reverse Proxy are, first of all load balancing. Indeed a Reverse Proxy distributes Client's requests across a group of servers in a manner that maximizes speed and capacity utilization while ensuring no one server is overloaded, which can degrade performance. Moreover it ensure security and anonymity, in fact a Reverse Proxy protects identities and acts as an additional defense against security attacks.

REST: REpresentational State Transfer
REST, or REpresentational State Transfer, is an architectural style for providing standards between computer systems on the web, making it easier for systems to communicate with each other. It is characterized by two main features: statelessness and the separation of concerns between Client and Server. In the REST architectural style, the implementation of the client and the implementation of the server can be done independently, that means that there can be done changes on one side without affecting the other side. As long as each side knows what format of messages to send to the other, they can be kept modular and separate. This improve the flexibility, the scalability and additionally the separation allows each component the ability to evolve independently. Systems that follow the REST paradigm are stateless, meaning that the Server does not need to know anything about what state the Client is in and vice versa. In this way, both the server and the client can understand any message received, even without seeing previous messages. REST requires that a Client makes a HTTP request to the Server in order to retrieve or modify data on the server.

MVC Design Pattern
Model–view–controller or MVC is a software design pattern commonly used for developing software systems. It specifies that the program logic of an application consist of three interconnected elements: model, view and controller. The *model* directly manages data and rules of the application, it contains no logic describing how to present the data to a user. The *view* presents the model's data to the user, it knows how to access the model's data, but it does not know what this data means or what the user can do to manipulate it. And finally the *controller* that stays between the view and the model, it listens to events and executes the appropriate reaction to these events.

Usually this approach supports the choice of the Thin Client, that places almost the entire model, view and controller logic on the Server. MVC allows full encapsulation of objects. Furthemore, MVC provides decopuling of its components, which means that multiple developers can work simultaneously on the model, controller and views.

Relational DBMS
A relational database is a digital database based on the relational model of data. This model organizes data into one or more tables of columns and rows, with a unique key identifying each row. A relational database organizes data into tables in a way

for which you can retrieve an entirely new table from data in one or more tables with a single query. Another benefit of the relational database approach is that this type of databases support SQL, which is an easy human-readable language that enables CRUD operations for the storage. All of this make Relational databases a good choice for applications that involve several transactions at a time. Relational databases allow you to change the schema or how you model data on the fly, and it also eliminate data redundancy. The choice is also reinforced by the fact that data are structured.

2.7 Other Design Decisions

2.7.1 The Queue Scaling Process

One of the focal points of CLup is the creation of a good Queuing Mechanism for Consumers in order to enter the various Stores. But what exactly is meant by 'good'? First of all, good means that it takes into consideration that each Store will have its own personal queue, we generally want the queue and the entrances to be made in such a way as to allow simultaneous entry only to a number of Consumers equal to Capacity of each Store. It is also very important to keep in mind that one of the intentions of the application is to give a reasonable and as correct as possible departure time to all those Customers who have booked a Virtual Ticket remotely and must leave from the place in which they are in time to get to the Store for their turn. Other factors to take into consideration for a good construction of our queue are:

- All those who have a Paper Ticket will not be able to have an update on their time of entry to the Store.
- Consumers who have booked a Visit must have priority of entry over those who have booked a Ticket.
- Consumers who have booked a Paper Ticket have higher priority over those who have booked a Virtual Ticket.
- People may take a different amount of time from the specified one to get out and the queue will have to react accordingly.
- Some Consumer may not show up or may be late, and the queue will have to react accordingly.

Taking these factors into consideration, the following Queuing Mechanism was created. It is recommended that the following mechanism will be used when implementing the application by the development team.

The queue is divided into Temporal Quantums. The beginning of the queue corresponds to the opening time of the Store, the end to its closing time. The time between the opening and closing hours is then divided into 5 minutes each. Each Temporal Quantum consists of a vector as long as the capacity of the Store. Whenever a certain Consumer requests a Ticket or a Visit, the Temporal Quantums corresponding to its residence time are filled with a space, reducing the space available for other Consumers. In this way, entry will not be allowed to a number of people exceeding the Capacity available at any single time the Store is open. All the amounts of time of shopping for a single Consumer must be filled in so that they are sequential to each other (we obviously do not want a Consumer to enter and exit the Store several times to complete their shopping). Consequently, when a request for space is made, a search

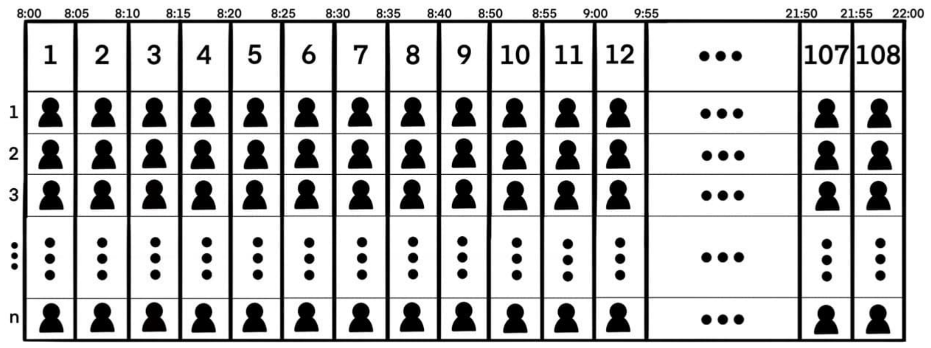


Figure 9: Example of a queue structure for one Store. The Store taken into consideration has as opening and closing hours 8:00-22:00. Consequently there are 108 Temporal Quantums of 5-minutes. Each Temporal Quantum can hold up to *n* people, with *n* equal to the store capacity. In the example shown, all the Temporal Quantum spaces have been occupied by a Consumer. No distinction was made between the various Consumers in this example.

must be made in the queue to check where the first sequential set of spaces, large enough to satisfy the request of the Consumer, is located. Therefore, such a mechanism does not focus on the temporal precedence of the various requests but on the consistency of the satisfaction of the request itself. In fact, it may happen that a Consumer at time t_1 requests a space of 3 Temporal Quantum which will be booked in the queue temporally subsequent to the space of 2 Temporal Quantum booked by another Consumer at time t_2, and this is evidently because a space of 3 Temporal Quantums was not available before, but one of 2 yes.

Until now, as long as Consumers come in and out at the appointed time, everything works fine. What does it mean to fill a series of Temporal Quantums? It means that at the start time of a certain Temporal Quantum the Store scanner expects to scan the QR Codes corresponding to the Consumers who have to enter in that Temporal Quantum. At the end time of the Temporal Quantum instead the Scanner expects to scan the QR Codes of all the Consumers who at that moment should have concluded their shopping as indicated. However, it may happen that a Consumer takes longer than it should to complete their shopping, or less. Let's consider the following cases.

In the case that a Consumer takes less than the pre-established time to exit the Store, basically no movement within the queue takes place. Their unused Temporal Quantums are simply freed up. In order for all Costumers with a Virtual Ticket to be provided with a good departure time, we do not want it to be possible for their filled Temporal Quantums to be anticipated. If so, it could happen that they are brought forward so much that they will no longer be able to leave on time from where they are to arrive on time for their turn. What happens is therefore simply a freeing of places within the unused Temporal Quantum, so that any other Consumers with new Tickets can occupy those spaces. In this way also the position of the Paper Tickets remains unchanged, and this is important as the entry time of the Paper Tickets cannot be updated because it is printed on a piece of Paper. In the event that a Consumer does not enter on time, that is in the event that his QR code is not scanned by the Scanner within the time included in his first Temporal Quantum filled, the consumer is considered late, and all his / her Temporal Quantums are released.

In the case that a Consumer takes too long to exit the Store, that is when the System realizes that its Qr Code has not been scanned out within the pre-established time, it is necessary to scale the other Consumers in subsequent Temporal Quantums, so that he is given more time to finish the shopping and the limit of people inside the shop is not exceeded, that is the **Queue Scaling Process**. We will show the solution to the problem through the following example.

Let's consider a Store with capacity 3 and opening hours 8:00, for which the following requests were made:

- Consumer 2 has booked a 10 minute Visit between 8:05 and 8:15.
- Before the opening of the Store Consumer 1 requires a Virutal Ticket with a shopping time of 10 minutes (corresponding to two Temporal Quantums).
- Before the opening of the Store Consumer 3 requires a Virtual Ticket with a shopping time of 20 minutes.
- At 8:05 am Consumer 4 requests a Paper Ticket with a shopping time of 15 minutes.
- At 8:05 am Consumer 4 requests a Paper Ticket with a shopping time of 15 minutes.
- At 8:06 am Consumer 5 requests a Virtual Ticket with a shopping time of 15 minutes.
- At 8:07 am Consumer 6 requests a Virtual Ticket with a shopping time of 15 minutes.
- At 8:15 am Consumer 7 requests a Virtual Ticket with a shopping time of 10 minutes.
- At 8:16 am Consumer 8 requests a Virtual Ticket with a shopping time of 10 minutes.
- At 8:17 am Consumer 9 requests a Virtual Ticket with a shopping time of 10 minutes.
- At 8:20 am Consumer 10 requests a Virtual Ticket with a shopping time of 10 minutes.
- and so on..

Let's consider the day time for which we arrive at the Temporal Quantum 4. At the end of this Temporal Quantum Consumer 3 (which we will call C_3) should be out. Instead the Scanner did not receive any scan of the QR code of C_3. This means that C_3 hasn't come out yet. What happens then inside the queue? C_3 didn't come out, so it needs more space. The queue will then give him a new Temporal Quantum. To do this it is necessary to check if there is an empty space at Temporal Quantum 5. If so, C_3 will take up that space. If this is not the case, it is necessary to choose a Consumer to be scaled. The rules for choosing the Consumer to scale, taking into consideration the previous considerations, are:

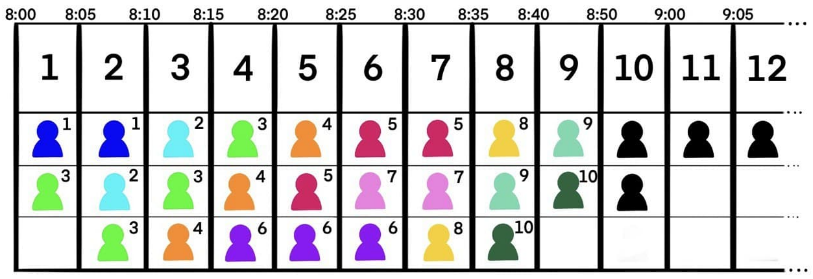


Figure 10: Queue Representation for the example described in this chapter. Each Consumer is characterized by its color and number in each space of the queue reserved for him.

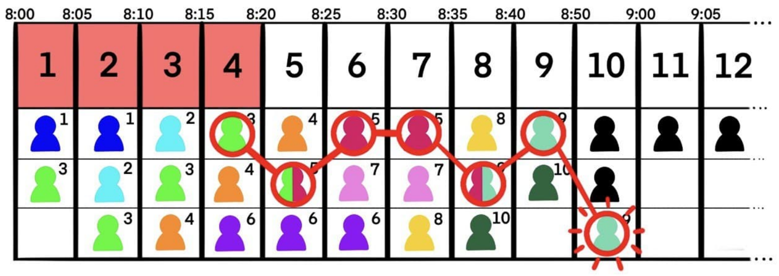


Figure 11: Representation of the queue for the example described in this chapter, specifically in the Temporal Quantum 4, highlighting the displacements caused by the Queue Scaling Process. A two-color character represents a Consumer passage that occupies that space. That is, the space that was previously occupied by the Consumer represented by the color on the right will now be occupied by the Consumer represented by the color on the left.

- Consumers are analyzed starting from the last reserved.
- A Consumer who has already entered cannot be chosen.
- We are looking for a Consumer with a Virtual Ticket to be scaled.
- If not, we are looking for a Consumer with a Paper Ticket to be scaled up.
- In general, a Cunsumer with a Visit must not be scaled. If there is no Consumer with a Virtual Ticket or Paper Ticket to be scaled, a Visit will be scaled (this hypothesis is very remote).

Taking these rules into consideration, let's look for a Consumer at the Temporal Quantum 5 to scale to make space for C_3. C_6 has already entered the Store at quantum 4, so it cannot be scaled. C_5 hasn't entered yet and has reserved a Virtual Ticket, so it's perfect. C_5 will 'chase' its reserved spaces in the following Temporal Quantums until it reaches the last one, at Temporal Quantum 7, in which we will have to find a Consumer to scale down to the Temporal Quantum 8 in order to let C_5 earn its place again. C_10 has not entered yet but has requested a Paper Ticket so we discard him for now. C_9 hasn't entered yet and has requested a Virtual Ticket, so it's perfect. At this point, C_9 behaves like C_5 for subsequent Temporal Quantums. C_9 finally find an empty space at Temporal Quantum 10, ending the scaling. If at Temporal Quantum 5 C_3 had not yet left the Store, the scaling would repeat itself following the same rules, making C_7 scale, then C_10 and so on, until we find an empty space as before.

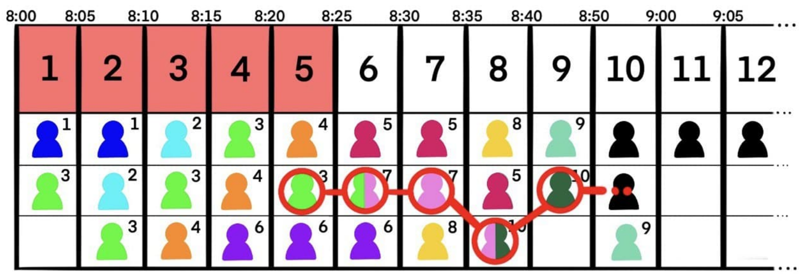


Figure 12: Representation of the queue for the example described in this chapter, specifically in the Temporal Quantum 5, highlighting the displacements caused by the Queue Scaling Process going on.

What happens if during the scaling we reach the end of the queue, i.e. there is no longer enough space to satisfy the Consumer request? In this case the Consumer request will be eliminated as it is no longer possible to satisfy it. The Consumer will then be notified of this (through the CLup application if he is a Customer, or through the Scanner if he has a Paper Ticket). In order to eliminate as few Reservations as possible, a final space of non-bookable spaces is left in the queue, corresponding to the last 30 minutes of opening hours of the Store. In this way the first scales will certainly have space.

The reason why a Visit is unlikely to be scaled is that the Manager can specify a percentage of Capacity of the Store that will be dedicated exclusively to reserve Tickets in every Temporal Quantum.

2.7.2 The Map Generation Process

In this section we give an example of how the map generation for the Visits could be generated. This map has two main objective: it wants to select a quick route for the Customer for him to go through all the departments that they have selected and it wants to minimize the possibility of overcrowding in the various departments. The creation of this map is delegated to the component Store Map Manager. This component is notified from the Queue Manager whenever a Visit is uploaded, deleted or modified (there has been a scaling in the queue). In this explanation we will call the entrance and exit points of the Store as Departments. These type of departments are special because cannot be selected from the Customer.

Moreover, we will assume that the Customer spends the same amount of Temporal Quantums in each *normal* department, and only one Temporal Quantum at the entrances and the exits (these two Temporal Quantums are also considered to shared with the other departments). We assume that the Customer will be quick to switch department, thus no Temporal Quantums will be assigned to check the movements between departments. This component will hold multiple data structure, which we will define as follow:

1. QD: Marix *#Temporal Quantums in the queue X #Departments in the Store*. In it there are written the number of people in a specific Departments for each Temporal Quantum in the Store (It is updated day by day, and there are only the Temporal Quantums of each single day). This matrix makes the algorithm much

more efficient in terms of run-time, but it is less efficient from the point of view of memory.

2. MAP: it is the map of the store, which is a graph with the departments (also with entrances and exits). The links between the various departments indicate the available paths between each department. Each arc has a weight equal to 1.

3. VCQ: it is a map (data structure), which maps the VisitID (primary key of each Visit), to its information. (The Map Manager will consider only the Visits in the current day, will delete the Visits of past days and will keep stored the future Visits). In particular there are the departments selected and a boolean IsMapped, to say if for this Visit has already been created a map.

4. SP: Matrix (obtained for example with the Floyd–Warshall algorithm) which contains the shortest distances between each pair of node of MAP.

This is how the Map Manager behaves after being notified by the Queue Manager:

1. Insert: if a Visit is inserted, it is saved in VCQ and IsMapped is set to false. If the Visit will be in the current day, the map is generated. The department in VCQ of this Visit will be updated: an entrance and an exit will be added and the all list of departments will be ordered to reflect the computed path. The boolean IsMapped will be set to true. QD is updated accordingly. (with +1s in the correct positions, which are given by the Temporal Quantums used by the Customer and the departments where they will be)

2. Update: if a Visit is updated (due to scaling), if IsMapped is false, then VCQ is simply updated and nothing else happens. If IsMapped is true, than QD is updated (with -1s), from the departments we delete the entrance and the exit and the map is re-computed using the new data (thus QD is updated again with +1).

3. Delete: if a Visit is deleted: if IsMapped is false, then the Visit is just deleted from VCQ. Otherwise, the Visit is deleted and QD is updated with -1.

Create Map

Let's define P[i, f]d as the average of the people in a department d from Temporal Quantum i to Temporal Quantum f (to compute it we just need the matrix QD).
Steps for the algorithm:

1. We compute the Temporal Quantum used in the Visit, using the information given by the time-slot. We will call the initial Temporal Quantum IQ and the final one FQ.

2. We add to the departments selected by the user an entrance point and an exit point. For the entrance: we check the value of QD at Temporal Quantum IQ for all possible entrances and we select the entrance with minimum value. For the exit: we check the the value of QD at Temporal Quantum FQ for all possible exits and we select the exit with minimum value.

3. We create a graph with only the departments selected by the Customer and the added entrance and exit as nodes. The graph is complete and each arc has a weight equal to the shortest path from one node to another in MAP (we can use the pre-computed SP to speed up the process)

4. We add an extra node which is connected to the entrance node and the exit node with arcs of weight very small (possibly 0) and we will call this node S. Now, the problem of finding the shortest path from route to entrance to exit can be reduced to a TSP problem.

5. Since TSP is an NP-hard problem, we use algorithms for its approximation (for example using MST, and NN greedy algorithm).

6. Having computed one or more sub-optimal path for solving the problem of the shortest path, we select one considering that we want to decrease the probability that a large amount of people will be in the same department in the same Temporal Quantums.

7. We can easily compute which Temporal Quantums will be assigned to each department (depending on their order in the path), since we assign to each selected department the same amount of Temporal Quantums.

8. We compute for each path the MAXP = maximum P[i, f]d (where i is the entrance Temporal Quantum in the department d and f is the exit Temporal Quantum of the department d).

9. Considering both MAXP and the length of each path we infer (during the implementation we will need to find a suitable formula) which is the best path, we remove the node S from it, and we save the result in VCQ, setting IsMapped to true and updating QD accordigly.

10. From this path we can easily compute the real path on MAP using, for example, Dijkstra algorithm to between each pair of ordered nodes.

Example of Map Creation
There has just been inserted a Visit which have:

- Departments: 6 and 8
- Day: Current day (e.g. 15/12/2020)
- TimeSlot: 8:00-8:40

The Store selected is represented in Figure [13], and it has 6 departments, plus the entrance department 0 and two exit departments 5 and 7. In Figure [14] there SP, computed from the Store Map Manager. Moreover in Figure [15] has been represented part of QD of the Store Map Manager.

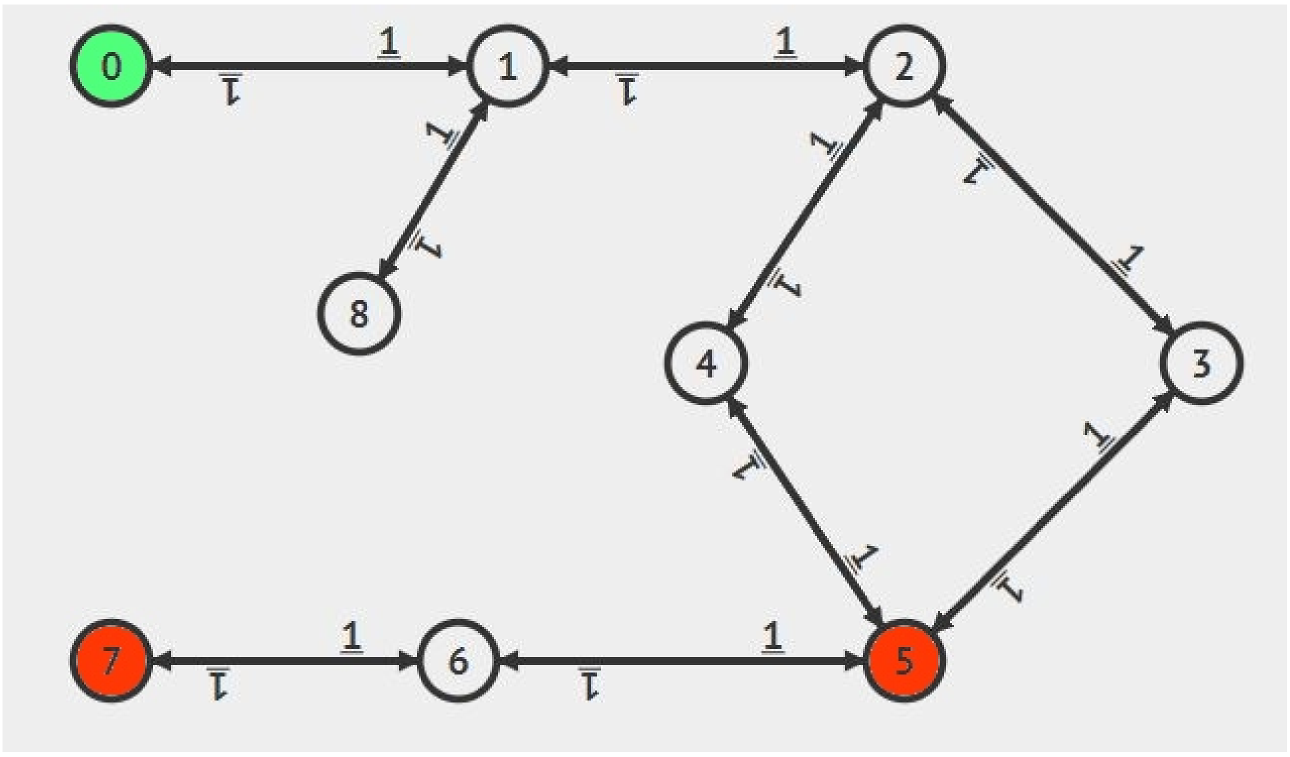
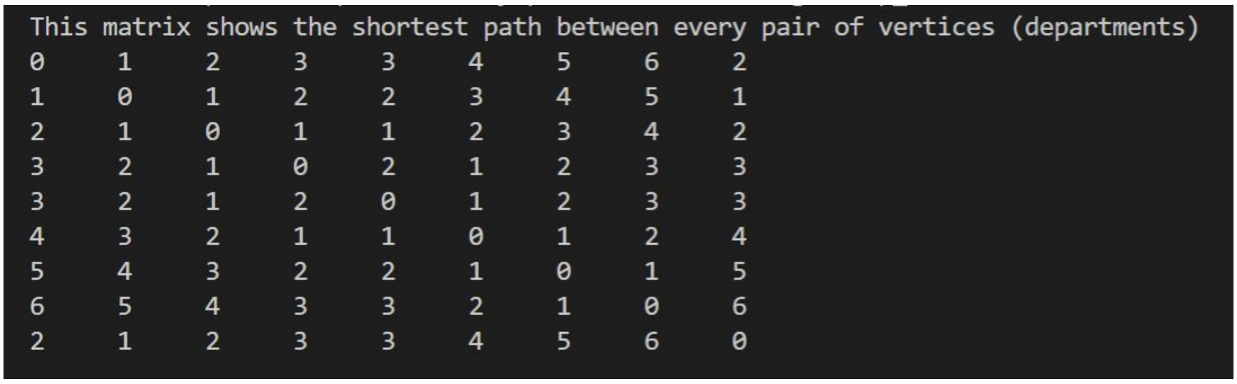
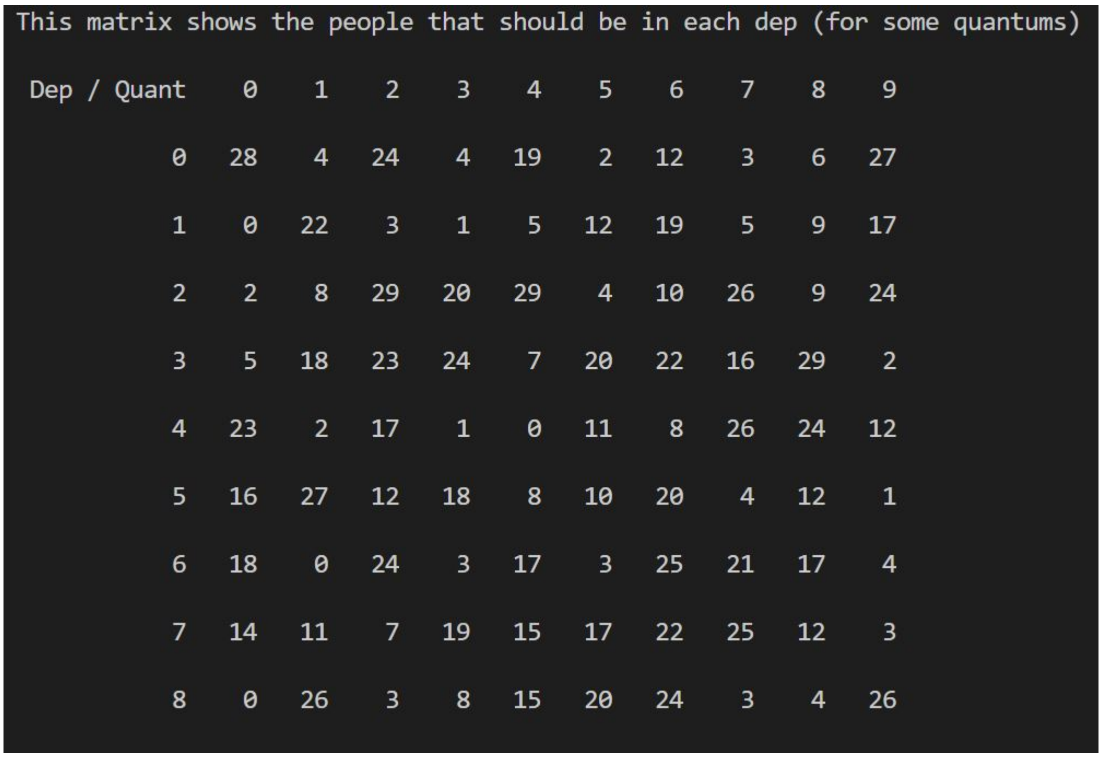


Figure 13: MAP: Map of the Store



```
This matrix shows the shortest path between every pair of vertices (departments)
0     1     2     3     3     4     5     6     2
1     0     1     2     2     3     4     5     1
2     1     0     1     1     2     3     4     2
3     2     1     0     2     1     2     3     3
3     2     1     2     0     1     2     3     3
4     3     2     1     1     0     1     2     4
5     4     3     2     2     1     0     1     5
6     5     4     3     3     2     1     0     6
2     1     2     3     3     4     5     6     0
```

Figure 14: SP: Shortest path between each pair of nodes in MAP



```
This matrix shows the people that should be in each dep (for some quantums)

 Dep / Quant    0    1    2    3    4    5    6    7    8    9

           0   28    4   24    4   19    2   12    3    6   27

           1    0   22    3    1    5   12   19    5    9   17

           2    2    8   29   20   29    4   10   26    9   24

           3    5   18   23   24    7   20   22   16   29    2

           4   23    2   17    1    0   11    8   26   24   12

           5   16   27   12   18    8   10   20    4   12    1

           6   18    0   24    3   17    3   25   21   17    4

           7   14   11    7   19   15   17   22   25   12    3

           8    0   26    3    8   15   20   24    3    4   26
```

Figure 15: QD: Estimated people in a specific department in a specific Temporal Quantum

Step for the algorithm:

1. Each Temporal Quantum has been defined to be 5 minutes-long, and that the time 8:00 is mapped to Temporal Quantum 0.Thus this Visit lasts 8 Temporal Quantums, from the 0^{th} to the $7th$. Since the Visit has only two departments, to each one of them will be assigned 8/2 = 4 Temporal Quantums.

2. We choose the entrance and exit dep. Since there is only one entrance department, 0 is selected. We have two exits. We select the exit which has the least amount of people at Temporal Quantum 7 (from figure [15]), and it is 5 with 4 people.

3. We create a graph with only the departments selected by the Customer and the added entrance and exit as nodes. The graph is complete and each arc has a weight equal to the shortest path from one node to another in MAP (we use the pre-computed SP (figure 14) to speed up the process). (figure [16])

4. We add an extra node which is connected to the entrance node and the exit node with arcs of weight very small (possibly 0) and we will call this node S. Now, the problem of finding the shortest path from route to entrance to exit can be reduced to a TSP problem. (figure [16])

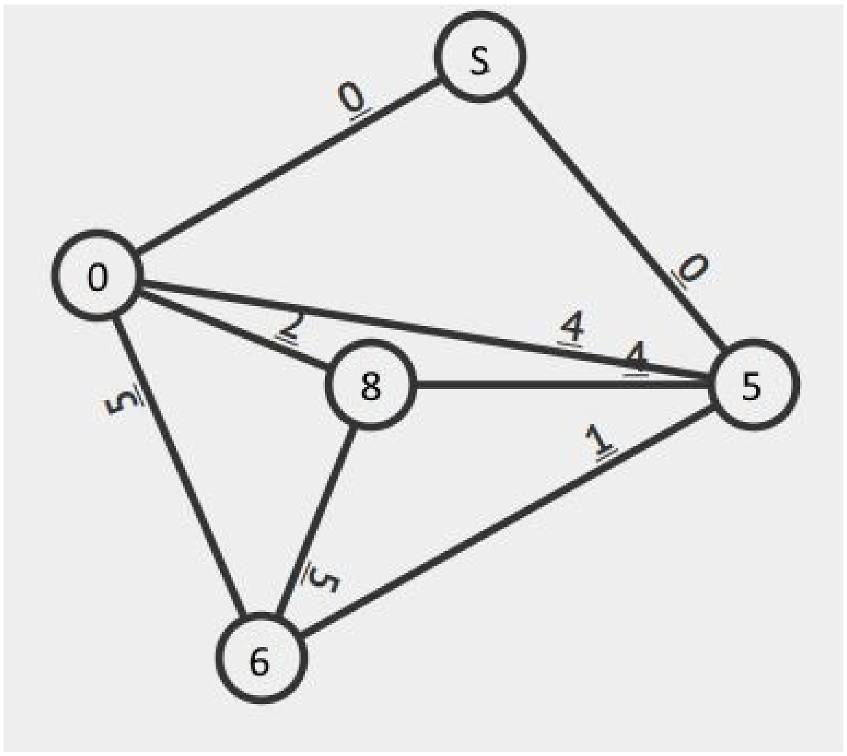


Figure 16: Preparing graph for TSP

5. In this example we just use the approximation of TSP using MST and we compute the MST using Prim's algorithm (Figure [17]).
   If we read the nodes of the constructed MST in preorder-walk adding 0 at the end, we get: 0 - S - 5 - 6 - 8 - 0. It's length is = 0 + 2 + 5 + 1 + 0 = 8. We can see that if we traverse the path backwards and we delete the first two nodes, we obtain the path: 0 - 8 - 6 - 5. Which is what we were looking for. (and it still has length 8)

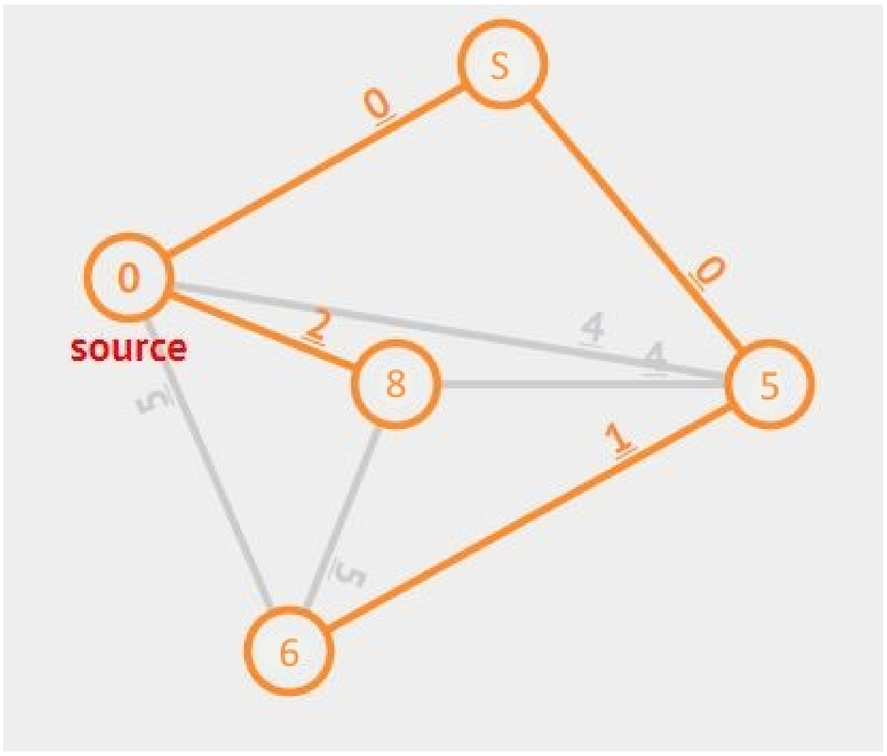


Figure 17: MST of graph in figure 16

6. Since in this example we compute a single path, we would not need to check P[i, f]d in each department, but to better understand the algorithm, we now compute MAXP anyway. Since the first real department to be visited is 8, to it will be assigned the Temporal Quantums from 0 to 3 included, and to department 5 will be assigned the Temporal Quantums from 4 to 7 included
$MAXP = \max \lfloor (0 + 26 + 3 + 8)/4 \rfloor, \lfloor (8 + 10 + 20 + 4)/4 \rfloor = \max 9, 10 = 10.$

7. Given the best path, we remove node S from it, and we save the result in VCQ, setting IsMapped to true and updating QD accordigly: (we add 1 to: (0, 0), (5, 7), (8, from 0 to 3 included), (6, from 4 to 7 included)).

8. From this path we can easily compute the real path on MAP using, for example, Dijkstra algorithm to between each pair of ordered nodes. (figure 18)

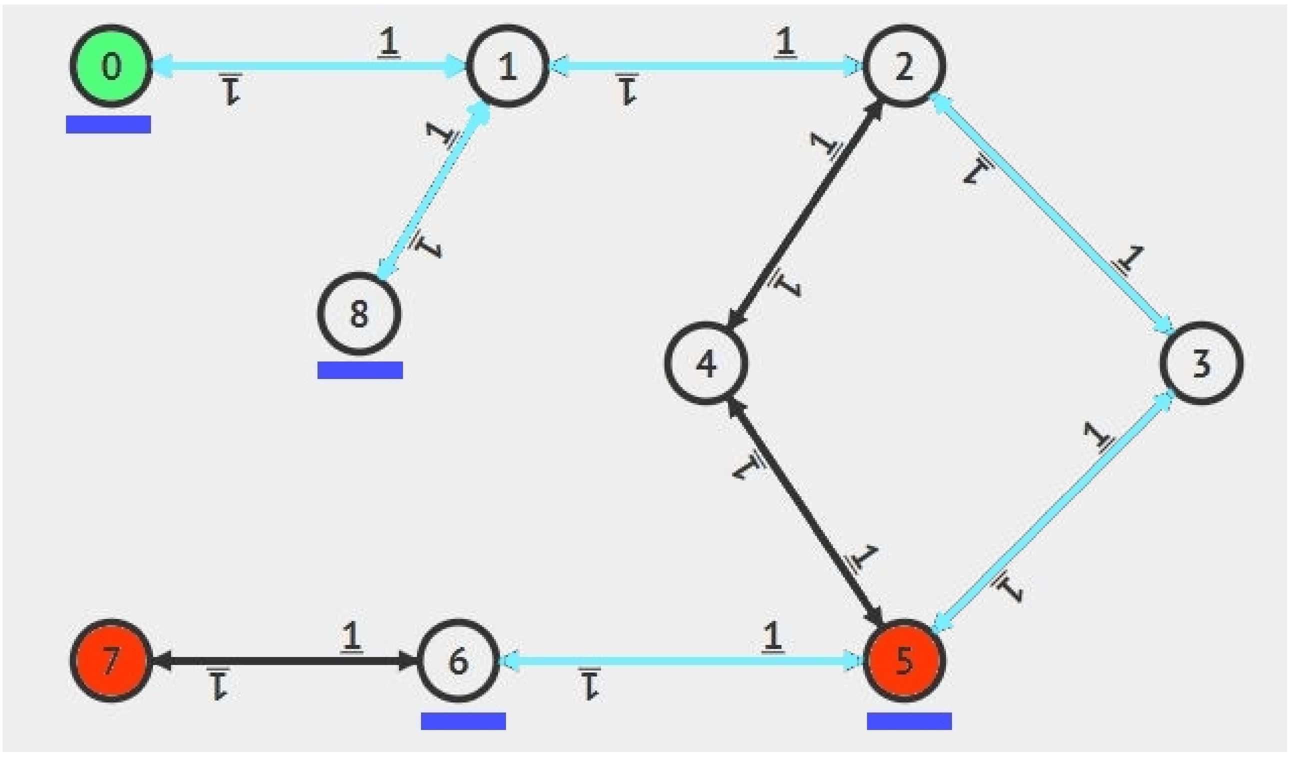


Figure 18: Created map for the Visit

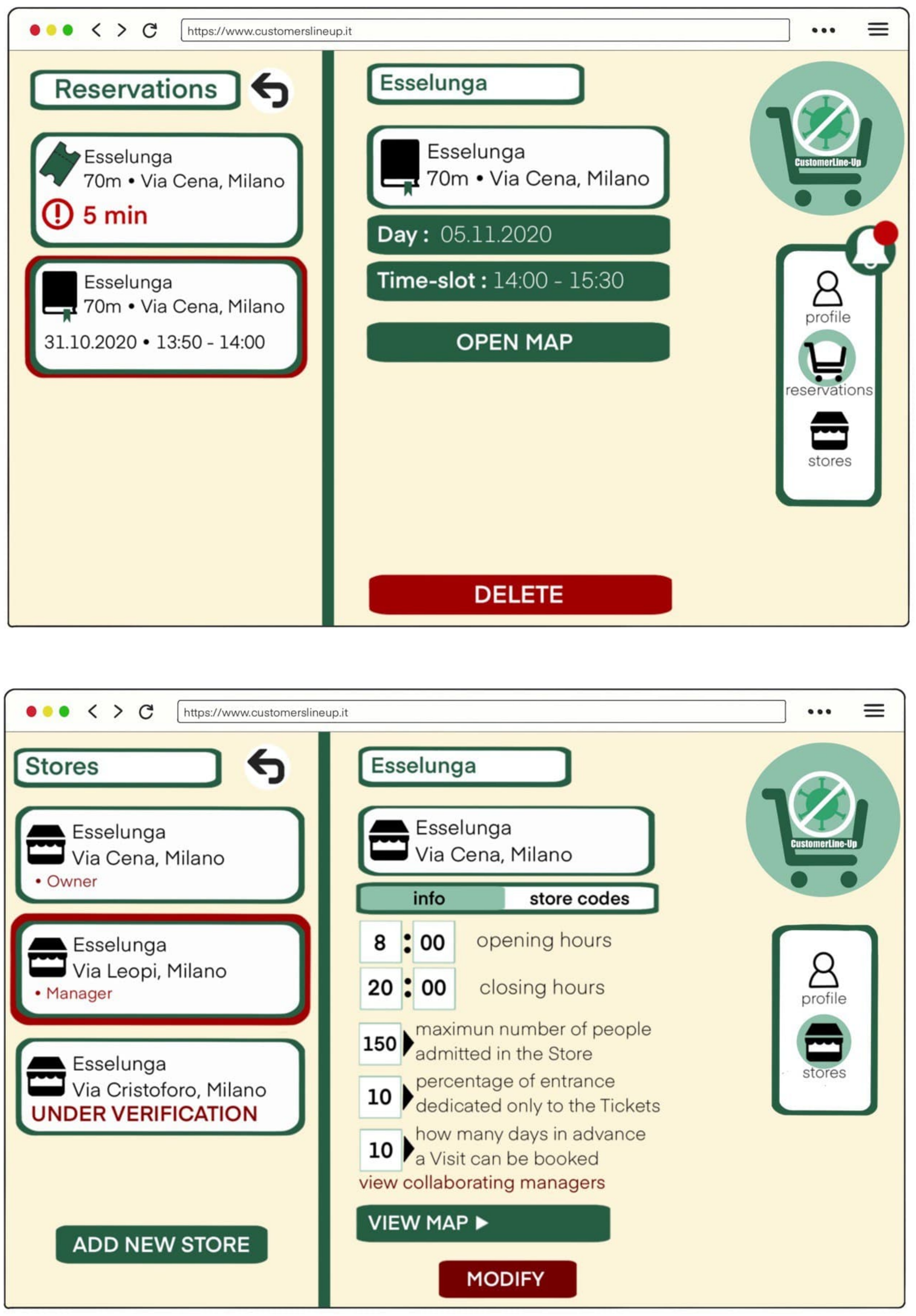
Final Considerations on the Map Creation

The approach used does not give as an optimal solution neither for the shortest path problem nor for the overcrowding one. It tries to give a sub-optimal problems. Since the overcrowding that we mostly want to avoid is the one at the entrances and at the exits (so that a minor number of queues is generated), both entrance and exit are computed looking only at the estimated amount of people that should use them at a given Temporal Quantum.

To achieve the optimal solution for the shortest path problem, we should have used a brute-foce algorithm with time complexity O(n!), which would constraint the possible number of departments that the user can select to 9-10. Since the system could be used by large Stores, we thought, that we should give a more efficient solution. It is though possible to apply different algorithms, basing the choice on the number of selected departments in the Visit (thus, it is possible to solve TSP chosing a different algorithm from the one in the example).

3 User Interface Design

3.1 Mockup


https://www.customerslineup.it

Reservations

Esselunga
70m • Via Cena, Milano
5 min

Esselunga
70m • Via Cena, Milano
31.10.2020 • 13:50 - 14:00

Esselunga

Esselunga
70m • Via Cena, Milano

Day : 05.11.2020

Time-slot : 14:00 - 15:30

OPEN MAP

DELETE

CustomerLine-Up

profile
reservations
stores

https://www.customerslineup.it

Stores

Esselunga
Via Cena, Milano
• Owner

Esselunga
Via Leopi, Milano
• Manager

Esselunga
Via Cristoforo, Milano
UNDER VERIFICATION

ADD NEW STORE

Esselunga

Esselunga
Via Cena, Milano

info | store codes

8 : 00 opening hours

20 : 00 closing hours

150 maximun number of people admitted in the Store

10 percentage of entrance dedicated only to the Tickets

10 how many days in advance a Visit can be booked

view collaborating managers

VIEW MAP

MODIFY

CustomerLine-Up

profile
stores

Figure 19: Web page graphics.

3.2 User Experience Diagram

To better define the most important Users action within the application, we reported in Figure 20 and in Figure 21 two User Experience diagrams (UX), one for the Customer and one for the Manager. User Experience diagrams are diagrams that display the complete path a User takes when using a product, it refers to the functions and respective interfaces of the application.

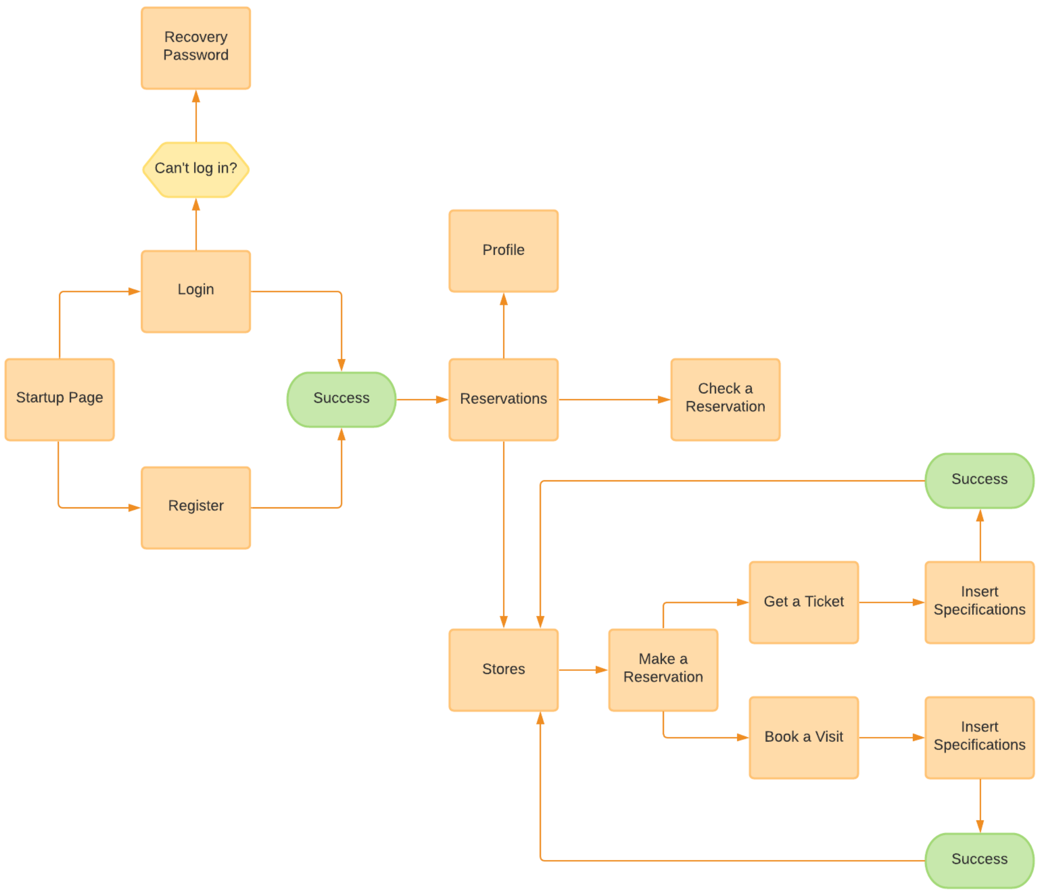


Figure 20: UX flow of a Customer User.

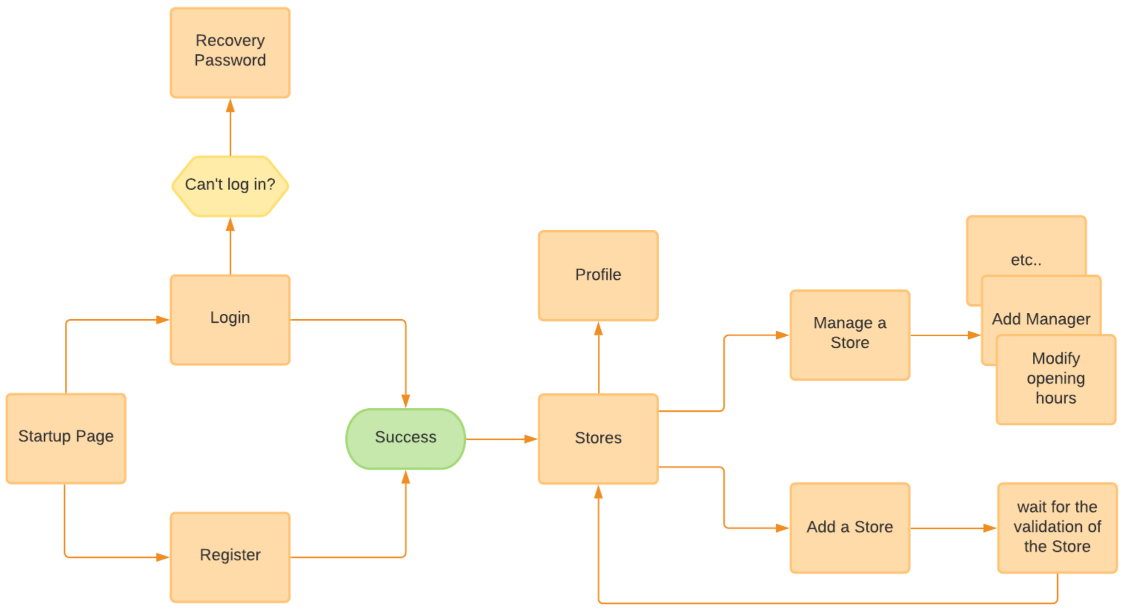


Figure 21: UX flow of a Manager User.

4 Requirements Traceability

Within this document, the design and architectural choices have been made in order to satisfy the requirements and the various specifications contained in the RASD. In particular, in this chapter we will show the link that the System components (that can be found in the DD) have with the requirements (that can be found in the RASD).

4.1 Functional Requirements

In the following table is shown which components are necessary to satisfy the requirements.

Auth Controller	R.1, R.2, R.3, R.4, R7
User Controller	R.5, R.6, R.8, R.17
Data Analytics Manager	R.14, R.19, R.31
Notification Manager	R.19, R.30
Ticket Interface	R.9
Ticker Controller	R.9, R.10, R.11, R.12
Visit Interface	R.13
Visit Controller	R.13, R.15, R.20
Storage Interface	R.1, R.4, R.5, R.6, R.8, R.9, R.10, R.13, R.22, R.23, R.24, R.26, R.27, R.28, R.21
Store Interface	R.22, R.23, R.24, R.26, R.27
Store Controller	R.22, R.23, R.24, R.26, R.27

Queue Manager	R.9, R.10, R.13, R.15, R.28, R.29
Store Map Manager	R.15
Mailing System Interface	R.3, R.6
Notification Service	R.19, R.30
Map Service	R.9, R.10, R.18
Totem	R.29
Scanner	R.28
Store Validity Interface	R.22, R.24
Stores Controller	R.9, R.13

Table 23: System's components and requirements mapping on goals.

In order for the table to be understood, we briefly report in the table below the reference Functional Requirements whose more precise and exhaustive description can be found in the RASD.

R.1	The User must be able to register or to log in if already registered.
R.2	Check if the User credentials are valid.
R.3	If the credentials are valid, the System sends a confirmation email, to let the User activate their account.
R.4	Allow to log in using personal credentials.
R.5	Allow to change credentials, if the new credentials are in the right format.
R.6	Allow to change password if it has been forgotten, through the personal email.
R.7	Provide to each Customer a unique QR code which will be used as identifier within the System.
R.8	Users must have the possibility to delete their accounts.

R.9	The Customer must be allowed to request a Ticket.
R.10	For Customers who are requesting a Ticket, the System should act to provide it.
R.11	Each Customer can request only one Ticket.
R.12	Every person on a Ticket can view the time in which they should enter the Store and the indicated duration of their shopping.
R.13	The Customer must be allowed to request a Visit.
R.14	For long time Customers, while booking a Visit or a Ticket, the System should infer, analyzing previous Reservations of that Customer, the expected duration of the Visit or Ticket.
R.15	The System should provide to the Customer a map of the Store.
R.16	Allow Customers to consult their pending requests.
R.17	The Customer must be allowed to delete a Visit or a Ticket from his/her pending requests.
R.18	Use third party services to enable the localization of Customers and Stores.
R.19	Notify Customers of available slots in a day-time range inferring information from their previous Visits.
R.20	Each Customer can book up to three Visits in a single day.
R.21	If a Reservation is expired, or it has served its purpose, or it has been deleted, it will be removed and will not be visible to the Customer anymore.
R.22	Allow Managers to create Stores, becoming the Owner of the Store.
R.23	Allow Owners to delete their Stores.
R.24	Verify the Store creation, acquiring specific credentials (for instance a Certified Email) that can be used to verify the validity of the Store and of the Manager.
R.25	Allow the Owner and Managers to edit the Store.
R.26	All the information about the Store can be updated by their Managers at anytime.
R.27	Allow Managers to upload a map of their Stores.
R.28	Provide a specific **Scanner App** to scan the QR codes of the Consumers and to interact with the Queuing Mechanism.
R.29	Provide a specific **Paper Tickets App** to generate paper tickets in the Store and to interact with the Queuing Mechanism.
R.30	Each Store has one and only one queue.
R.31	Suggestions are based on the habits of the Customers, inferred by the Data Analytics Manager.

4.2 Non-Functional Requirements

The Non-Functional Requirements of the application are defined in the Sections 3.3 and 3.5 of the RASD. Design choices like the Three-Tier Architecture acts to improve and respect the performances needed and in general to improve the software quality attributes.

5 Implementation, Integration and Test Plan

As previously illustrated, the System is composed by many components that can be divided into the following categories:

- **Frontend components:** Client application
- **Backend components:** User Controller, Auth Controller, Data Analytics Manager, Notification Manger, Ticket Manager, Ticket Interface, Ticket Controller, Visit Manager, Visit Interface, Visit Controller, Store Interface, Queue Manager, Store Map Manager, Stores Controller.
- **External components:** Mailing system, Notification service, Store validity, Map service and the DBMS.

In order to implement, integrate and test the System, a bottom-up strategy will be used. The System has been divided in smaller subsystems (which are defined later on in this document) to support this strategy. From the implementation and the testing of each subsystem it will be possible to integrate them to obtain a reliable, decoupled and modular System. Components of external subsystems don't need to be tested because they are considered reliable. The implementation and integration process will be performed in two phases:

1. Implementation and integration of different components of the same subsystem
2. Integration of different subsystems

The following table lists the main features offered to the customer, together with their importance and implementation difficulty, in order to better understand the decision about implementation, testing and integration taken.

Functionality	Importance for the customer	Difficult of implementation
Registration and Login	Low	Low
Get a ticket	High	High
Book a visit	High	High
Receive notifications	Medium	Medium
Shopping Path	Medium	High
Receive suggestions	Medium	Medium

Regarding the first point, components integration of the same subsystem will be made only for the backed components. In detail, components to be integrated and tested togheter are:

- User Controller and Storage Interface composing **User subsystem**
- Authentication Controller and Storage Interface composing the **Authentication subsystem**
- Ticket Interface, Ticket Controller, Notification Manager,Store Interface, Store Controller, Stores Controller, Storage Interface and Queue Manager composing **Ticket Subsystem**
- Data Analytics Manager, Storage Interface, Notification Manager composing **Recommender subsystem**

- Visit interface, Visit Controller, Storage Interface, Store Interface, Store Controller, Stores Controller, Queue Manager, Store Map Manager composing **Visit subsystem**

For the second point the only subsystem integration that will be performed is between Business Logic and Client subsystem. It's important that the verification and validation processes starts as soon as the development of the system begins in order to find errors as quickly as possible. As specified before, will be used and incremental approach for integration process in order to isolate bugs in single subsystems and don't propagate them. Each component of each Subsystem is tested using **Unit Testing**. In particular, one possible approach could be TDD. This, in combination with good tests, designed, allows to develop stable components. Good tests are measured in *How much they stress corner case and/or particular cases*. Moreover, since this value is not measurable, is possible to watch also to test coverage that should be $>$ 90%. Sometimes this value can be lower, due to the fact that some component, expecially the one from Model, are made up mostly by getter and setter.

Due to the fact that the frontend and the external subsystems are independent, the external services will be used in place of stubs to test the interfaces. At the end, when the integration system is completed, the whole system is tested: for this purpose a fundamental type of test is the performance test that is useful to identify bottlenecks affecting the response time, the utilization and the throughput. It also find out all the inefficiency of the system, from algorithms that could be optimized in the implementation, to network issues.

5.1 Sequence of Component Integration

The following diagrams describe the process of implementation, integration and testing. The arrow goes from the used component/subsystem to the component/subsystem that uses it.

Integration of the backend subsystem All the components are first implemented and unit tested. Once finished this phase, components are integrated and will be performed integration testing.

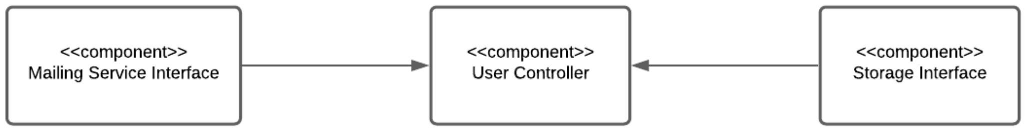

<<component>> Mailing Service Interface
<<component>> User Controller
<<component>> Storage Interface

Figure 22: User Subsystem

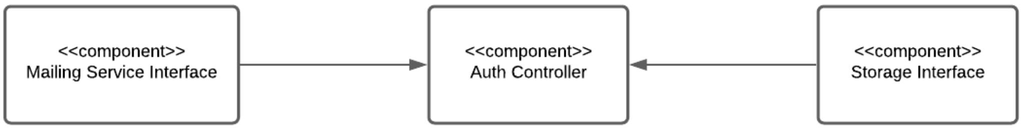


Figure 23: Auth Subsystem

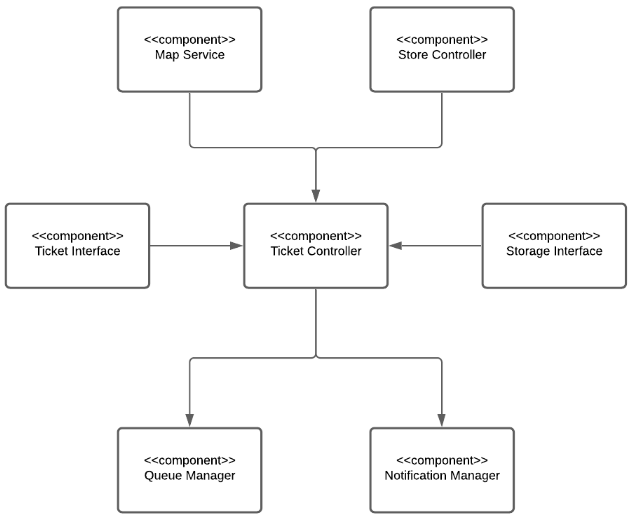


Figure 24: Ticket Subsystem

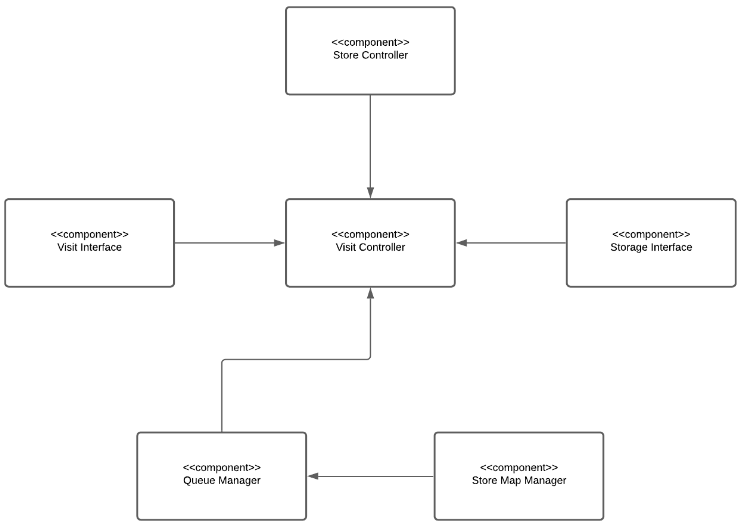


Figure 25: Visit Subsystem

Integration of the backend and frontend subsystem
Once all the components of the backend are implemented and well tested , the frontend will be integrated and tested with the backend. In particular the subsystem to be integrated with the Client subsystem are the User subsystem, Auth subsystem, Ticket subsystem and Visit subsystem.

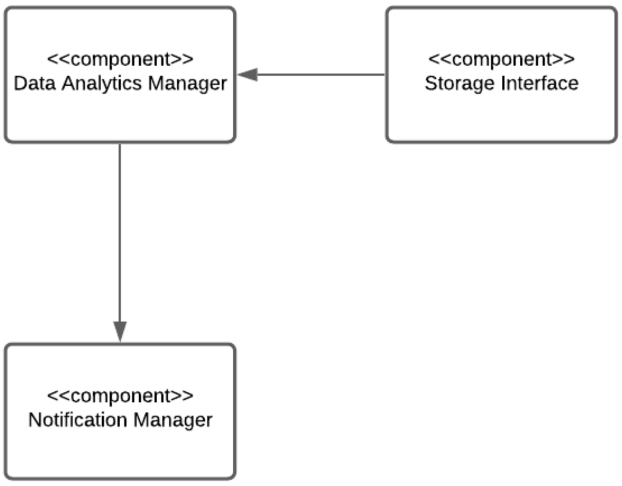


Figure 26: Recommender Subsystem

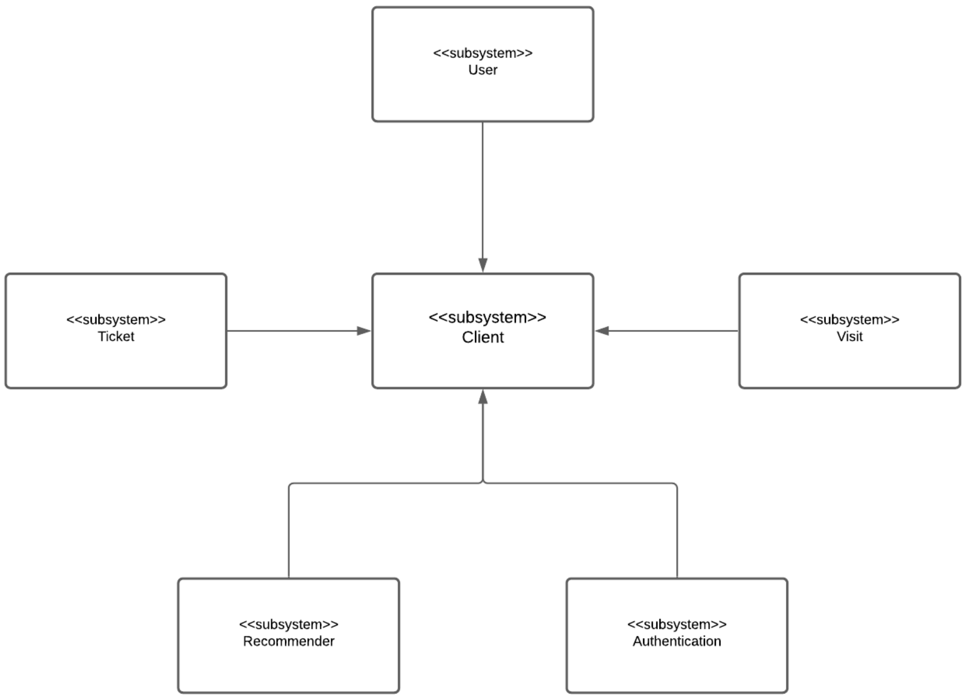


Figure 27: Integration Backend and Frontend

System Integration At the very end, all the components of the frontend, backend and external service are integrated and tested. Once again, this is done in order to check the behavior of the components when they work together.

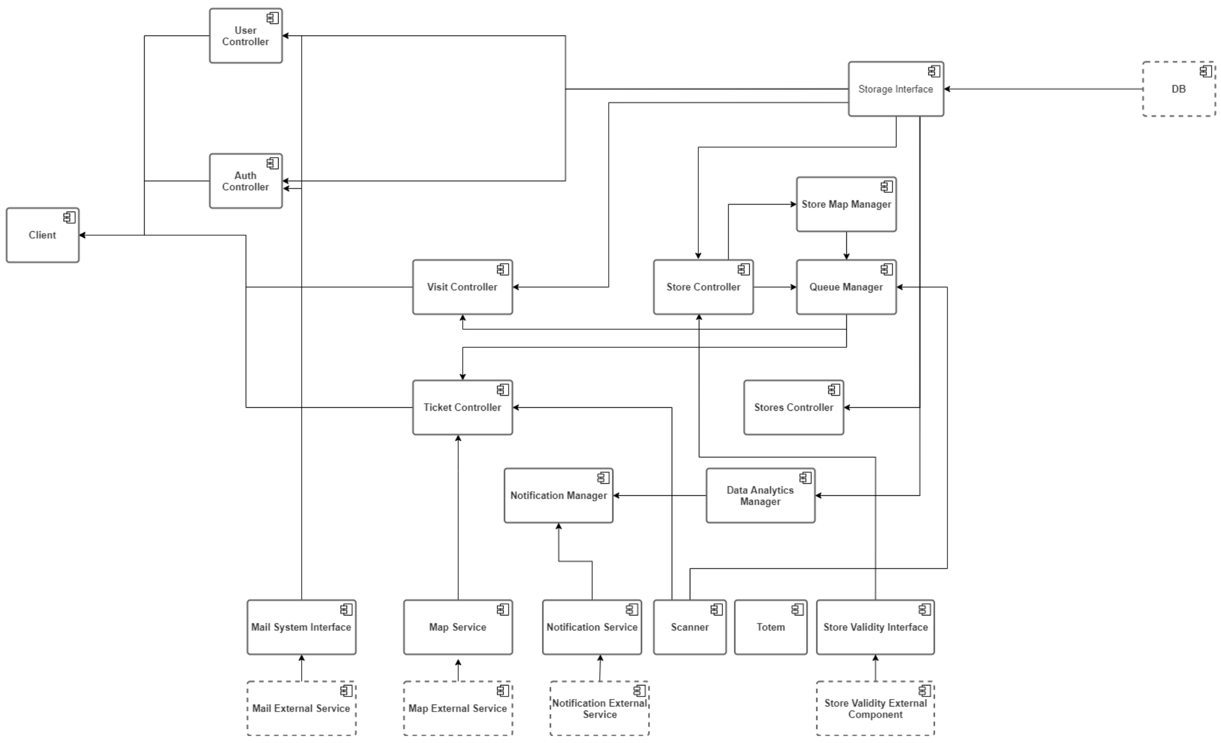


Figure 28: System Integration

6 Effort Spent

Lorenzo:

	Description	Hours
1	minor revisions	0.5
2	map algorithm	5
3	map algorithm example	4
4	map algorithm revision of some mistakes	0.5
5	revisions on the document	0.5
6	runtime view registration and login	2.5
7	map creation runtime view	2
8	revision, refactor, update REST API	1
9	revision, refactor, update traceability matrix	1.2
10	runtime view access store	1.5
11	Final call	3
12	Other calls	8
13	**TOTAL**	29.7

Yasmin:

	Description	Hours
1	Introduction and Purpose	0.7
2	High level components overview	0.7
3	Component View	3
4	Queue Scaling Process	5
5	Description of the selected architectural patterns	2.5
6	User interfaces	1
7	User Experience Diagrams	0.6
8	First approach Functional Requirements	2.5
9	Finished Functional Requirements	1.8
10	Revision	0.5
11	Modified UX	0.4
12	More Revision	0.4
13	Start of Component View Diagram	1
14	Finish of Component View Diagram	1.5
15	Func requirements adjustments and other adjustments	1.2
16	Final call	3
17	Other calls	8
	TOTAL	33.8

Giovanni:

	Description	Hours
1	Document revision and deployment diagram	1
2	Deployment View, start Component Interface	3
3	Implementation, Integration and Test Plan and some revision	7
4	System Integration diagram	0.4
5	Review first chapter and some additions	1.4
6	Revision of some chapters and edit of some diagrams	2
7	Final call	2
8	Other calls	8
	TOTAL	24.8