



POLITECNICO MILANO 1863

Design document

<i>Davide Li Calsi</i>	<i>10613807</i>
<i>Andrea Alberto Marchesi</i>	<i>10577090</i>
<i>Marco Petri</i>	<i>10569751</i>

Professor: Matteo Giovanni Rossi
Academic year: 2020/2021

Contacts:

Davide Li Calsi	davide.li@mail.polimi.it
Andrea Alberto Marchesi	andreaalberto.marchesi@mail.polimi.it
Marco Petri	marco.petri@mail.polimi.it

Table of contents

PART 1: INTRODUCTION	1
1.1 Purpose	1
1.2 Scope	1
1.3 Definitions, acronyms and abbreviations	1
1.4 Revision history	2
1.5 Document structure	2
PART 2: OVERALL DESCRIPTION	4
2.1 Overview	4
2.2 Component view	6
2.2.1 Front-end services component	9
2.2.2 Store service	11
2.2.3 Booking service description	12
2.2.4 Queue service	13
2.2.5 User service	14
2.3 Deployment view	15
2.4 Runtime view	17
2.4.1 User login	17
2.4.2 Customer registration	17
2.4.3 Ticket check-in	18
2.4.4 Ticket check-out	18
2.4.5 Physical queue	19
2.4.6 Enter online queue and notification	20
2.4.7 Queue exit	21
2.4.8 Visit booking	22
2.4.9 Booking deletion	23
2.4.10 Store manager's parameters' modification	23
2.5 Component interfaces	24
2.5.1 Front-end services	24
2.5.2 Back-end services	25
2.5.3 Data access objects	27
2.6 Selected architectural styles and patterns	28
PART 3: USER INTERFACE DESIGN	29
3.1 Web app	29
3.1.1 Login	29
3.1.2 Checkpoint controller	30
3.1.2.1 Checkpoint controller at entrance	30
3.1.2.2 Checkpoint controller at exit	30
3.1.1 Customer	32
3.1.1.1 Registration	32

3.1.1.2 Home	32
3.1.1.3 Enter a queue	33
3.1.1.4 Exit a queue	33
3.1.1.5 Book a visit	34
3.1.1.5.1 Select store	34
3.1.1.5.2 Select date and time	35
3.1.1.5.3 Select items and categories	36
3.1.1.5.4 Select group size	36
3.1.1.6 Delete a visit	36
3.1.2 Store manager	37
3.1.2.1 Setting of parameters	37
3.2 Mobile app	38
3.2.1 Login	38
3.2.2 Checkpoint controller	38
3.2.2.1 Checkpoint controller at entrance	38
3.2.2.2 Checkpoint controller at exit	39
3.2.3 Customer	39
3.2.3.1 Registration	39
3.2.3.2 Home	40
3.2.3.3 Enter a queue	40
3.2.3.4 Exit a queue	40
3.2.3.5 Book a visit	41
3.2.3.5.1 Select store	41
3.2.3.5.2 Select date and time	42
3.2.3.5.3 Select items and categories	42
3.2.3.5.4 Select group size	43
3.2.3.6 Delete a visit	43
3.2.4 Store manager	44
3.2.4.1 Setting of parameters	44
PART 4: REQUIREMENTS TRACEABILITY	45
PART 5: IMPLEMENTATION, INTEGRATION AND TEST PLAN	50
5.1 Implementation plan	50
5.2 Integration plan	53
5.3 Testing plan	57
PART 6: EFFORT SPENT	61
PART 7: REFERENCES	62

1

Introduction

1.1 Purpose

The purpose of this document is the detailed description of the design of the lining up mechanism described in the RASD for that system. This document comprehends the definition of high-level architectures used for the system and the models that compose the system. The document describes interfaces and the traceability of the requirements using the traceability matrix to track how requirements are fulfilled by the DD. Then there is the integration, implementation and test plan which specifies how these phases of the development are executed.

This document is mainly addressed to development and design stakeholders.

1.2 Scope

The project is developed using both a web application and mobile application for customers, store managers and checkpoint controllers. The presence of both web application and mobile application is intended to reach the highest number of customer due to user preferences. The web application shall be responsive, then it will be usable both on computers and mobile devices.

The application shall be unique and there should not exist different application for different users. The application uses RBAC to verify which functionalities included in the RASD offer to the logged user.

The system is going to be developed with the usage of a DBMS to store information provided by the users and every information needed to implement accordingly with the requirements of the application. Both mobile application and web application front-end interact with the same business components (i.e. there exists only one DBMS and only one Database) and so a user may choose by its own to use the mobile application or the web application.

1.3 Definitions, acronyms and abbreviations

Definitions:

1. **Check-in:** the act of scanning a ticket by a checkpoint controller at entrance, in order to validate it;
2. **Check-out:** the act scanning a ticket by a checkpoint controller at exit, in order to register the fact that the owner is gone;

Acronyms:

1. **DBMS**: database management system;
2. **DD**: design document;
3. **RASD**: requirement analysis and specification document;
4. **RBAC**: role-based access control;
5. **S2B**: system to be;
6. **UML**: unified modelling language.

1.4 Revision history

Main version	Subversion	Additions and modifications
0	1	Purpose, scope of the document and document structure added to the document. Some acronyms have been inserted. An overview of the high-level components and their interaction has been inserted in chapter 2.
1	0	Component diagram, deployment diagram, runtime views, design patterns, user interface, requirements traceability, implementation, integratiton, testing
	1	Revision and some parts rewriting

1.5 Document structure

Chapter 1: this chapter contains the introductory information to the document. It contains the needed information in order to understand and read the whole document. This chapter includes a list of the revisions done to the document and a list of the documents cited and used as reference to design the system.

Chapter 2: this chapter includes software architectures used to design the system. The chapter uses extensively the UML to define architectural components and architectural views. The chapter once it has defined the entities and the components of the system then defines how components interacts with each other. At the end of the chapter there are specified the selected design patterns and other design decisions which are not described using UML.

Chapter 3: this chapter extensively describes how the interface should be realized in the two types of applications which are the mobile application and the web application. This part specifies what is contained in the users' views using images and describes what is possible to do on the views.

Chapter 4: this chapter is strongly connected with the RASD because it specifies how the requirements for the application are fulfilled by the design document. There is an analysis of the system using the traceability matrix.

Chapter 5: this chapter contains three different detailed plans on how the software components are going to be implemented, integrated and tested.

Chapter 6: this part contains information about references used in the document.

2

Architectural design

This chapter describes the principles and architectural design choices that we adopted for the S2B. Our focus is mainly on the server-side of the application.

At first, we provide a brief overview that highlights the core parts of our system. Then, we show in-depth the components that provide the functionalities illustrated in the RASD. Such abstracts objects and their relationships are described at various levels of abstraction and granularity in section 2.2, where we also explain the functions provided by each component and its interactions.

Section 2.3 contains a diagram of how the application should be deployed, i.e. the physical infrastructure that should support the system. After having provided a static view of the S2B, in section 2.4 we cover in detail the application dynamics. Some runtime diagrams explain in detail how the components must interact to accomplish the main tasks needed for the software to work properly.

Section 2.5 explains in greater detail the interfaces of each component and illustrates how they can communicate with the rest of the system. A general diagram gives an overview of every interface, though text-based explanations are provided afterward too.

Finally, in section 2.6 you can find a list of the main design patterns and styles that we implicitly or explicitly adopted in this document, accompanied by a short explanation about why we chose them.

2.1 Overview

The main component is the central application server. Such high-level component is in charge of receiving requests from both Customers, Store Managers and Checkpoint Controllers and decides whether to accept or reject them. Its responsibilities include:

- **Queue real time management:** monitors the state of a queue, adds customers to the requested queue, removes them when necessary and calls a new number whenever it is needed.
- **Bookings management:** keeps track of booked visits for each store that subscribed to the service. It handles requests for visits and their deletion.
- **Account management:** allows for registration by Customers. It also provides data verification functionalities whenever it receives a login request.

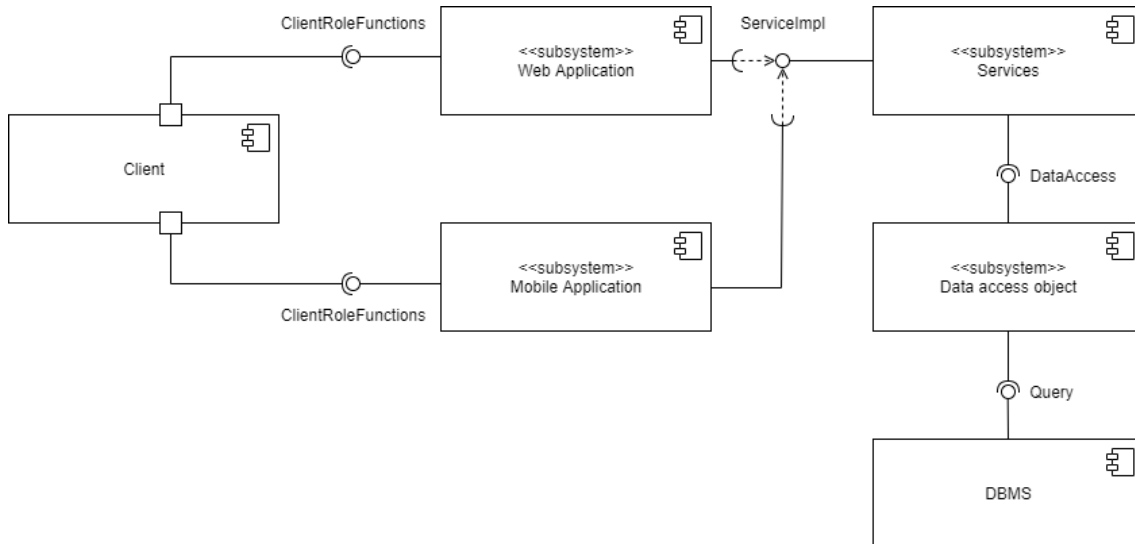
Another important component is the central Database. Information concerning the accounts of every user, the monitored stores, and records of every visit is stored in such database. This component is normally accessed only via transactions performed by the central application

server. However, the DBMS can be directly accessed whenever a sys-admin needs to create accounts for store managers or checkpoint controllers, apart from maintenance and configuration needs.

Users can communicate with the central server by either using a mobile app or an appropriate web application. They will be able to send the intended requests to the main server and receive responses. The main server also exchanges messages with ticket machines, which notify when a customer has printed a paper ticket for instance.

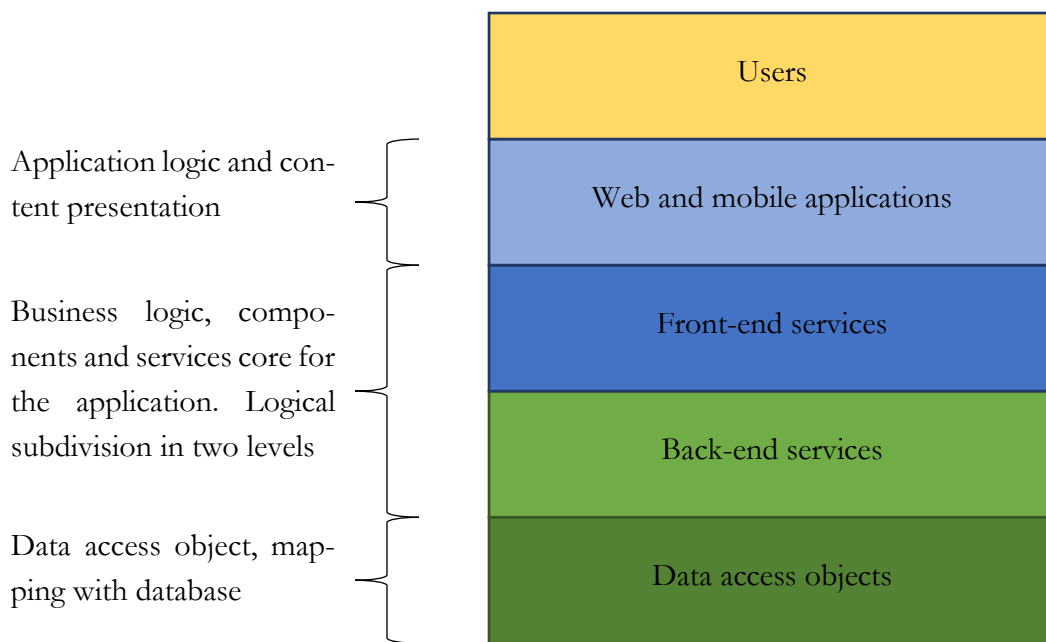
2.2 Component view

The application needs several components of different types in order to fulfil the requirements given in RASD. In the component view there are different layers of abstraction since components can contain other components and they can be black-box, glass-box, grey-box and white-box. Here follows a top-down analysis starting by the highest possible level of abstraction for the S2B.



This is a component diagram describing high-levels components in terms of subsystems. The system is composed of a client which has a role and can access its role functions through interfaces both by the web application subsystem and the mobile application subsystem. Services components realize the business of the application exposing APIs to the application level (Web and Mobile application). Data Access Objects realize the connection with the DBMS exposing APIs to allow Services to access the DBMS data without directly contacting the DBMS (doing so, Services do not need to know about the DBMS' internal state). Here the DBMS is specified as component since it is third-party software. In the rest of the section, it will be not uncovered specifying its subcomponents. Data access objects are not specified too since they are strictly dependent on the technology chosen by the developing team.

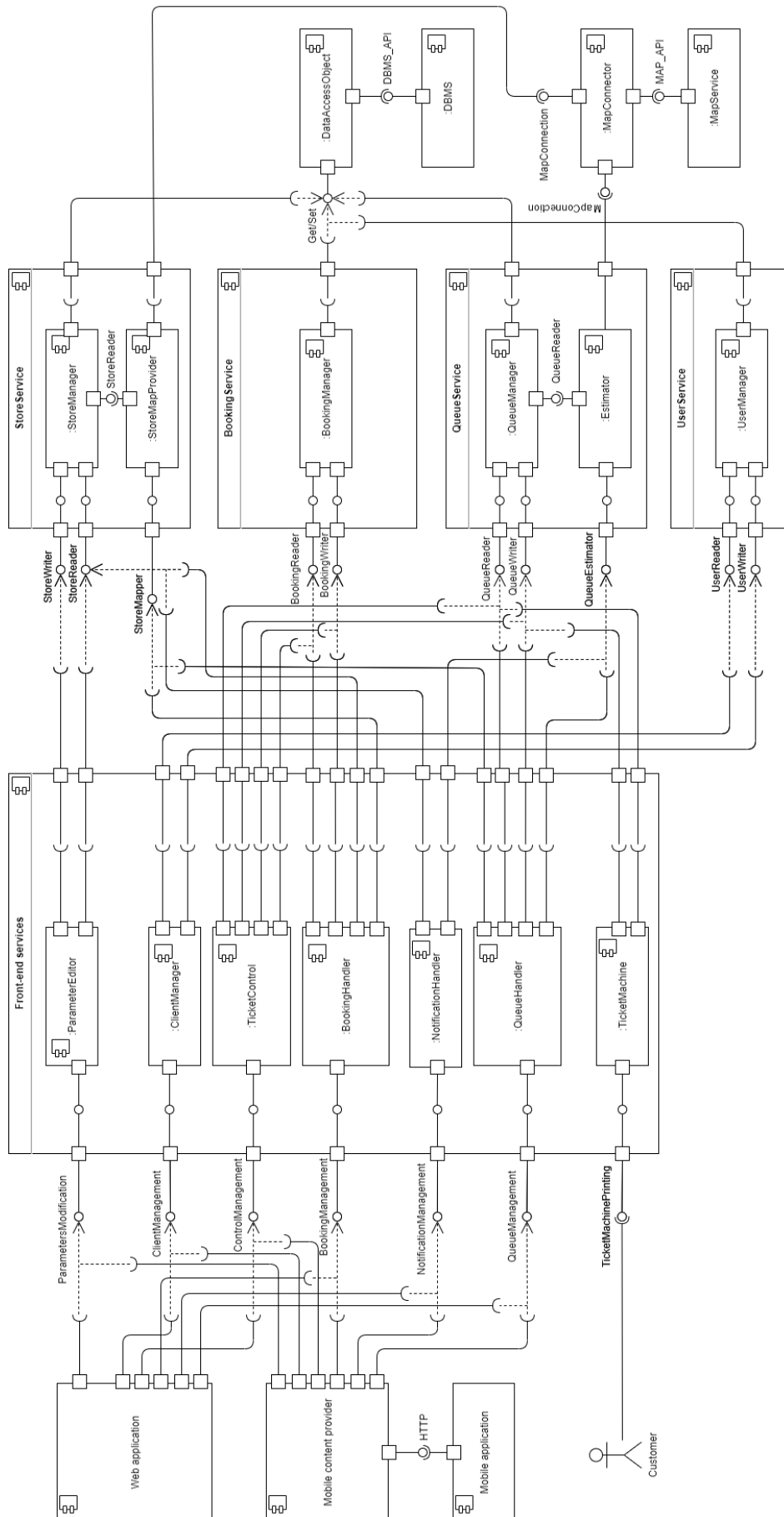
The presentation layer and part of the application logic (field correctness checking) is contained in the web application and mobile application. The business logic is contained in the services and it has been split in two different layers that we call respectively **front-end services** and **back-end services**. Their main difference is the abstraction level: the front-end services are directly called by the web application and the mobile application. Front-end services use back-end services which are nearer to the database and are closer to data. They allow the front-end services to inspect data and obtain some complex data evaluated directly on the values of objects mapped from the database. Behind the back-end services there are DAOs, which are the lowest application level components and are simply the programming abstraction of database information. To better explain what we mean a graphic is shown here below.



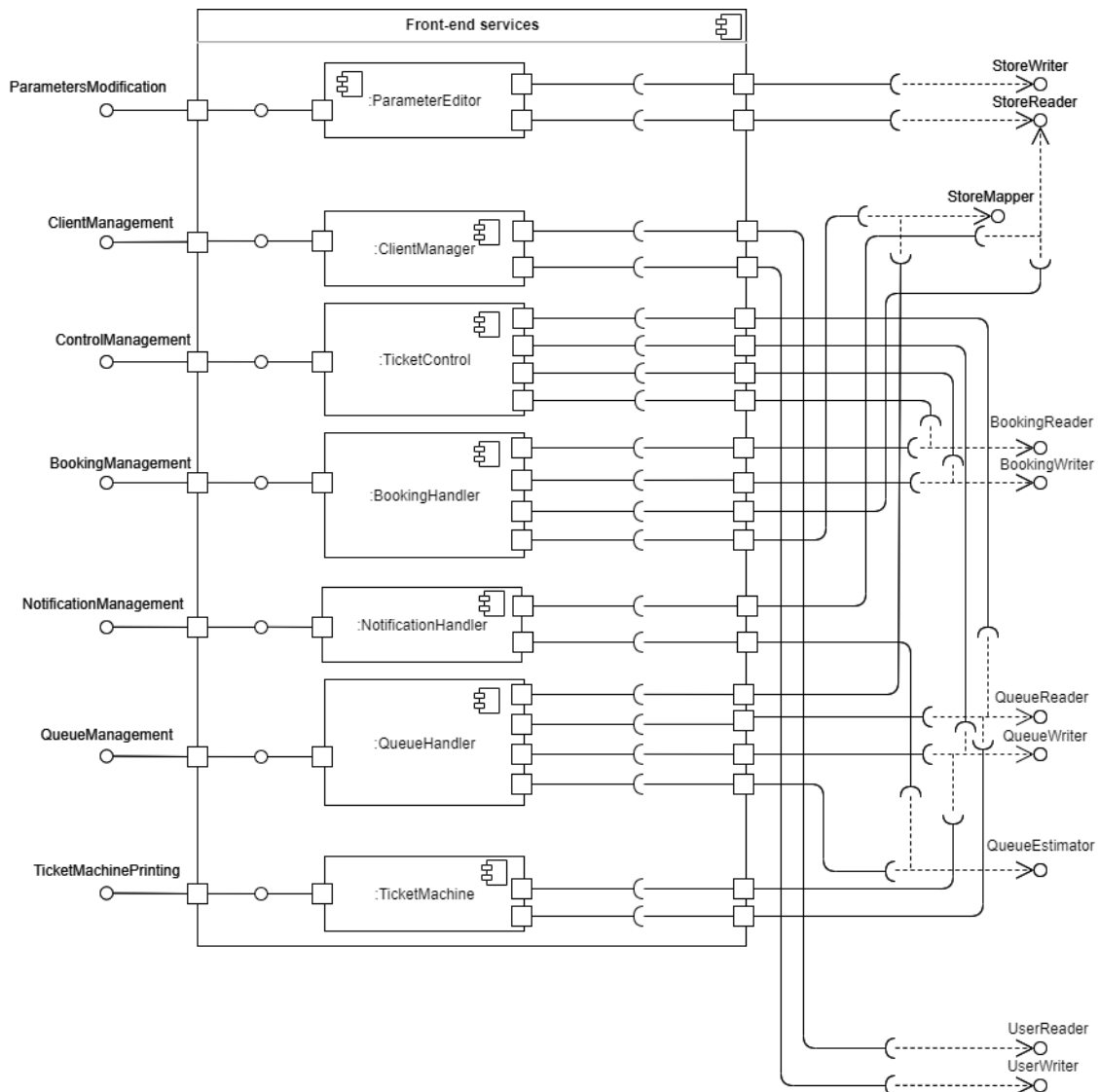
We do not talk about DAOs at a finer granularity simply because they depend on the Database schema, which is not specified in this document. We also do not provide a finer description of both the Mobile App and the Web App because we prefer to focus more on the server side, which is the core of the system.

The previously described system is going to be developed and assembled to be a 4-tier software system. This system is developed as 4-tier software being inspired by the web pattern of 3-tier client-server approach. As shown in the complete component diagram, there are 4 levels of abstraction: application level (web and mobile application), high business level, low business level and data level. The application level contains the presentation and the management of the clients' requests. The server at the application level uses the high business level (called front-end here). High-level business components provide services usable by the clients, using and retrieving complex data from the low business level. The latter is in charge of extracting information from the DB and elaborates it to produce more complex data, which have to be inferred from the DB because they are not fields of tables. The low business level (called back-end in this document) elaborates and writes complex data to the database using DAO pattern. *Here we use term front-end and back-end referring to the business level*, i.e. high-business level is the front-end component of business to communicate with the application level, it does not communicate directly with the client.

In the following pages we provide the general component diagram in order to have a general view of the system, then we explain singularly each subsystem.



2.2.1 Front-end services component



Front-end services are those services used by the clients through the application level to access functionalities. Front-end services are of different type and are intended to be atomic, so they do not contain other components. Because front-end services are atomic, we grouped them in a component called front-end component.

A *central component for the application* which allows to realize RBAC is the **ClientManager**. This manages client's session usually. The **ClientManager** is vital for the application, it is in charge of managing the session and to recognize the role of the client using the application. Different clients use different services, this is achieved using authentication and the client's session.

A **store manager** can access a service of store's parameter modification for one of the stores he manages. The application offers a service, the **ParameterEditor**, for editing some store's parameters. Such service uses the **StoreService** to get and modify the selected parameters.

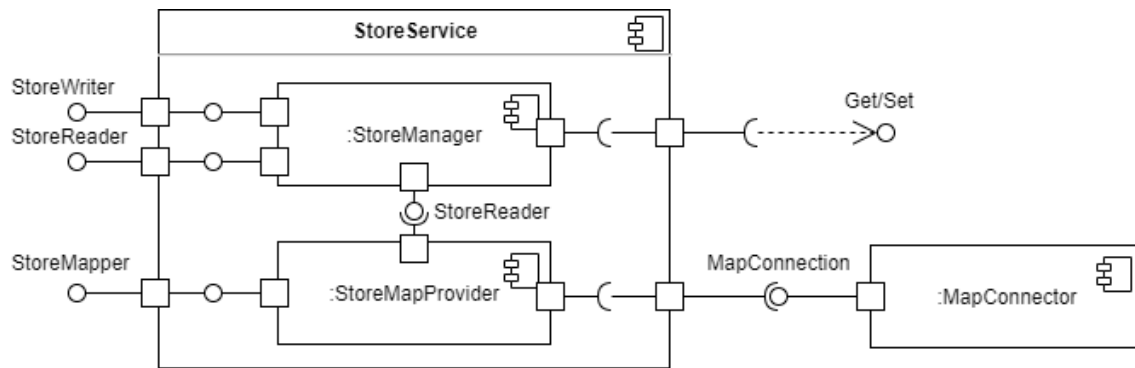
A **checkpoint controller** can access the application to control tickets using the **TicketControl** component.

A **customer** can book a visit or queue for a store. The components related to customer's activities are the **QueueHandler**, **BookingHandler**, **TicketMachine** and **NotificationHandler**. The **QueueHandler** manages the operations which can be performed by a customer on a queue: the customer can join a queue for a specific store and can leave the queue whenever they want. Furthermore, the **QueueHandler** component allows the customer to get a precise estimation of the waiting time to get in the store. These three functionalities are realized using the **QueueService** interfaces which modify the queue and inspect it to compute an estimation.

A customer can use the **BookingHandler** component to make a reservation or to delete one of its existing future visits. It uses the **BookingService** component's interfaces to create or delete bookings from the database, and uses the **StoreService** to retrieve the store's availability while creating a booking, letting the customer choose their favourite time between all the possible time slots.

A customer can receive notifications regarding a queue or the availability of time slots to book a visit. Since we are going to use a client-server approach, we immediately observe that the implementation cannot be directly of a notification sent from the server to the client since this is impossible. If the client has a mobile application, the application will periodically check if there exists a notification and if so, it will prompt out a notification on the smartphone of the client (if the application can throw a notification and the user has not disabled the functionality through system's settings, otherwise a notification can be received only after entering the application). In the case of a web application, if the web page is still open a sound can be played to make the customer aware of the notification (easily achieved using web technologies like Javascript).

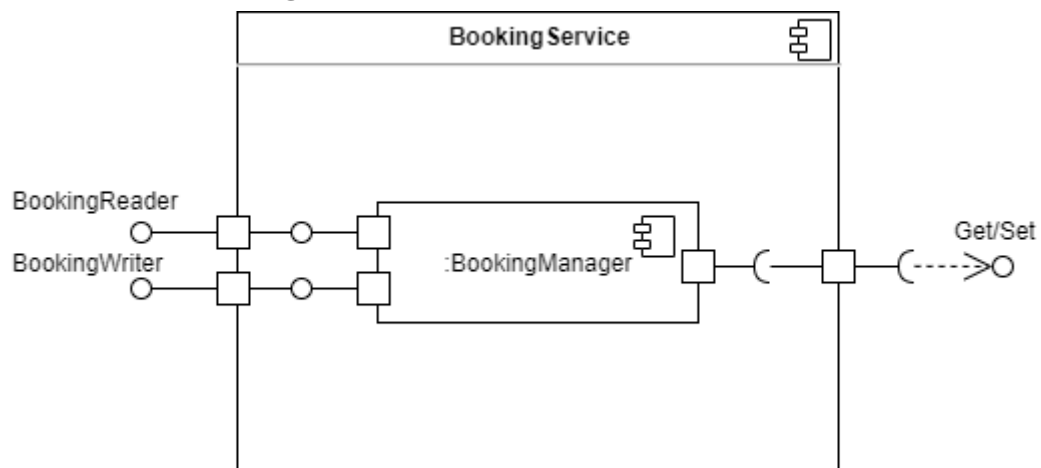
2.2.2 Store service



Store service is the component used to inspect and edit store parameters for what regards the S2B. The Store service is a component which directly communicates with the DAOs. The communication is not specified in terms of interfaces with such components since they strictly depend on the chosen technology.

Store service is composed of two subcomponents which are the **StoreManager** and the **StoreMapProvider**. The **StoreManager** is the component used to retrieve information about a store and to modify the information of a store. The **StoreMapProvider** component is used to obtain a map of stores in a certain area. The **StoreMapProvider** uses the interface **MapConnection** to connect with the **MapConnector** component. This structure is the component view of the Façade pattern, using the interface **MapConnector** to use the **MapService**. Doing so the **StoreService** has no code dependant on the specific map API chosen, it only calls the **MapConnector** which implements the communication. To provide maps the **StoreMapProvider** needs to read data of a store. This is done by using the store manager's **StoreReader** interface.

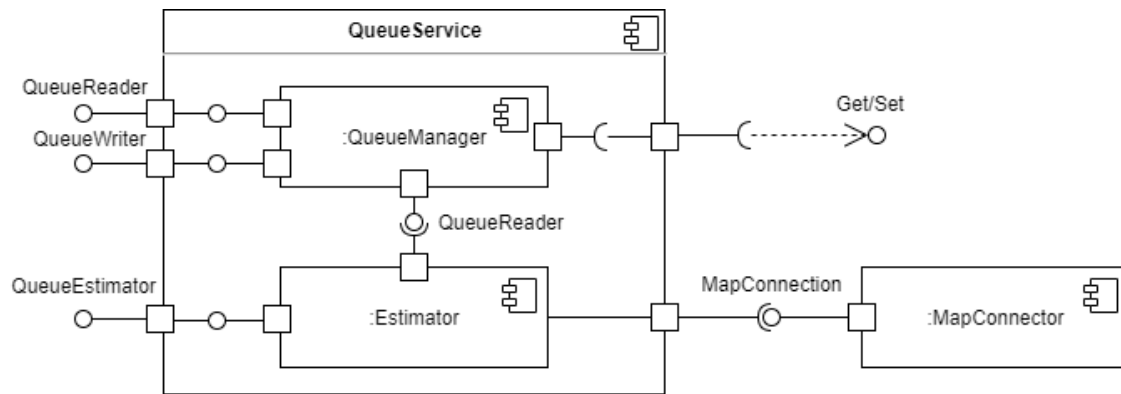
2.2.3 Booking service description



Booking service is the component used to create and delete a booking for what regards the S2B. It directly communicates with the DAOs. The communication is not specified in terms of interfaces with such components.

The **BookingManager** component creates and deletes bookings. It also extracts information regarding them by the simple invocation of DAOs.

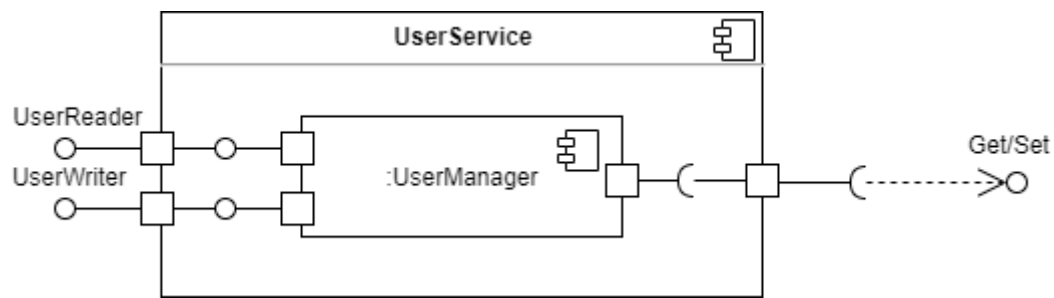
2.2.4 Queue service



Queue service is the component used to get in and get out of a queue. It directly communicates with the DAOs. The communication is not specified in terms of interfaces with such components.

Queue service is composed of two main sub-components: **QueueManager** and **Estimator**. The **QueueManager** component is in charge of adding and removing clients to or from a queue, as well as inspecting a queue. These operations are realized calling DAOs methods. When a client is in a queue, it can get an estimation of the waiting time. The **Estimator** component is the component that estimates the waiting time for the customer to get into the store. The **Estimator** uses **MapConnector** to estimate the travel duration for the customer to reach the store. The latter component allows to retrieve maps information using Façade pattern as previously said.

2.2.5 User service



User service component is used to retrieve data of a user. Such information can be used to log in and to register, in order to access services. This back-end service is connected to DAOs to get the database's information needed to retrieve data about users.

The only component of the user service is the **UserManager**. This component exposes the two interfaces of **UserReader** and **UserWriter**, allowing the front-end services to read and write user's data.

2.3 Deployment view

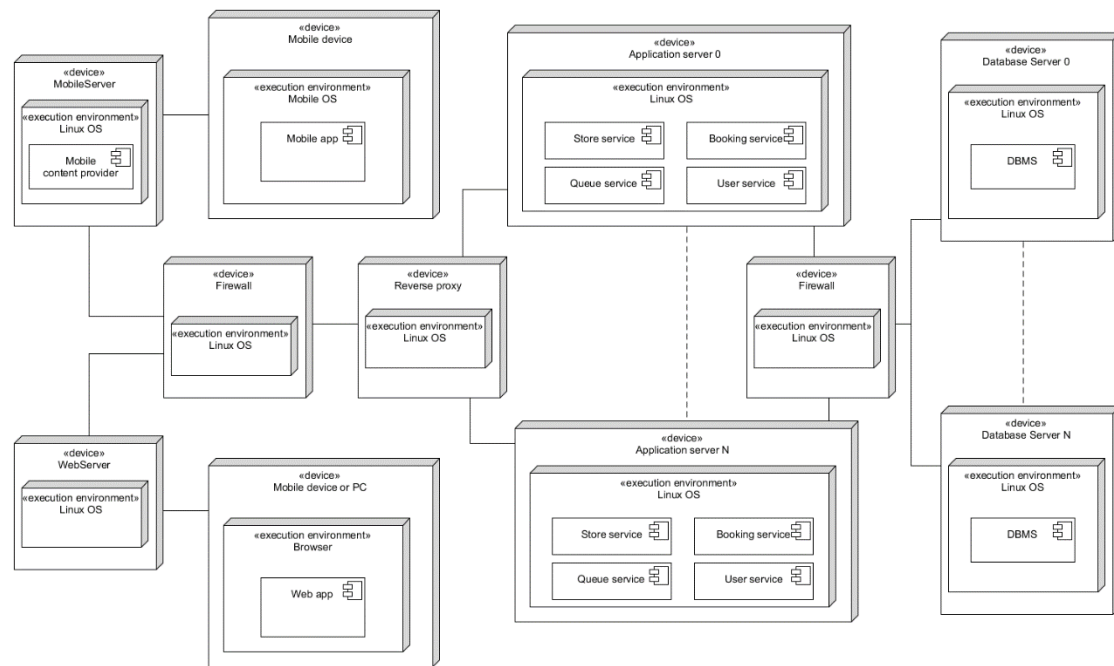
The diagram below shows how the system will be deployed. A web server hosts the web app and the client can access it using his favourite browser. A server offering the mobile content provider service will interact with the mobile app. This component replies to HTTP requests sent by the mobile application. The HTTP response contains the needed content to build the presentation. In order to prevent overloading, the third tier's functionalities will be distributed among some application servers. Cloning the application servers is chosen over partitioning. This is because in case some failure occurs in one application server, the other servers can do its work without interrupting the service. Their number will be proportional to the business tier's computational load, such that there is always a significant margin of available computing capacity. In case the computational load is much lower than the available capacity, and is expected to remain that way, some servers may be shut down to save on maintenance costs.

A reverse proxy will handle work distribution. If an excessive amount of internet traffic is slowing down the system, its load balancing functionality will distribute the traffic over one or more servers to improve overall performance. In addition, it can compress inbound and outbound data, as well as cache commonly requested content, both of which speed up the flow of traffic between clients and servers. It will be secured by a firewall placed before it, as it typically is.

An additional firewall will filter the access to the DBMS. This is to provide a stricter security layer to the DBMS, which has a higher sensitivity level than the rest of the system.

The DBMS will be hosted on different servers such that maintenance work, hardware problems, security breaches and so forth do not necessarily impact the whole platform. Moreover, scaling up multiple machines will yield more performance benefits than scaling up one big one. Their number will depend on the system's amount of stored data.

Linux is chosen to host the systems components because compared to other OS it is more: stable and reliable, secure, flexible, and is generally cheaper in terms of licensing fees, software/hardware purchase, and maintenance costs.

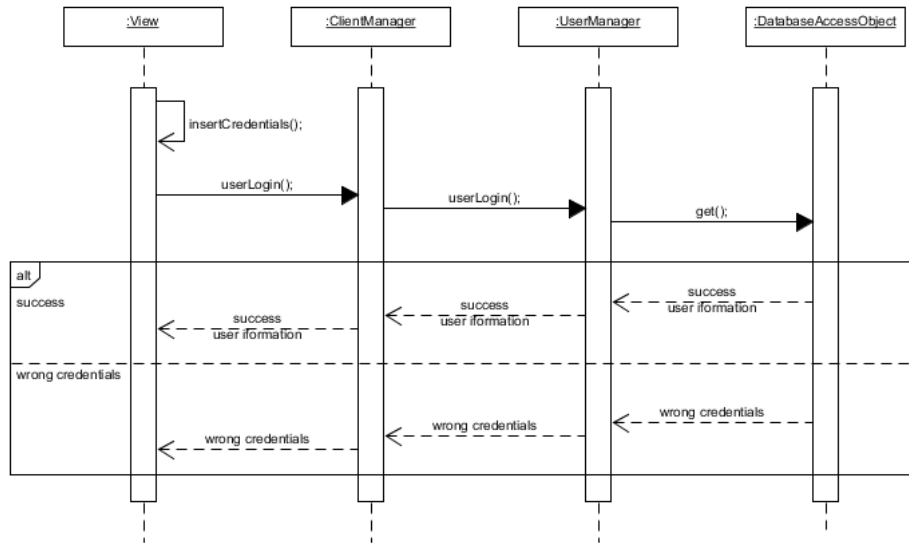


2.4 Runtime view

The sequence diagrams of the most significant events are reported below. In order to avoid clutter the web application component and the mobile application component have been condensed into the View entity, as they perform the same actions in the same circumstances.

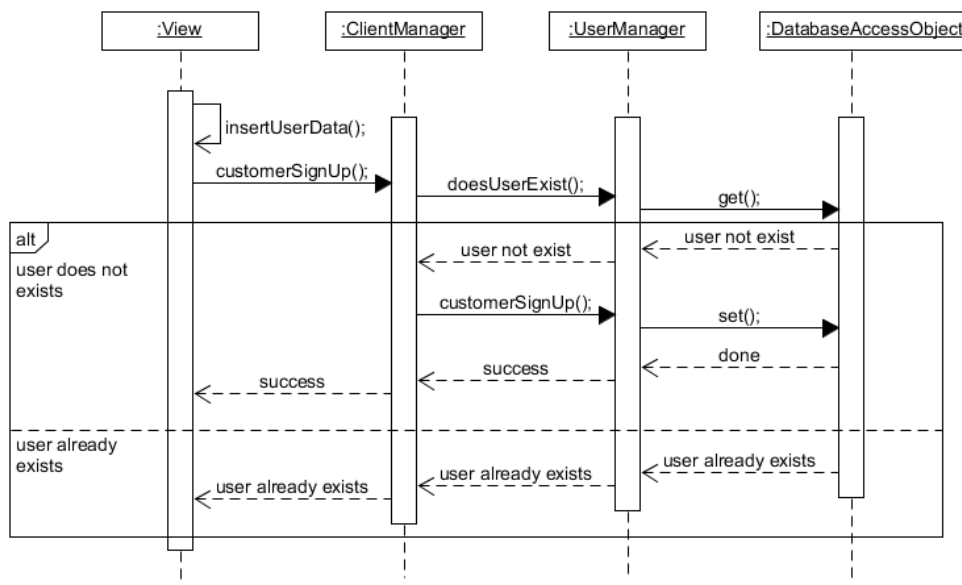
2.4.1 User login

To login every user needs to first insert their credentials. The credentials will then be sent all the way to the **UserManager** that will be responsible for validating them.



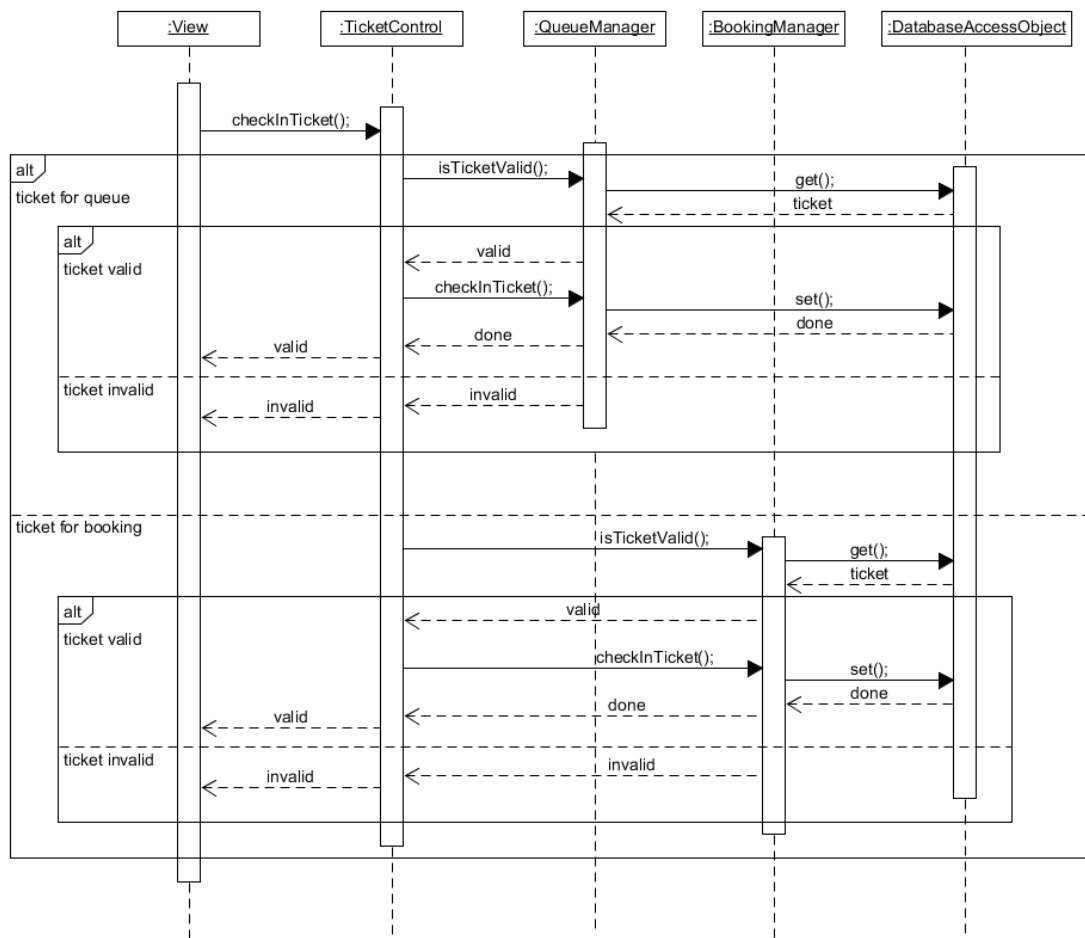
2.4.2 Customer registration

The registration through the view is only available the customers. To register they need to insert their registration data. That data will reach the user manager that after verifying that the user to be created is new will create it by saving it to the DBMS.



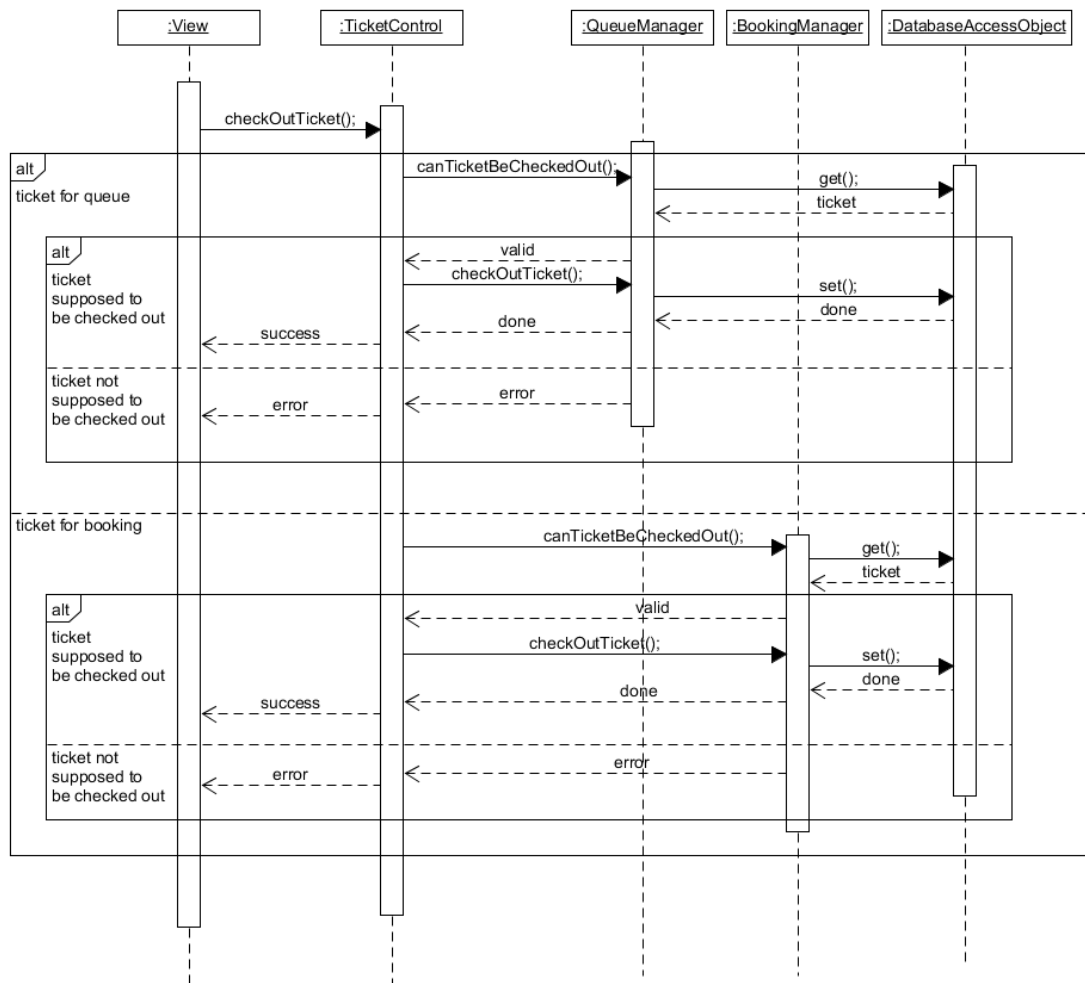
2.4.3 Ticket check-in

Checkpoint controllers at the entrance of the store will check in customers by scanning their tickets. The ticket control component will inquire to either the booking manager or queue manager, depending on the ticket's type, the ticket validity. The booking and queue managers will elaborate it after retrieving ticket's information from the database. The **TicketControl** component is able to decide if the ticket is a queue ticket or a booking ticket, based on the information present on the ticket.



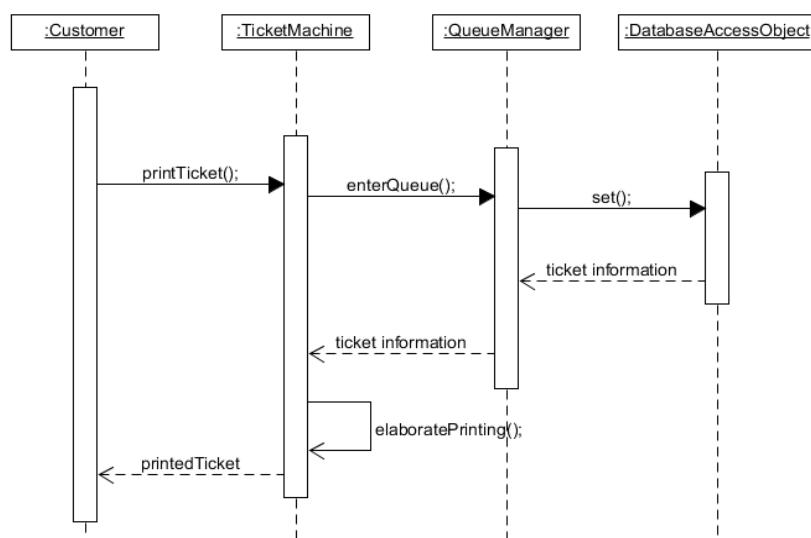
2.4.4 Ticket check-out

Checkpoint controllers at the exit of the store will check out customers by scanning their tickets. The ticket control component will tell to either the booking manager or queue manager, depending on the ticket's type, to check out the ticket. The booking and queue managers will check out the ticket, updating the fact that the customer has left the store, if the ticket was supposed to be checked out.



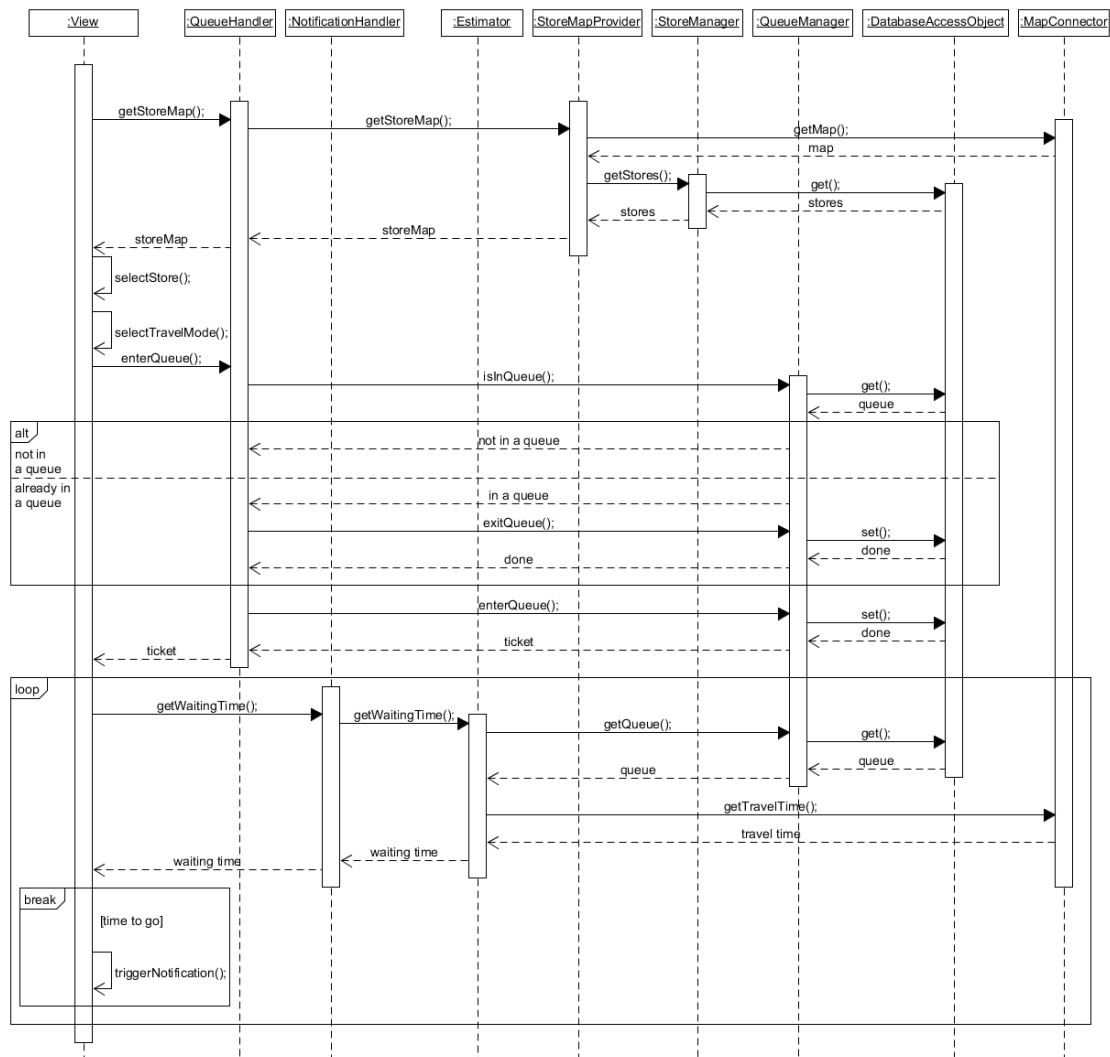
2.4.5 Physical queue

The ticket machine component is responsible for providing tickets to customers in store. No identification is required to print tickets. The queue manager is responsible for creating the new ticket and sending it to the ticket machine component that will print it.



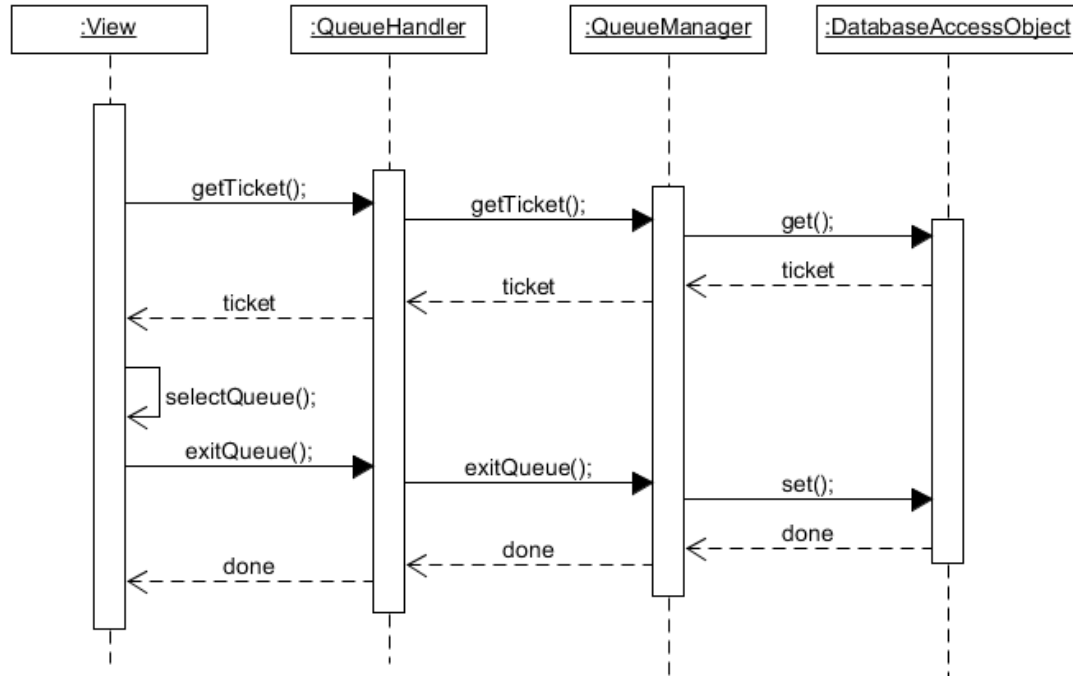
2.4.6 Enter online queue and notification

First the view receives the stores' map from where the customer will choose the store. Such map is provided by the store map provider, it will need to receive the map by the map connector and the stores by the store manager. Then after selecting store and travel mode the queue manager will create the queue ticket for the customer. The customer's view will then keep asking the notification handler for the waiting time. Such waiting time is the time to wait before heading to the store. It will be calculated by requesting the travel time to the map connector and the queue status to the queue manager. When receiving it the view will trigger a notification if the waiting time is over and will stop the waiting time requests.



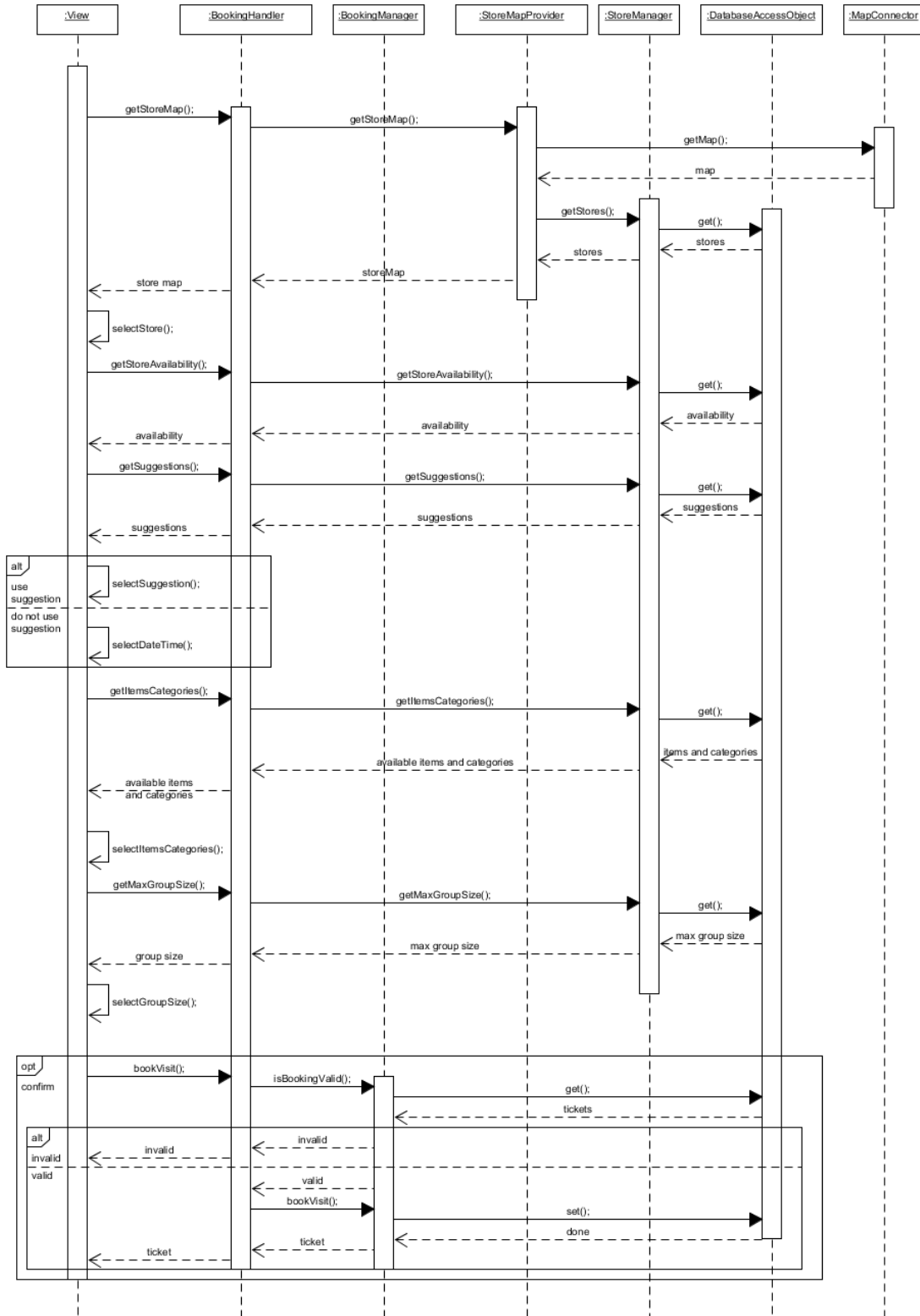
2.4.7 Queue exit

The view will first receive information on the current queue ticket. Assuming that there is a queue ticket, otherwise it would not be possible to exit the queue, the customer will select first the queue. Then, he exits the queue and the queue manger will finalize this operation.



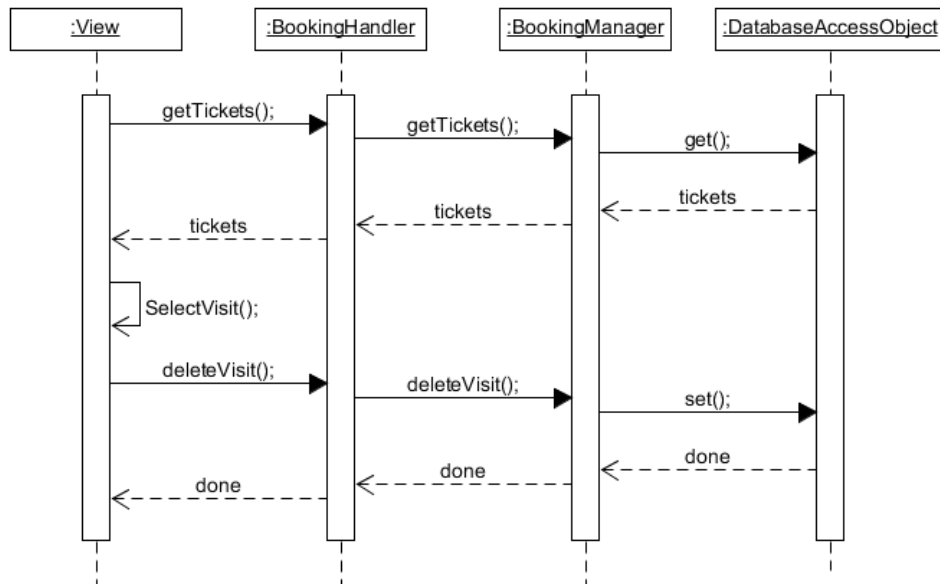
2.4.8 Visit booking

First the view receives the store map as explained in the enter online queue diagram. Then the view will set the specified parameters after getting information about them from the store manager component. At the end if the customer confirms it the booking manager will finalize the booking and provide the ticket to the customer.



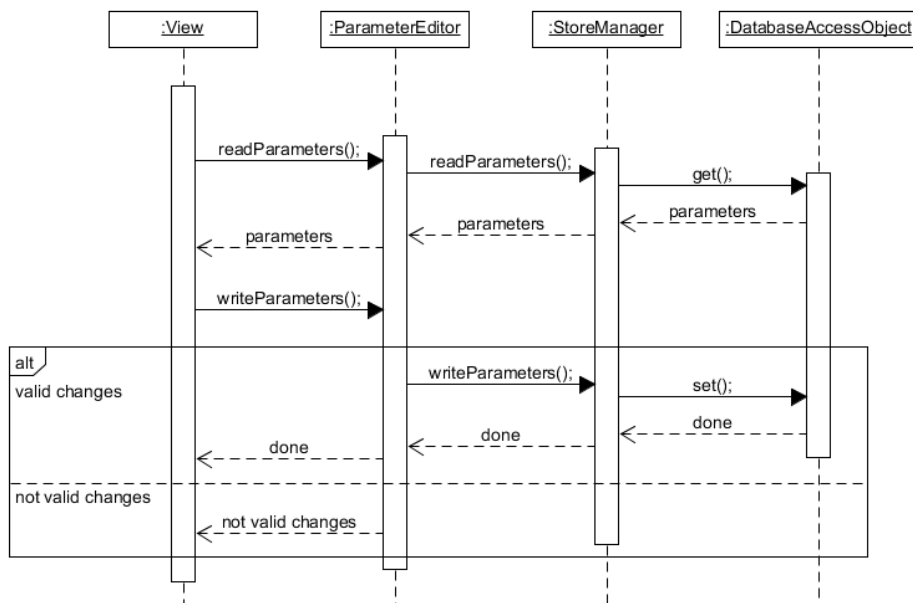
2.4.9 Booking deletion

The view will first receive information on the current booking tickets. Assuming that there is at least one booking ticket, otherwise it would not be possible to delete a visit, the customer will select to delete the visit and the booking manger will finalize this operation.



2.4.10 Store manager's parameters' modification

After receiving the current parameters, the store manager user will modify them and on save they will be sent to the parameter editor. It will check their validity and in case they are valid the store manager will save them on the database.



2.5 Component interfaces

This section presents the components' interfaces' methods. The methods are defined in an abstract way, independent of the actual technology that will be used during the implementation. Some of these abstract methods might then be mapped in the implementation to specific get and set request. Keeping this level of abstraction is vital to not constraint the implementation team.

The interfaces' methods are first presented by the diagram below, and then described more in detail in the following list.



2.5.1 Front-end services

- **ParametersModification**
 - **readParameters()**: reads any parameter of the store.
 - **writeParameters()**: writes any parameter of the store if the changes are valid.
- **ClientManagment**
 - **customerSignUp()**: creates a new account for a customer given registration data like name or email as input. If it succeeds it returns success. If the user already exists it returns an error.
 - **userLogin()**: takes credentials as input and tries to authenticate the user. If it succeeds it returns success and information on whether the user is a customer, a checkpoint controller, or a store manager, so that the view can display the appropriate UI. It returns an error if the credentials are wrong.

- **ControlManagment**
 - **checkInTicket()**: controls if the ticket given as input is valid either for booking or for queueing. If it is, it checks in the ticket, if it's not it returns the reason for its invalidity.
 - **checkOutTicket()**: checks out a ticket.
- **NotificationManagment**
 - **getWaitingTime()**: returns the time to wait before heading to the store.
- **BookingManagment**
 - **getTickets()**: return the active visits' tickets of the customer.
 - **deleteVisit()**: deletes a visit.
 - **bookVisit()**: books a visit and returns a ticket for it.
 - **getStoreMap()**: returns the store map used by the user to select the store.
 - **getStoreAvailability()**: returns information on the store's availability.
 - **getSuggestions()**: returns suggestions containing both a suggested time and a suggested store.
 - **getItemsCategories()**: returns the items and categories of a store.
 - **getMaxGroupSize()**: returns the max visiting people per ticket of a store.
- **QueueManagment**
 - **getTicket()**: if the customer is in a queue it returns his ticket for that queue.
 - **enterQueue()**: enters the customer in the queue of the store given as input and returns a ticket for it.
 - **exitQueue()**: exits the customer from the queue of the store given as input.
- **TicketMachinePrinting**
 - **printTicket()**: prints a ticket.

2.5.2 Back-end services

- **StoreWriter**
 - **writeParameters()**: given a store as input, writes any parameter of the store.
- **StoreReader**
 - **readParameters()**: given a store as input, reads any parameter of the store.
 - **getStores()**: returns the store of a given area.
 - **getStoreAvailability()**: returns the availability of store.
 - **getSuggestions()**: returns suggestions containing both a suggested time and a suggested store.
 - **getItemsCategories()**: returns the items and categories of a store.
 - **getMaxGroupSize()**: returns the max visiting people per ticket of a store.
- **BookingWriter**
 - **deleteVisit()**: deletes a visit.

- **bookVisit()**: given all the booking parameters as input, saves the visit and returns a ticket for it.
- **checkInTicket()**: given a ticket and a store as inputs, saves that it has been used and returns that is valid.
- **checkOutTicket()**: given a ticket and a store as inputs, checks out the ticket.
- **BookingReader**
 - **getTickets()**: given the customer as input, it returns the active visits' tickets of that customer.
 - **canTicketBeCheckedOut()**: returns whether the ticket can be checked out or not.
 - **isTicketValid()**: returns whether a ticket is valid or not.
 - **isBookingValid()**: returns whether a booking has valid parameters or not, and can consequently be finalized to generate a ticket.
- **QueueWriter**
 - **enterQueue()**: given a customer and a store as inputs, first checks if the customer is already in a queue. If he is not, he's added to the store's queue. If he is, he's removed him from the queue he's in and added to a new one.
 - **exitQueue()**: given a customer and a store as inputs, removes the customer from the store's queue.
 - **checkInTicket()**: given a ticket and a store as inputs, saves that it has been used and returns that is valid.
 - **checkOutTicket()**: given a ticket and a store as inputs, checks out the ticket.
- **QueueReader**
 - **getTicket()**: given the customer as input, if the customer is in a queue it returns his ticket.
 - **isInQueue()**: given a customer as input returns whether he is in a queue or not.
- **QueueEstimator**
 - **getWaitingTime()**: returns the time to wait before heading to the store.
- **UserWriter**
 - **customerSignUp()**: creates a new account for a customer given registration data like name or email as input.
- **UserReader**
 - **userLogin()**: takes credentials as input and tries to authenticate the user. If it succeeds it returns success and information on whether the user is a customer, a checkpoint controller, or a store manager, so that the front end can display the appropriate UI. It returns an error if the credentials are wrong.
 - **doesUserExist()**: returns whether the user given as input exists or not
- **MapConnection**

- **getTravelTime()**: given a starting point, and end point, and a travel mode as inputs, returns the travel time to go to the end point from the starting point by using the travel mode.
- **getMap()**: given a location as input returns the map of the area of that location.

2.5.3 Data access objects

- **DataAccessObjectInterface**
 - **set()**: perform any write operation on the database.
 - **get()**: perform any read operation on the database.

2.6 Selected architectural styles and patterns

The architecture of the system has been strongly influenced by the client-server approach both for the deployment of the components and the layer approach. The S2B has been modelled as a 4-tier application in order to reduce the concept contained in components and trying to separate meaning of actions. To do so, we decided to design a 4-tier application. The components of the application have been replicated in different application servers. Those servers are necessary in order to reduce the probability of unavailability since the application has to be highly available and reliable, as said in sections 3.5.1 and 3.5.2 of the RASD.

While designing the application, we chose to use the following design patterns:

- **Façade pattern:** the façade pattern is used to connect to external services like the `MapService`. We use a class called `MapConnector` which exposes some functions used to retrieve map data, doing so the integration between the application and the external map service is confined in this component and a future change of the service will be easier. The mobile application does not communicate directly with the front-end services for safety reasons. The only components which can access the front-end services are web applications. The mobile application sends requests to a web application which interrogates the front-end services acting as a façade for the mobile system;
- **DAO pattern:** the DAO pattern is used to connect components of the back-end services to the data layer of the database through a DBMS. This pattern needs to be used for every interaction with the database using the API to access it;
- **Model-view-controller:** model-view-controller pattern is chosen to develop the mobile application. The view of the application is totally embedded in the mobile application, which uses a web application to retrieve needed data using HTTP requests.

3

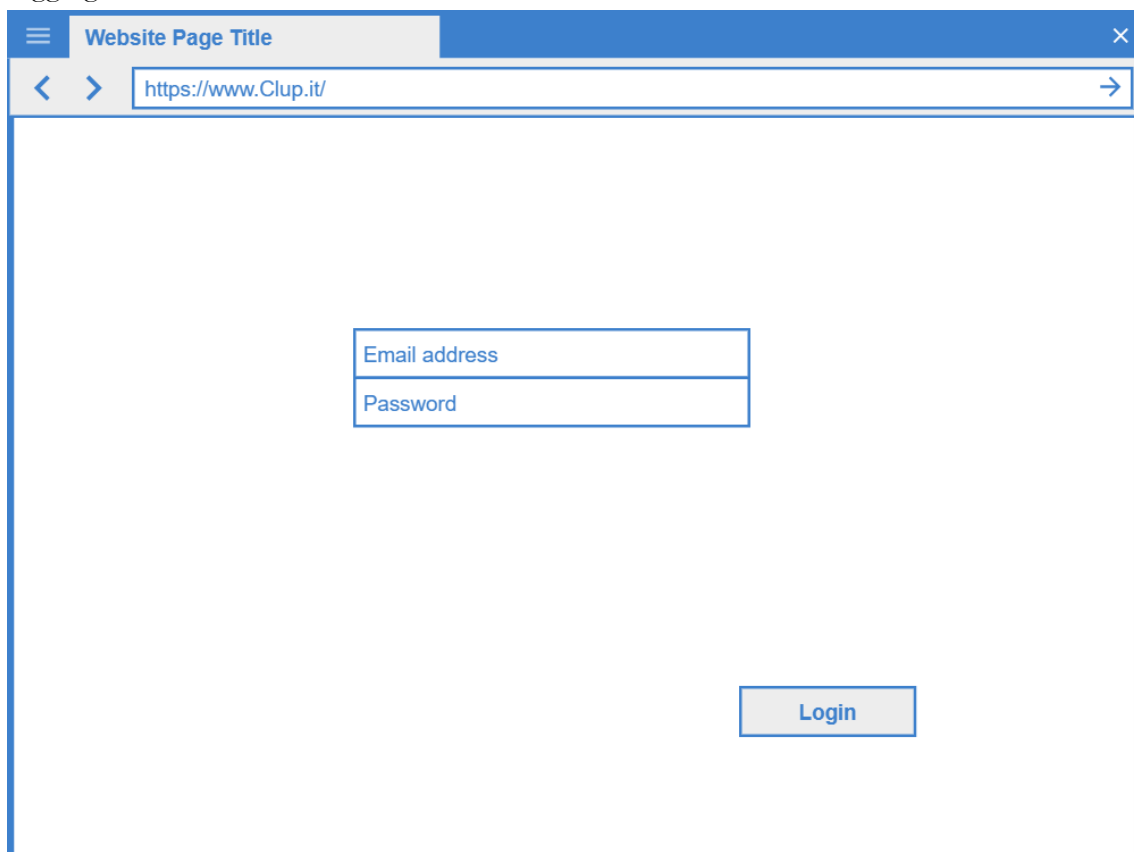
User interface design

Mock-ups for the web app and mobile app are presented below. Since the mobile app UI works in the exact same way as the web app UI only the web app interfaces are commented.

3.1 Web app

3.1.1 Login

The login page is the same for customers, checkpoint controllers and store managers. After logging in each one of them will interact with his own UI.



The image shows a mock-up of a web browser window. The browser's address bar displays the URL <https://www.Clup.it/>. The page content is centered and consists of two stacked text input fields. The top field is labeled "Email address" and the bottom field is labeled "Password". Below these fields, centered on the page, is a rectangular button labeled "Login". The browser window has a blue header bar with a menu icon on the left and a close icon on the right. The page title "Website Page Title" is visible in the header.

3.1.2 Checkpoint controller

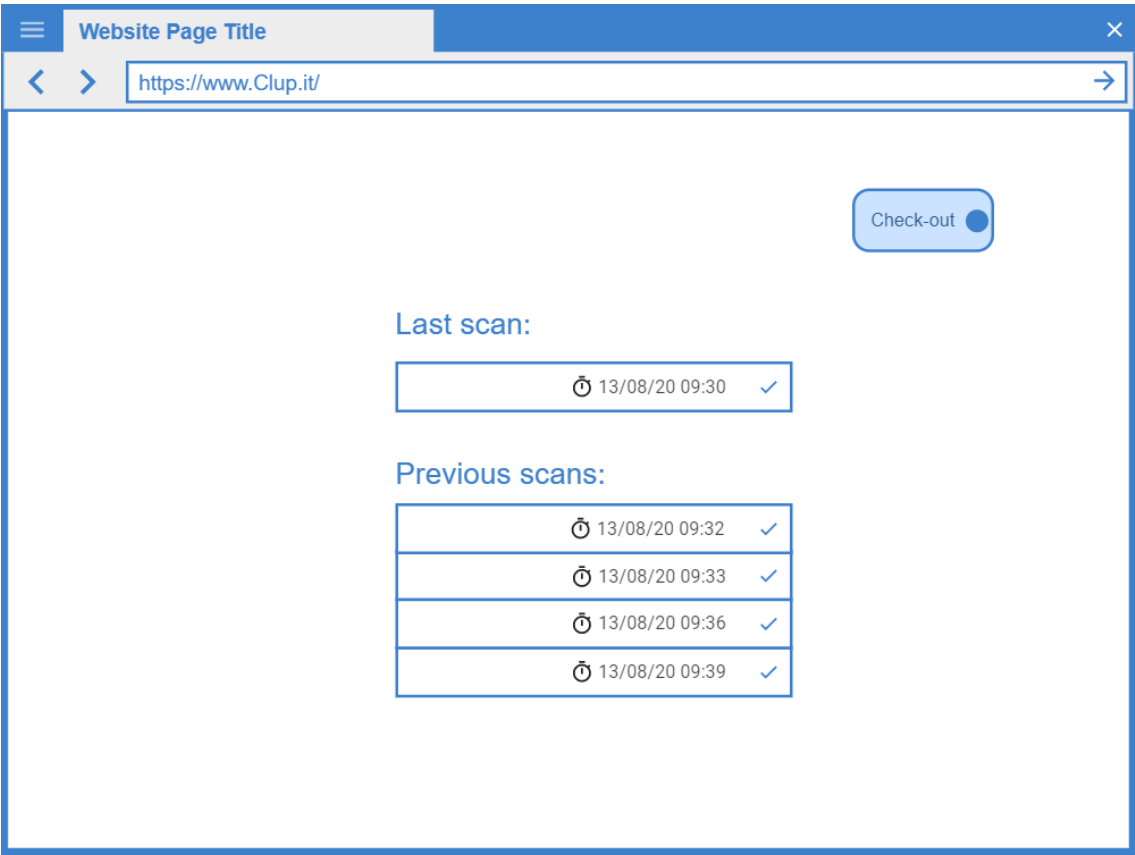
3.1.2.1 Checkpoint controller at entrance

The checkpoint controller can see the last scanned ticket, and the previous scanned tickets. The checkmark means that the ticket is valid, the warning sign means that the ticket is invalid and the invalidity reason is reported. The checkpoint controller can switch between checking in and checking out by using the toggle on the top right.

The screenshot shows a web browser window with the title 'Website Page Title' and the URL 'https://www.Clup.it/'. The main content area features a 'Check-in' toggle button in the top right corner. Below it, the 'Last scan:' section displays a single entry with a clock icon, the time '13/08/20 09:30', and a checkmark. The 'Previous scans:' section displays a table with four rows of scan data.

Previous scans:		
	🕒 13/08/20 09:32	✓
invalidity reason	🕒 13/08/20 09:33	⚠️
	🕒 13/08/20 09:36	✓
invalidity reason	🕒 13/08/20 09:39	⚠️

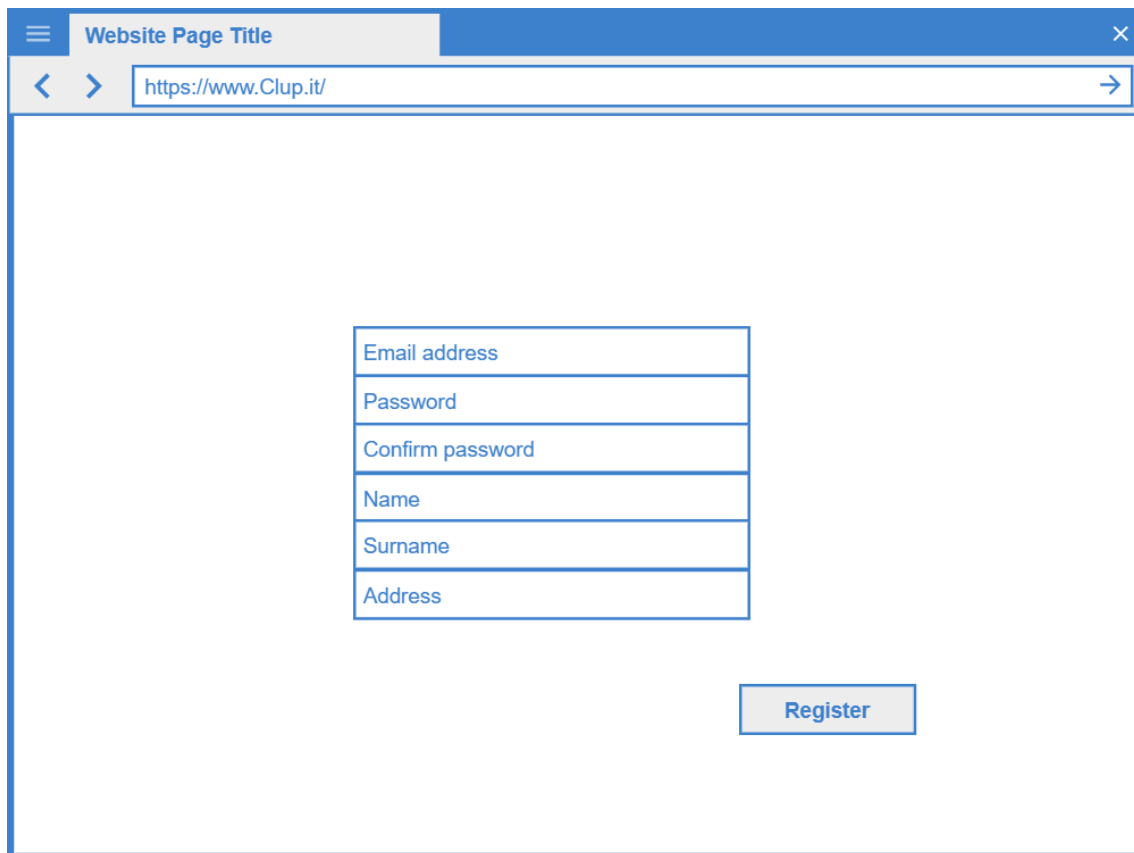
3.1.2.2 Checkpoint controller at exit



3.1.1 Customer

3.1.1.1 Registration

The customer can register into the system by inserting registration data.



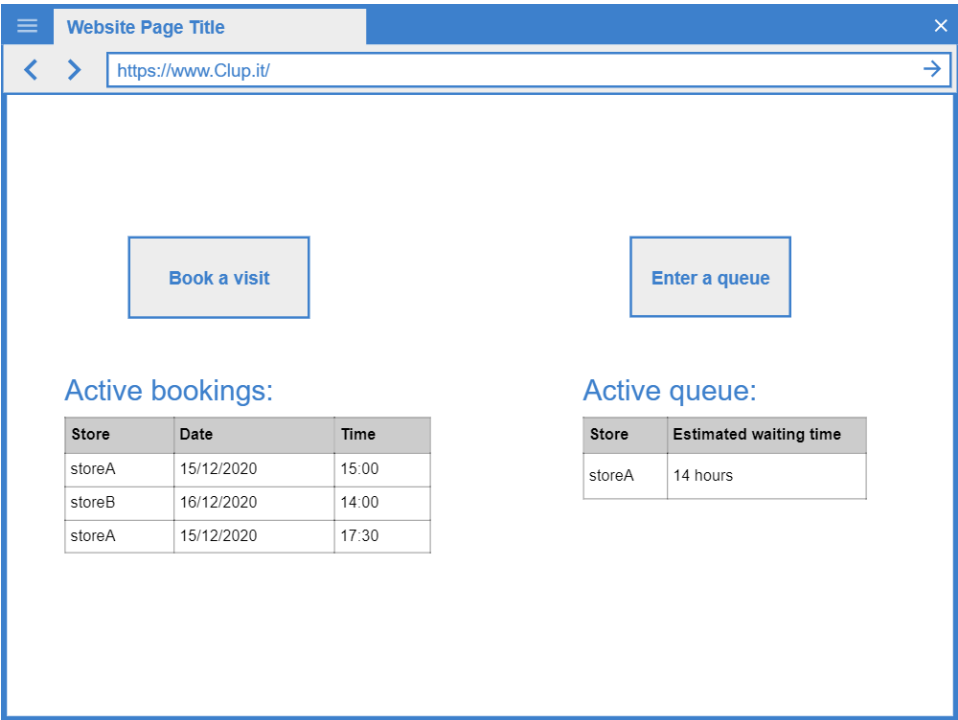
The screenshot shows a web browser window with the title "Website Page Title" and the address bar displaying "https://www.Clup.it/". The main content area contains a registration form with the following fields:

Email address
Password
Confirm password
Name
Surname
Address

Below the form, there is a "Register" button.

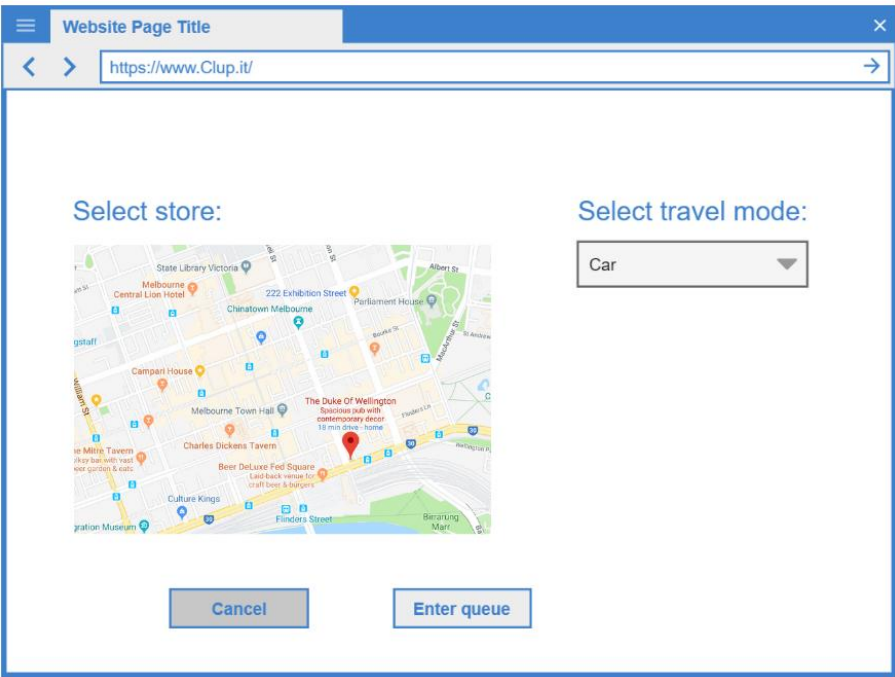
3.1.1.2 Home

This is the customer's default screen. From here he can book a visit, enter a queue or access his tickets. By accessing a ticket he also has the option to delete its visit or exit its queue.



3.1.1.3 Enter a queue

After pressing the enter a queue button the customer selects the store from the map and the travel mode from the combo box and can then enter a queue.



3.1.1.4 Exit a queue

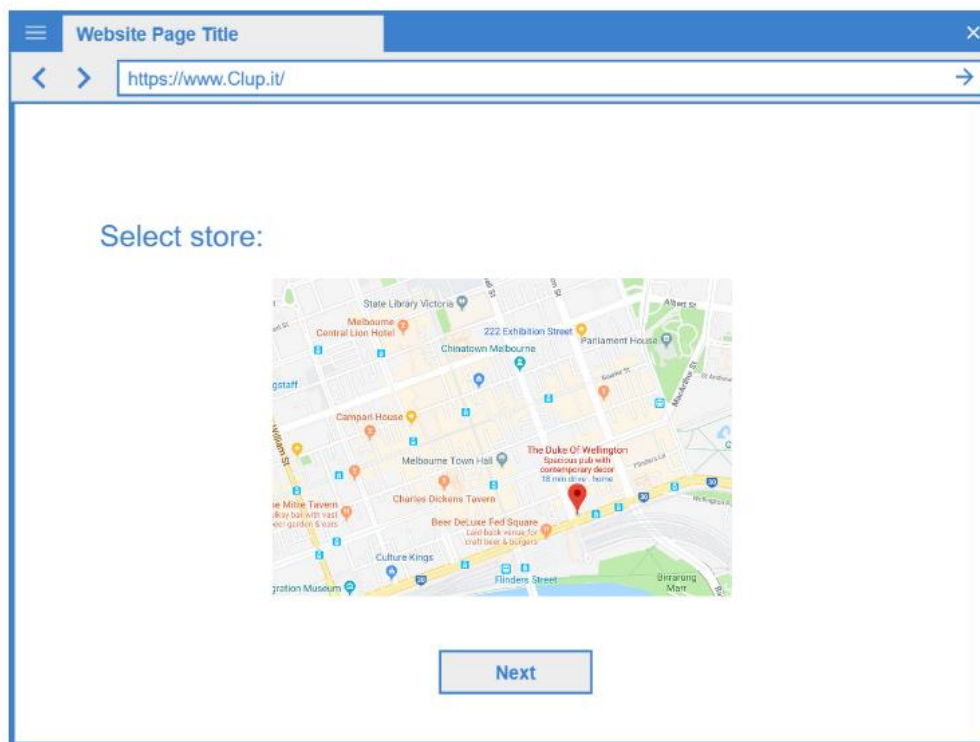
After selecting a queue ticket the customer can click on the exit queue button the exit from that queue.



3.1.1.5 Book a visit

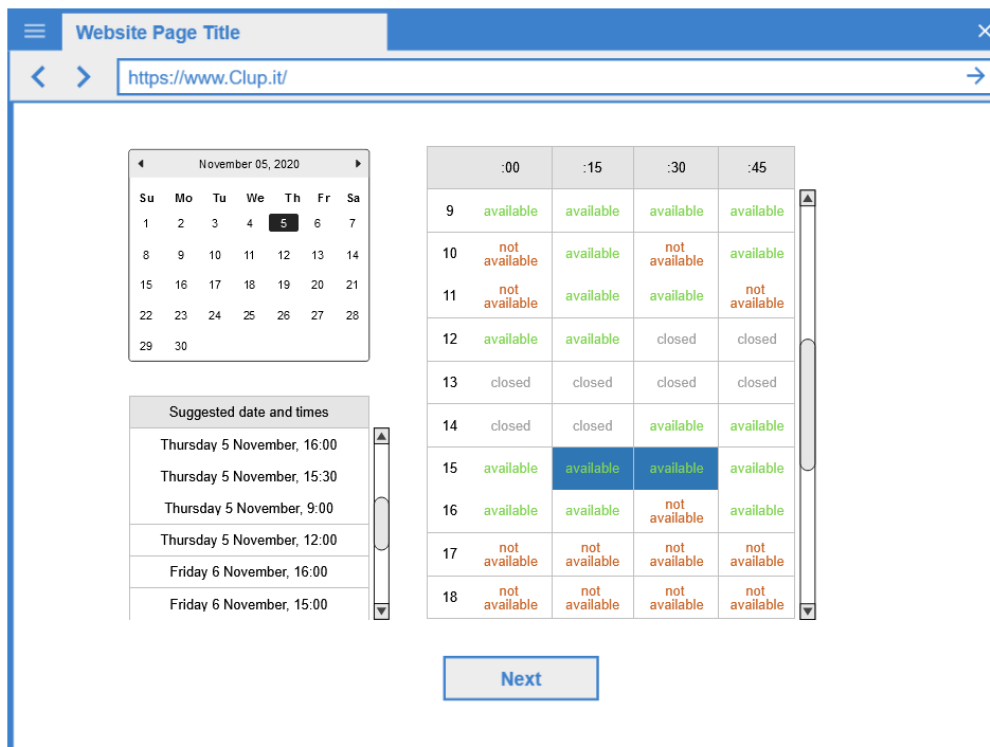
3.1.1.5.1 Select store

After pressing the book a visit button the customer can select the store from the map.



3.1.1.5.2 Select date and time

After selecting the store the customer can either select a suggestion of a date, time, and a store, or select the date and time on his own. To specify the date and time he must select free consecutive time slots. They are here shown in blue.



3.1.1.5.3 Select items and categories

After selecting the date and time or a suggestion the customer selects the items and categories he wish to shop for by searching them from the search bar.

The screenshot shows a web browser window with the address bar displaying 'https://www.Clup.it/'. The page title is 'Website Page Title'. The main content area has a heading 'Select item or category:'. On the left, there is a table titled 'selected items and categories' with the following items: beer brand A, meat, and dairy. On the right, there is a search bar with the text 'be' and a magnifying glass icon. Below the search bar is a list of suggestions: beer brand A, beans, beats, beacon brand B, and beef. To the right of the list is a circular button with a plus sign. At the bottom center of the page is a 'Next' button.

3.1.1.5.4 Select group size

After selecting items and categories the customer selects the number of people that will visit the store with him and can then either confirm the booking or cancel the operation.

The screenshot shows a web browser window with the address bar displaying 'https://www.Clup.it/'. The page title is 'Website Page Title'. The main content area has a heading 'Select the number of people that will visit the store:'. Below the heading is a numeric input field with the value '1' and up/down arrows. At the bottom of the page are two buttons: 'Cancel' and 'Submit'.

3.1.1.6 Delete a visit

After selecting a booking ticket the customer can click on delete visit to delete that ticket's visit.



3.1.2 Store manager

3.1.2.1 Setting of parameters

Here the store manager can set the following parameters: the maximum number of people allowed in a store sector at the same time, the length of each time slot, the maximum permitted duration of a visit, opening and closing time of the store. He can also select what store he intends to set the parameters for. He can click save once he is satisfied with his changes, cancel otherwise.

The screenshot shows a web browser window with the title 'Website Page Title' and the address bar displaying 'https://www.Clup.it/'. The main content area has a blue border and contains the following elements:

- max people in store:** A text input field with the value '30' and a dropdown arrow.
- time slot length: (in minutes):** A text input field with the value '10' and a dropdown arrow.
- max visit duration: (in minutes):** A text input field with the value '40' and a dropdown arrow.
- Select store:** A button with a dropdown arrow.
- Calendar:** A calendar for August 2016. The days of the week are labeled S, M, T, W, T, F, Sa. The dates 1 through 31 are displayed in a grid. The date 1 is highlighted in red.
- opening time / closing time:** A table with two columns: 'opening time' and 'closing time'. The first row shows '8:30' and '12:30'. The second row shows '13:30' and '18:30'.
- Buttons:** 'set for schedule', 'set for a day', 'Cancel', and 'Save'.

3.2 Mobile app

3.2.1 Login

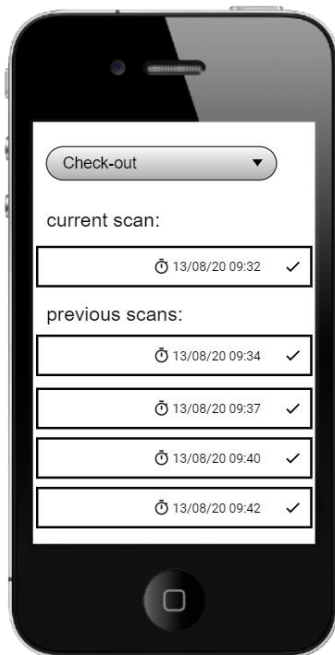


3.2.2 Checkpoint controller

3.2.2.1 Checkpoint controller at entrance

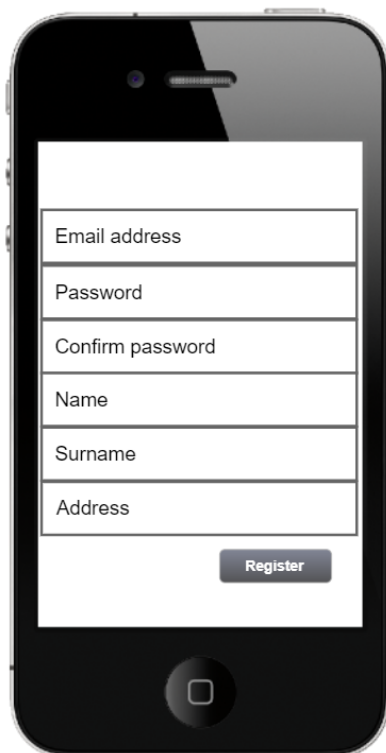


3.2.2.2 Checkpoint controller at exit

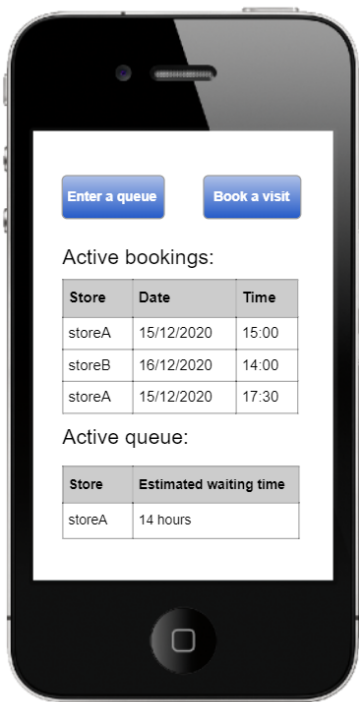


3.2.3 Customer

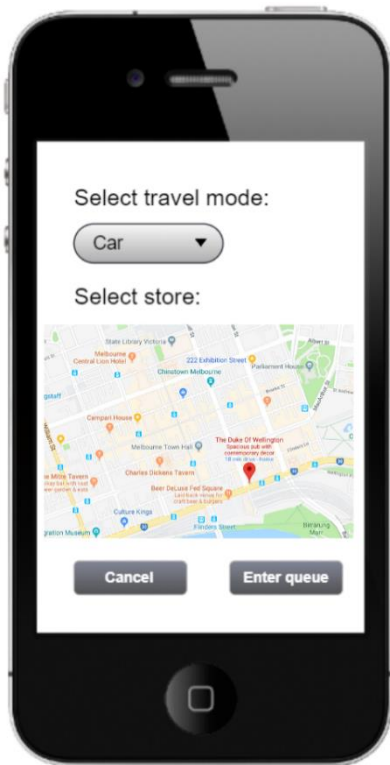
3.2.3.1 Registration



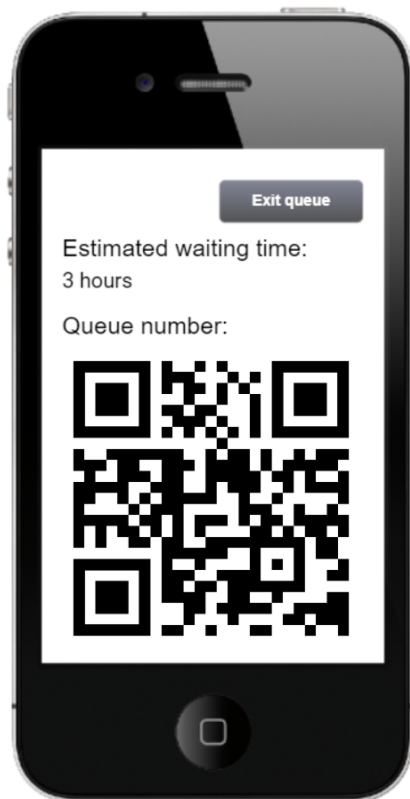
3.2.3.2 Home



3.2.3.3 Enter a queue

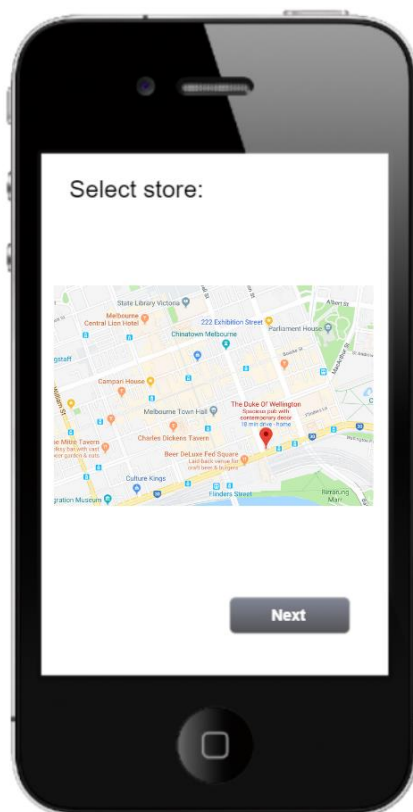


3.2.3.4 Exit a queue

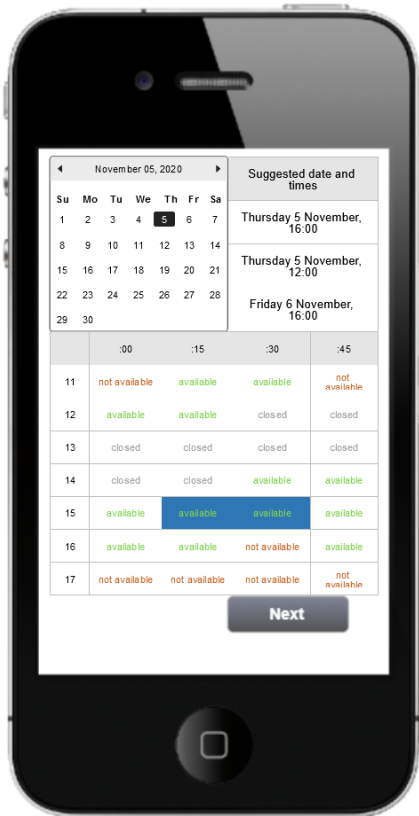


3.2.3.5 Book a visit

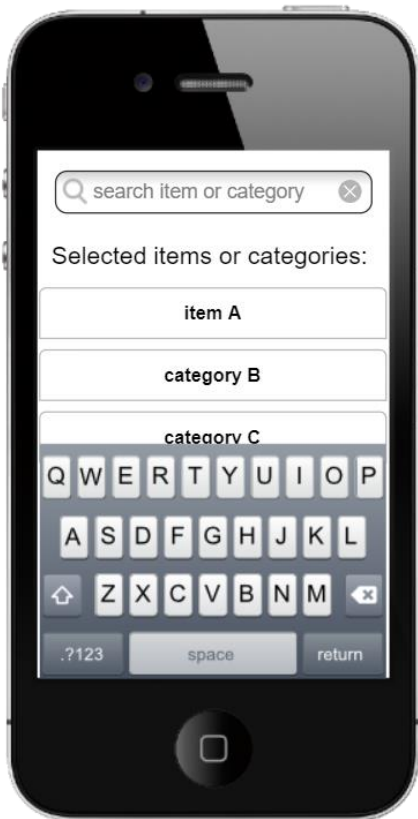
3.2.3.5.1 Select store



3.2.3.5.2 Select date and time



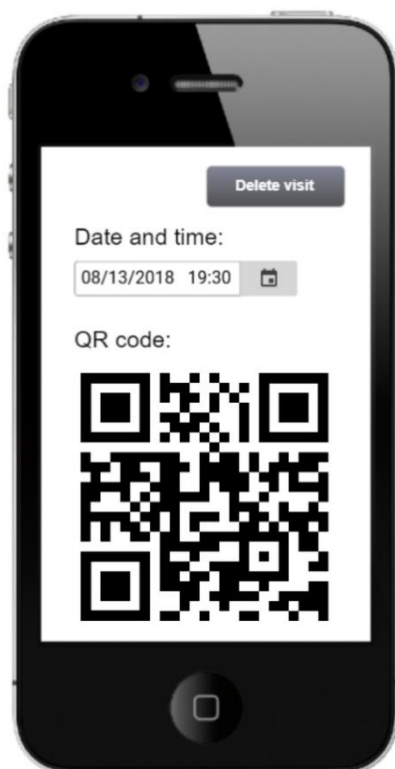
3.2.3.5.3 Select items and categories



3.2.3.5.4 Select group size

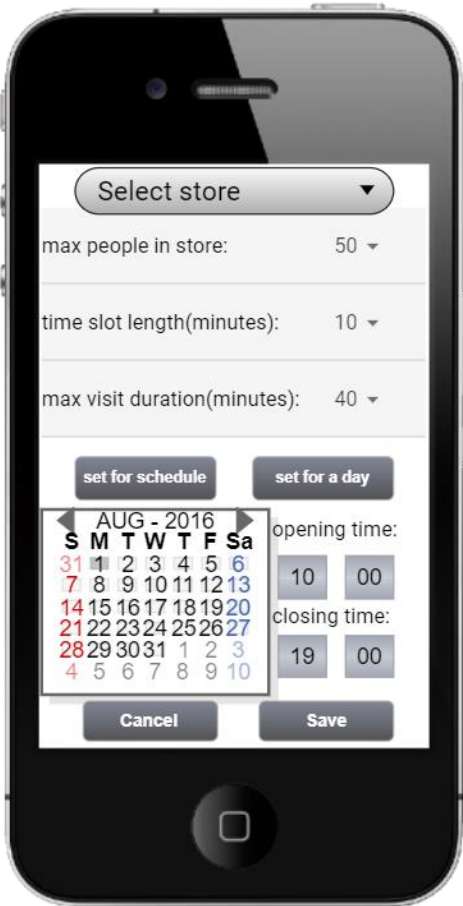


3.2.3.6 Delete a visit



3.2.4 Store manager

3.2.4.1 Setting of parameters



4

Requirements traceability

The DBMS and the `DataAccessObject` are needed for every requirement, as for each requirement there is the need to read or write the database. Therefore, they will not appear on this table to avoid clutter.

	Requirement	Components
R1	A user who wants to use the system online as a customer must register for free. Further uses require customers to login with valid credentials	<ul style="list-style-type: none">• ClientManager• UserManager
R2	Store managers and checkpoint controllers need to log in as well, but their accounts are created by a sys-admin. Thus they do not need to register as store managers or controllers	<ul style="list-style-type: none">• ClientManager• UserManager
R3	After logging in, a user can only access the functionalities that are specific of their role	<ul style="list-style-type: none">• ClientManager• UserManager
R4	The system will display a digital map with all the available stores in their area. Customers can select a store where they intend to make a purchase, among the displayed ones	<ul style="list-style-type: none">• MapService• MapConnector• StoreMapProvider• QueueHandler• BookingHandler
R5	Customers can join a digital FIFO queue for a store. In order to join such queue, they also need to specify how they intend to reach the store. After joining a queue, the system will send the user a digital ticket. A user can have at most one valid ticket for a given store at any time, i.e. it is not allowed to get another ticket for a store S while being in the digital queue for S ₂ : their requests for another ticket is simply overwritten (if the selected store is the same one for which the customer is already queuing, the request is simply denied).	<ul style="list-style-type: none">• QueueHandler• QueueManager
R6	A digital ticket consists of a number representing the position in the queue and a QR code. For any	<ul style="list-style-type: none">• QueueManager

	two customers waiting in the same queue, their waiting numbers are not equal	
R7	If a customer A is in a digital queue, the system will display an interval (ti, tf) representing the estimated waiting time. Such interval is calculated as a 95% confidence interval for the real waiting time T.	<ul style="list-style-type: none"> • QueueHandler • Estimator • MapService • MapConnector
R8	When the estimated time calculated at point 7 is less or equal than the estimated travel time for a customer A, they will receive a notification telling them to reach the store	<ul style="list-style-type: none"> • QueueHandler • Estimator • MapService • MapConnector • NotificationHandler
R9	Customers can retrieve a paper printed ticket at the store. Such ticket is identical to a digital ticket for what concerns its validity constraints	<ul style="list-style-type: none"> • QueueManager • TicketMachine
R10	The system will “call” waiting numbers by displaying them on a monitor at the entrance of the store	<ul style="list-style-type: none"> • QueueManager • QueueHandler
R11	Assuming that at most N people are allowed to be in the store at the same time, and M people are currently in the store, the system will call N-M numbers sequentially. If $N-M > 1$, then the calls will have a temporal distance of two minutes	<ul style="list-style-type: none"> • QueueManager • TicketControl
R12	A non-scanned ticket for a queue is valid for store S if and only if it was issued for S and its number has not been called by S yet or if the call happened no longer than 2 minutes before. In any other case, the ticket is marked as invalid, and the next waiting number will be called	<ul style="list-style-type: none"> • QueueManager • TicketControl
R13	A customer may also ignore the digital queue and book a visit for a store instead	<ul style="list-style-type: none"> • BookingManager • BookingHandler
R14	If the functionality specified at point 13 was selected, the system will display a time table on the customer’s device, plus a set of pre-calculated suggested visits. Each day is divided in time slots of equal length. The customer may either select a finite number of contiguous and free time slots for their visit, or one of the suggested visits	<ul style="list-style-type: none"> • BookingHandler • StoreManager
R15	In case the customer chooses to specify their own time interval, the total time they specify must not exceed a time limit established by the store manager. Furthermore, customers can only choose	<ul style="list-style-type: none"> • BookingHandler • StoreManager

	time slots that start and end after the opening time and before the closing time of the selected store. Finally, the visit must not overlap with any other visit booked by the customer.	
R16	While booking the user may input a list of items and categories of items that they intend to buy. Their app will show a list of categories and items that they can choose	<ul style="list-style-type: none"> • BookingHandler • StoreManager
R17	Then the system will send the user a QR code that certifies their booking	<ul style="list-style-type: none"> • BookingManager • BookingHandler
R18	A non-scanned QR code for a visit is valid for store S if and only if it was issued for S and either the selected time of arrival has not arrived yet or no more than 5 minutes have passed since the selected time of arrival	<ul style="list-style-type: none"> • BookingManager • TicketControl
R19	Checkpoint controllers can scan a customer's QR code upon letting them into the store. The system will check the validity of the code, and will display an error message if the token has lost its validity. After being scanned and approved, the token is marked as invalid, and thus cannot be reused for entrance	<ul style="list-style-type: none"> • BookingManager • QueueManager • TicketControl
R20	A store manager who has previously logged in may set parameters for their store only. Such parameters are: the maximum number of people allowed in a store sector at the same time, the length of each time slot, the maximum permitted duration of a visit, opening and closing time of the store	<ul style="list-style-type: none"> • StoreManager • ParameterEditor
R21	If a customer has used their app for longer than one month, the system will send them a notification once every two days. Such notification contains a list of 3 stores that are within their area and that have free time slots. The stores whose data is sent are always the 3 stores for which the sum of free time intervals at the time of sending is maximum	<ul style="list-style-type: none"> • NotificationHandler • StoreMapProvider • MapService • MapConnector
R22	A checkpoint controller can also scan the customer's QR code when they are exiting the store. However, this time the system will not check the validity of the token	<ul style="list-style-type: none"> • TicketControl • QueueManager • BookingManager

R23	Each booking requires the specification of the number of people that will make the purchases within the same group	<ul style="list-style-type: none"> • BookingManager • BookingHandler • StoreManager
R24	Customers are allowed to delete a booking at most 30 minutes before the expected time of the booking. A customer can also exit their queue at any time if they wish.	<ul style="list-style-type: none"> • BookingManager • BookingHandler • QueueManager • QueueHandler

In order to fit all the requirements the following matrix has been split in two tables:

	requirements											
components	1	2	3	4	5	6	7	8	9	10	11	12
ParameterEditor												
ClientManager	X	X	X									
TicketControl											X	X
BookingHandler				X								
NotificationHandler								X				
QueueHandler				X	X		X	X		X		
TicketMachine									X			
StoreManager												
StoreMapProvider				X								
BookingManager												
QueueManager					X	X			X	X	X	X
Estimator							X	X				
UserManager	X	X	X									
DataAccessObject	X	X	X	X	X	X	X	X	X	X	X	X
DBMS	X	X	X	X	X	X	X	X	X	X	X	X
MapConnector				X			X	X				
MapService				X			X	X				

	requirements											
components	13	14	15	16	17	18	19	20	21	22	23	24
ParameterEditor								X				
ClientManager												
TicketControl						X	X			X		
BookingHandler	X	X	X	X	X						X	X
NotificationHandler									X			
QueueHandler												X
TicketMachine												
StoreManager		X	X	X				X			X	
StoreMapProvider									X			
BookingManager	X				X	X	X			X	X	X
QueueManager							X			X		X
Estimator												
UserManager												
DataAccessObject	X	X	X	X	X	X	X	X	X	X	X	X
DBMS	X	X	X	X	X	X	X	X	X	X	X	X
MapConnector									X			
MapService									X			

Since every row and column in the matrix has at least an X no component is useless, and every requirement needs some component to be realized.

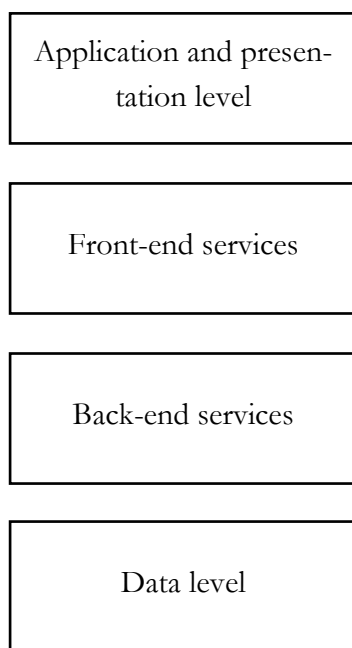
5

Implementation, integration and test plan

The software system must be implemented and integrated following some specific plans in order to reduce the effort in writing correct code. An implementation, integration and test plan should help developers to build safer and cleaner code. Implementation and integration plans are strictly correlated; however, they're presented separately.

5.1 Implementation plan

The implementation of the application has to follow a bottom-up approach mixed with a thread approach. You should start from the lowest software levels, i.e. the data level, then rise up progressively to the application and presentation level.



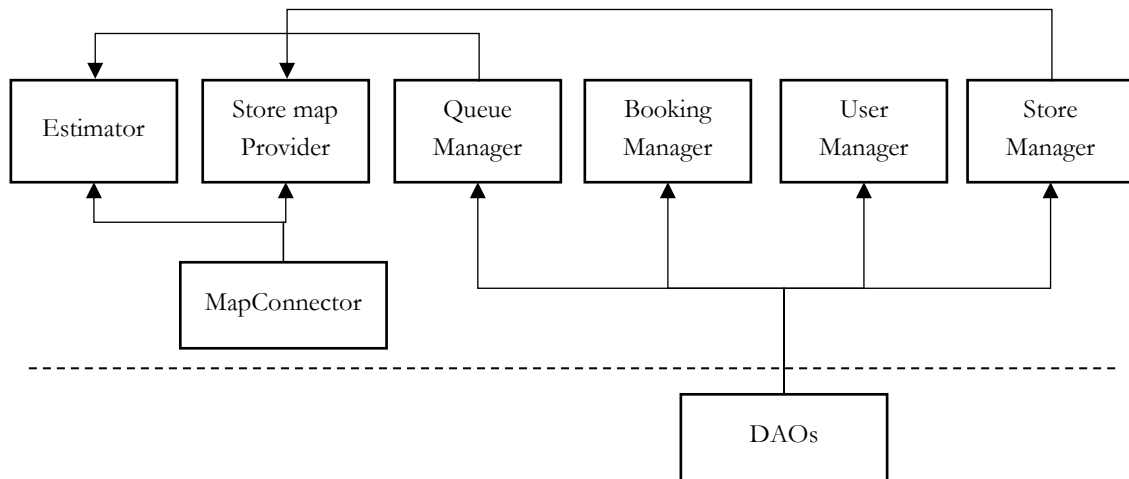
The implementation plan chosen for this application is to implement the whole data level, then implement the immediately higher level, which is the back-end services, and so on. During the implementation of a level, the approach is still bottom-up. First, we implement in parallel classes that are not dependent on other classes. Then, when those classes are ready, we step to the classes that use only the already implemented ones. While implementing these other classes we implement them in parallel again. When we have finished, we again step to the next classes using the already implemented ones, and so on until the program is fully implemented. Then, when the interfaces required by a class are all implemented,

you can start implementing the class itself.

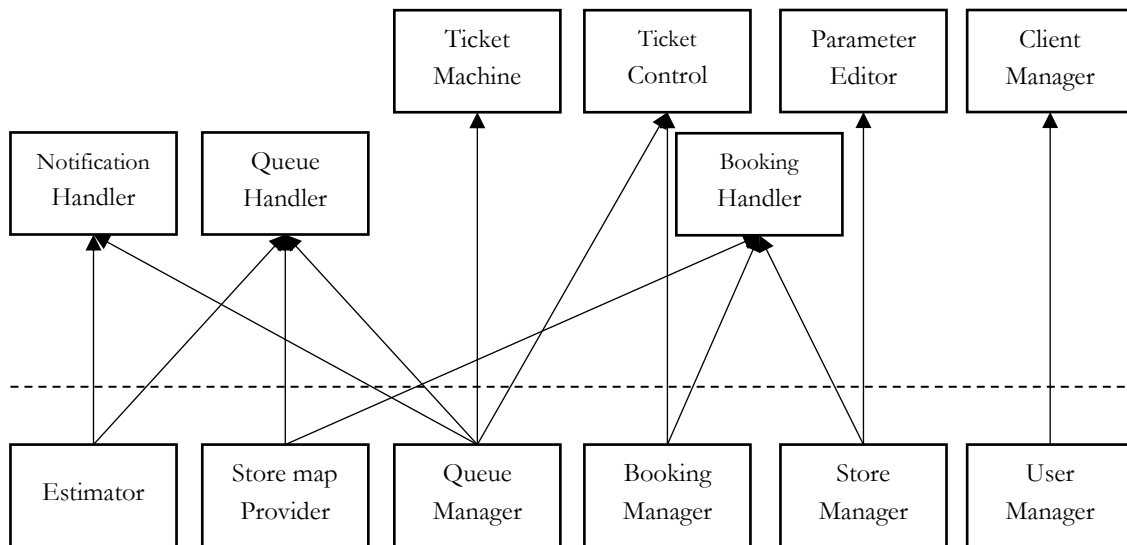
In this document we do not directly specify which classes should be written, but only use the concept of component to specify the implementation plan. However, it is important to notice that the general methodology must be maintained at a finer level (the implementation level

where classes are specified). Here is introduced a diagram explaining all dependencies between components. In the diagrams the direction of the arrows is the dependency between modules: if module A points to B, then B needs A to be implemented first.

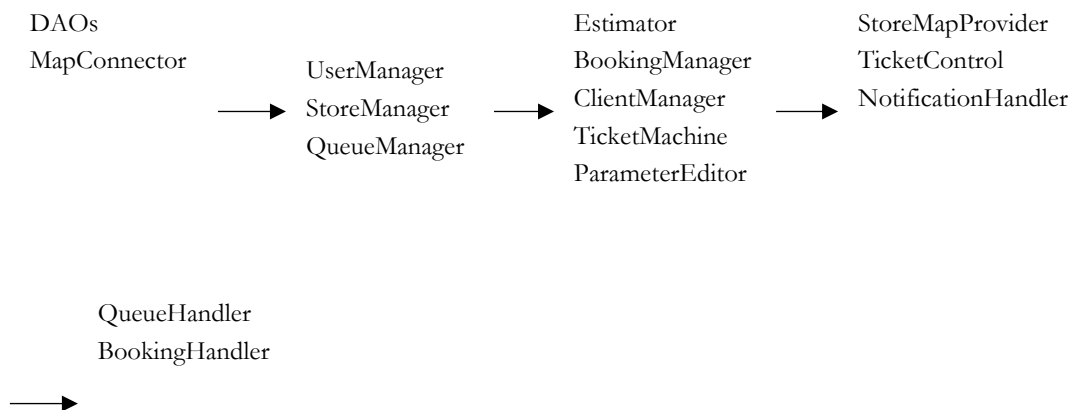
High attention should be paid to the fact that the DBMS is not implemented (since we use some external implementation), although it requires the database schema. The schema should be designed and implemented before the implementation of software components. However, *an agile approach is strongly suggested* to cope with the possible errors and bugs which can be found.



These are the dependencies between the data level components and back-end services and between back-end services (**MapConnector** is a back-end service). The dependencies between back-end services and front-service are fairly complex, this is the reason why we show them in a separate diagram (the following one).



The dependency between the front-end services and the web-application components is not specified directly, in fact the dependency is connected with the decision of the language and the framework to use. The implementation plan, more than implementing all the back-end services and then the front-end services, establishes an order of implementation of back-end components so that front-end components and back-end components may be developed at the same time. Some front-end components use only one back-end services and some only use two. This led us to observe that an approach similar to the thread approach can be applied. A suggested implementation plan is the following.



With that approach the implementation of the component can be still using a bottom-up approach mixed with a sort of thread approach, which gives the possibility to use some functionalities of the system before it is finished. After the implementation of the server components, the web application and the mobile application are implemented.

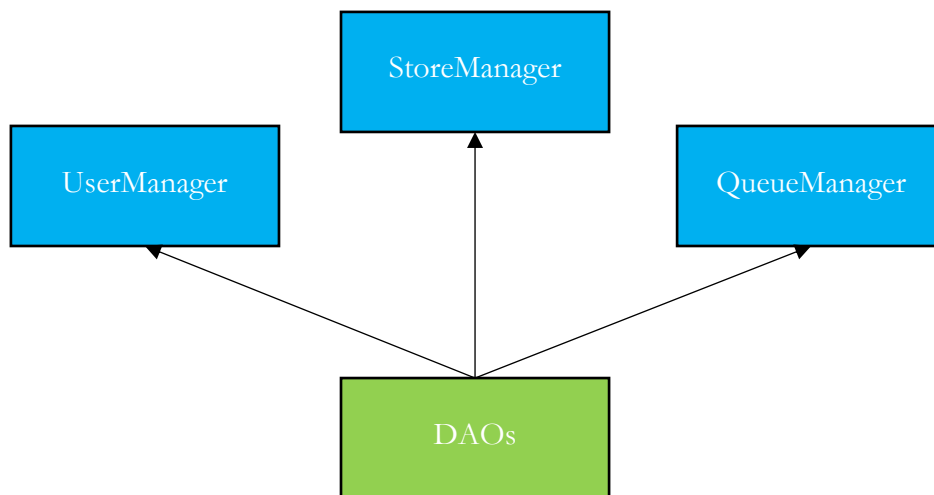
5.2 Integration plan

Integration plan is fully guided by the implementation plan, since the integration plan describes the way in which the components have to be integrated to build up the entire system. Since the implementation plan is a bottom-up approach mixed with a thread-like approach, integration is done quite easily. The integration between different components is going to be done incrementally. Components are developed mostly sequentially; this leads to apply sequentially the integration between on-development component with already built components. The overall integration consists of four phases for the business and data layers. There should be another phase consisting in the integration between the web and mobile applications and the front-end services. Web and mobile's components should be uncovered in another finer document like an implementation document.

Graphics used to express dependencies in the implementation phases follow this concept: green boxes represent already implemented components and blue components represent the components being implemented and imposed to integration.

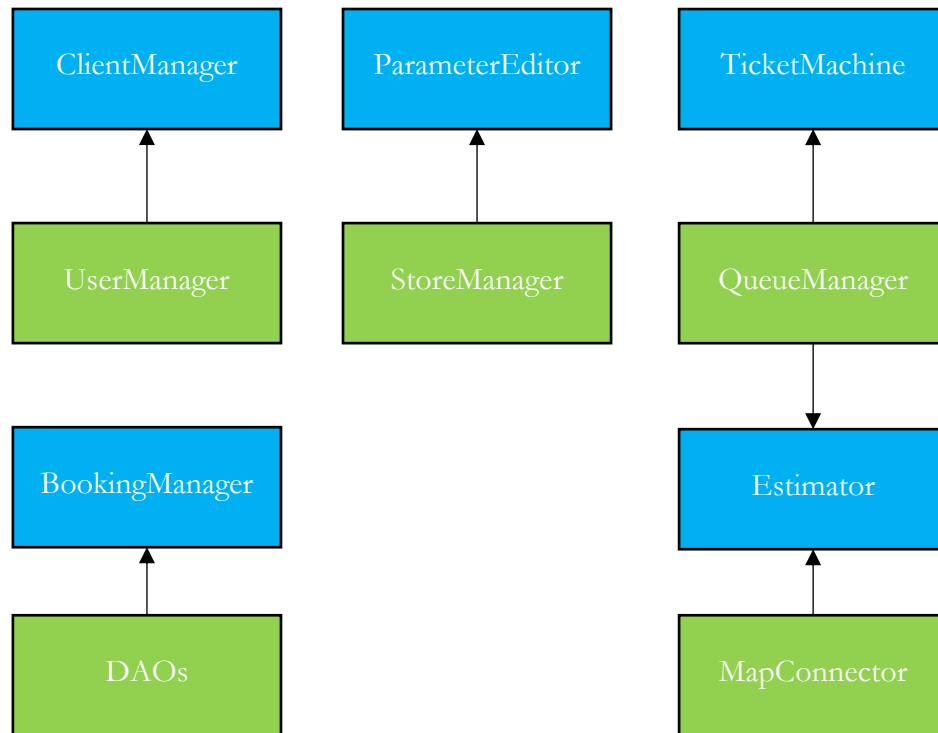
Phase I:

The first phase (which has to be handled during the development of the system) is the integration between the first back-end services of the business level and the components of the data-layer. Moreover, the integration between the `MapConnector` and the other back-end services is not happening since in the first phase neither `Estimator` nor `StoreMapProvider` are implemented. This integration must be carried out during the implementation of the back-end services and not only at the end of their implementation.



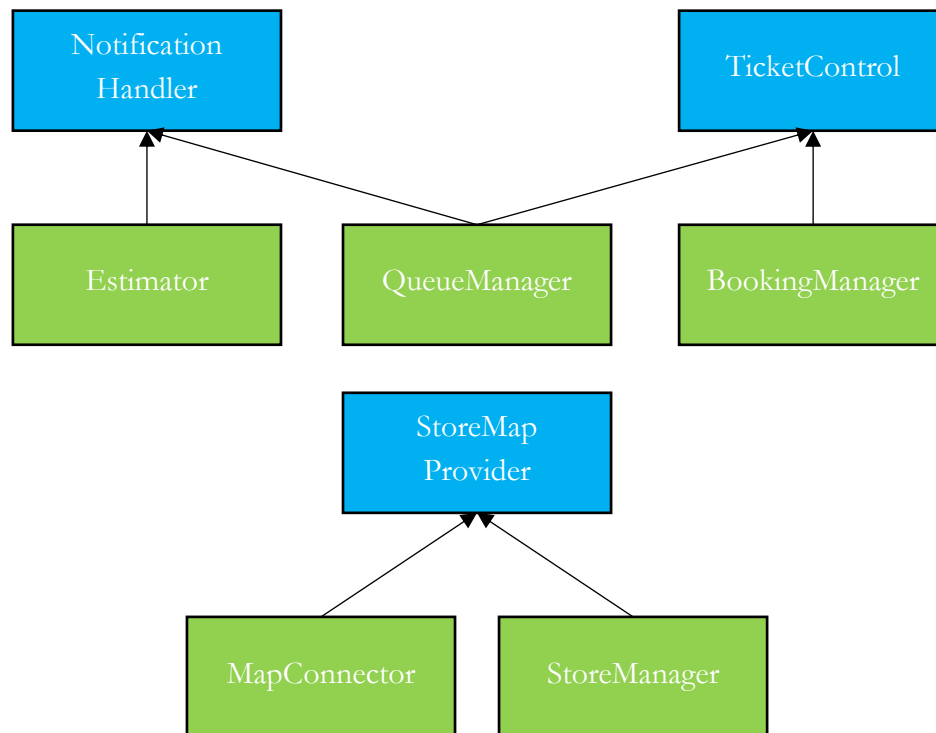
Phase II:

The second integration phase is one of the most critical. It is the first phase in which a front-end component is integrated with the back-end component it uses. This part should be followed by the implementation of a part of the web application to allow the stakeholders to explore the system provided. This part involves both the implementation between back-end services and data layer and front-end services and back-end services. As in the other phases the integration should be done incrementally.



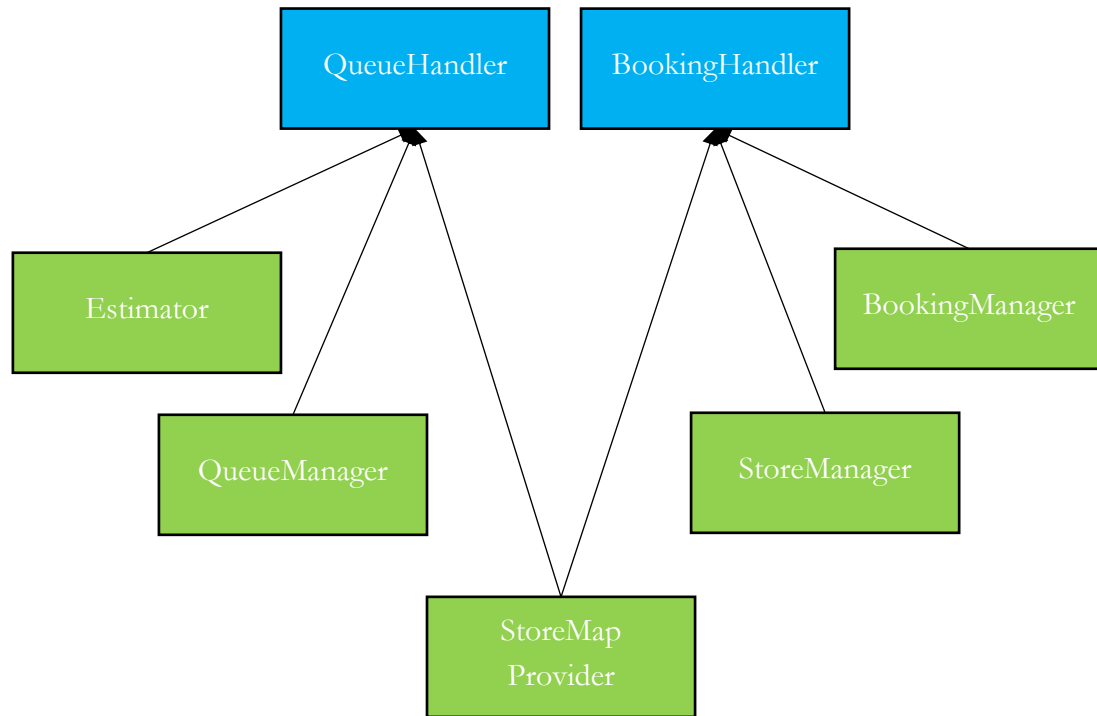
Phase III:

This integration's phase closes the integration between data layer and business layer with the integration between the last back-end service implementation and implements other two important front-end services for the application: `TicketControl` and `NotificationHandler`. This phase should be extended in further documents stating the integration between the front-end components integrated in phase II and the components of the web application with the chosen language and framework.



Phase IV:

The fourth phase is the last integration phase of the business components. In this phase the integration should be majorly passed to integrate front-end services with the web-application and mobile-application implementation framework dependant. The latest front-end services implemented only depends by back-end services. Here we have a diagram of the integration to perform enhancing the dependencies as in the other diagrams.



5.3 Testing plan

In this plan we describe how to perform the following types of testing:

- Functional testing;
- Performance testing.

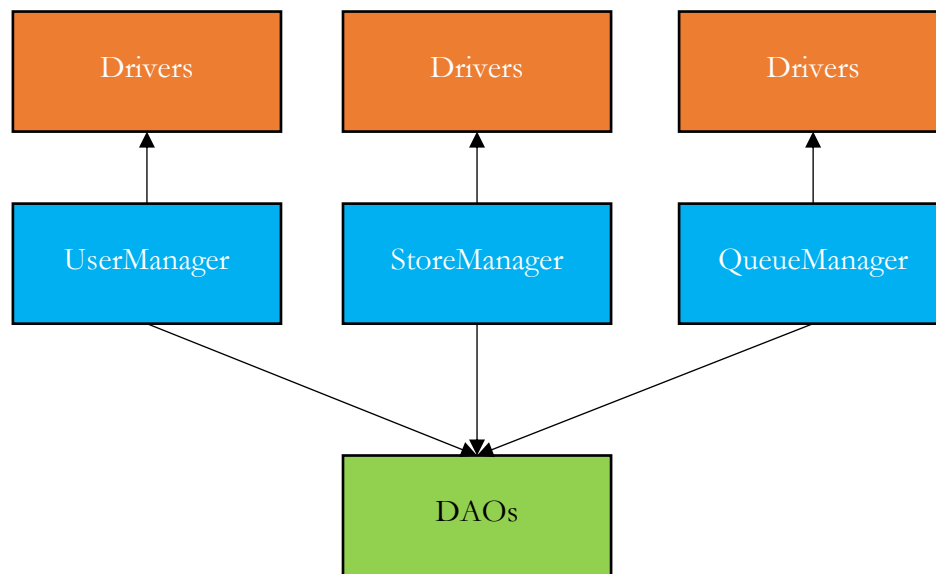
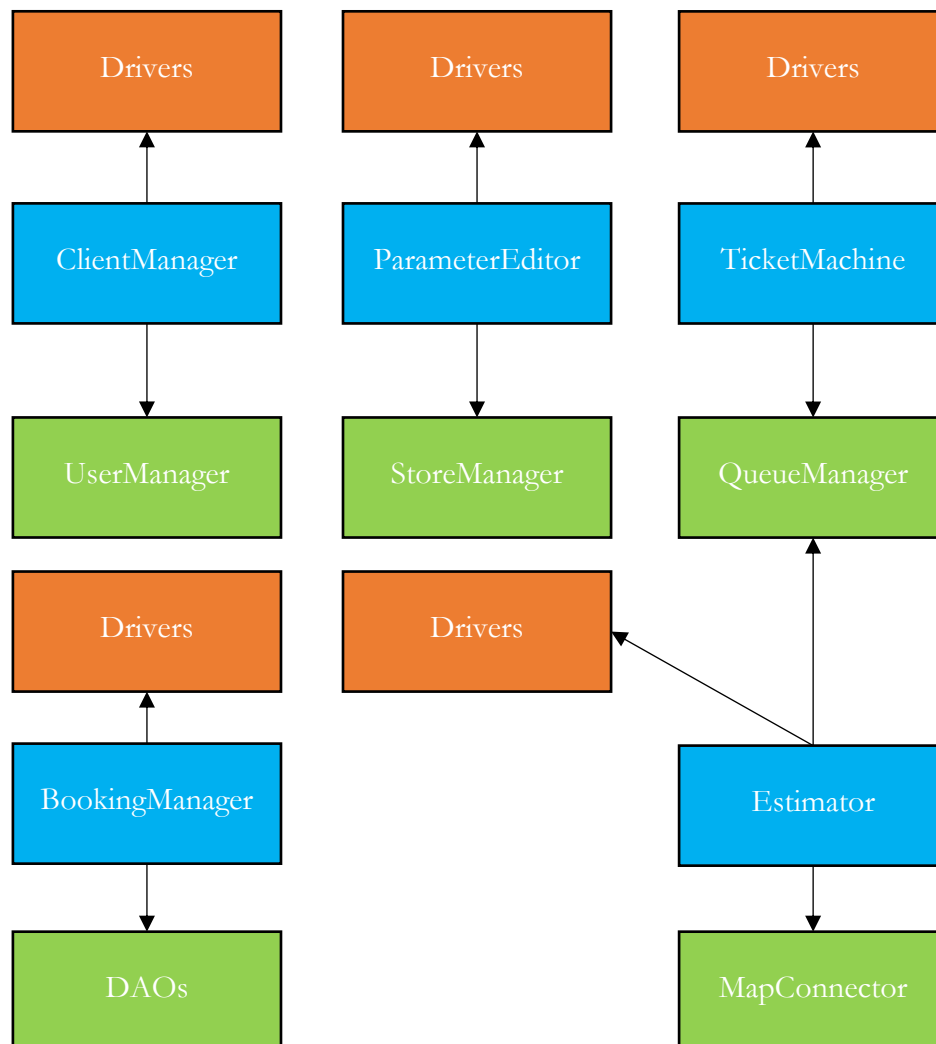
Testing must be performed as soon as possible in the application and agile approach is strongly suggested for the development and the lifecycle of the development. The testing performed should be automated whenever it is possible and test cases must be written right after the source code. We can abstract the concept of unit testing and “Unit test” every component before integrating it with other components. When a component code has to be modified, it needs to be tested automatically before sharing changes also to other team members. No untested software’s change can be applied. Saying that no untested software’s change can be applied, we mean that every test case related to the code must be performed. Performance testing needs to be tested on functioning code and must be performed at the end of the day when the code changed. The needing of effectuate performance testing at the end of a working day is required since performance test is a lot slower than functional testing. Here is described how test is carried out during the integration phases. Unit testing is not expressed in terms of diagrams since using a bottom-up approach implies you always need drivers and never need stubs.

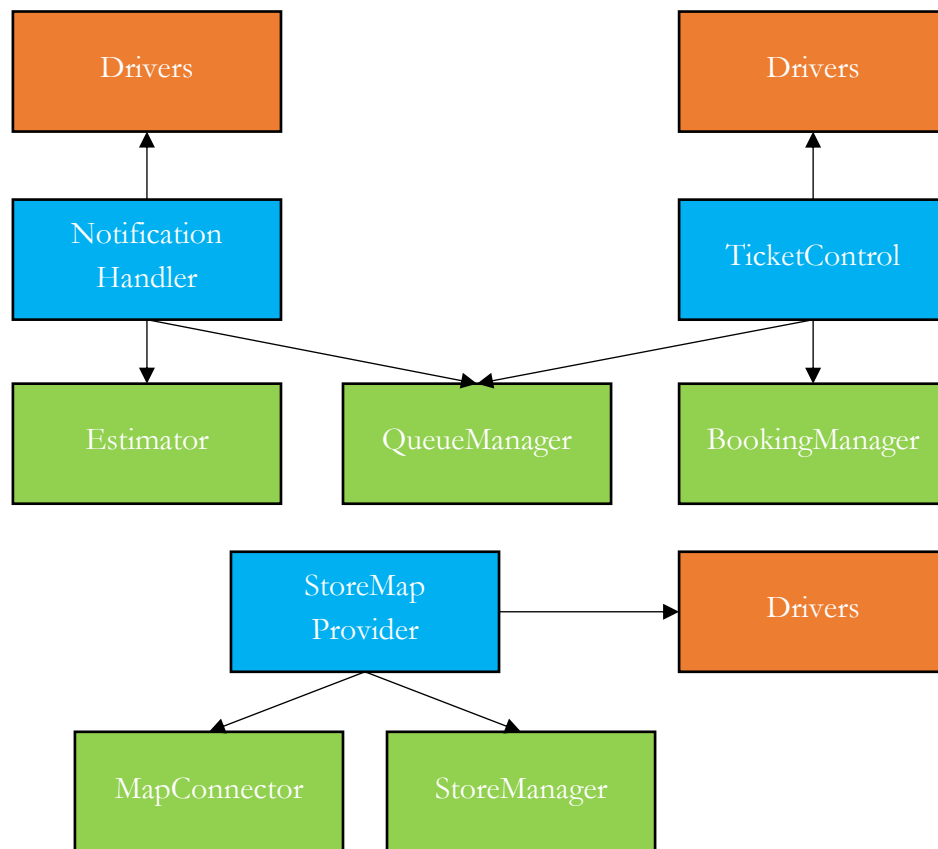
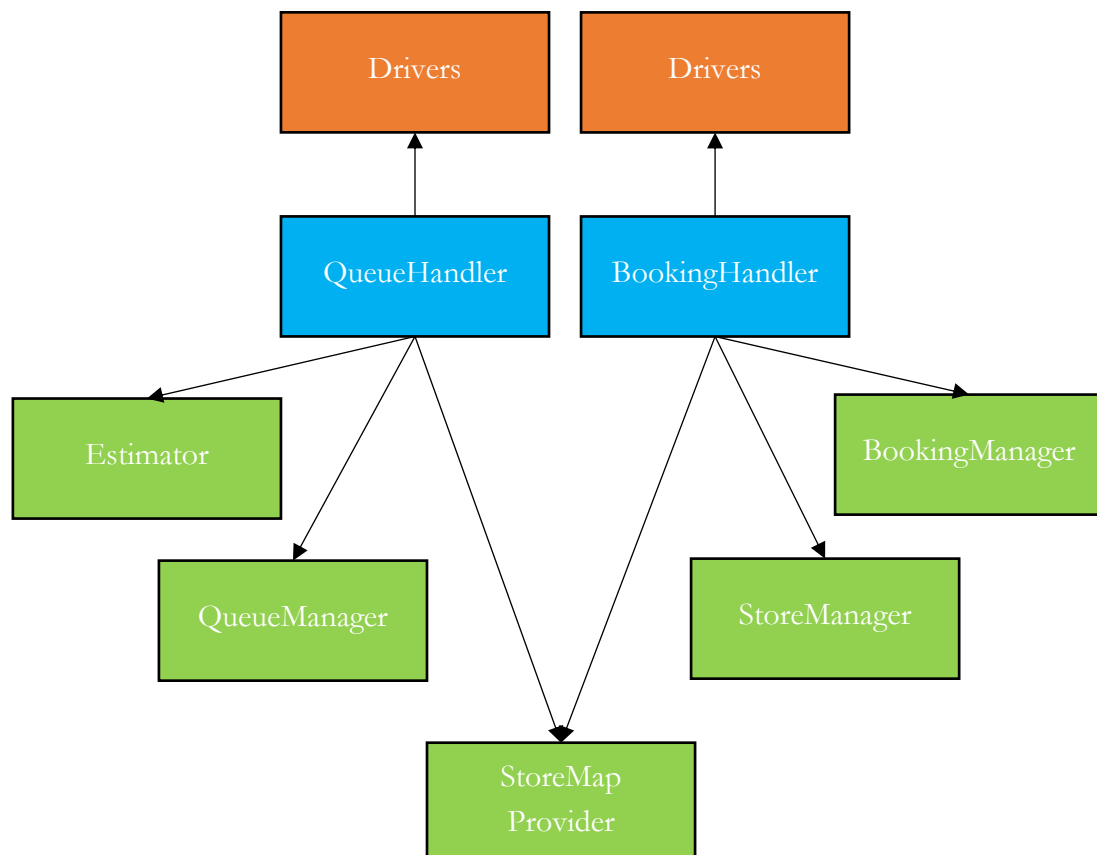
Testing phases are not described since they’re the representation of integration phases including how drivers are organized.

For the sake of simplicity here we do not list all the modules which use the tested components since there are dependency diagrams in the implementation and integration part. We refer to the set of all drivers when using “Drivers” in these diagrams. Here arrows represent usage relationship, not dependencies.

Phase 0:



Phase I:**Phase II:**

Phase III:**Phase IV:**

6

Effort spent

This part summarizes the effort spent by each member of the team in the documentation building process.

Davide Li Calsi

Introduction	0hrs
Overall description	5hrs
User interface design	0hrs
Requirements traceability	0hrs
Implementation, integration and testing plan	0hrs
Revision	7.5hrs

Andrea Alberto Marchesi

Introduction	0hrs
Overall description	15hrs
User interface design	22hrs
Requirements traceability	3hrs
Implementation, integration and testing plan	0hrs
Revision	1hrs

Marco Petri

Introduction	3hrs
Overall description	12hrs
User interface design	0hrs
Requirements traceability	0hrs
Implementation, integration and testing plan	4hrs
Revision	2.5hrs

All

Meetings	12hrs
----------	-------

7

References

In this chapter every reference which has been used to produce this document. We can identify two main types of source of information: documents and other sources. Other sources can be websites, videos and other, the meaning of document is intended to be clear. Here we cite tools and external references such websites and other sources of information:

1. UML diagrams (used as quick reference): <https://www.uml-diagrams.org/component-diagrams.html>;
2. UML component diagrams: https://www.ibm.com/support/knowledge-center/SS8PJ7_9.7.0/com.ibm.xtools.modeler.doc/topics/ccompd.html;
3. UML subsystem component: https://www.ibm.com/support/knowledge-center/SS8PJ7_9.7.0/com.ibm.xtools.modeler.doc/topics/csubsys.html.

Now that other sources of information have been cited, here the list continues with the list of documents used as reference to produce this document:

4. IEEE 1016-2009: <https://standards.ieee.org/standard/1016-2009.html>;
5. Software engineering – Hans Van Vilet pp. 285, 605-614: ISBN 9780470031469.