CLup project
Luca De Martini, Alessandro Duico

Design Document

Deliverable:	DD
Title:	Design Document
Authors:	Luca De Martini, Alessandro Duico
Version:	1.0
Date:	December-2020
Download page:	https://github.com/luca-de-martini/DeMartiniDuico-sw2
Copyright:	Copyright © 2020, Luca De Martini, Alessandro Duico - All rights reserved

Contents

List of Figures

List of Tables

1 Introduction

1.1 Purpose

This application aims to provide a digital alternative to the "handing numbers" approach and improve it by allowing people to take their number online, this way people only need to go to the shop when it is their turn to enter and they will not need to hang around the building.

The product will also allow shop managers to effectively monitor the entrances by scanning a QR code associated with the "number" to ensure the safety limits are being followed.

Additionally to the lining up mechanism the application will allow customers to book a visit in the future, which will improve the distribution of customers during the day and the week, since they will be able to choose a less crowded time slot.

CLup will also allow customers to choose the approximate time of their visit and the category of items they wish to buy, which allows making more accurate predictions of the waiting times and, by knowing the areas that they will visit, to better utilize the space in the building, while respecting health and safety measures.

1.2 Scope

The CLup system aims to implement the following functionalities:

- Registration and Login
- Staff checks a token
- Staff emits a substitute ticket
- Customer books a visit
- Customer gets a ticket
- Manager adds and configures a Shop
- Manager checks Shop occupancy

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

Third party company A company that wants to use our services

Ticket A digital proof of presence in the waiting list

Booking A digital proof of reservation for a visit to the Shop in a specific time slot

Token Either a ticket or a booking, it's associated with a code which can also be represented as a QR code

Staff Company personnel which is responsible of checking entrances

Shop Point of sale of the Third party company

1.3.2 Acronyms

RASD Requirement Analysis and Specification Document

DD Design Document

API Application Programming Interface

REST Representational state transfer

HTTP Hypertext Transfer Protocol

HTML Hypertext Markup Language

UML Unified Modeling Language

1.3.3 Abbreviations

[Gn] n-th goal.

[Dn] n-th domain assumption.

[Rn] n-th functional requirement.

1.4 Revision history

v. 1.0 - 9/01/2021 Initial release

1.5 Reference documents

BEEP channel - Mandatory Project Assignment

Luca De Martini, Alessandro Duico - RASD

1.6 Document structure

This document provides a specification of the architecture of the *CLup* system, mainly intended for developers and testers. It complements the RASD with a more in-depth description of the components and their interaction, and the plan for implementation, integration and testing. Eventually, the solutions adopted to satisfy each of the requirements of the RASD are explained in detail. This document is structured in five chapters, that are briefly described here:

1. **Introduction**: the purpose of the document is presented, together with prerequisites for the correct interpretation of the document;

2. **Architectural Design**: shows the main components of the System and their relationships. This section also focuses on design choices and architectural styles, patterns and paradigms, along with a complete explanation of the reasons for their employment;
3. **User Interface Design**: expands the indications on the user interfaces and user experience, starting from those provided in the RASD;
4. **Requirements Traceability**: provides a map of the requirements presented in the RASD to the design choices of the DD;
5. **Implementation, Integration and Test plan:** presents the plan for the implementation and integration of the components and for the testing of the system.

2 Architectural Design

2.1 Overview

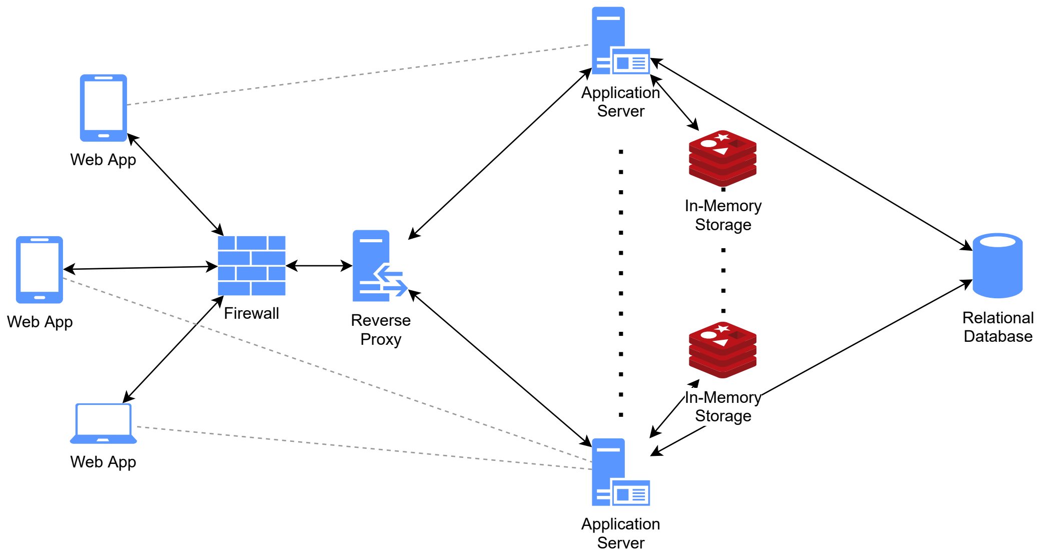


Figure 1: Architecture overview

The figure above shows an high level representation of the *CLup* system architecture. Two parts can be easily distinguished, separated by the firewall: the Client Web Application and the Application Server with its data storage. Further details on the System components and their interactions will be examined in the following sections.

2.2 Component view

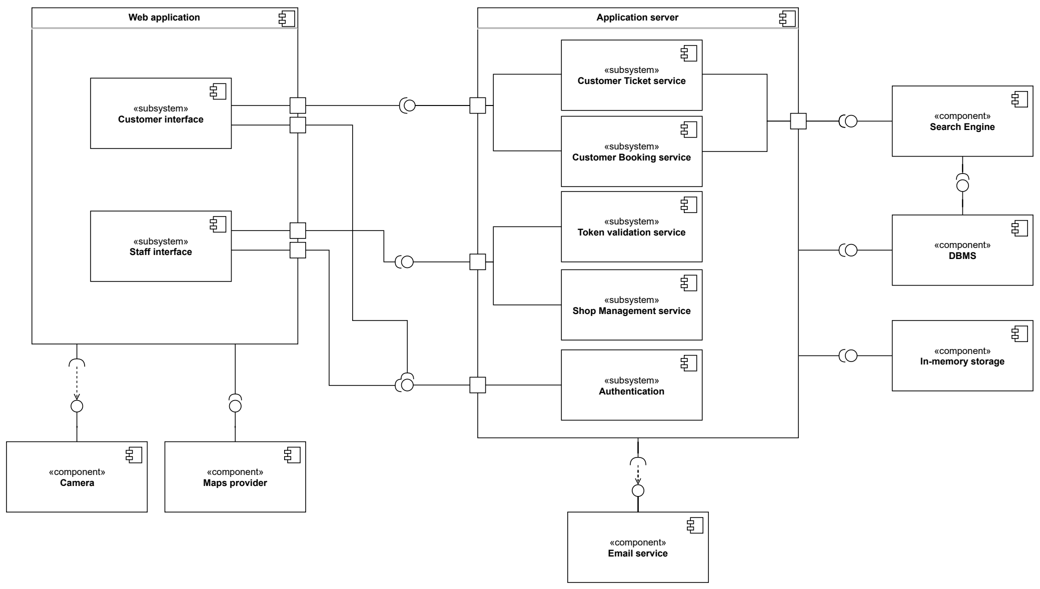

Web application
«subsystem» Customer interface
«subsystem» Staff interface
Application server
«subsystem» Customer Ticket service
«subsystem» Customer Booking service
«subsystem» Token validation service
«subsystem» Shop Management service
«subsystem» Authentication
«component» Search Engine
«component» DBMS
«component» In-memory storage
«component» Camera
«component» Maps provider
«component» Email service

Figure 2: Component diagram of the full system

In this section, an UML component diagram shows the internal structure of the *CLup* system. The architecture is simple, there are a small number of components and few external service providers. It can be divided in two higher level components: a Web Application and an Application Server, with the respective dependencies. As for component dependencies, the Web application uses a Camera module to scan the QR codes, and it can use an optiona Map service provider to show directions. The Application server uses a DBMS, an in-memory data store and a search engine. The relationships and interfaces between the components are represented through assembly connectors and delegation connectors. A detailed description of the components is included in the following paragraphs:

Web application
CLup is a distributed system, therefore part of the logic is included in the client. The Web Application client consists of two main components and some external interfaces:

- **Customer interface** is what Customers use to interact with the service. It processes the inputs, transparently querying the application server with transformed data, and it transforms the repsonses into a user frierndly representation. The application guides the user through the creation of a Ticket or a Booking while performing input validation to reduce the workload on the application server. It can interface with a *Maps provider* to display a Map with directions to the Shop.
- **Staff interface** is what the Staff uses to interact with the service. The behaviour is similar to that of the Customer interface, however the functionalities allow checking tokens and managing the shops. It interfaces with a *Camera* to scan the QR codes associated with tokens

Application server
The application server is accessed through a REST API. Its internal structure follows the Actor model (2.5.3) so it can be seen as a set of components which have no shared state. The actors can be organized and seen as three groups based on their functions:

- **Authentication** handles registrations and logins. It interfaces with the DBMS to validate and update credentials and with the Session Store to create and update Sessions. It can verify or update login credentials and authenticate a user Session.
- **Customer Ticket service** interface with the Search engine to allow users to find Shops. It uses Session storage to check authentication and update session data. It interacts with the DBMS to retrieve the persistent data necessary for the services. It allows users to create Tickets, search Shops and receive time estimates.
- **Customer Booking service** interface with the Search engine with the Ticket Service and has a similar policy to the Session storage and the DBMS. It computes the availability of Shop departments to decide whether to accept Booking requests.
- **Token validation service** It uses Session storage to check authentication and update session data. It interacts with the DBMS to retrieve the persistent data necessary for the services. It allows the Staff to validate Tokens.
- **Shop Management service** interfaces with the DBMS. It allows the Managers to add and edit Shops, check tehir current status and occupancy.

Application server external Interfaces
The application server has interfaces for its internal components in the form of REST API. The web application can interact with the Application Server by sending HTTP queries to the endpoints described in the API (2.5)

Search Engine
This component generates real-time search results to allow Customers to find a Shop by name.

DBMS
Stores the long term persistent data regarding Shops, Bookings, Tickets and Account data in a relational database.

In-memory Session Store
Keeps track of Sessions and stores data that should persist throughout the lifetime of a session. It provides a fast in-memory data structure that the Application server can use for non critical data to reduce the frequency of accesses to the relational database.

Email service
It provides the capability to send emails to new Customers to validate their email address.

2.3 Deployment view

The deployment view shows the organization of the distributed devices that make up the Clup service with a UML deployment diagram. The first tier is the *Presentation tier*, represented by the web application. The web application can be accessed either through a web browser as a normal website, or installed on a smartphone as a *Progressive Web App*. The middle tier, is the *Logic tier*, composed of the application server running the Clup artifact, the reverse proxy serving static pages, and the Redis session store. The last tier is the *Data tier* running a DBMS.

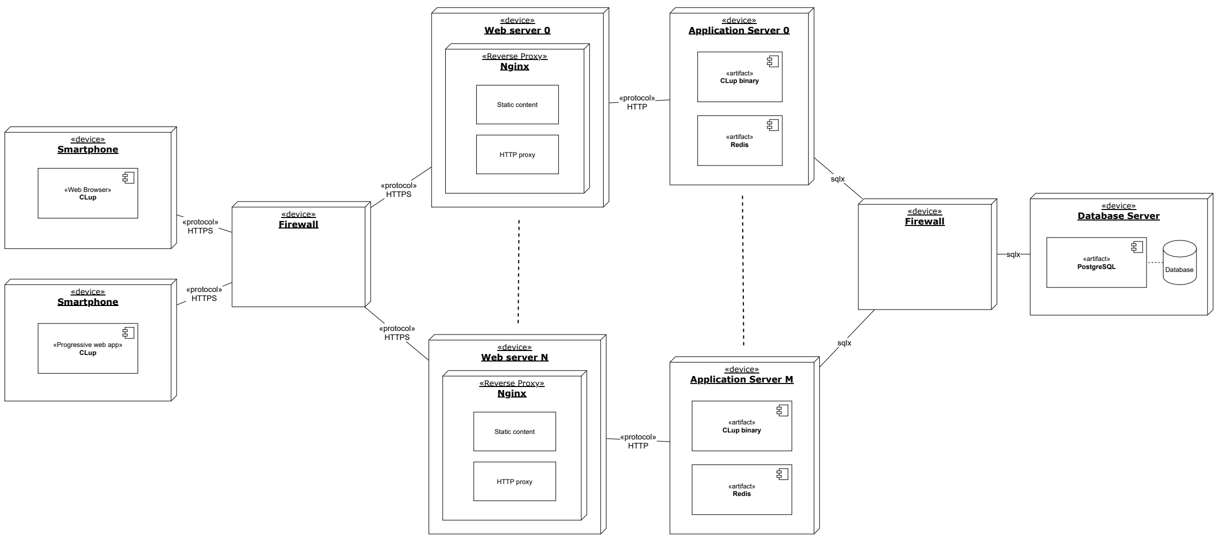


Figure 3: Deployment Diagram

CLup can also be deployed in a containerized environment as stated in the portability section of the RASD. This deployment scenario is easier to setup and adapts to many different device layouts. It can be deployed on a single machine running docker, a docker cluster or in the cloud. Configuring the service in a containerized environment is easier since installing dependencies and connecting components is automated.

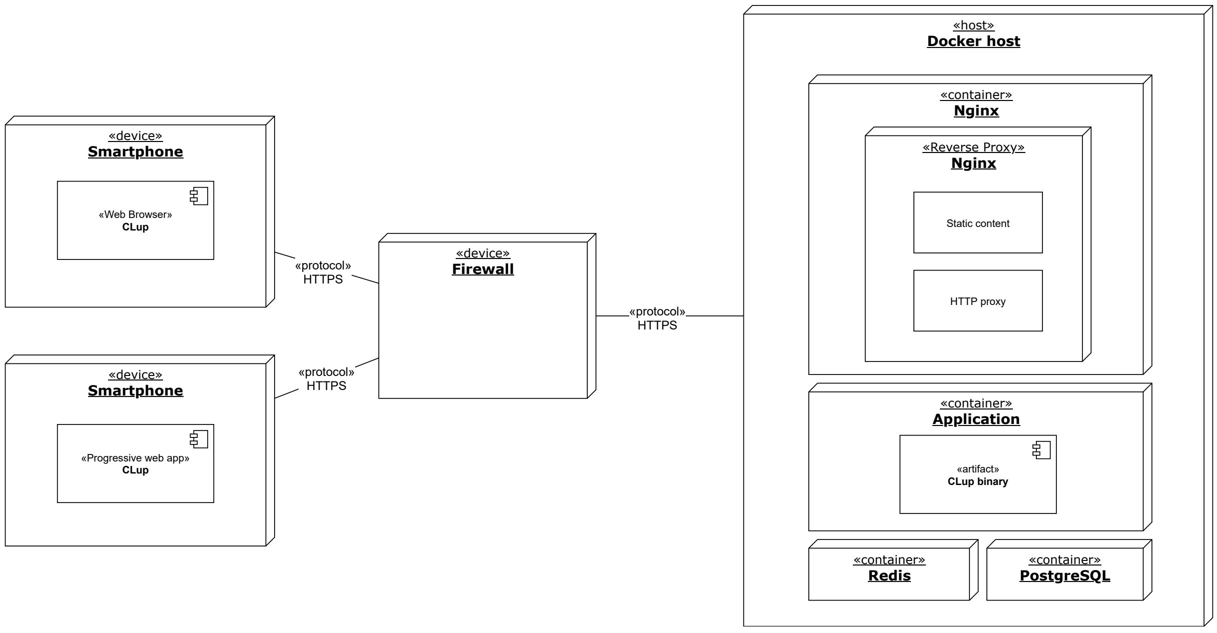


Figure 4: Deployment Diagram using Docker containers

The CLup application server should be deployable as separate artifacts for the components shown in Figure 2 to allow updating the services one at a time and limit service degradation in case of failures.

2.4 Runtime view

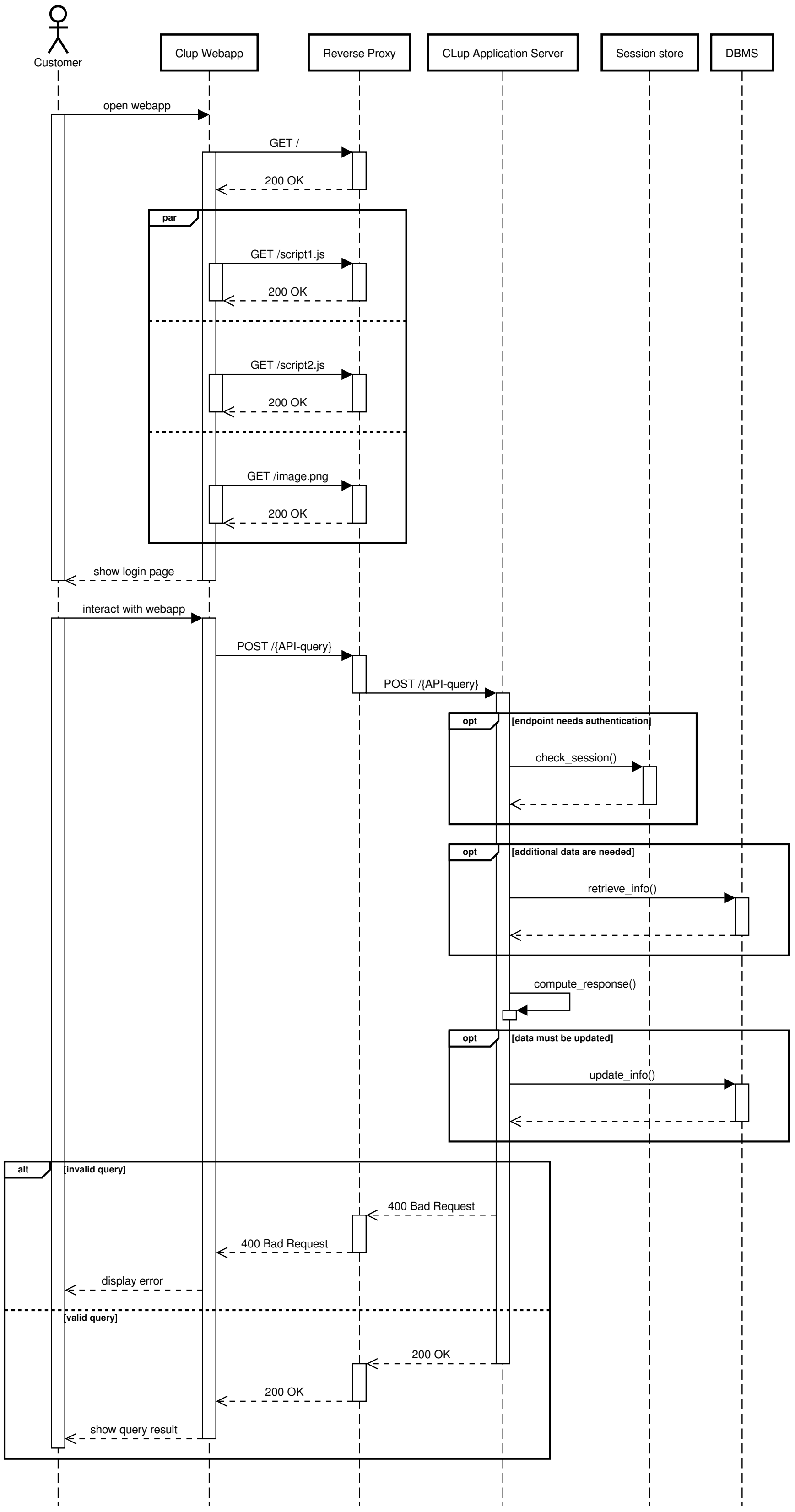


Figure 5: Generic API request (with detail of the Reverse Proxy)

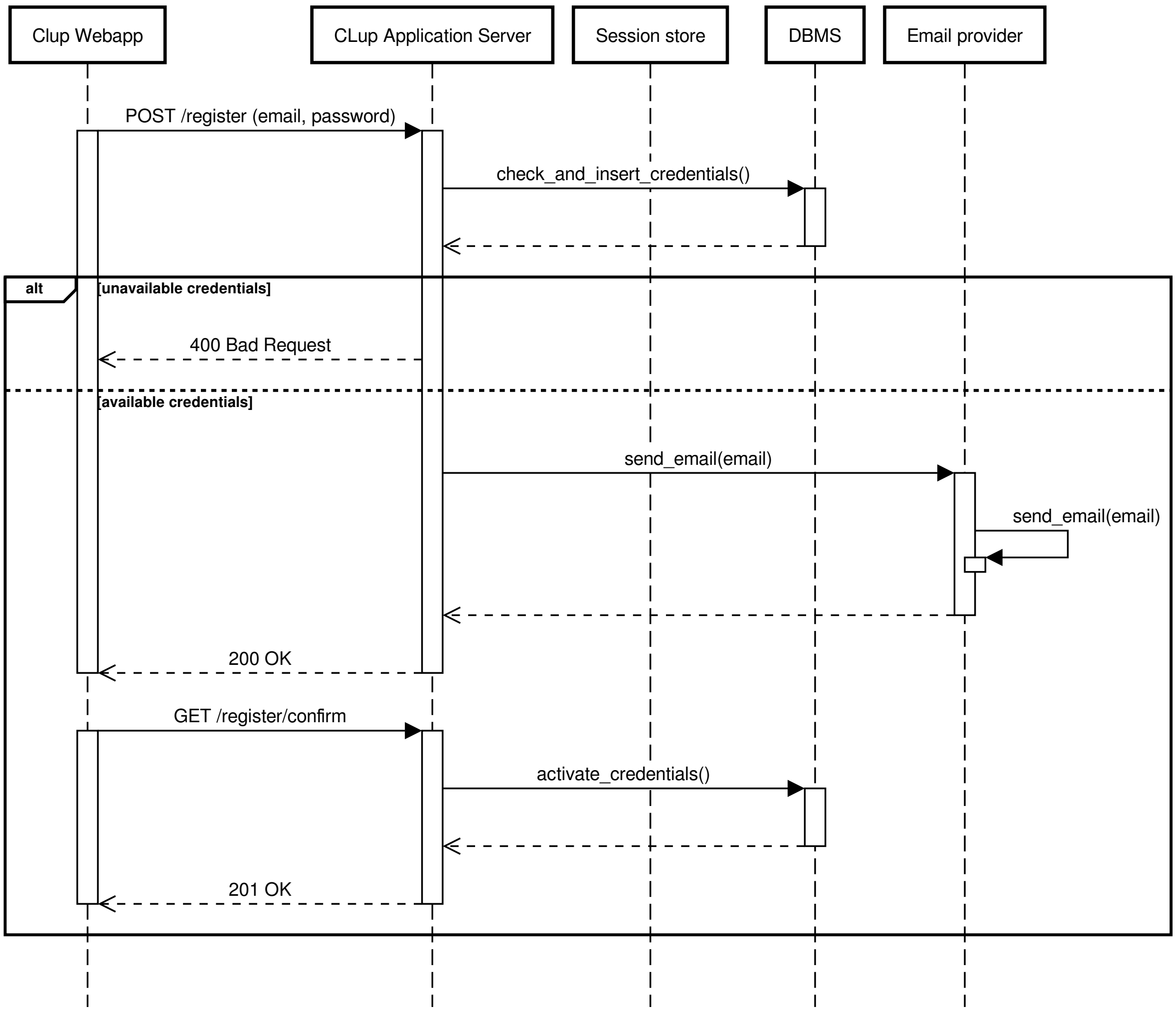


Figure 6: Customer registering

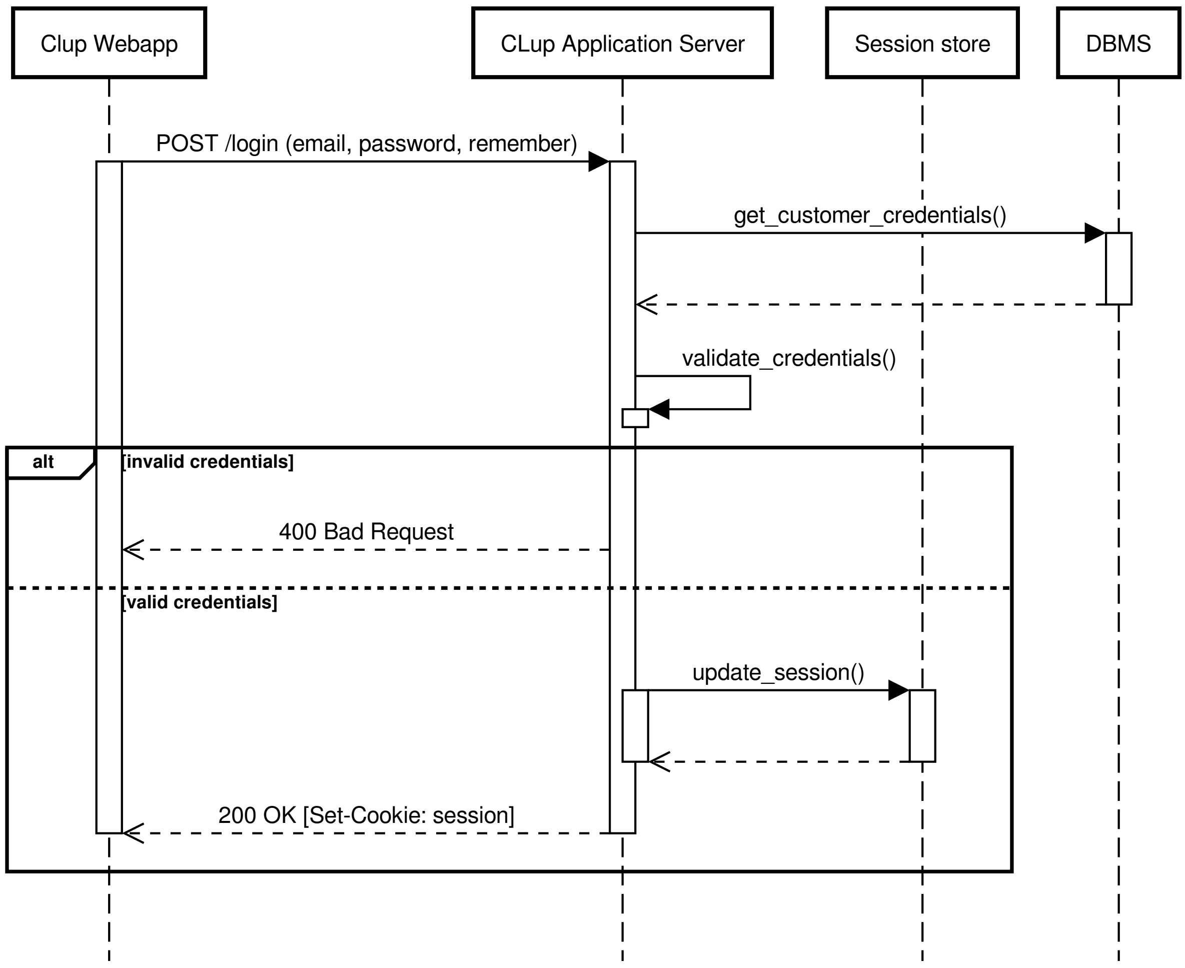


Figure 7: Customer logging in

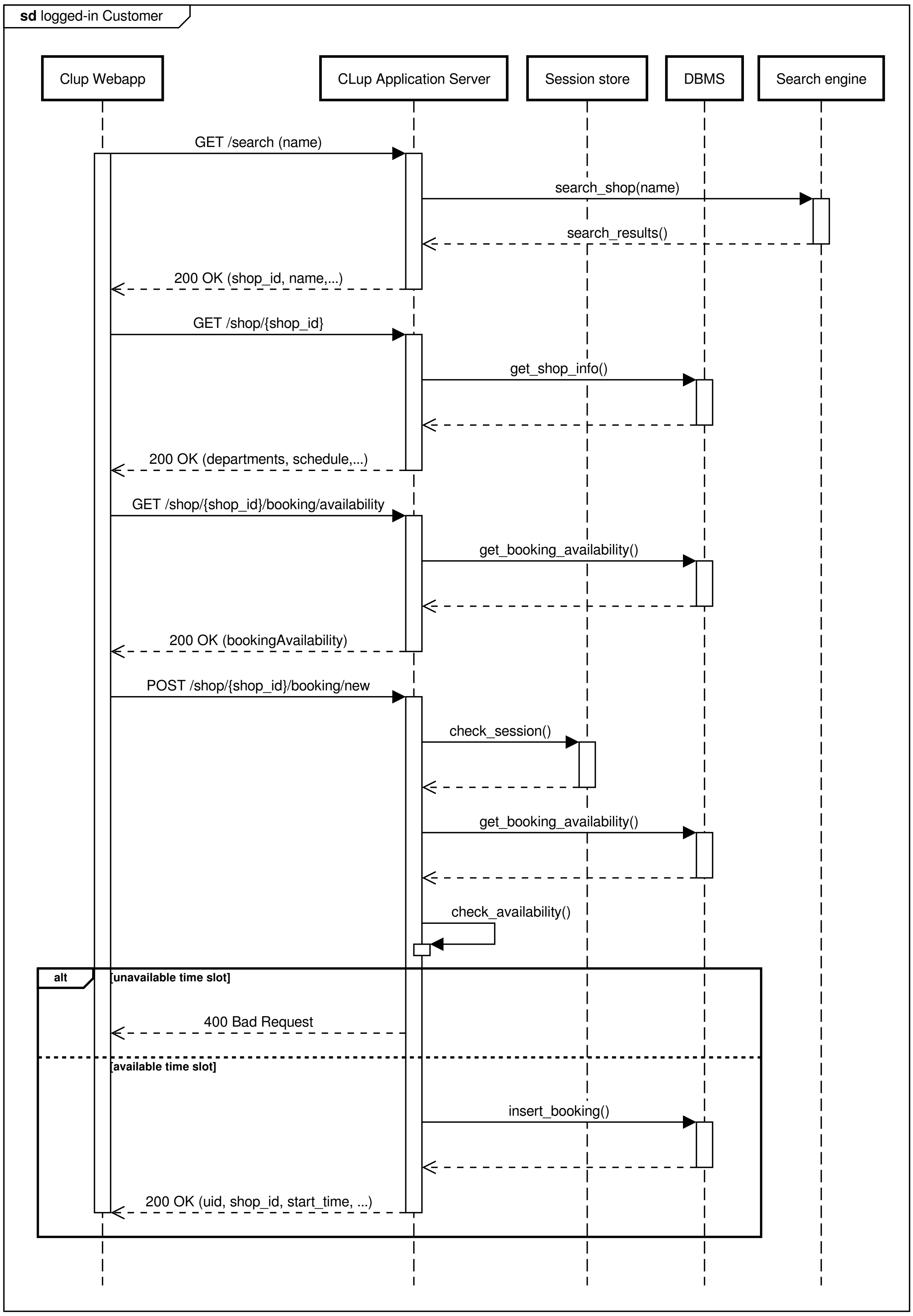


Figure 8: Customer generating a Booking

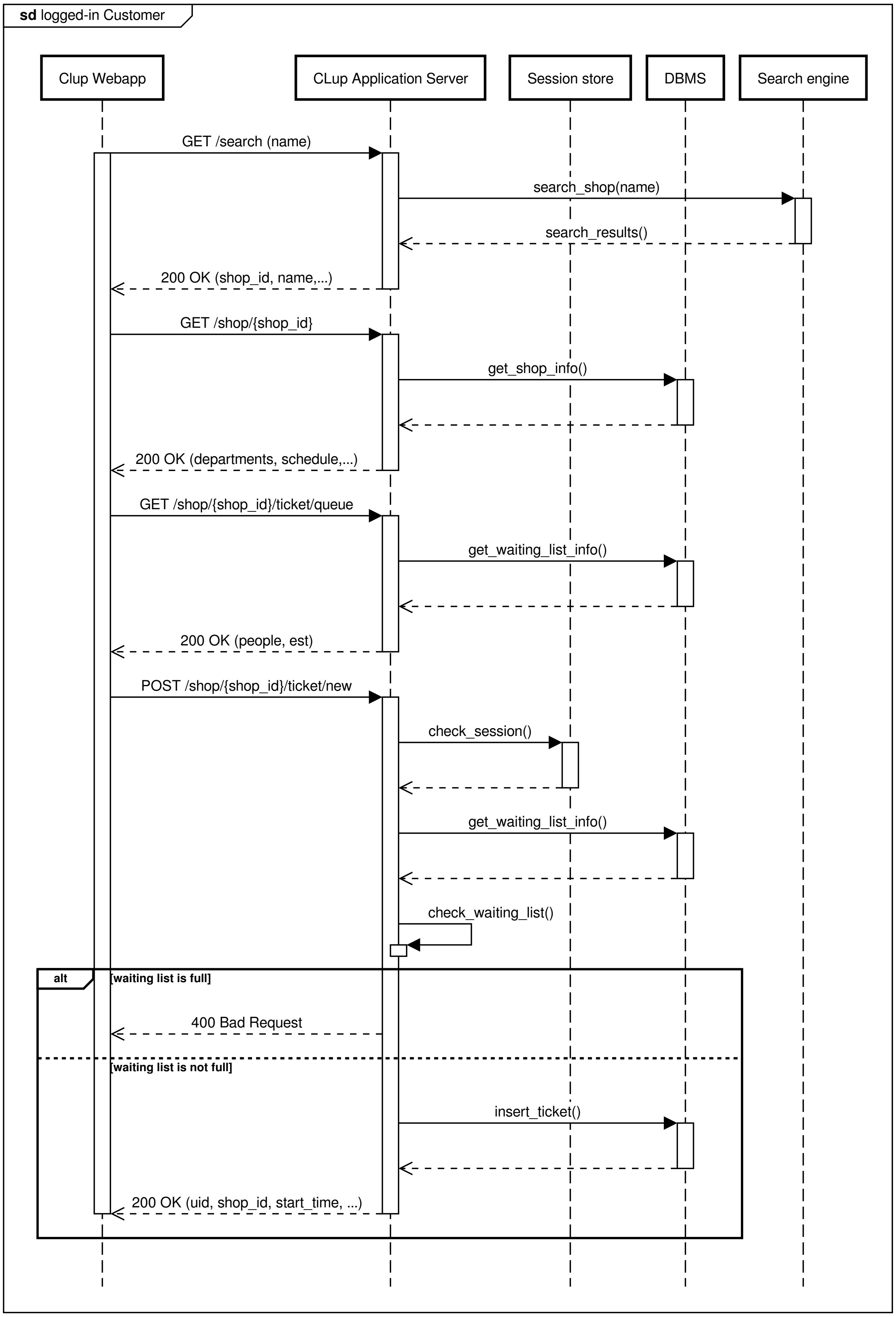


Figure 9: Customer generating a Ticket

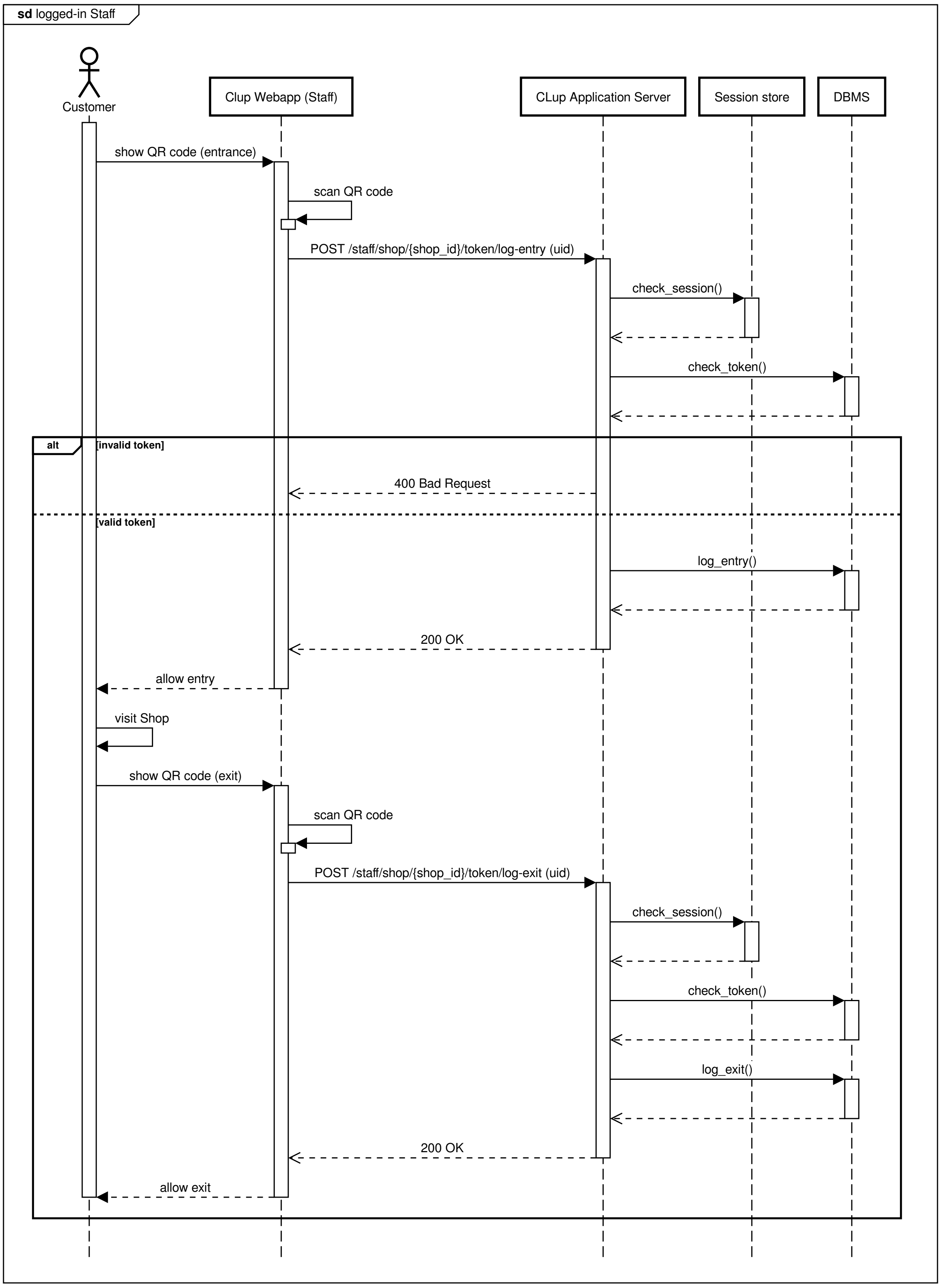


Figure 10: Staff validating either a Ticket or a Booking

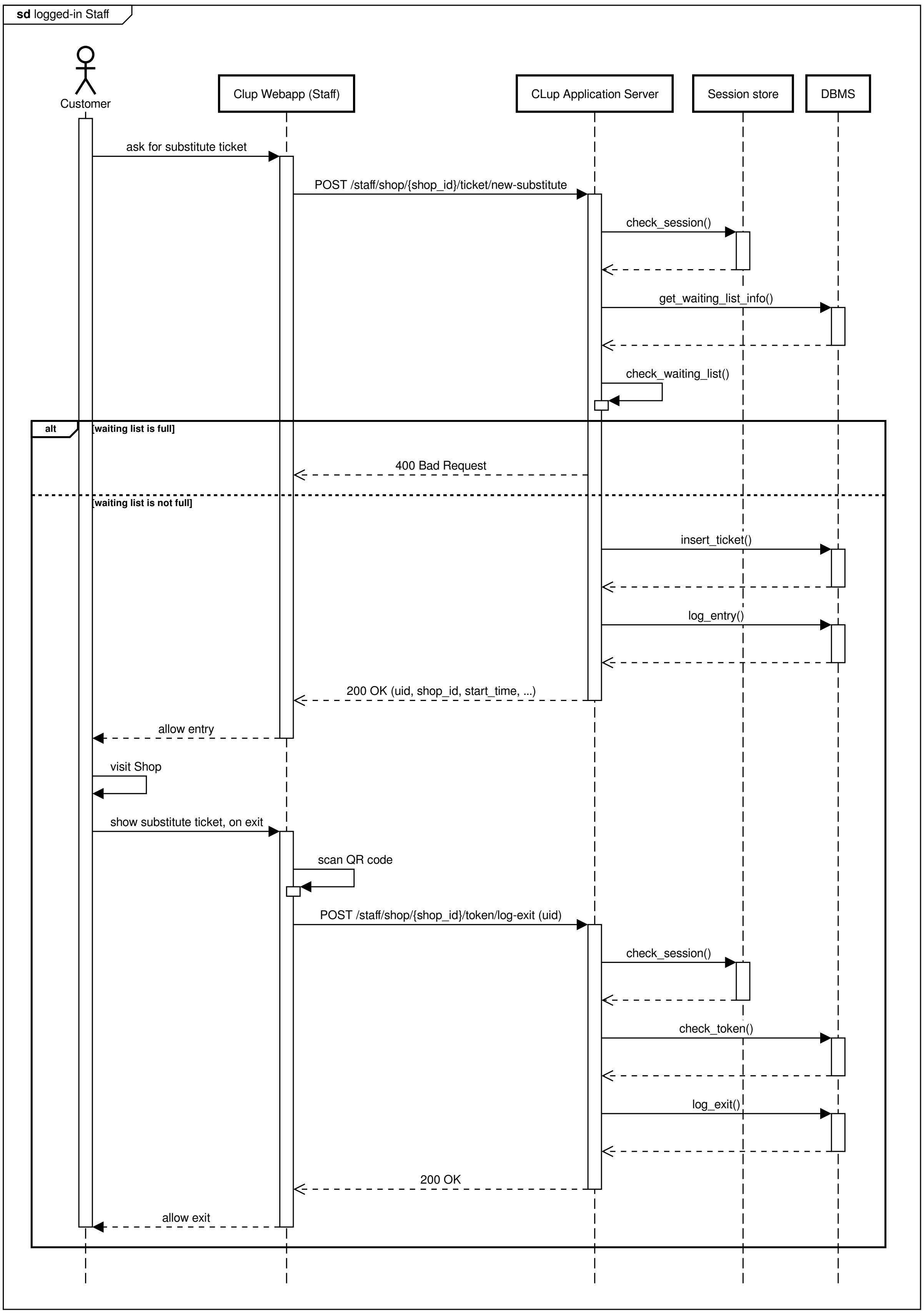


Figure 11: Staff generating a substitute Ticket

2.5 Component interfaces

2.5.1 API

Class	HTTP request	Description
account	**POST** `/login`	Logs in and returns the authentication cookie
account	**POST** `/register`	Request creation of a new account
account	**GET** `/register/confirm`	Submit email confirmation
booking	**POST** `/shop/{shop_id}/booking/new`	Request a booking for a shop
booking	**GET** `/shop/{shop_id}/booking/availability`	Get information about the time slot availability for bookings
manage	**POST** `/staff/manage/create-account/{shop_id}`	Generate a temporary staff account for activation
manage	**POST** `/staff/manage/shop/add`	Add a new shop to the database
manage	**POST** `/staff/manage/shop/list`	Get a list of all managed shops
manage	**POST** `/staff/manage/shop/{shop_id}/edit`	Edit an existing shop listing
manage	**POST** `/staff/manage/shop/{shop_id}/show`	Make shop public and reachable from customers
manage	**POST** `/staff/manage/shop/{shop_id}/hide`	Make shop private and hidden from customers
shop	**GET** `/search`	Search for a shop by name
shop	**GET** `/shop/{shop_id}`	Get available shop information
staff	**GET** `/staff/shop/{shop_id}/ticket/queue`	Get detailed information about the current queue status
staff	**GET** `/staff/shop/{shop_id}/booking/list`	Get detailed information about the current and future bookings
staff	**GET** `/staff/shop/{shop_id}/token/info`	Get token information and validity

Class	HTTP request	Description
staff	**POST** `/staff/shop/{shop_id}/ticket/new-substitute`	Request creation of a substitute ticket
staff	**POST** `/staff/shop/{shop_id}/token/log-entry`	Log entry, consume the token and update shop occupancy information
staff	**POST** `/staff/shop/{shop_id}/token/log-exit`	Log exit, update shop occupancy information
staff-account	**POST** `/staff/login`	Log in and return authentication cookie
staff-account	**POST** `/staff/register`	Activate staff account and set password
ticket	**GET** `/ticket/est`	Get the estimated waiting time for a ticket
ticket	**POST** `/shop/{shop_id}/ticket/new`	Request a ticket for a shop
ticket	**GET** `/shop/{shop_id}/ticket/queue`	Get information about the queue status and approximate waiting time
token	**GET** `/tokens`	Get active tokens for the customer

2.5.2 Endpoints

POST	`/login`		
Description	Logs in and returns the authentication cookie		
Body	`RequestLogin`		
Responses	**Code**	**Description**	**Body**
	200	OK	
	400	Invalid Credentials	

POST	`/register`		
Description	Request creation of a new account		
Body	`RequestRegistration`		
Responses	**Code**	**Description**	**Body**
	200	OK	
	400	Invalid data	

GET	`/register/confirm`		
Description	Submit email confirmation		
Parameters	**Name**	**Type**	
	code	`string`	
Responses	**Code**	**Description**	**Body**
	201	successful operation	
	400	Invalid code	

POST	`/shop/{shop_id}/booking/new`		
Description	Request a booking for a shop		
Parameters	**Name**	**Type**	
	shop_id	`string`	
Body	`RequestBooking`		
Responses	**Code**	**Description**	**Body**
	200	OK	`TokenBooking`
	400	Error	

GET	`/shop/{shop_id}/booking/availability`		
Description	Get information about the time slot availability for bookings		
Parameters	**Name**	**Type**	
	shop_id	`string`	
	day	`string`	
Responses	**Code**	**Description**	**Body**
	200	OK	`BookingAvailability`
	400	Error	

POST	`/staff/manage/create-account/{shop_id}`		
Description	Generate a temporary staff account for activation		
Body	`string`		
Responses	**Code**	**Description**	**Body**
	200	OK	`StaffTempAccount`
	400	Invalid email	

POST	`/staff/manage/shop/add`		
Description	Add a new shop to the database		
Body	`string`		
Responses	**Code**	**Description**	**Body**
	200	OK	`Shop`

POST	`/staff/manage/shop/list`		
Description	Get a list of all managed shops		
Responses	**Code**	**Description**	**Body**
	200	OK	`Array<SearchResult>`

POST	`/staff/manage/shop/{shop_id}/edit`		
Description	Edit an existing shop listing		
Parameters	**Name**	**Type**	
	shop_id	`string`	
Body	`Shop`		
Responses	**Code**	**Description**	**Body**
	200	OK	

POST	`/staff/manage/shop/{shop_id}/show`		
Description	Make shop public and reachable from customers		
Parameters	**Name**	**Type**	
	shop_id	`string`	
Responses	**Code**	**Description**	**Body**
	200	OK	

POST	`/staff/manage/shop/{shop_id}/hide`		
Description	Make shop private and hidden from customers		
Parameters	**Name**	**Type**	
	shop_id	`string`	
Responses	**Code**	**Description**	**Body**
	200	OK	

GET	`/search`		
Description	Search for a shop by name		
Parameters	**Name**	**Type**	
	q	`string`	
Responses	**Code**	**Description**	**Body**
	200	OK	`Array<SearchResult>`

GET	`/shop/{shop_id}`		
Description	Get available shop information		
Parameters	**Name**	**Type**	
	shop_id	`string`	
Responses	**Code**	**Description**	**Body**
	200	OK	`Shop`

GET	`/staff/shop/{shop_id}/ticket/queue`		
Description	Get detailed information about the current queue status		
Parameters	**Name**	**Type**	
	shop_id	`string`	
Responses	**Code**	**Description**	**Body**
	200	OK	`Queue`

GET	`/staff/shop/{shop_id}/booking/list`		
Description	Get detailed information about the current and future bookings		
Parameters	**Name**	**Type**	
	shop_id	`string`	
Responses	**Code**	**Description**	**Body**
	200	OK	`BookingList`

GET	`/staff/shop/{shop_id}/token/info`		
Description	Get token information and validity		
Parameters	**Name**	**Type**	
	shop_id	`string`	
	uid	`string`	
Responses	**Code**	**Description**	**Body**
	200	OK	`TokenTicket`
	400	Ticket does not exist	

POST	`/staff/shop/{shop_id}/ticket/new-substitute`		
Description	Request creation of a substitute ticket		
Parameters	**Name**	**Type**	
	shop_id	`string`	
Body	`RequestTicket`		
Responses	**Code**	**Description**	**Body**
	200	OK	`TokenTicket`
	400	Error	

POST	`/staff/shop/{shop_id}/token/log-entry`		
Description	Log entry, consume the token and update shop occupancy information		
Parameters	**Name**	**Type**	
	shop_id	`string`	
Body	`TokenCode`		
Responses	**Code**	**Description**	**Body**
	200	OK	
	400	Error	

POST	`/staff/shop/{shop_id}/token/log-exit`		
Description	Log exit, update shop occupancy information		
Parameters	**Name**	**Type**	
	shop_id	`string`	
Body	`TokenCode`		
Responses	**Code**	**Description**	**Body**
	200	OK	
	400	Error	

POST	`/staff/login`		
Description	Log in and return authentication cookie		
Body	`RequestLogin`		
Responses	**Code**	**Description**	**Body**
	200	OK	

POST	`/staff/register`		
Description	Activate staff account and set password		
Body	`StaffRequestRegistration`		
Responses	**Code**	**Description**	**Body**
	200	OK	
	400	Invalid credentials	

GET	`/ticket/est`		
Description	Get the estimated waiting time for a ticket		
Parameters	**Name**	**Type**	
	uid	`string`	
Responses	**Code**	**Description**	**Body**
	200	OK	`QueueEst`
	400	Invalid or expired code	

POST	`/shop/{shop_id}/ticket/new`		
Description	Request a ticket for a shop		
Parameters	**Name**	**Type**	
	shop_id	`string`	
Body	`RequestTicket`		
Responses	**Code**	**Description**	**Body**
	200	OK	`TokenTicket`
	400	Error	

GET	`/shop/{shop_id}/ticket/queue`		
Description	Get information about the queue status and approximate waiting time		
Parameters	**Name**	**Type**	
	shop_id	`string`	
Responses	**Code**	**Description**	**Body**
	200	OK	`QueueEst`

GET	`/tokens`		
Description	Get active tokens for the customer		
Responses	**Code**	**Description**	**Body**
	200	OK	`Tokens`

2.5.3 Models

BookingAvailability: `Array<{dept_id: string, time: TimeSpan, available: integer,}>`

BookingList: `Array<TokenBooking>`

Department

Field	Type
uid	`string`
description	`string`
capacity	`integer`

Queue: `Array<TokenTicket>`

QueueEst

Field	Type
people	`integer`
est	`string`

RequestBooking

Field	Type
shop_id	`string`
department_ids	`Array<string>`
start_time	`string`
end_time	`string`

RequestLogin

Field	Type
email	`string`
password	`string`
remember	`boolean`

RequestRegistration

Field	Type
email	`string`
password	`string`

RequestTicket

Field	Type
shop_id	`string`
department_ids	`Array<string>`

Schedule: `Array<TimeSpan>`

SearchResult

Field	Type
uid	`string`
name	`string`
image	`string`
description	`string`

Shop

Field	Type
uid	`string`
name	`string`
description	`string`
image	`string`
location	`string`
departments	`Array<Department>`
weekly-schedule	weekly-schedule

StaffRequestRegistration

Field	Type
email	`string`
token	`string`
password	`string`

StaffTempAccount

Field	Type
email	`string`
token	`string`

TimeSpan

Field	Type
start	`string`
end	`string`

TokenBooking

Field	Type
uid	`string`
shop_id	`string`
department_ids	`Array<string>`
start_time	`string`
end_time	`string`

TokenCode

Field	Type
uid	`string`

TokenTicket

Field	Type
uid	`string`
shop_id	`string`
department_ids	`Array<string>`
creation	`string`
expiration	`string`
state	`string`

Tokens

Field	Type
tickets	`Array<TokenTicket>`
bookings	`Array<TokenBooking>`

2.6 Selected architectural styles and patterns

The main design pattern characterizing the Application server is the **Actor Model**[2]. It's a model that allows building highly concurrent applications, without the need to synchronize access to shared memory. An **Actor** is an indipendent virtual processor with its own local state. Each actor has a **Mailbox**. When a message appears in the mailbox and the actor is idle, it kicks into life and processes the message[1]. Actors process one message at a time and can only interact with other actors with messages.

In the *CLup* Application server, every endpoint of the API is an independent actor with no shared state.

On a higher level of abstraction, *CLup* follows the **Model, View, Controller** design pattern, where the controller is split between the Application Server and the Web Application.

The Web Application implements the **MVVM** (Model-View-ViewModel) pattern, where every property of the view is exposed by the view model through a binding. This accomplishes a better separation between the UX and the business logic of the Application, which makes development in a team more efficient.

3 User Interface Design

As described in section *3.1 User Interfaces* of the RASD, the web application displays to the Users a different page for each of this actions:

- Register
- Login
- Show the generated Tokens
- Search for Shops
- Get a Ticket or Booking for a Shop

and, similarly, to the Staff:

- Register (Activate account)
- Login
- Manage Shops
- Manage Shop departments (and other attributes)
- Validate a Ticket or Booking
- Generate Substitute Ticket

The mockups will not be shown again in this document; for reference, see those included in the RASD.

3.1 UX diagrams

After enumerating the pages of the User Interface, this section focuses on the connections between them. The UX diagrams provide an abstract representation of the possible routes of the interaction of the User with the Web Application: the first one focuses on the Customer; the second focuses on the Staff. Some upstream links are omitted for the sake of clarity.

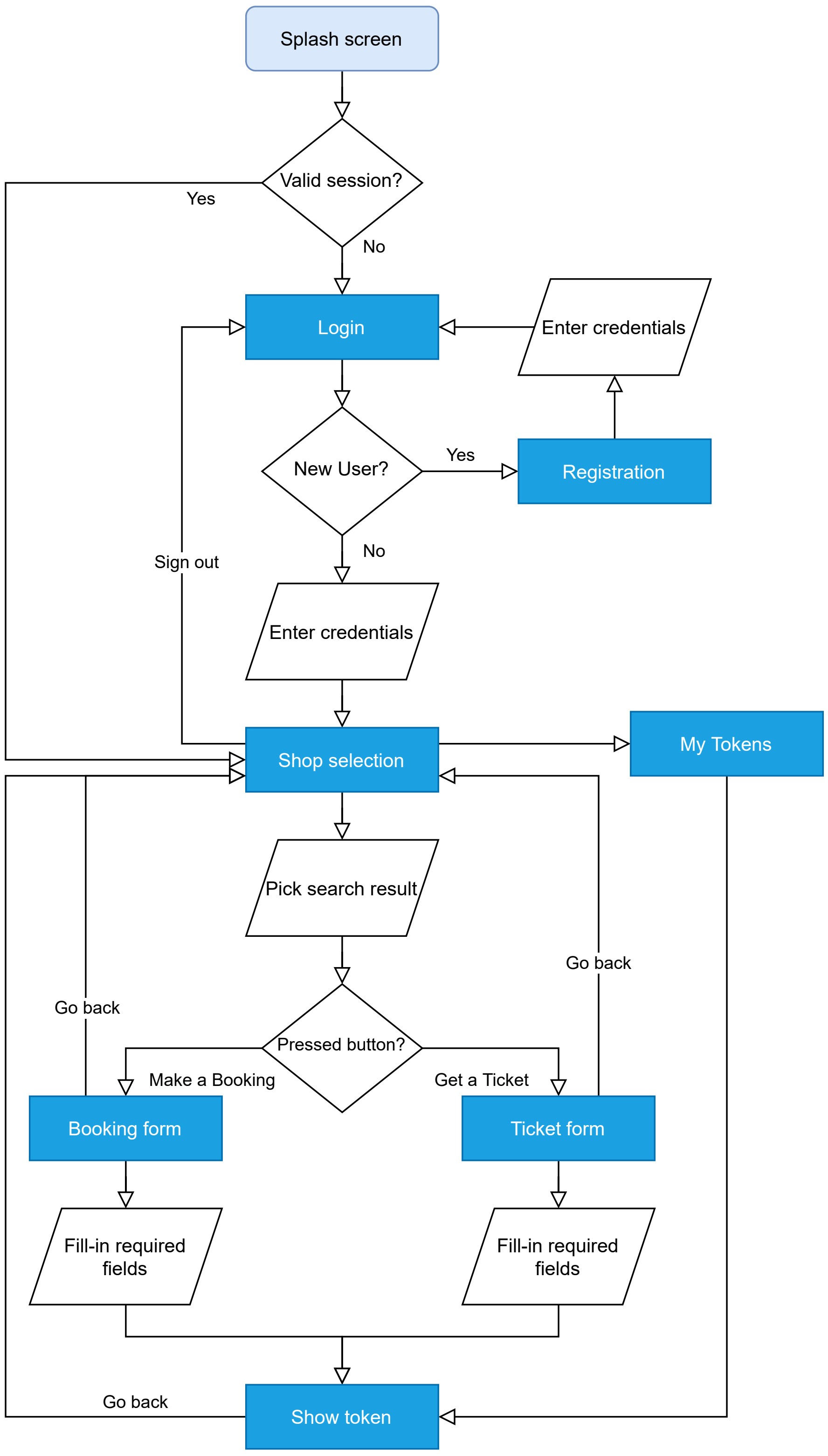


Figure 12: UX flowchart for a Customer

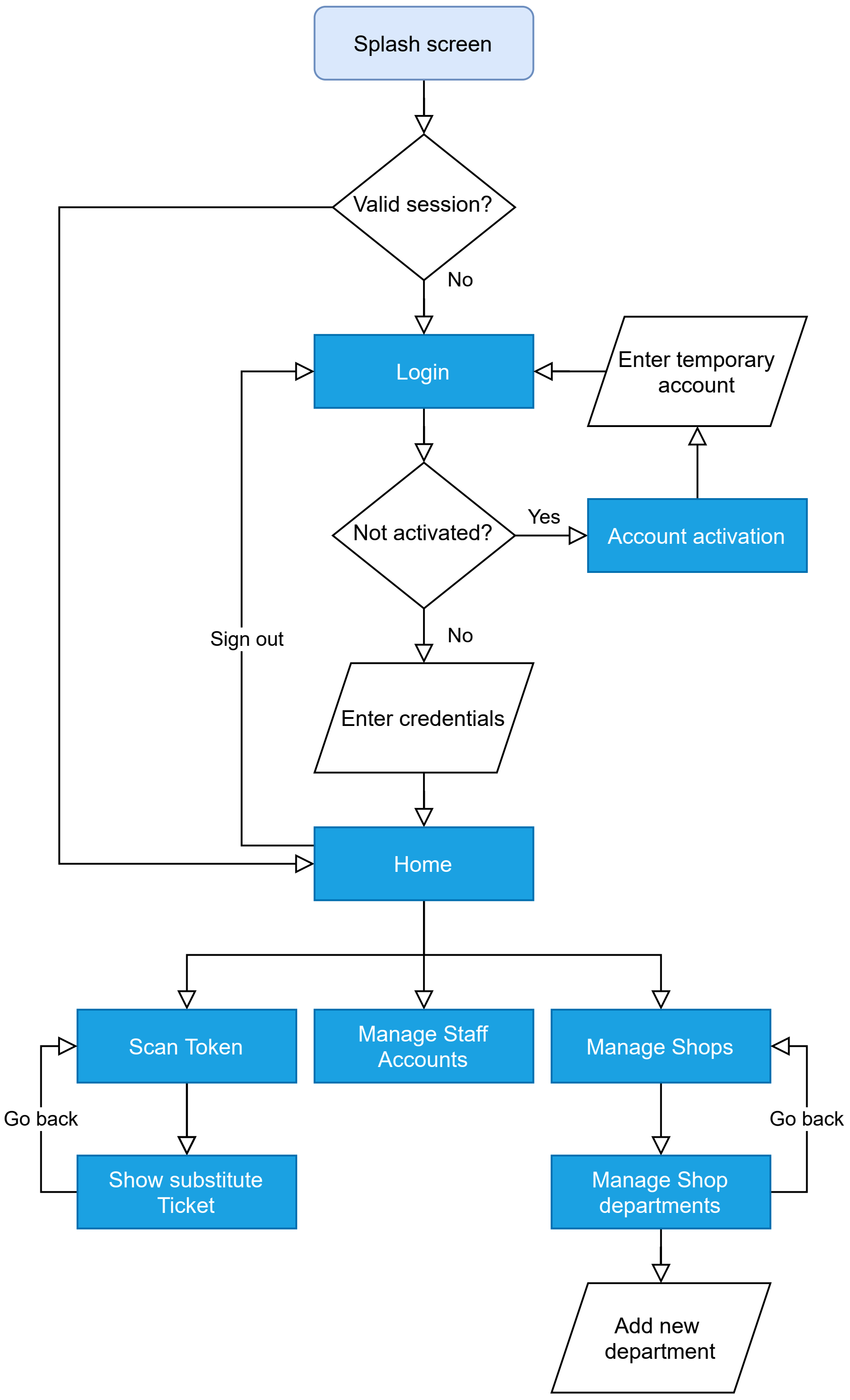


Figure 13: UX flowchart for a Staff member

4 Requirements Traceability

Table 2: Mapping of the Requirements (from the *RASD*) on the Components.

Component (DD)	**Requirements (RASD)**
Customer interface (Web application)	• **[R2]** The system shall allow customers to request the right to visit a shop as soon as possible • **[R3]** The system shall give Customers a Token associated with their position in the waiting line • **[R10]** The system shall allow customers to choose a time in the future in which they wish to visit a Shop • **[R11]** The system shall give Customers a Token associated with their booking • **[R12]** The system shall ask Customers to specify the approximate duration of their visit • **[R14]** The system shall give Customers an estimate of the waiting time remaining before it's their turn • **[R17]** Customers shall be able to specify the categories of items they intend to buy
Maps provider	• **[R16]** The system shall be able to connect to a Maps service to show information about travel time
Staff interface (Web application)	• **[R4]** The Staff shall be able to scan Customer generated Tokens using a camera • **[R5]** The Staff shall be able to scan Customer generated Tokens using a textual input • **[R6]** Given a Token, the Staff application shall be able to verify its validity • **[R7]** Given a Token, the Staff application shall be able to verify its position in the waitingline • **[R8]** Given a Token, the Staff application shall be able to mark it as used and update the list of Customers currently inside the Shop • **[R19]** The system shall allow managers to enlist a Shop and edit the details of an existing Shop

Continuation of Table 2	
Component (DD)	**Requirements (RASD)**
Camera	• **[R4]** The Staff shall be able to scan Customer generated Tokens using a camera
Customer Ticket service	• **[R1]** The system shall keep track of the list of Customers waiting to visit each Shop • **[R2]** The system shall allow customers to request the right to visit a shop as soon as possible • **[R3]** The system shall give Customers a Token associated with their position in the waiting line • **[R12]** The system shall ask Customers to specify the approximate duration of their visit • **[R13]** The system shall automatically infer an estimate visit duration for returning customers • **[R14]** The system shall give Customers an estimate of the waiting time remaining before it's their turn • **[R15]** The system shall notify Customers when their turn is about to come • **[R17]** Customers shall be able to specify the categories of items they intend to buy

Continuation of Table 2

Component (DD)	**Requirements (RASD)**
Customer Booking service	• **[R9]** The system shall keep track of the Customers that want to visit a Shop in the future • **[R10]** The system shall allow customers to choose a time in the future in which they wish to visit a Shop • **[R11]** The system shall give Customers a Token associated with their booking • **[R12]** The system shall ask Customers to specify the approximate duration of their visit • **[R13]** The system shall automatically infer an estimate visit duration for returning customers • **[R17]** Customers shall be able to specify the categories of items they intend to buy • **[R18]** The system shall keep track of the number of customers visiting the Shop on a Department basis
Token validation service	• **[R6]** Given a Token, the Staff application shall be able to verify its validity • **[R7]** Given a Token, the Staff application shall be able to verify its position in the waitingline • **[R8]** Given a Token, the Staff application shall be able to mark it as used and update the list of Customers currently inside the Shop • **[R18]** The system shall keep track of the number of customers visiting the Shop on a Department basis • **[R19]** The system shall allow managers to enlist a Shop and edit the details of an existing Shop
Shop management service	• **[R17]** Customers shall be able to specify the categories of items they intend to buy • **[R18]** The system shall keep track of the number of customers visiting the Shop on a Department basis • **[R19]** The system shall allow managers to enlist a Shop and edit the details of an existing Shop

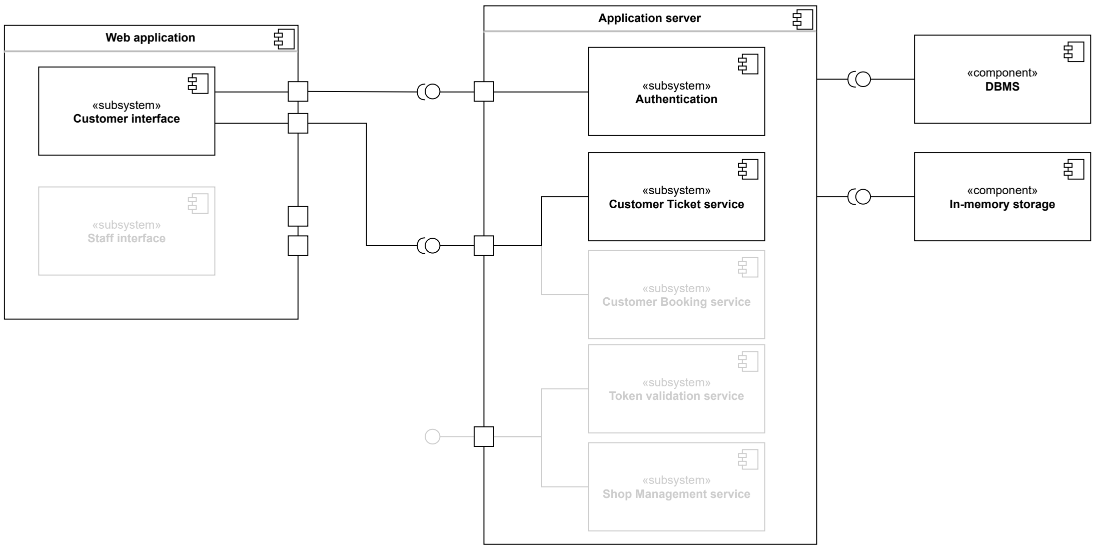
5 Implementation, Integration and Test plan

5.1 Recommended Implementation

- **Web Application**: the web application should be written in a web framework such as *Vue.js* or *Angular* to create a consistent web interface with cross platform support.
- **Application Server**: the recommended implementation for the server uses the *Rust* language, the *Actix*[1] actor framework and *SQLx*[2]. The Rust language provides a safe, concurrent alternative over C and C++ while providing comparable performance, allowing high reliability and throughput for the service.
- **In-Memory data store**: the in memory data store employed may be *Redis Enterprise*
- **DBMS**: the relational database manager software may be *PostgreSQL 13*
- **Reverse Proxy**: the reverse proxy may be *NGINX*

5.2 Implementation Plan

The implementation should follow a *Thread* (also known as *Tracer bullet*[1]) approach.



The system should be written starting from a subset of functionality implemented from the start to the end. Then the other functionalities will be added alongside what was already written.

The first thing to be programmed should be the authentication and account creation, interfacing with the DBMS and the memory storage and exposing a minimal API.

This way, the system can reach partial functionality with a structure representative of the final product in a small amount of time. Then other components can rely on the existing structure for integration.

[1]https://actix.rs/
[2]https://github.com/launchbadge/sqlx

The Ticket service should be implemented and integrated right after the Authentication, adding part of the core functionality.

The web application can be developed in parallel by referencing the proposed REST API for the first stages of development.

At this point a partially working version of the system is complete and can be presented to the stakeholders for feedback.

The next components should be created and integrated in the following order:

- Staff interface, with the associated Token Validation service
- Booking service
- Shop management service

The Search engine and Maps service interfaces can be added in parallel after the first working version, since they have essentially zero coupling to the other parts of the system.

5.3 Testing plan

Testing should be done throughout all stages of development, following a bottom up approach.

Unit tests should be written starting from the basic component so that tests for other components that use them can be written relying on the fact that the subcomponents are already tested. This speeds up debugging since it's easy to locate at which level of the component hierarchy there is a problem.

Tests for the application server which interact with the database should be run on an indipendent instance of the service and not on the main network.

5.4 Integration plan

The main branch of the version control should be reserved for working and potentially deliverable, versions of the system. The web application and the application server should have two separate repositories, in order to make the commit history more readable. *Feature branches* should be created as necessary, to work on adding new features to the system.

Testing should be enforced through a *Continuous Integration* platform such as GitLab CI, Travis CI or GitHub Actions. This way every commit will be tested and marked, so that broken commits will be highlighed and will not be merged into the main branch.

6 Effort Spent

Luca De Martini

Date	**Hours**	**Description**
2020-12-28	3h	First Meeting
2020-12-29	3h	Introduction
2020-12-30	4h	API
2021-1-1	2h	API
2021-1-2	2h	API & Deployment
2021-1-3	2h	Architecture
2021-1-4	3h	Review & API
2021-1-5	5h	Review & API
2021-1-6	3h	Implementation Plan
2021-1-7	3h	Integration & Testing Plan
2021-1-8	2h	Review

Alessandro Duico

Date	**Hours**	**Description**
2020-12-28	3h	First Meeting
2020-12-29	3h	Introduction
2020-12-30	3h	Component view
2020-12-1	4h	Deployment diagram
2021-1-2	1h	Runtime View
2021-1-4	3h	Runtime View
2021-1-5	3h	Review
2021-1-6	3h	User Interface Design
2021-1-7	3h	Requirements Traceability
2021-1-8	2h	Review

References

[1] Andrew Hunt David Thomas. *The Pragmatic Programmer, 20th Anniversary Edition*. Addison-Wesley Professional, 2019.

[2] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.