POLITECNICO
MILANO 1863

CLup ITD

CLup Implementation and Test Deliverable

Authors:

David Leonardo Gomez 10787907
Martín Anselmo 10789796
Vinay Krishna Munnaluri 10780972

Software: [Required Software List] **(on readme.md file)**

Source Code: https://github.com/mfanselmo/AnselmoGomezMunnaluri

Document Version: 1.0

Date: 25 January 2021

Professor: Elisabetta Di Nitto

Contents

1 Introduction

1.1 Purpose

This document describes the Integration and Testing Plan Document for CLup virtual lining system. A correct integration and testing are essential in order to guarantee that all the subsystems function according to the requirements specified in the RASD, the Design on DD and without unexpected behaviors or errors. The purpose of this document is describing the integration testing activities for all the components, and ultimately the whole system.

In the following sections we are going to provide:

- A description of the chosen testing strategy and the reasons that led to its adoption.
- The entry criteria that must be met in order to properly begin the testing.

• A list of all the components and their subsystems involved in the testing activities.

• The order in which the components and their subsystems will be integrated.

• A description of the planned testing activities for each integration step, including all the expected inputs and outcomes.

1.2 Scope

CLup is a software designed to handle crowds and lines in stores during the COVID-19 pandemic. It consists in a digital queueing system so customers do not have to go and stay in line for long periods of time, putting their health in risk and wasting their time.

The three main functionalities offered by CLup are:

1. Lining up: A customer can request a number (ticket) for a virtual queue for entering the supermarket. They will be given a QR code that will be scanned when entering and exiting the store.
2. Book a visit: A customer will be able to request the booking of a visit to the supermarket, so they don’t have to wait in a queue if they know the time they are coming. In addition to this, the customer will be able to tell the system the approximate duration of their visit and what they will buy, so that the system can better manage the people inside of a store.

3. Alternatives for the customer: CLup will offer the customer suggestions in case the queue for a particular store is too long (going to a different store or in a different time).

1.3 Definitions

- Ticket: The QR code generated for entering to a store
- Invalid Ticket/code: A QR code which is not valid to enter a store in that moment (Because it is not the time to enter, or because it already expired)
- Manager: The person in charge of a store
- Store: A branch or location of a supermarket.

1.4 Abbreviations

- DD: Design Document
- RASD: Requirements Analysis and Specifications Document
- REST: Representational State Transfer
- API: Application Programming Interface
- SPA: Single Page Application
- PWA: Progressive Web App

1.5 Revision history

• 1.0 Initial Version

1.6 Reference Documents

• Assignment Document: “A.Y. 2020-2021 Software Engineering 2 I&T assignment goal, schedule, and rules”

2 Implementation of Functionalities

2.1 Functionalities Implemented.

CLup has 3 main functionalities and 2 of them have been implemented: **Lining up and Booking visit**. These functionalities were split into several requirements which most of them must be accomplished for the system to work how it's intended.

1. Users can log in and out of the platform.
2. Registered customers can get tickets.
3. Managers can produce tickets for customers.
4. Non registered customers can get tickets through the platform.
5. Registered customer can check the status of his code using the system.
6. The system reads QR codes from the customers.
7. The system shows an alert if a customer shows an invalid code.
8. Customers can book shopping visits.
9. A manager can check how many people are currently in his/her store.

The functionalities implemented share a lot of logic behind them, because of their nature. To mimic a queueing system, we took an approach of slots. These are subdivisions of the day in 30 minutes intervals in which customers can get a ticket in. For the lining up, the request is made using the current time, so the next available slot is given to the user, and for the booking request, the user tells the system which time he wants to get a ticket in.
This approach assures the customers that most of the time (except when someone takes a little longer in their shopping), that they will be able to enter the store in the time the ticket says.

Note, we skipped the sms sending functionality for this implementation project for practical reasons. In the real world, this functionality is one of the most important ones in CLup. It was not implemented because of costs, each of the members of the group is currently living in different countries, and because it is out of the scope of this implementation.

2.2 Resources Used.

2.2.1 Languages

CLup is currently using:

· HTML

· SCSS (Compiles to css)

· Javascript (with Node.js as runtime)

· Python3

· JSON format (for data communication)

2.2.2 Frameworks

As the system is using a Client-Server model, it has been divided into a frontend, made with ReactJs and a backend, made with Django.

The business tier (backend of the system) was implemented using the Django REST framework, an easy to use toolkit for building web APIs. This framework handles the interaction with the dbms, using an Object Relational Mapping (ORM) approach. Authentication and permissions (for handling customers and managers) are also easily implemented using tools provided in the framework and by python free libraries.

- asgiref: ASGI is a standard for Python asynchronous web apps and servers to communicate with each other, This package includes ASGI base libraries.
- dj-database-url: This simple Django utility allows you to utilize the 12factor inspired DATABASE_URL environment variable to configure your Django application.
- Django: a high-level Python Web framework.
- django-cors-headers: A Django App that adds Cross-Origin Resource Sharing (CORS) headers to responses.
- django-seed: Django-seed uses the faker library to generate test data for your Django models.
- djangorestframework: Provide a single full-time position on REST framework.
- Faker: Faker is a Python package that generates fake data for you.
- gunicorn: Gunicorn ‘Green Unicorn’ is a Python WSGI HTTP Server for UNIX.
- psycopg2: PostgreSQL database adapter.
- python-dateutil: Date utilities for python database management.
- pytz: Brings the Olson tz database into Python.
- six: Six is a Python 2 and 3 compatibility library.
- sqlparse: A non-validating SQL parser for Python.
- text-unidecode: The most basic port of the Text::Unidecode Perl library.

- whitenoise: With a couple of lines of config WhiteNoise allows your web app to serve its own static files.

React is a Javascript library for creating user interfaces based on components. Since the logic is handled in the business tier, React only handles the presentation of the user interfaces, with all the data being retrieved using HTTP and showing it to the user. Many users will be on mobile phones while using CLup, so all pages are built with the concept of mobile first. In addition to this characteristic, the frontend has been developed as a Progressive Web App (PWA), this allows the page to be installed as an application on modern devices (iOS, Android, Windows, MacOs), allowing for much faster response rates and much more.

The application developed here is then compiled into static html files which get uploaded to a simple server (in our case, Netlify).

Some of the elements provided by the React framework and Node.js runtime that we used are:

- Create React App: A toolkit that gives the foundations to the web application, with everything needed for basic development. When running this tool, a Single Page Application (SPA) is created.
- React Router Dom: A library used for handling declarative routers within the web application.
- Context api: A library used within React for handling state, such as keeping the user credentials.
- Ant design: A React UI library.
- Jest and React Testing library: The libraries used for testing the frontend.
- Netlify: A serverless platform for deploying the web application.

2.2.3 APIs

This section is divided in the domain of the APIs, and will only contain the specification of internal APIs, the external APIs will only be listed.

A. Sessions

a. Login [POST]

For users to log in into the application, the user exists on system.
Returns the token that will be used to verify him.

- Request:

```
https://url/session
HEADERS:
Content-Type:application/json
{
  "user": {
    "phone_number": "",
    "password": ""
  }
}
```

- Response (Code 200):

```
Content-Type:application/json
{
  "authentication_token": "",
  "is_manager": true / false
}
```

- Response (Code 400):

```
If an auth token is not returned
```

b. Logout [DELETE]

Takes the token of the user and makes it invalid.

- Request:

```
https://url/session
HEADERS:
Content-Type:application/json
{
  authentication_token: ""
}
```

- Response (Code 200):

```
{
  message: "success"
}
```

B. Users

a. Get current user [GET]

Returns the current user, with all of the information related to him: Phone number, Active tickets and Bookings.

- Request:

```
https://url/user
HEADERS:
Content-Type:application/json
{Authorization: "Token {user_token}"}
```

- Response (Code 200):

```
Content-Type:application/json
{
  is_manager: bool,
  active_tickets: [],
  bookings: [],
  managed_store: [list of store_ids in the case the user is manager]
}
```

- Response (Code 400):

```
Content-Type:application/json
{
  message: "User not logged in"
}
```

b. Create new user [POST]

Creates a new account.

- Request:

```
https://url/user
HEADERS:
Content-Type:application/json
{
  phone_number: "",
  password: "",
  email_address: "",
  username: ""
}
```

- Response (Code 200):

```
{
  authentication_token: "",
  isManager: bool
}
```

- Response (Code 400):

```
Content-Type:application/json
{
```

```
  message: "Passwords do not match"
}
```

C. Stores
    a. Get all stores [GET]

Gets a list to show the available stores and gets all of their information.

- Request:

```
https://url/store
HEADERS:
Content-Type:application/json
```

- Response (Code 200):

```
Content-Type:application/json
{
  [{
    store_id,
    location: {
      address,
      latitude,
      longitude
    },
    max_customers,
    current_customers,
    name
  },
  {}
  ]}
}
```

D. Slots
    a. Get next Slots

- Request:

```
https://url/slots
HEADERS:
Content-Type:application/json
BODY:

{
  time_of_visit: "2021-02-06 14:00:00",
  store_id: ""
}
```

- Response (Code 200):

```
Content-Type:application/json
```

```
{
  "available_slot": "2021-02-06 14:30:00"
}
```

- Response (Code 400):

```
Content-Type:application/json
{
  message: "No more slots available today"
}
```

E. Tickets

a. Create new ticket [POST]

New ticket (before creating a ticket, we check the next available slot and only create a ticket for then)

- Request:

```
https://url/ticket
HEADERS:
Content-Type:application/json
BODY:
{
  "phone_number": "",
  store_id: "",
  time_of_visit: ""
}
```

- Response (Code 200):

```
Content-Type:application/json
{
  ticket_id: "",
  approximate_enter_time: "2021-02-06 14:30:00"
}
```

- Response (Code 400):

```
Content-Type:application/json
{message: "You already have a ticket in this slot"}
{message: "This slot is not available"}
```

b. Ticket status [GET]

Description

- Headers:

```
https://url/ticket
HEADERS:
```

```
Content-Type:application/json
{
  "ticket_id":
"4f75482e100691087c02a69e97bb5be12952cf027ad9895113ed0b56c8957294",
}
```

- Response (Code 200):

```
{
  "approximate_enter_time": "2021-01-31 10:21:00+00:00",
  "store_name": "CONAD",
  "lat": 93.887,
  "lon": 87.453,
  "address": "via tartini",
  "categories_to_visit": "groceries",
  "currents_customers": 5,
  "max_customers": 5,
  "status": "New",
  "store_id": 1,
  "allowed_in": false,
  "reason": "you are not assigned to current slot please wait for your
turn"
}
```

- Response (Code 400):

```
Content-Type:application/json
{
  message: "invalid ticket"
}
```

c. Ticket cancel [DELETE]

- Request:

```
https://url/ticket
Content-Type:application/json
HEADER:
BODY:
{
  "ticket_id":
"4f75482e100691087c02a69e97bb5be12952cf027ad9895113ed0b56c8957294",
}
```

- Response (Code 200):

```
Empty response text
```

- Response (Code 400):

```
Content-Type:application/json
{
    message: "Not valid ticket"
}
```

d. Ticket scan [POST]

- Request:

```
https://url/scanticket
Content-Type:application/json
HEADER:
{
  Authorization: "Token {manager_token}
}
BODY:
{
  "ticket_id": "",
}
```

- Response (Code 200):

```
{
  ticket_id:"",
  message: "Ticket scanned succesfully",
  ticket_status: "scanned (or completed)"
}
```

- Response (Code 400):

```
{
  detail: "Invalid manager auth token"
}
```

F. Booking

- Request:

```
https://url/booking
HEADERS:
Content-Type:application/json
BODY:
{
  "store_id": 2,
  "time_of_visit": "2021-01-23T17:19:13.582Z",
```

```
  "categories_to_visit": [
    "fruits",
    "vegetables"
  ]
}
```

- Response (Code 200):

```
Content-Type:application/json
{
    "ticket_id":
"4f75482e100691087c02a69e97bb5be12952cf027ad9895113ed0b56c8957294",
}
```

- Response (Code 400):

```
Content-Type:application/json
{
    message: "Something went wrong"
}
```

a. Booking status [GET]

Description

- Request:

```
https://url/booking
HEADERS:
Content-Type:application/json
BODY:
{
    "ticket_id":
"4f75482e100691087c02a69e97bb5be12952cf027ad9895113ed0b56c8957294",
}
```

- Response (Code 200):

```
Content-Type:application/json
{
  "enter_time": "2021-01-23T17:19:13.582Z",
  "store_id": 1,
  "categories_to_visit": [
    "fruits",
    "vegetables"
  ]
}
```

- Response (Code 400):

```
Content-Type:application/json
{
```

```
  message: "Not valid ticket"
}
```

b. Booking cancel [DELETE]

- Request:
- Response (Code 200):
- Response (Code 400):

G. Manager

a. Get available stores [GET]

Description

- Request:

```
https://url/manager
HEADERS:
Content-Type:application/json
```

- Response (Code 200):

```
Content-Type:application/json
{
  data: [
    {
      "store_id": 1,
      "lat": 100,
      "lon": 100,
      "address": "Viale Piave, 38/B, 20129",
      "estimated_waiting_time": 120 // minutes
      "people_in_store": 25,
      "people_in_line": 15,
    },
  ]
}
```

b. Ticket scan [POST]

Gets a list of the current manager' stores (current logged manager)

- Request:

```
https://url/manager
HEADERS:
Content-Type:application/json
BODY:
```

```
{
  "ticket_id": "",
}
```

- Response (Code 200):

```
Empty response text
```

- Response (Code 400):

```
Content-Type:application/json
{
  message: "This ticket is not able to come in to the store right now,
try later"
}
```

H. External APIs

2.2.4 Middleware

The system in the current state is not using middlewares, but it's designed to use a CDN when the system grows.

2.3 Source Code Structure

In the sections below we will show the reader the structure of the source code. Please note that this is divided into 2 separate projects, the frontend for the presentation tier and the backend for the business tier.
As a convention, we will use the notation `/directory_name` for directories and `file_name.extension` for files.

2.3.1 Backend

The top level of the backend directory contains some important elements:

- `.settings.py`: Contains the configuration of the backend, libraries and parameters.
- `.manage.py`: The main file which manages how the django server will run.
- `/ticketingsystem`: Contains the generated html code which is uploaded to netlify.
- `/api`: Where the major part of the backend is handling all the requests
    - `/migrations`: Contains auto generated data of database and model links.
    - `/fixtures`: Contains sample data for testing purposes.
    - `models.py`: Database relations for converting into python objects.
    - `urls.py`: Navigation description on backend administration.

    - `views.py:` Render of the admin backend pages.

2.3.2 Frontend

The top level of the frontend directory contains 5 important elements:

- /`build`: Contains the generated html code which is uploaded to netlify.
- /`public`: Contains static files that will be bundled on compiling
- /`node_modules`: Contains all dependencies handled by node.js (Not uploaded to github)
- `/src:` Contains all the code of the application
    - `/api:` Contains helper functions that interact with the backend api
    - `/components:` Contains independant and reusable components
    - `/constants:` Contains that don't change throughout the application, such as the route names.
    - `/context:` Contains the code for the React context api, this is, the main state that is carried out through the session
    - `/pages:` Contains the components for each of the views. Each time the web page is loaded, the user only sees a header and one of these pages.
    - `/router:` Contains the code that handles the permission levels of the routes, and the logic for switching views in this SPA.
    - `/tests:` Contains the code for the tests
    - `App.js:` Main component of the web app
    - `App.test.js:` Contains the high level tests (rendering is ok)
    - `index.js:` Entry point of the web app.
    - `index.scss:` Contains the entry point for the style sheets.
- `.env`: Contains the environment variables
- `.package.json`: A general description of the application, the dependencies and the commands used. Standard in node.js applications

2.4 System Installation Instructions

2.4.1 Backend

Installing

For running the backend locally, the python interpreter and its package manager must be installed, python can be installed here and sqlite here.

- **Python**: v3.6 (used to develop).

- **Pip (python package installer)**: On version 3.6, it comes with python installation.
- **SQLite 3 Database**: v3.34.1.

First, install SQLite, adding it to the environment variables.
Then, clone the repository and inside the `/backend` and install all the dependencies and required libraries, using the command:

- `pip install -r requirements.txt`

You can also set up a virtual environment following this guide's steps 2 and 3.
Then you will have to run the migrations to the database, do this with the commands below:

- `python manage.py makemigrations`
- `python manage.py migrate`

Then, if you want to access and modify the database easely using django-rest-framework built in admin panel, you need to create a super user:

- `python manage.py createsuperuser`

Running

To run the project simply run the following command, which will set up the database and the backend server in port 8000.

- `python manage.py runserver`

Then, the backend server will be running on `localhost:8000`

Use a set up database

If you want to make use of a database provided by us, you can simply start with the sqlite file given in the repository, or you can run the following commands. Note, you will have to create a new superuser after this.

- `python manage.py flush` (deletes the database)
- `python manage.py dumpdata`
- `python manage.py loaddata users-data.json`

This will load the data described in the fixture users-data.json, which contains 3 stores and 3 managers. To login with these managers, simply enter the register page, and complete all the fields with the same values given in the file, as well as the passwords you want to use.

When you have the project running, you can access the superuser panel in the /admin page.

2.4.2 Frontend

Installing

For running the frontend locally, there are a couple of requirements that need to be satisfied. Both can be installed here.

- **Node.js**: v12.16.1 (used to develop) or higher.
- **NPM (node package manager):** v6.13.4 (used to develop) or higher
    - Any other package managers such as **yarn** should work, but we only tested with npm

Then, clone the repository and inside the `/frontend` and set the .env file with the correct environment variables (copy the `.env.example` into a `.env` file):

- REACT_APP_BACKEND_URL=http://localhost:8000 (if running the backend locally)
- REACT_APP_BACKEND_URL=http://167.172.0.34:8000 (if you want to connect to the real backend server)

The next step is to install the project locally and download all required dependencies). Run the following commands inside the `/frontend` directory

- `npm install`

Running

Finally, to run the project simply run the following command, which will set up the project in port 3000.

- `npm start`

Then go to `localhost:3000`

Testing

With everything installed, run:

- `npm run test`

Building

With everything installed, run

- `npm run build`

Which will generate the static html in the `/build` directory

For running the compiled html, the only requirement would be a http server such as this node.js one or this one for python, run in the build directory. (note, because of the environment variables getting set in the building stage, if running this make sure to have a version compiled with the correct ones.)

2.5 Deployment

Currently, CLup is deployed on two different servers, one for each tier. And they communicate through HTTP methods.
For the business tier, we are using a Digital Ocean instance, with an Ubuntu 20.04 virtual machine. This is open for requests on the IP address http://167.172.0.34:8000/, and offers an admin interface http://167.172.0.34:8000/admin for managing manually the database (this interface is given by the django REST framework). In this case, we used sqlite for development, as it is easy to set up and to use. In the real world, a more robust solution would be needed, such as mySQL or PostgreSQL, but for the purpose of keeping things simple, we stuck with sqlite here.

For the presentation tier (frontend), the system is deployed on Netlify, a serverless platform for static sites, in the url: https://clup.netlify.com. It is also set up with Continuous Integration (CI), as each push to the main branch on github triggers a deploy. For this deploy, all tests are run (static, unit and integration), and in the case they fail, the deploy is rejected. The status of the deploy can be seen on the readme of the application, with a badge given by Netlify.
Note for the reader. Currently, the API is exposed in port 8000, and it doesn't allow https (only http). In the real world, the VPN would be set up with a reverse proxy, exposing port 443 so all connections are secure. Something more interesting, is that the frontend is set up through a https connection, but connecting through http with the backend, how does it not raise a security error on browsers? Netlify does not allow insecure communications such as this one, so what is happening, is that all calls are actually being redirected through an internal reverse proxy, giving the illusion of a secure connection, while it is not. Check the file `/frontend/public/_redirects` for this set up. Again, in the real world the backend needs an https connection so all interactions are secure.

3. Integration Strategy

The integration is done with a core component that is being tested by itself working with its subcomponents alone, moving forward making the connections adding a not complete component that needs any part of the ones already tested and integrated, then the component is fully developed. Before adding another component to the testing the latest complete component is being tested along with the ones already tested, when the system is all mounted a full functionality test is done.

3.1 Entry Criteria

Integration testing should start as soon as the main components of the system are developed, and some conditions are satisfied before beginning the verification and testing activities.

- The RASD and DD documents must be complete and corrected in order to provide all the guidelines of the development, design and system architecture.
- For testing specific features of some components, external modules must have access to the APIs for a proper usage, in particular:
    - The DBMS should be configured and operative in order to allow testing over all the components that make usage of system data.
    - The testing of the AlternativesManager module requires the StoreManager module and most of the external modules are fully operative as this will take an important role in all the concurrent operations and any unimplemented section could lead to accumulative inconsistencies or deadlocks.
    - The SMS module must be connected and operative in order to maximize the queue system efficiency.
- For testing the integration between components, the other components must have their main functionalities and must have been isolated-tested and tested with the external modules before the integration.
- At least this functionalities completion is needed in order to integrate:
    - StoreManager: The core component, APIs, Ticket service, Booking Service and Queue service must be operational with sample data.
    - UserManager: Manager Service ready to check stores.
    - AlternativesManager: Business logic to handle re-booking with DB.
    - Redirector: API connection between components.
    - WebApp: Web pages and forms ready to send data to the backend server.

3.2 Elements to be Integrated.

Front-end components: Web Application for Customers and Web Application for Managers

Back-end components: StoreManager, UserManager, AlternativesManager and Redirector.

External components: DBMS

None of the components of the frontend has connection directly to the external modules, and by the architecture of the system, the backend components cannot be fully tested until integration with another one and its related external modules.

The integration of StoreManager, UserManager and the DB are the core of CLup.

3.3 Integration Testing Strategy

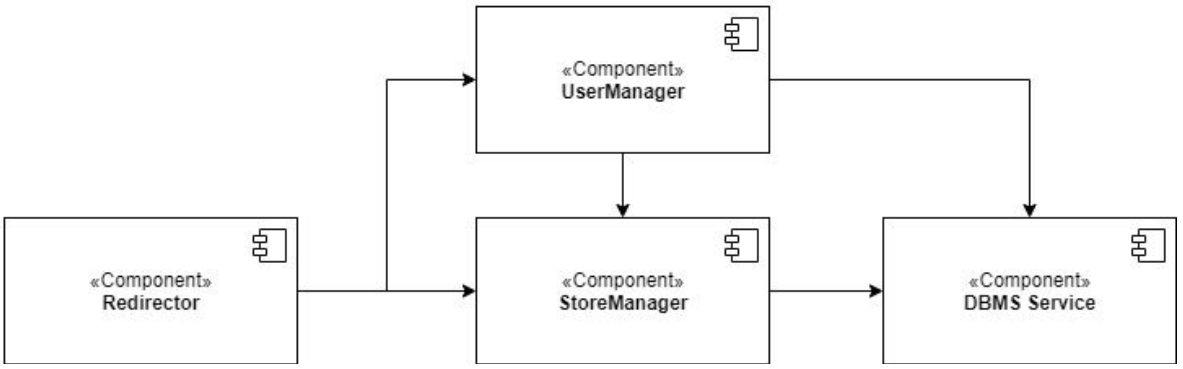
The team will follow a bottom-up strategy as the core of the system is based on a few components, the development will start with the core backend components, and parallely the frontend web page, and gradually integrate and test components to the core.

3.4 Sequence of Component/Function Integration

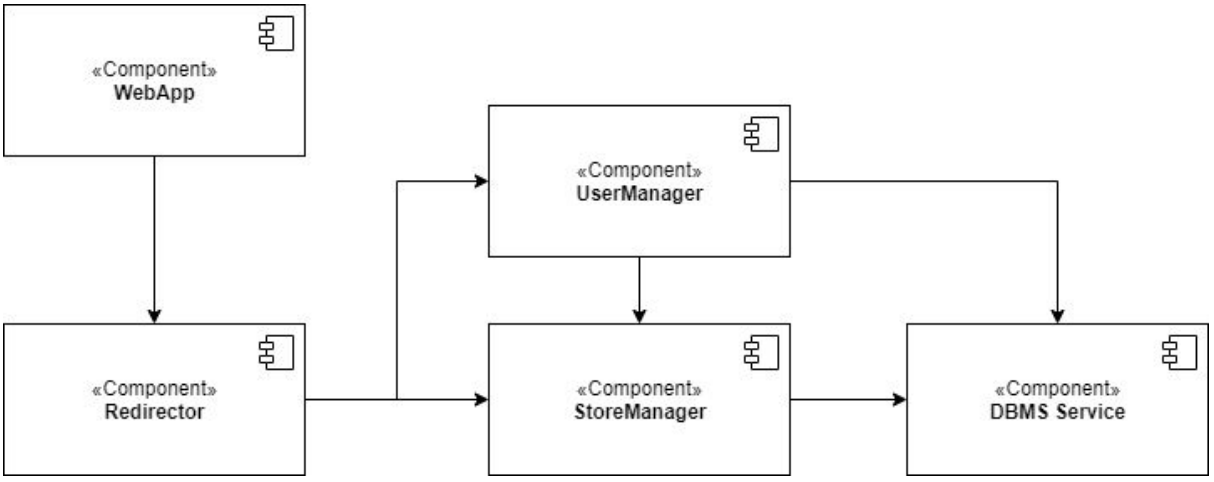
1. StoreManager component in parallel with the frontend's WebApp, focusing mainly on the DBMS connection and API testing.

2. Later on, after the UserManager's component ManagerService is developed, it will be integrated with the StoreManager and completed.

3. The redirector will be developed and connected to existing backend APIs.



4. In this point, the WebApp can be connected with the redirector and the redirector will be integrated to the StoreManager for a testing experience close to what the user will do and in the state of the system, no new backend technical errors should show.



5. With all the system components integrated, a full system test can be made. This system test was done manually, using the user flows described in the RASD, as the scenarios.

4 Individual Steps and Test Description

4.1 Backend

For the backend, mostly integration tests of the components with APIs were done, the methodology used was using an external tool “postman” to test the APIs responses and using Django UnitTest module to test the database connection and updates.

4.1.1 Store Manager - Integration Test Case 1

Test ID	IC1 - T1
Components	StoreManager, DBMS
Input Specification	Get Stores List
Output Specification	The DB returns all the store information inside
Description	StoreManager sends a read query to the DBMS in order to search for all the existent stores in system
Environmental Needs	-

Test ID	IC1 - T2
Components	StoreManager, DBMS
Input Specification	Update Store data
Output Specification	A confirmation of update is received
Description	StoreManager sends a update query to the DBMS in order to change some values inside a store
Environmental Needs	-

4.1.2 User Manager - Integration Test Case 2

Test ID	IC2 - T1

Components	UserManager, DBMS
Input Specification	Register New user credentials
Output Specification	If the user phone number does not exist on system, the new account is created on the database ● Exception: Invalid input for number/password (see Test Data Section), the user should not be created
Description	UserManager sends an update query to the DBMS in order to search for a existing user with provided credentials, if no similar credentials found, user is notified through web app
Environmental Needs	-

Test ID	IC2 - T2
Components	UserManager, DBMS
Input Specification	User Login Credentials
Output Specification	If the user phone number and password matches with one existent on the system,, an auth token is returned. ● Exception: Invalid input for number/password (see Test Data Section), no auth token will be received.
Description	UserManager sends an read query to the DBMS in order to search for a user with provided credentials, if it's found, the user gets authenticated, if no similar credentials are found, user is notified through web app that an error on number/password was committed.
Environmental Needs	- (if needed another test to be completed before)

Test ID	IC2 - T3
Components	UserManager, StoreManager

Input Specification	Booking test
Output Specification	If the user booking data matches with the store id, and the store has the available slots, a ticket code is generated and returned. ● Exception: Invalid user session. ● Exception: Invalid date/time received. ● Exception: Not found store. ● Exception: Unavailable slots at visit date.
Description	A booking request is sent to the StoreManager, the store manager verifies the user information with the UserManager component, if the information matches, it verifies the store data on success, a ticket code is created .
Environmental Needs	- (if needed another test to be completed before)

4.1.3 Redirector - Integration Test Case 3

Test ID	IC3 - T1
Components	Redirector, StoreManager
Input Specification	Booking data
Output Specification	If the StoreManager verifies all the data provided, a ticket is returned to the redirector
Description	StoreManager goes through the booking module and sends data to UserManager, on success, returns ticket as response.
Environmental Needs	FStore - T1 and IC2 -T3 tests successfully completed

Test ID	Pending arrangement (Integration/Function/Full System - #)
Components	Redirector, UserManager

Input Specification	Login user message
Output Specification	If the user phone number and password matches with one existent on the system, an auth token is returned from the UserManager.
Description	UserManager does receive a call from the redirector, then it returns the response depending on the credentials.
Environmental Needs	IC2 - T2 test successfully completed

4.1.4 Store Manager - Stores - Functionality Test

Test ID	FStore - T1
Components	StoreManager
Input Specification	Create ticket
Output Specification	A ticket code is returned by the component.
Description	A data stub with booking is received by the StoreManager as verified data, a ticket code is returned by the component.
Environmental Needs	-

4.1.5 User Manager - Sessions - Functionality Test

Test ID	FUser - T1
Components	UserManager
Input Specification	Logout
Output Specification	A confirmation message is returned from the component and the auth token is deleted.
Description	UserManager receives a logout message and it closes the session of the current user, deleting the auth token.
Environmental Needs	IC2 - T1 and IC2 - T2 tests successfully completed.

4.2 Frontend

For the frontend web app, mostly functionality tests of the components were done. The methodology was to use the testing library Jest in addition to the React-testing-library, both of which gives us the functionality to mock server calls and user events.

4.2.1 WebApp - Integration Test Case 4

Test ID	IC4 - T1
Components	Redirector, WebApp, UserManager, StoreManager
Input Specification	Register new user
Output Specification	If the user phone number does not exist on system, the new account is created on the database ● Exception: Invalid input for number/password (see Test Data Section), the user should not be created
Description	UserManager sends an update query to the DBMS in order to search for a existing user with provided credentials, if no similar credentials found, user is notified through web app
Environmental Needs	-

4.2.2 WebApp - Main rendering integration

Test ID	ICWeb - T1
Components	Web app
Input Specification	User loads the web application
Output Specification	If the user is not authenticated, he can only see what non authenticated users can see (line up, login) The same for customers (authenticated users) and managers.
Description	The web app router handles this directly, and only renders what the user should see.

Environmental Needs	User loads the web application in a browser

4.2.3 WebApp - Stores selection

Test ID	ICWeb - T2
Components	Web app
Input Specification	User loads line up page, where he can select a store to get a ticket
Output Specification	The user should see the stores available from the API
Description	A mock API is set, and the system loads the component rendering the stores. The test passes if the name of the stores in the API are shown to the user.
Environmental Needs	User loads the line up page

4.2.4 WebApp - Request a ticket

Test ID	ICWeb - T3
Components	Web app
Input Specification	User requests a ticket while being anonymous
Output Specification	The confirmation page is loaded if the data input (phone number) is correct.
Description	The system tests that if the user did not provide a correct phone number, no ticket gets created and it is shown an alert. If the phone number is correct, the ticket is created and the confirmation page is loaded.
Environmental Needs	Anonymous user loads the line up page

Test ID	ICWeb - T4
Components	Web app
Input Specification	User requests a ticket while being authenticated
Output Specification	The confirmation page is loaded, and a ticket gets created with the user's phone number automatically
Description	The system loads the user's phone number automatically, and creates a ticket. The test passes if the ticket gets created with the correct phone number
Environmental Needs	Customer (registered user) loads the line up page

4.2.5 WebApp - Check ticket status

Test ID	ICWeb - T5
Components	Web app
Input Specification	User loads the page after scanning a ticket to check the status, with a valid ticket.
Output Specification	The confirmation page is loaded, and the ticket's data is shown
Description	The test checks if the ticket's data is shown on screen for the user to see.
Environmental Needs	User scans (or enters url directly) a ticket.

Test ID	ICWeb - T6
Components	Web app
Input Specification	User loads the page after scanning a ticket to check the status, with an invalid ticket.

Output Specification	An error message is shown, and the user gets redirected to the home page.
Description	The system calls the api with an invalid ticket id, and returns an error.
Environmental Needs	User scans (or enters url directly) an invalid ticket.

5 Program Stubs and Test Data Required

For the testings related to the backend, a testing data set is provided inside the fixtures folder, containing sample data, also the database is initialized with basic sample data for the full system testing.

In the case of the frontend, we needed to start working in parallel meanwhile the backend was not 100% completed. We discussed the API format, and worked based on it. For making the requests to the API, we used the axios library, so what was done in the meantime, was that we separated the logic of this calls with everything else, and just returned what was supposed to be returned. Later we integrated correcting small mistakes made during the process.
For the testing, we took a different approach. Using the Mock Service Worker library, we mocked an API, so the components that made the requests did not have to change.

6 Effort Spent

Topic	**Hours**
Initial setup of document	1hr
Section 1	1hr
Section 2	3hr
Section 3	4hr
Section 4	9hr

Section 5	1hr
Final review of document	3hr

7 References