Software Engineering II 2020/2021

February 2021

Implementation and Test Deliverable

CLup: Customers Line-up Software

Link to GitHub repository

Version: 1

Authors:

R. Cedeno, D. Cumming, A. Pugliese

Contents

1. INTRODUCTION

The need of avoiding physical interactions in order to prevent the spreading of the COVID-19 disease has put a hard constraint on the businesses that are based on in-person shopping since there are safety restrictions on how many people can be inside of a certain building while there must be a distance of at least one meter between them. To address these constraints, the implementation and testing of the Customers Line-Up Software is presented in this document. The software functionalities are guided by the requirements presented in the project's RASD document and the development process was steered by the project's DD.

1.1 Scope

The implementation process carried out in this project had the aim of guaranteeing an stable and high-quality software prototype with essential functionalities that are proposed to allow users to line-up to virtual queues to enter a store, guaranteeing safe conditions in terms of social distancing inside the building. Additionally, the software will allow users to book a queue number on a specific time slot.

As elaborated in the design process, the application considers a web, map-based and mobile-friendly user interface that explicitly indicates the degree of availability of the geo-located stores around the user. The philosophy behind the design is based on minimizing the number of different application pages and user clicks in such a way that each one of the functionalities can be addressed in a user-friendly way.

In this document, a general overview of the considered functionalities that are developed in the prototype is elaborated as well as a justification of each one of the design choices, including from the code building to the development frameworks. Additionally, the technical details about the testing procedure that provides completeness and error detection is presented.

1.2 Definitions, Acronyms, Abbreviations

1. RASD: Requirements and Specifications Document
2. DD: Design Document
3. FCFS: First Come First Served
4. OSM: OpenStreetMap.
5. API: Application Programming Interface.
6. JWT: JSON Web Token.

7. IaaS: Infrastructure as a Service.

8. ORM: Object-relational mapping - It is a programming technique for converting data between incompatible type systems using object-oriented programming languages.

9. QRCode: Quick Response Code: optically readable image that in an arrangement of black squares makes available the representation of information.

2. IMPLEMENTED REQUIREMENTS

The requirements/functions that are actually implemented in the software (with motivations for including them and excluding others if applicable).

The functionalities that were planned and designed for this project were already specified in the RASD and DD. Here we describe the main advantages and drawbacks associated with the functionalities that were actually implemented and an explanation of why we choose them.

2.1. Registration & Login

Authentication is provided because a certain state needs to be set for each user (e.g. waiting, shopping). As detailed in the next section, this functionality is handled by built-in secure libraries for this purpose.

2.2. Map

The map is considered in this prototype because, as specified in the DD, it is the core in which the rest of the functionalities are acting on in such a way that operate as the main default page of the application. The map functionality is fundamental for allowing the users to interact with the application in a way that makes them geo-locate their preferences, so they are able to choose their preferences based on their convenience. The map presents all of the supermarkets available for booking or line up in a 2.2km (0.02deg) radius around the user location. Each supermarket is marked with a color (green, yellow or red) depending on their availability (i.e. waiting times). With this last functionality, the user will be able to select the supermarket according to their needs or convenience. The map also presents the functionality of filtering, which helps the user to refine their search and have better results to what they are looking for.

The main advantage of the map is that it is very intuitive to use and interact with, it presents all the available information in a fast and visual way; it is very practical and there is no need to access the information in any other way.

2.3. Line-up

The line-up allows the users to have a number in the queue for a supermarket to enter as soon as possible. This is the most basic functionality that was expected in the requirements. The queueing of the users is approached with a FCFS heuristic.

The main advantage of this functionality is that the user will be able to queue in a supermarket if they have the need to enter ‘as soon as possible’. In case the user selects a supermarket with low waiting times (marked in ‘green’ on the map) they will not need to wait any time to enter. The main drawback of this functionality is that, if there is no waiting time, the user will only have 5 minutes to reach the supermarket and enter, if this doesn’t happen then the QRCode will expire and they would need to request a new one. The second drawback is that if there is a waiting time, the user will not be able to enter the supermarket immediately.

2.4. Book

The booking has the functionality of allowing the users to enter the store on a specific time slot that they can choose from a number of date and time options. After selecting the specific time that the user would like to book and confirming the request, they will be redirected to a page to visualize an automatically generated QR code to be scanned at the entrance of the store. There might be cases in which several users want to register into the same date and time, in this case the queueing of the users works in the same way as the line-up, with a FCFS heuristic.

The main advantage of this functionality is that it permits the user to plan its activities and not queue while waiting to enter the supermarket, instead they will be able to get access on the exact time that was requested. The main drawback of this functionality is that the user will not be able to queue immediately in the supermarket, bookings can only be made after one hour of the current time; instead the user would need to use the line up functionality..

2.5. QR code

For each line-up or booking request, the backend generates a secret token that is unique to each booking and stores it in the DataBase as part of the information pertaining to each line-up or booking. Whenever a user requests a line-up or a booking they will be redirected to the QRCode page where the secret token is afterwards requested. Through a functionality present in the front end a QRCode is generated according to the token number. This code plays an important role in the interaction with the application, since it has the functionality to let the user enter the supermarket whenever it is scanned.

2.6. Waiting Time

The waiting time is a crucial element for the application to be fully functional. It is responsible for calculating the time that the user will need to wait until the user will be permitted to use the QRCode to enter the supermarket. This waiting time doesn't change for the people that booked a specific date and time, but it is modified for the line-up users according to the number of people present in the queue, as well as the entering and exit times to the supermarket.

The main advantage of waiting time is that it shows in a clear way what is the waiting time for them to be able to enter the supermarket. This is important because without this visual aid, the user would never know when it is their turn to be able to access the supermarket.

3. DEVELOPMENT FRAMEWORKS

3.1 Programming Languages

The frameworks and programming languages that were used to implement this application were already specified in the Design Document in section 2.4. Here we will talk about the advantages, the disadvantages and the reason why we chose this stack.

In the following subsections, the stack is composed of Vue.js, Flask, and SQLAlchemy.

3.1.1 Vue.js and Nuxt.js

Vue.js was chosen among other popular web frameworks because of its simplicity, flexibility, and its well-defined ecosystem. Then, there is Nuxt.js, a high-level framework that is built using Vue.js which simplifies our job generating the folder structure, managing routing, and using Nuxt.js predefined functionalities.

The disadvantages, in general, of Vue.js, are that it is relatively new, compared to other frameworks, so it has a smaller community which means less documentation and plugins. Regardless of this, we could find enough documentation and adequate resources for our necessities.

3.1.2 Python and Flask

Python was chosen as the backend programming language because of the familiarity of all the team members with it. Then, the chosen framework was Flask, mainly because of its minimalism as it is a micro web framework that has minimal dependencies on external libraries, written in Python, which was formed for a faster and easier use, and also has the ability to scale up to complex applications.

The disadvantages of using Flask are mainly seen in the production phase when many requests are made at the same time as Flask, like many other frameworks, handles the requests one at a time. There are two options for this, one is to migrate the backend to a more robust framework and the other is to enable auto-scaling in the IaaS where the backend is hosted with its respective load balancer.

3.1.3 SQLAlchemy and PostgreSQL

PostgreSQL is a relational database management system (DBMS) emphasizing extensibility and SQL compliance. It will contain all the tables where the data will be inserted and queried. The communication with the web server will be by migrating the database using SQLAlchemy, meaning also that the tables' columns will be defined using models in Python.

SQLAlchemy is an open-source SQL toolkit and object-relational mapper (ORM) for Python. It is used through Flask-SQLAlchemy. The main advantage of using this is that all the models, queries, and interactions with the database are written in Python on the server side.

3.2 Middlewares

3.2.1 JSON Web Token (JWT)

JWT is an open standard that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed.

It is used for authentication through Flask-JWT. It securely manages the users' information and the sessions using tokens that are required to use the application in general. The advantage of this approach is that it is not necessary to worry about sessions and cookies, the Flask server sends a JWT that is valid for a specific time and is used to automatically authenticate to use the application services.

There are also some disadvantages about JWT that include the expiration at specific intervals and not based on a specific time after the last user-application interaction, two known exploits: the "none" algorithm and the HMAC hack, and the fact that they are not easily revocable. Despite all this, JWT is a good option to manage sessions in this application. Furthermore, some of the disadvantages can be "fixed" by implementing also well-known solutions in case of production.

3.3 Application Programming Interfaces (APIs)

The APIs used in this project

3.3.1 Vue GeoLocation API

It is a Geolocation Web API used in the frontend to retrieve the user's location in order to show the nearby supermarkets. The user's location is never sent to the server, neither stored in the database.

3.3.2 Overpass Turbo

Overpass turbo is a web-based data mining tool for OpenStreetMap (OSM) which runs overpass API queries and shows the results on an interactive map. It is possible to export the results as JSON, GeoJson, KML, and others.

Using this API we retrieved a JSON containing the list of Milan's supermarkets with the fields id, latitude, longitude, and name:

```
{
  "id": 250969620,
  "lat": 45.447585,
  "lon": 9.1420016,
  "name": "LIDL"
 }
```

This list is used in the server to populate the table Supermarket in the database. This population is mainly done for developing and testing purposes since, in a real scenario, only the supermarkets that decide to use the application will be included in the database.

In figure 1, we can see the location of the supermarkets retrieved from Overpass turbo.

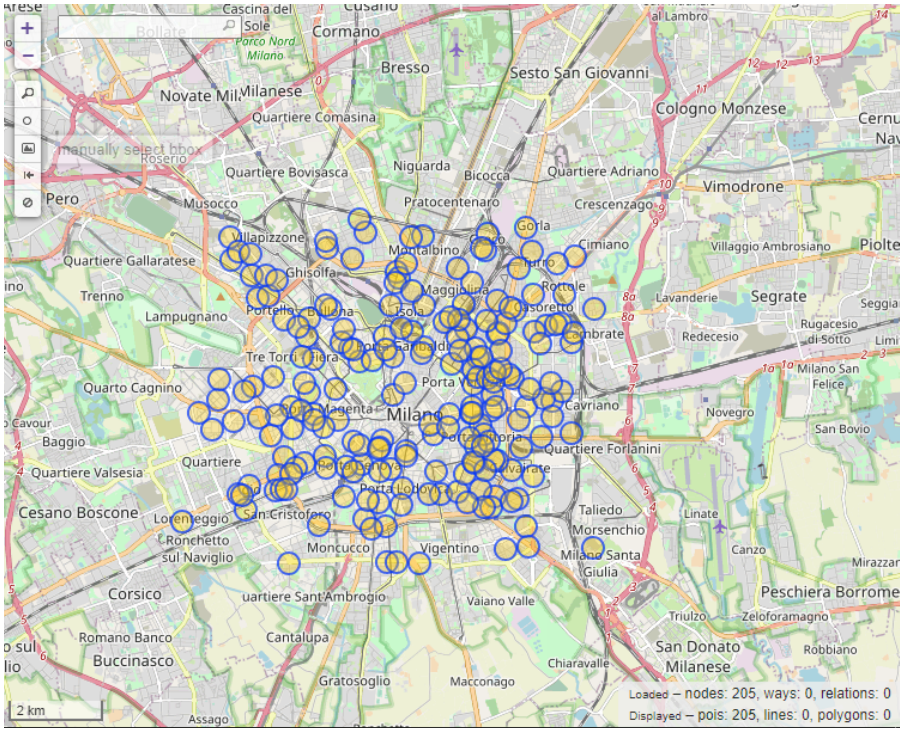
Loaded – nodes: 205, ways: 0, relations: 0
Displayed – pois: 205, lines: 0, polygons: 0
manually select bbox
2 km

Figure 1: Milan's supermarkets from OSM

4. STRUCTURE OF THE SOURCE CODE

As specified in the design document, the code is splitted into two subsystems: the server (back-end) and the client (front-end).

4.1. Server

The server is in charge of most of the logic that is behind the software functionalities, as well as the communication with the database. We can highlight the two most important pieces of code in the server:

- main.py: This piece of code is where the application is built. It contains all the APIs that the client components need to access the DB, as well as most of the logic that is required to perform the functionalities, including a function that runs every 5 seconds to update waiting times of the queues and expire timed-out requests if exist. The model of entities that is stored in the DB is called from this piece of code.

- models.py: In this code, the definition of the entities that are stored in the DB is contained. It is specified each one of the attributes of each class as well as how the initialization of them is performed. The implemented classes in the DBMS of the software are illustrated in the Entity-Relationship Diagram presented in figure 2. The entity User identifies each account that is able to perform a request that will be stored as a Waiting class that is associated with the Supermarket class (specific supermarket that the request was made for). The queueing of these requests will be managed by the logic of the application in main.py. When the turn is conceded, the users are able to scan the QR code in the supermarket in order to get into the supermarket. In this moment the system will move from Waiting into a Shopping instance to specify that the User entered the specific Supermarket. Finally, when the Users leave the Supermarket after Shopping, they scan the QR code and the system removes them from the Shopping table. The times of entering and leaving the Supermarket are registered in the Record table for future waiting time estimation purposes.

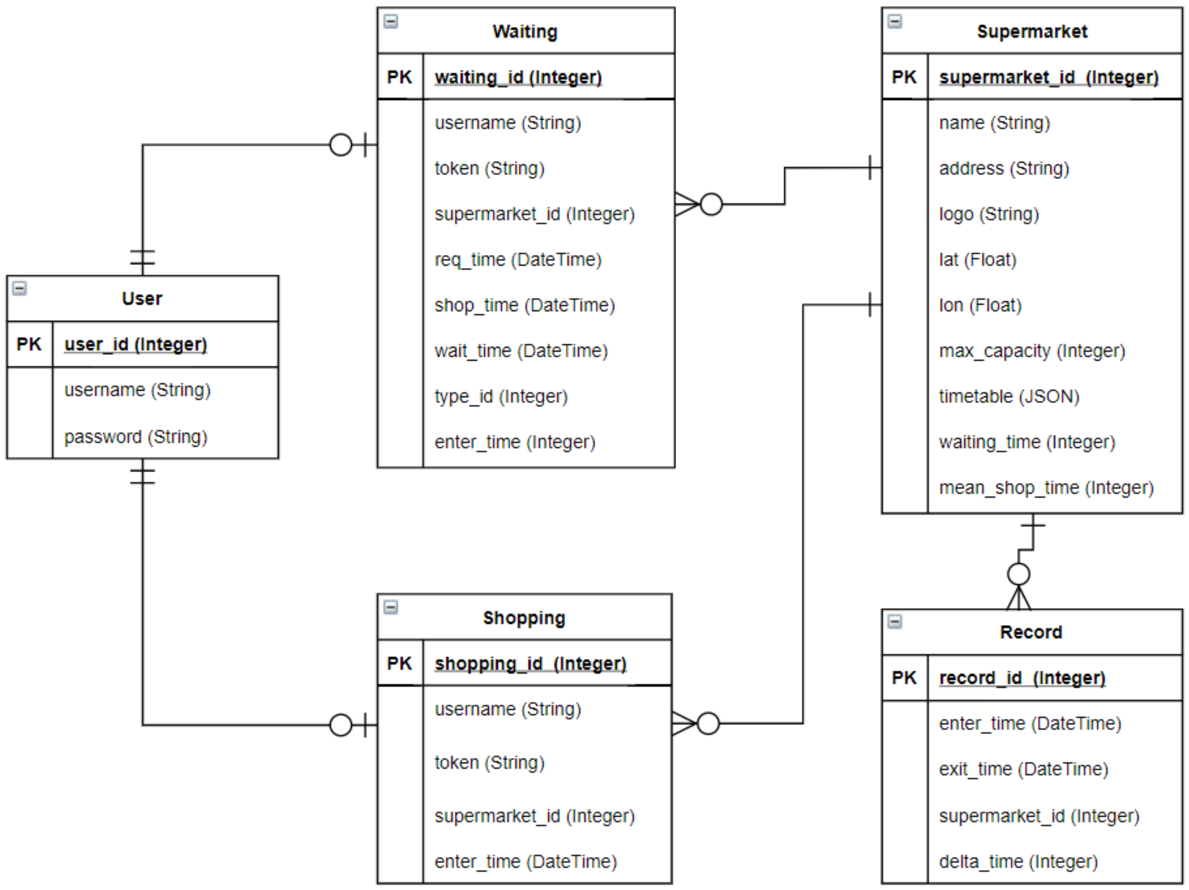

Figure 2: Entity-Relationship Diagram

4.2. Client

The structure of the client is based on the pages of the web application, so each one of them has its own code, whose internal structure is splitted into the template (HTML), script (JavaScript) and style (CSS) into a single code file. The implemented pages are the following:

- login.vue & register.vue: Pages in charge of allowing the user to complete the form to create an account in the system or authenticate in order to proceed to the software functionalities.
- index.vue: Map user interface through which all the software functionalities can be accessed and the availability of each supermarket in the map can be seen. The supermarkets, requests and logs (e.g. users waiting its turn or shopping in store) are fetched by means of APIs in the back-end approached by functions in this front-end through HTTP requests. Moreover, the logic behind the filtering of the supermarkets is contained in this piece of code.
- booking.vue: a user interface based on a calendar and time selection is provided to let the users select a future time slot for booking.
- list.vue: list of supermarkets ordered by availability. The users can select them in order to perform line-up. Suitable for users that want to shop as soon as possible.
- qrcode.vue: this piece of code retrieves the token associated with a request and encodes it as a QR code. Moreover, it also fetches the waiting time associated with a request from the back-end and displays it so the user can see the time to be able to enter the supermarket.

5. TESTING

5.1. Backend - Pytest

Pytest is the main library for python testing and is widely used among the programming community.

For the backend tests we created a secondary testing database that erases all the data after each test session, meaning that we tested our endpoints and functions on a clean database instead of the development or production database.

The testing database configuration was made using the Flask setting variables and SQLAlchemy connection variable. It is a replica of the development database, the same schema.

We tested, in general, each table of the database model, some tests more relevant than others depending on the constructor of each model.

Regarding the functionalities, we defined and tested some common and relevant tests in the application usage that will be formally defined as follows.

Case 1: Load supermarkets

GIVEN a Flask application configured for testing

WHEN the '/' page is requested (GET)

THEN check that the response is valid

Case 2: Rejected get in

GIVEN user 1 and user 2 lining up

WHEN user 2 tries to enter. The '/getin' page is requested (POST)

THEN It should be rejected because it is not his turn

Case 3: Accepted get in

GIVEN user 1 and user 2 lining up

WHEN user 1 tries to enter the '/getin' page is requested (POST)

THEN It should be accepted because it is his turn

Case 4: Full supermarkets

GIVEN user 1 and user 2 in the supermarket and user 3 lining up

WHEN user 3 tries to enter the '/getin' page is requested (POST)

THEN It should not be accepted because the supermarket is full

Case 5: Now available

GIVEN user 1 and user 2 in the supermarket and user 3 lining up

WHEN user 1 or user 2 gets out and user 3 tries to enter the '/getin' page is requested (POST)

THEN It should be accepted because the supermarket is not full anymore

Case 6: Booking not available

GIVEN user 1 and user 2 booked supermarket at the same time

WHEN user 3 tries to book for the same time '/booking' page is requested (POST)

THEN It should not be accepted because the supermarket is already full at that time

5.2. Frontend - Jest

Jest is the main testing library for javascript and is one of the default options for testing in Nuxt.

Vue also has some helping libraries for testing like vue test-utils, that allow to create mocking data, a testing mock server, set the data for the vue components and run functions inside each component.

In Jest we test that all the pages are rendering correctly by checking any keyword in each page. Then, one of the most relevant functionalities in the frontend are the filter and sending the filtered supermarkets list to the “/list” view. For the second case we implemented a test to prove that the list of supermarkets was correctly displayed in order from the least to the maximum waiting time.

One relevant fact of the tests and the application in general is the usage of the vuex store module that allows us to share data between components, this is also

6. DEPLOYMENT

The deployment, for both the backend and frontend application, was made in Heroku. The database was also deployed in Heroku in the same app as the backend as an add-on.

We used the free tier of Heroku that is enough to host the current state of the application. Heroku also allows auto-scaling as the traffic grows, by adding more dynos.

6.1 Frontend deployment

For frontend we created a Heroku application that runs a heroku/nodejs buildpack, meaning that the web server that is serving the application is Node.js.

Nuxt is already preconfigured in the buildpack, so we had to add only the source files and a Procfile, that are the Heroku instructions for building the apps, with the command

```
web nuxt start
```

The builder automatically installs the dependencies from the *package.json* and Heroku uses git as the version control system and as the way to upload the files.

After the build is complete, Heroku hosts the application in:
https://clup-frontend.herokuapp.com/

Note that also Heroku adds https.

6.2 Backend deployment

For the backend we created a Heroku application that runs a heroku/python buildpack. This allows us to run a wsgi application using a widely used library for deployment called *gunicorn*.

We had to add a file called *wgsi.py* as the entrypoint for the application to work with gunicorn and add a Procfile with the instruction to start the app:
web `gunicorn wsgi:app`

Also, we added the *requirements.txt* for the builder to install the dependencies, and the source code of the backend.

The builder automatically installs the dependencies and recognizes the app as a wsgi application.

After the build is complete, Heroku hosts the application in
https://clup-server.herokuapp.com/

The database was added to the backend application as a postgresql add-on, that hosts a database and gives us the connection parameters.

With the connection parameters we connected the deployment-ready backend application to the production database, migrated the tables and added the supermarkets data.

This database is the free tier of Heroku Postgresql that allows for 10.000 rows, 500MB of space and 20 concurrent connections, enough for hosting the application without problems for low traffic.

7. INSTALLATION INSTRUCTIONS

To run the application locally, the installation instructions are specified in the readme files of each folder.

As the application is web-based, it does not need to be installed to use it, only click on this link where it was deployed:

https://clup-frontend.herokuapp.com/

8. EFFORT SPENT

Jesus Rodrigo Cedeño Jimenez	80 hours
Diego Andres Cumming Cortes	80 hours
Angelly de Jesus Pugliese Viloria	80 hours

9. REFERENCES

1. vue-geolocation-api. (2021). Retrieved 1 February 2021, from https://www.npmjs.com/package/vue-geolocation-api

2. Overpass turbo - OpenStreetMap Wiki. (2021). Retrieved 1 February 2021, from https://wiki.openstreetmap.org/wiki/Overpass_turbo

3. JWT.IO - JSON Web Tokens Introduction. (2021). Retrieved 1 February 2021, from https://jwt.io/introduction

4. Flask-JWT — Flask-JWT 0.3.2 documentation. (2021). Retrieved 1 February 2021, from https://pythonhosted.org/Flask-JWT/