CLup

Customers Line-Up

ITD

Implementation and Testing Document

Davide Merli - 10578363
Dario Passarello - 10566467

Deliverable:	Implementation Document
Title:	Implementation Document
Authors:	Davide Merli, Dario Passarello
Version:	1.0
Date:	07 February 2021
Download page:	https://github.com/davidemerli/MerliPassarello/
Copyright:	Copyright © 2020, Davide Merli, Dario Passarello – All rights reserved

Contents

List of Figures

1 Introduction

1.1 Scope

This document contains all information regarding the development and the testing of the CLup prototype application and system.

The ITD is a follow-up to the **RASD** (Requirement Analysis and Specification Document) and the **DD** (Design Document) which define in detail the system as a whole, all the planned features and the design choices.

1.2 Document Structure

Chapter 1 explains the purpose of this document and the relation between this Implementation and Testing Document and the RASD and the DD. Acronyms and Definitions used through the whole document are listed and explained. It provides the history of the document revisions and this very description of all the chapters, together with the documents used as a reference.

Chapter 2 lists all the defined Requirements in the other documents and provides details about the fullfilling of those requirements. Considering that presented application is a prototype, not every functionality has been implemented.

Partially implemented features are also discussed in detail.

Chapter 3 presents all the adopted Frameworks both on the backend and on the frontend. The implementation is put into perspective of the defined and analyzed system structure showed in the Design Document. Every choice about existing software is provided in this chapter.

Chapter 4 describes the structure of the source code for both the frontend and backend. Useless information about autogenerated code/configs is omitted if not needed to explain implementation aspects.

Chapter 5 analyzes all aspects regarding the testing of the system. Adopted methodologies and achieved results are all presented in this chapter, together with the test coverage.

Chapter 6 provides insights on how the system has been deployed for manual testing, how to run locally the server or the client application, and which scripts are provided to do so.

Chapter 7 shows the time spent from each member of the team for writing this document.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Acronyms

- **S2B**: Software To Be

- **RASD**: Requirement Analysis and Specifications Documents

- **REST**: REpresentational State Transfer

- **API**: Application Programming Interface

- **UX**: User Experience

- **UI**: User Interface

- **SSO**: Single sign-on

- **QR code**: Quick Response code

- **OS**: Operating System

- **RAM**: Random Access Memory

- **LAN**: Local Area Network

- **GPS**: Global Positioning System

- **GB**: GigaByte

- **TCP/IP**: Transmission Control Protocol/Internet Protocol

- **HTTPS**: Hypertext Transfer Protocol Secure

- **IoT**: Internet of Things

- **MQTT**: Message Queuing Telemetry Transport

- **RAID**: Redundant Array of Independent Disks

- **UML**: Unified Modeling Language

- **TDD**: Test-driven development

- **WSGI**: Web Server Gateway Interface

1.3.2 Definitions

- **Access controller**: a subsystem that permits the entrance of customers into the store. It can be a device like a smart turnstile that reads customers tickets or just a person of the store staff manually scanning tickets.

- **Business account**: a CLup account that is destined to store managers or operators and therefore the 'business' side of CLup

- **In-Site ticket**: a ticket that is taken by a customer near the store. It can be both a virtual paperless ti an emitter near the store premises

- **Virtual ticket**: a ticket issued through the CLup application

- **Physical/Paper ticket**: a printed physical ticket, emitted by a printer near the store premises

- **Valid Ticket** (at time X): a ticket that has a code recognized by the CLup system and valid for the specified time

- **Time slot**: a time delta that is associated with a number of bookable tickets (which varies and is customizable from store to store)

- **People-Counting System**: a subsystem that permits the counting of the number of people inside the store. It can comprehend a device like a proximity sensor or a smart turnstile, or it can be a person of the store staff manually counting people.

- **Customer Application**: the CLup mobile application destined to customers that want to shop inside stores adopting CLup

- **Operator Application**: the CLup mobile application destined to store staff to monitor entrances and statistics

- **Store Main System**: the store main server that communicates directly with CLup servers. All store subsystems and smart devices should communicate with it through an Intranet

- **Geocoding API**: Geocoding converts addresses into geographic coordinates to be placed on a map. A Geocoding API allows the use of their services to permit translation between textual addresses and Latitude/Longitude coordinates

- **Map API**: An external services that provides operations of geographics maps and the download of map information, usually of the places in proximity of given geographics coordinates

- **Hashed Password**: When a password has been "hashed" it means it has been turned into a scrambled representation of itself. A user's password is taken and – using a key known to the site – the hash value is derived from the combination of both the password and the key, using a set algorithm.

- **Time to market**: is the length of time it takes from a product being conceived until its being available for sale.

- **Alpha Test**: a trial of machinery, software, or other products carried out by a developer before a product is made available for beta testing.

- **Beta Test**: a trial of machinery, software or other products in the final stages of development, carried out by a party unconnected with the development process.

- **DevOps**: is a set of practices that combines software development (Dev) and IT operations (Ops). It aims to shorten the systems development life cycle and provide continuous delivery with high software quality.

- **Quality Assessment**: is the data collection and analysis through which the degree of conformity to predetermined standards and criteria are exemplified.

1.4   Revision History

- 1.0: 07 January 2021 - First Release

1.5   Reference Documents

- R&DD Assignment A.Y. 2020-2021 - Elisabetta di Nitto, Matteo Giovanni Rossi

- Software Engineering II slides and material - Elisabetta di Nitto, Matteo Giovanni Rossi

- CLup RASD - Davide Luca Merli, Dario Passarello

- CLup DD - Davide Luca Merli, Dario Passarello

- Material Design Guidelines

- Convetional Commits Guidelines

- Git - versioning system

1.6   Used Tools

- Umlet - for UML diagrams

- Github - for code hosting and version control

- LaTeX - to write this entire document

- Visual Studio Code + Latex Workshop - as a LaTeX environment

- Visual Studio Code + Flutter & Dart plugins - as a Flutter environment

- Postman - to test the REST API

- pgAdmin - to manage the postgres test dataset

- Amazon Web Services - to deploy the backend

- Github Pages - to deploy the web application

1.7   Team Contacts

- Dario Passarello: dario.passarello@mail.polimi.it

- Davide Merli: davideluca.merli@mail.polimi.it

2 Requirements and Functionalities Implemented

The prototype functionalities focus on the 'line-up' feature: the customer can only retrieve a ticket to enter the store as soon as there is space. The 'book a visit' feature is not implemented. The scope of the prototype is to show how the system works from the point of view of the customer. For this reason the prototype focuses on the User Experience. The management side is kept at a bare minimum providing only the store operator functions to call customer to the entrance, scan tickets and view real time data about the store crowdedness. The system does not provide functions to insert new stores, neither to create new operator accounts; some random data is provided to the testers in order to be able to test the system, and the implemented features.

2.1 Component and Deployment View

With respect to the Design Document, to ease and speedup the task of developing a prototype some components were moved or removed.
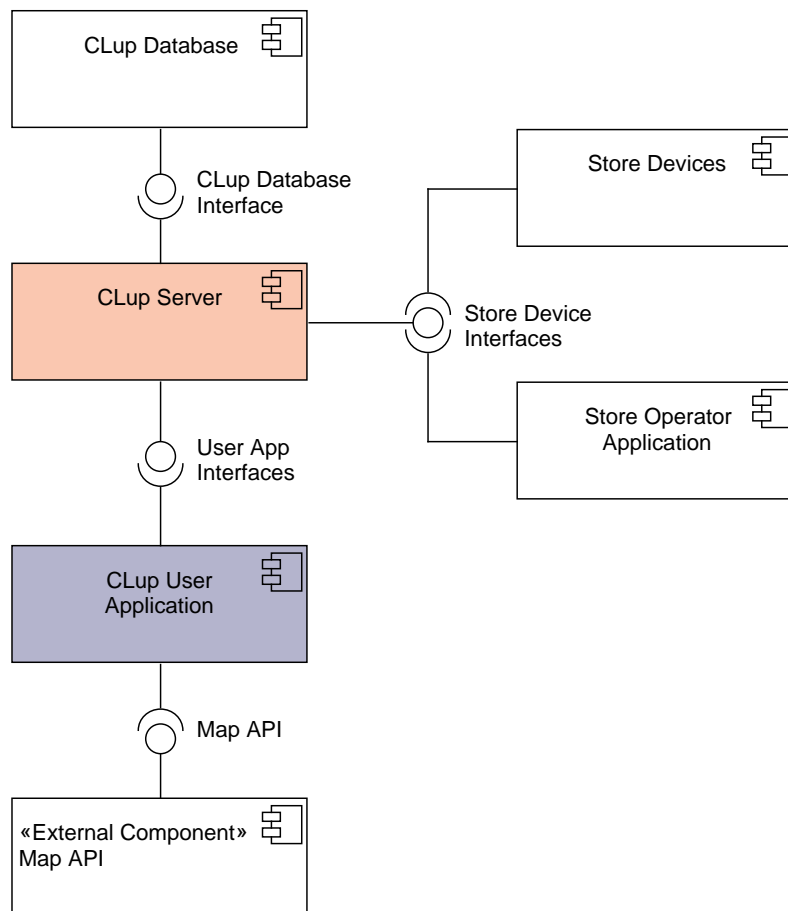
Figure 1: Implemented Component Diagram

In the DD the original component diagram moved the functions handled by the store 'outside' of the CLup server. This is convenient when the two systems handle different types of tickets and different type of collected statistics, but in the prototype case there is only one type of ticket to handle. In order to avoid overengineering the prototype, the two macro components were merged in a unique backend macro-component called CLup Server in the diagrams.

Figure 2: Prototype Deployment Diagram

Due to the merge of the two components the backend now can run in a single machine isolating the different modules in containers.

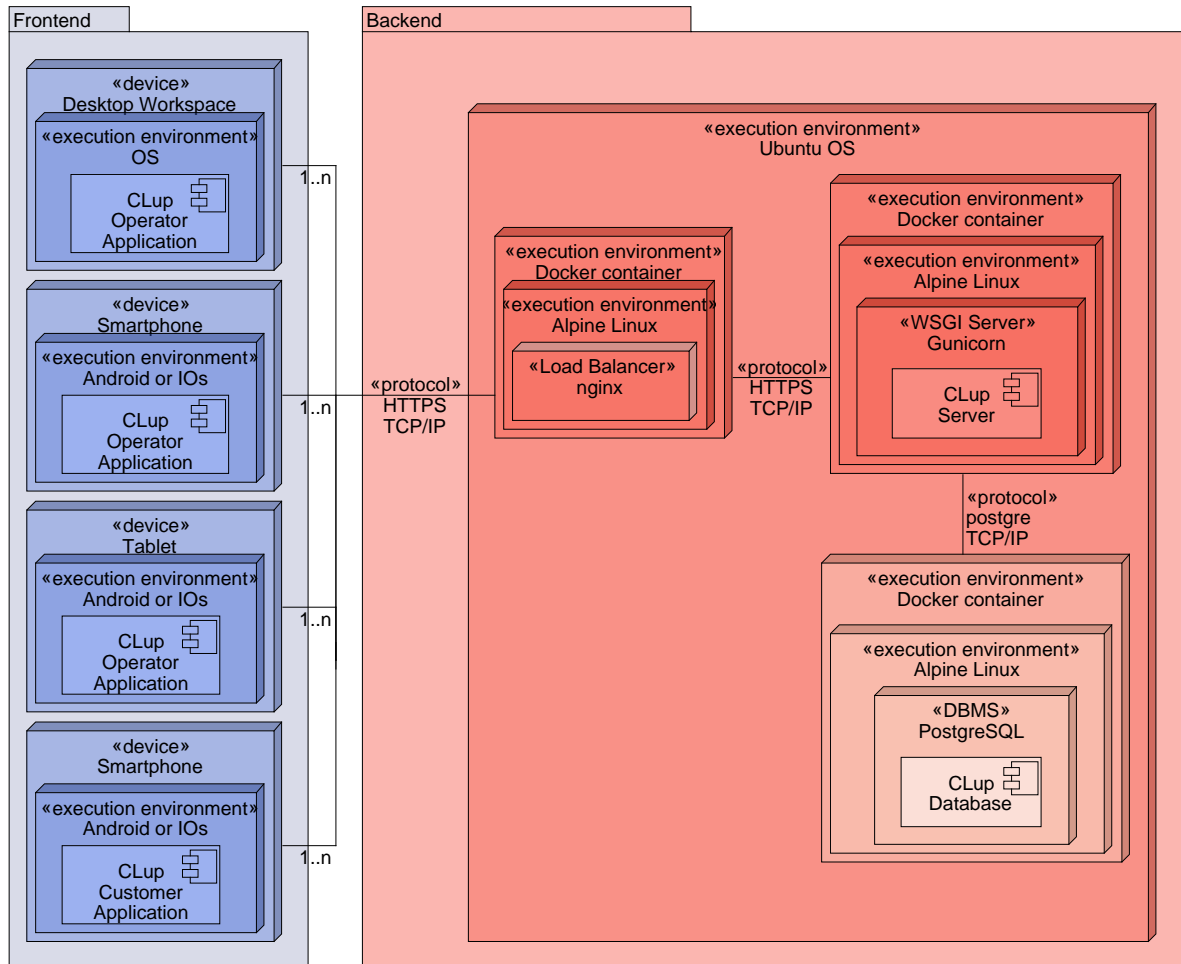| Label	Requirement description	Implemented?
R1	The system must keep general information and contacts about the store chains adopting CLup	No
R2	The system must provide each store a store-admin account	No
R3	The store-admin account must allow the creation of store-operator accounts	No
R4	For each store the system must allow the users to retrieve information about location and business hours	Partially
R5	The system stores information about capacity of each market	Yes
R6	The system won't let anyone enter the store if the maximum capacity has been reached	Yes
R7	The system will let a customer enter the store if and only if they have a a valid ticket	Yes
R8	The system will use the occupancy data retrieved from the store to control the store access	Yes
R9	The system must provide an interface to communicate with the store access control	Yes
R10	The system must provide an interface for user to compile a shopping list	No
R11	The system must take in consideration shopping list data and historic data from previous user visits to reduce store crowdedness per department	No
R12	The system must allow the store-admin account to create and edit entrance time intervals	No
R13	Each time interval must have a number of bookable slots fewer than the store capacity	No
R14	The system must allow authenticated users to book a visit in a desired time interval	No
R15	The system must not allow a user to book a slot in an already full time interval	No
R16	The system must not allow a user to book a visit if he has already reserved another visit	Yes
R17	The system must allow a customer to create a numbered virtual queue ticket and notify them if he can enter immediately (if the store is not full) or provide them a waiting time estimation	Partially
R18	The system must notify the customers with a virtual queue ticket when it's time to approach the store entrance	Partially
R19	The store operator application must allow an authenticated operator to manually admit customers	No

Label	Requirement description	Implemented?
R20	The system must ask the customer to provide the estimated visit time when booking a time slot	No
R21	The system must allow stores to hand out numbered physical queueing tickets to those that do not use the CLup application	Partially
R22	The system must allow the access to the store to customer with numbered tickets using a 'First Come First Served' logic, treating virtual and physical ticket owner equally	Yes
R23	The system must try to estimate waiting time based on store capacity, reservations and the current number of people with numbered tickets waiting in line	No
R24	The system should interface with an screen placed at the entrance of the store to notify customer which ticket numbers will enter in the next called batch	Partially
R25	The system must let the store-admin accounts retrieve statistics collected from CLup regarding their store	No
R26	The system must record periodically and store statistics about the occupancy of each store	Yes
R27	The customer CLup application must show brief statistics about average occupancy of each stores during different days of the week	No
R28	The operator CLup application must show to an authenticated operator the real time occupancy of the store	Yes
R29	The customer CLup application must be cross-platform and must work on the majority of the devices	Yes
R30	The stores adopting CLup must be displayed on a map	Yes
R31	The CLup customer application allows user to mark stores as favorite in order to access them quickly	No
R32	The CLup customer app after the login allows immediately to book tickets right from the homepage	Yes
R33	The system must provide an interface for automated control devices to communicate to CLup data about store entrances, store leavings and crowdedness in the various departments	Partially
R34	The system must push notifications to user devices with update information on the store he has a ticket for	Partially
R35	The system must associate tickets with line numbers	Yes
R36	The system allows customers to register an user account	Yes
R37	The system allows registered customers to authenticate	Yes

2.2   Partially Implemented Features

- **R4 For each store the system must allow the users to retrieve information about location and business hours**: The store location feature is implemented (in order to display stores in the map). The business hours are not provided to the user, this feature can be implemented using an external API with opening times or storing these opening times in the database

- **R17 The system must allow a customer to create a numbered virtual queue ticket and notify them if he can enter immediately (if the store is not full) or provide them awaiting time estimation**: The waiting time estimation is not provided. To estimate the waiting time an accurate model based on historical data can be used.

- **R18 The system must notify the customers with a virtual queue ticket when it's time to approach the store entrance**: The push notification component is not implemented in the prototype so the server can't push message to the user application. However the client can currently check if the ticket has been called by pulling data from the server API periodically; the application will show alert dialogs accordingly.

- **R21 The system must allow stores to hand out numbered physical queueing tickets to those that do not use the CLup application**: The CLup server provides an API for this feature but for satisfying this requirement a physical ticket printer is required.

- **R25 The system should interface with a screen placed at the entrance of the store to notify customer which ticket numbers will enter in the next called batch**: Similar to R21

- **R33 The system must provide an interface for automated control devices to communicate to CLup data about store entrances, store leavings and crowdedness in the various departments**: Similar to R21. The departments' crowdedness control feature is not implemented in the prototype.

3   Adopted Frameworks

The implementation of the CLup prototype is divided in two parts: Frontend and Backend.

3.1   Backend Frameworks and Languages

3.1.1   CLup Server and API

The framework chosen for the backend is Flask. Flask is a lightweight WSGI web application framework written in python. Flask is very good for prototyping web application and APIs because it is very easy to set up (it can be installed with a single command from the python package installer PIP) and comes with a debug mode that speeds up the debugging process. Along with flask, some other libraries were used to build the CLup Server Api.

- **SQLAlchemy** is a Object Relational Mapper library (ORM). An ORM allows to map the database entities and the relationship between them to objects and references in the application code. The ORM will translate operations on the objects to SQL queries that will be executed on the database. The ORM decouples the application from the database. In this way the application code is independent from the underlying DBMS, and the latter could be changed without modifying the application code.

- **Marshmallow** is a marshalling library for python, well integrated with flask. Marshalling consists in converting objects from the memory in a format ready for storage or transmission. Marshmallow makes the validation the request inputs a lot simpler so the programmer doesn't need to write a lot of error-prone boilerplate code for the input validation

- **Flask-RESTful** The CLupServer API uses the RESTful architectural style. Flask-RESTful simplifies the code writing of this type of APIs, providing the programmer a software interface to declare each endpoint as a class, containing the different HTTP methods accepted by the endpoints. For example, to implement an endpoint that allows to create ticket with a POST request, it is sufficient to declare a class 'CreateTicket' that extends the class Resource (provided by FlaskRestful), and implement the post() method.

- **Flask-Jwt-Extended** is a library for handling the authentication in flask using JWT tokens. A JWT token is an encoded string generated by the server and given to the user after checking its credentials. The JWT contains the user e-mail, an expiration timestamp and an hash for checking integrity. The JWT token is generated when the user makes a login request. Every request done by the user (that requires authentication) must contain this token, then flask-jwt-extended will check the validity of the token at each request. With flask-jwt-extended, allowing the access to an API endpoint to only the authenticated user is very straightforward, it's enough to put a python annotation before the endpoint methods for which authentication is needed.

3.1.2 Data layer

Regarding the data model a SQL relational database was preferred with respect to a noSQL one, because the data to persist has a well defined Entity-Relation structure (i.e. Tickets, Users, Stores. . .).

For the Data layer a postgreSQL DBMS was adopted. PostgreSQL is a production ready open-source relational SQL database. It's stable and used in a lot of commercial applications so it's a good fit for the CLup prototype.

3.1.3 Development Tools

Different development tools are used when writing and debugging the backend code.

- **Poetry** is a tool that helps managing the dependencies of a project, setting them up in an isolated python environment. Isolating the execution environment is essential to enhance the portability and the maintainability of the software. When poetry is set up it will create automatically a virtual python environment for the project and resolve the dependencies.

- **Black** is an automatic python code formatter. Black formats the code according to the PEP-8 standard, enhancing the readability.

- **pytest** is the official python testing framework. (More about the testing on section 5)

3.1.4 Deployment Tools

Docker is an useful tool to automatically build and ship applications. A Docker application is made of different containers each one running a different application, these application can communicate using an internal network.

For this prototype, docker was employed to build and deploy the backend using a single command. Without docker, each component (i.e. CLup Server Application, Database, Load Balancer) should have been set up manually.

To deploy the application in production it is not recommended to use the flask development WSGI server. So the CLup Server Application container runs a **gunicorn** server.

Nginx is a widely used load balancer. Due to the low usage of a prototype application, a load balancer is not strictly needed, but it is included for enhancing the stability of the gunicorn server and can be useful to have better performance when stress testing the system.

3.2 Frontend

3.2.1 Flutter

For the application frontend we decided to use the Flutter framework considering its many advantages:

- Flutter allows to natively compile applications for every major platform, including Android, IOs, Web and Desktop applications. By using a single code base it is possible to deliver the same experience to every platform, and without the need to define different teams for every version.

- Natively compiling applications ensures good performances even if not writing native code for the specific target platforms.

- Flutter provides lots of predefined Widgets[1] for every kind of situation. This allows for faster development since every common behavior in a classic application is already usable out of the box

- There are lots of libraries (both official and third party libraries) that further populate the list of usable Widgets (i.e. libraries to draw graphs with statistics, map integration, HTTP requests, QR code scanning/generation)

- Flutter allows for 'Hot Reloading', which instantly rebuilds the application while debugging, and without necessarily restarting the application, speeding up the development process runs on runs on

- Flutter is written with the Dart programming language, which is very flexible, has very powerful features (like Mixins and Extensions Methods, but also a very polished null safety implementation) and resembles Java in the general syntax (which is a known language to team members)

- Team members already had some experience with the framework

On the other hand, Flutter has some downsides:

- Flutter Web is still in beta developement, so it has not been tested enough to be considered 'stable', but the developement of the framework is very active, and behind the project there is Google.

- Some low level features, for example the interaction with background tasks in the various operating systems running the application, are not yet supported natively (in Dart code) but may require small amounts of codes to be written in the respective native languages (i.e. Java/Kotlin for Android, Swift for IOs)

---

[1]Every component in a Flutter application is a Widget, and every displayed page is a Tree of Widgets, one inside the other; working with Widgets allows to only rebuild the parts of the application that need to be updated, which improves performance.

3.2.2 External Libraries

To speed up the developement process some external libraries were used, here are listed the most important ones:

- **font_awesome_flutter**: Font Awesome is a font and icon toolkit that is widely used for the vaste collection of icons it provided, for almost every use case.

- **dio**: Dio is a powerful Http client for Dart, which supports Interceptors, Global configuration, FormData, Request Cancellation, File downloading, Timeout etc. It is very useful for making and handling requests to the CLup API.

- **google_maps_flutter**: A Flutter plugin that provides a Google Maps widget. It is an official library from the Flutter team, and (along with **google_maps_flutter_web**) allows the CLup application to interact with the map, put marks where the stores are located, etc.

- **qr_flutter**: QR.Flutter is a Flutter library for simple and fast QR code rendering via a Widget or custom painter. It was used to render the ticket information in a QR Code for an easy scan of the tickets.

- **barcode_scan**: A flutter plugin for scanning 2D barcodes and QR codes. It is a discontinued library but was the fastest to setup and very easy to use, should be substituted with another library for the final product. Integrates with Android and IOs but not with Flutter web, so on the webapp it is only possible to scan tickets by typing the ticketID.

- **geolocator**: A Flutter geolocation plugin which provides easy access to platform specific location services. On the webapp (which can be used on desktop devices without GPS sensors) the location is obtainaed through an API request that geolocalizes with IP addresses.

4 Source Code Structure

4.1 Backend Source Structure

All the backend code is stored in the ITD/CLupServer folder.

```
CLupServer\

|−− clup−server\
|−− data\
|−− db_config\
|−− docker−compose.yml
|−− docker−compose−testing.yml
|−− build.sh
|−− test.sh
```

- **data**: Contains nginx configuration files.

- **db-config** Contains a SQL script that runs when the PostgreSQL is created. This script creates the databases (clup, clup-testing)

- **docker-compose.yml**: docker-compose configuration file for setting up the production server. Docker-compose will read this file and build/set-up the containers accordingly.

- **docker-compose-testing.yml**: docker-compose configuration file for running the tests. Docker-compose will read this file and build/set-up the containers accordingly, containerizing all components but not the flask server.

- **build.sh**: A shell script that builds (if not already done) and starts the docker containers of the production server.

- **test.sh**: A shell script that starts the docker containers and runs pytest.

```
CLupServer\clup−server\

|−− clup_server\
|−− tests\
|−− CLup.py
|−− wsgi.py
|−− config.py
|−− data.json
|−− docker−entrypoint.sh
|−− Dockerfile
|−− poetry.lock
|−− pyproject.toml
|−− README.rst
|−− requirements.txt
```

- **clup-server**/: Contains the application python package

- **tests**/: Contains the integration tests

- **CLup.py**: The startup script for the flask application. This script is used to run
  the project when flask is not containerized. Some optional flag could be passed to
  this script (preceded by two dashes):

  - dev: For setting up Flask in development mode

  - drop: For dropping the database content on startup

  - populate: For populating the CLupUser and Store tables from the sample data
    stored in data.json

- **wsgi.py**: The startup script for the application when it is run from a production
  WSGI server (i.e. Gunicorn). This is startup script is used when the app is con-
  tainerized.

- **config.py**: Stores the configuration variables for flask for Development mode and
  for Production mode. The JWT secret key for the Production server is taken from
  the environment variables to avoid publishing the secret in the repository. The
  secret should be set to a random string (using export) before starting the server
  (This is automatically done from the build.sh script).

- **data.json**: Contains fake Stores and Users data to populate the database.

- **docker-entrypoint.sh**: A shellscript executed from Docker when it starts the ap-
  plication container. This script starts gunicorn.

- **Dockerfile**: Contains the instructions to allow Docker to build the application
  container.

- **pyproject.toml**: Configuration file for poetry. Contains the list of all the library
  dependencies needed to run the project.

- **requirements.txt**: Configuration file for pip. Used from the application container to retrieve all the project dependencies.

  ```
  CLupServer\clup−server\

  |−− __init__.py
  |−− models.py
  |−− schemas.py
  |−− routes.py
  |−− orm.py
  |−− auth_manager.py
  |−− information_provider.py
  |−− queue_manager.py
  |−− ticket_manager.py
  ```

- **__init__.py** is the package initialization file, executed by Python when the clup-server package is imported from another source file. Here is implemented the application factory method createApp(). This method allows multiple instances with different configurations to be created. The createApp methods will import all the other modules in the package and then starts to inizialize all the needed resources. For example it will start the database connection, configure the API routes...

- **models.py** contains all the models class declarations. Each model class is mapped to a database table, and SQLAlchemy provides the translation to a SQL statement for every operation made on a instance of one of the model classes. Each class contains also methods to decouple the data layer management code from the business code executed at each API call

- **routes.py** registers on start up all the API routes exposed to be accessed by the application

- **auth-manager.py** contains the APIs and the business logic for authenticating customers and operators.

- **information-provider.py** contains the APIs that provide information about the stores.

- **queue-manager.py** contains the APIs and the business logic used by the store operators(or automated control systems) to get the status of the queue and manage it

- **ticket-manager.py** contains the APIs to create, view and cancel tickets.

4.2 Frontend Source Structure

All the frontend code is stored in the ITD/clup_application folder.

```
clup_application\

|—— android\
|—— assets\
|—— fonts\
|—— ios\
|—— lib\
|—— test\
|—— web\
|—— .gitignore
|—— .metadata
|—— pubspec.lock
|—— pubspec.yaml
|—— README.md
```

- **android**/: folder containing mostly configuration files to correctly build the android application. Most of these are autogenerated files, with little tweaks, for example to setup Android permissions.

- **assets**/: here are stored all the needed assets files (in this case only the CLup logo).

- **fonts**/: contains the collection of fonts used in the application.

- **ios**/: configurations to build the IOs application. has not been tweaked since the team had hardware limitations and could not test IOs builds.

- **lib**/: contains the whole Flutter code of the application.

- **web**/: contains configuration files for the webapp build.

- **.gitignore**: autogenerated by Flutter, avoids pushing build and other local configuration files to the git repo

- **.metadata**: file used by Flutter to track the properties of the Flutter project. Autogenerated and should not be manually edited.

- **pubspec.lock**: autogenerated, used by Flutter to store information about dependencies

- **pubspec.yaml**: main configuration file for the dependencies of the Flutter project.

- **README.md**: text to be displayed in the git repo to provide info about the folder.

```
lib\

|── api\
|────── authentication.dart
|────── information_provider.dart
|────── operator_utils.dart
|────── ticket_handler.dart
|── conditional_deps\
|────── key_finder_stub.dart
|────── keyfinder_interface.dart
|────── mobile_keyfinder.dart
|────── web_keyfinder.dart
|── gps\
|────── gps_component.dart
|── pages\
|────── login_page.dart
|────── map_page.dart
|────── operator_page.dart
|────── signup_confirm_page.dart
|────── signup_page.dart
|────── store_view_page.dart
|────── ticket_page.dart
|── configs.dart
|── generated_plugin_registrant.dart
|── main.dart
```

- **api/**: this package contains all the calls to the CLup API tranformed in simple utils as in a Facade pattern, for easy access throughout the whole application.

- **conditional_deps/**: this packages contains different implementations of local storage management, to avoid a problem when working with crossplatform applications in Flutter; it is not possible to ship an application with web libraries as dependencies, so the application instantiates different behaviours to correctly match the underling environment.

- **gps/**: in this packages is present the code that is used to retrieve the user location. The method determinePosition() controls them location permissions to use the gps sensor, and in case of no sensor or no permissions, makes a call to an external API to geolocalize using the ip address.

- **pages/**: this package contains all the pages of the application, and the majority of the code takes care of the layouts, the visual componenents, and the buttons. There is very little application logic, since all the work is done on the backend.

- **configs.dart**: contains global variables like URLS and custom colors, to be easily accessed throughout the whole application.

- **generated_plugin_registrant.dart**: autogenerated by google_maps_flutter_web, not to be edited manually.

- **main.dart**: code to create and launch the application, here is defined the theme of the CLup app, the hierarchy structure of the pages, and utils function to write/read from local storage (implementing the correct keyfinder from the conditional dependencies).

5 Testing

5.1 Backend

The testing in the backend is performed using pytest. Pytest is the official testing framework for python. The testing metodology adopted for the backend is the AAAC pattern. AAAC stands for Arrange Act Assert Cleanup.

- **Arrange**: In this phase the resource needed for the test (fixtures) are created. For example the connection with the database is created, the test data are loaded, the mocks are instantiated.

- **Act**: The function to test is executed and its results are collected

- **Assert**: The results collected from the previous phase are compared with the expected results. If there is something different than what was expected an Assertion Error is raised, and the test fails.

- **Cleanup**: After the test has been executed the resourced are dropped, side effects from the tests are restored (i.e. rollback database transactions).

5.1.1 Fixtures

A **fixture** is a resource that is needed for setting up the test (Arrange phase).

Example of fixtures are a file with test inputs, a connection with an API or a database...

Pytest allows to give a different scope to each fixture. A fixture could last for the entire test session, or could be used only for a single test or for a group of tests.

Here is an example of a pytest fixture

```
#Scope could be function, class, module...
@pytest.fixture(scope=session)
def a_fixture(fixture_dependency_1, fixture_dependency_2,...):
    #Arrange fixture
    ...
    yield fixture
    #Cleanup fixture
```

As shown a fixture could depend from other fixture. For example in the CLup prototype the database connection depends on the start of the flask application, and the test data to write in the database depends from the database connection.

5.1.2 Flask testing

Flask is really easy to test. A flask application in development mode offers a test client that could simulate the request sending to the client. In this prototype this client is used as fixture to test all the different API endpoints.
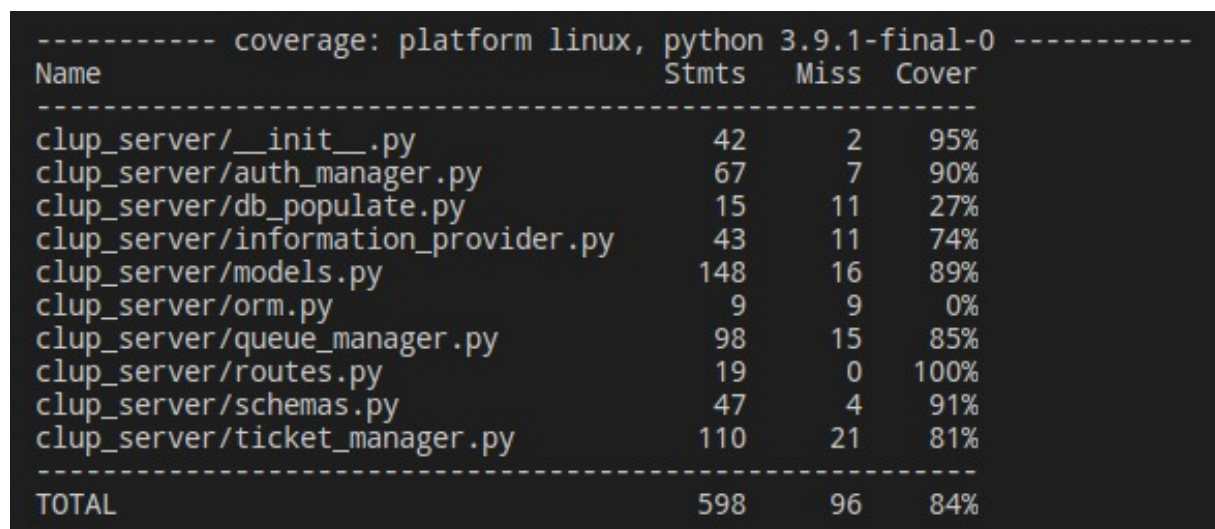
5.1.3 Unit tests

Unit tests aren't really useful for a simple application like this. Writing a unit test requires to make a stub for the database component that is already tested by the postgreSQL team. This requires too much effort compared to the benefits, so it turned out not to be so useful. Instead, this implementation performs integration tests with the CLup Server component and the CLup database.

5.1.4 Integration Tests

The integration test is performed using the test client provided by flask and aims to test the integration between the CLup Server and the database. There are different small tests for the edge cases for most of the APIs and a bigger test that simulates the complex evolution of the queue in a supermarket, performing all the possible operations in the queue. The tests were performed locally on a test database different from the production database using test data stored in 'data.json'.

5.1.5 Statement Coverage

The statement coverage is calculated using pytest-cov, a plugin for pytest. The coverage of the statements is about 85%, which is an acceptable value considering that there are a some statements that are not accessible because they are executed only during the first setup of the system.

```
---------- coverage: platform linux, python 3.9.1-final-0 ----------
Name                                    Stmts   Miss  Cover
--------------------------------------------------------------
clup_server/__init__.py                    42      2    95%
clup_server/auth_manager.py                67      7    90%
clup_server/db_populate.py                 15     11    27%
clup_server/information_provider.py        43     11    74%
clup_server/models.py                     148     16    89%
clup_server/orm.py                          9      9     0%
clup_server/queue_manager.py               98     15    85%
clup_server/routes.py                      19      0   100%
clup_server/schemas.py                     47      4    91%
clup_server/ticket_manager.py             110     21    81%
--------------------------------------------------------------
TOTAL                                     598     96    84%
```

Figure 3: Statement Coverage

5.2 Frontend

The flutter application was tested manually, integrating the app with the rest of the system. Automated unit tests were not performed due to the difficulty of testing an application with a Graphical User Interface, that in a prototype varies a lot during the course

27

of the developement. Moreover, Flutter provides a lot of pre-tested small components as Widgets, and the application is built on top of those.

5.3 Sample Data used for testing

The database is populated with some testing Data. Here are provided all the account credentials to allow the tester to use all the features of the application.

Customer Accounts

- e-mail: customer<N>@CLup.com

- password: customer<N>@CLup.com

N is a number from 1 to 100 (example: customer1@CLup.com, for both email and password)

It is always possible to create a new customer account using the Sign Up functions in the application.

Operator Accounts

Each store has one store operator account linked to that store with this credentials.

- e-mail: operator<ID>@CLup.com

- password: operator<ID>@CLup.com

To find the store ID assigned to each store use the stores.txt file available in the ITD folder.

For example to access to the operator account for the store located in Carnischio use the email operator1@CLup.com. For the store located in Ivrea use the email operator2@CLup.com. . .

6 Installation

6.1 Backend

The backend server has been deployed with Docker containers on AWS.

6.1.1 Starting the server

An instance of the server has already been deployed online using AWS, and the API is available at https://clup.waifocus.com so it's not needed to run it locally. If you want it is still possible to run the server locally on Linux or on Windows Subsystem of Linux (WSL)

Installation steps

1. Install Docker

2. Install Docker Compose

3. Check if Docker service is running. If not start it with the command

   ```
   sudo systemctl start docker
   ```

4. Open build.sh and change the JWT_SECRET_KEY string to another value. When running the application in a production environment keep this value secret.

5. Run the script with

   ```
   ./build.sh
   ```

   this script will create the containers by invoking docker-compose. If you didn't add your user in the docker group you have to run the script with sudo

6. You can do requests the address localhost:8000

6.1.2   Running tests

For running pytest a startup script that doesn't containerize the application is available.
Installation steps

1. Install Docker

2. Install Docker Compose

3. Check if Docker service is running. If not start it with the command

```
sudo systemctl start docker
```

4. Install Poetry

5. Open a linux shell and go in CLupServer/clup-server

6. Run the command

```
poetry install
```

7. Run the test script with

```
./test.sh
```

The script sets up all the containers and the flask application in testing mode and the it runs the Pytest. If you didn't add your user in the docker group you have to run the script with sudo

8. (Optional) If you want to start the application out of the container run the command

```
poetry run python3 CLup.py − −dev − −populate
```

6.2   Frontend

6.2.1   Flutter Mobile Application

The application has been developed and tested only on Android, due to limitation in the available hardware.

To install the application we provide two .apk packages:

- RELEASE Version: A smaller package containing the prototype as it would be released in a production environment, downloadable here

- DEBUG Version: A bigger package containing the debug version of the prototype, which is useful for checking potential error messages, full content of API calls, etc. Downloadable here

6.2.2 Flutter Web Application

A release version of the web application has been deployed using Github Pages, and it is available at https://clup-mp.github.io, without the need to install anything.

We also provide a package containing all files to deploy the web application, downloadable here

To test locally the web application it is possible to setup a simple web server, for example by using python:

```
python3 -m http.server
```

Then opening a browser on http://localhost:8000

7 Effort Spent

7.1 Dario Passarello

- Meetings on how interface backend with frontend: 4 hrs

- Backend project setup: 4 hrs

- Learning new skills in backend development: 8 hrs

- Coding backend: 22 hrs

- Writing backend tests: 7 hrs

- Deploying backend: 6 hrs

- Final system test: 3 hrs

- Writing ITD documentation: 8 hrs

Total: 62 hours

7.2 Davide Luca Merli

- Meetings on how interface backend with frontend: 4 hrs

- Frontend project setup: 1h

- Documenting on Flutter features, best practices, external libraries: 8h

- Frontend development: 40h

- Deploying WebApp: 2 hrs

- Final system test: 4 hrs

- Writing ITD documentation: 6 hrs

Total: 65 hours