ATD –
Acceptance Test Deliverable

***Project**: CLup*
***Authors**: Vincenzo Riccio, Giancarlo Sorrentino, Emanuele Triuzzi*

POLITECNICO
MILANO 1863

Immordino Alessandro – 969549
Pala Riccardo – 969598
Polvanesi Giacomo – 971083

Politecnico di Milano

A.A. 2020/21

Table of Contents

1 Project to be analysed

Title: CLup

Authors:

- Vincenzo Riccio
- Giancarlo Sorrentino
- Emanuele Triuzzi

Download page: https://github.com/SirGian99/RiccioSorrentinoTriuzzi

Referenced documents:

- *V. Riccio, G. Sorrentino, E. Triuzzi,* "CLup – Requirements Analysis and Specification Document".
- *V. Riccio, G. Sorrentino, E. Triuzzi,* "CLup – Design Document"
- *V. Riccio, G. Sorrentino, E. Triuzzi,* "CLup – Implementation and Testing Document"
- *M. G. Rossi, E. Di Nitto,* "I&T assignment goal, schedule, and rules"

2 Installation instruction

This section includes all steps followed during the installation process, in order to be able to carry out all tests needed for the acceptance.

For what concerns the environment setup:

- Download and installation of IntelliJ IDEA Ultimate Edition 2020.2.3 for Windows;
- Configuration of JDK 15 as SDK in IntelliJ IDEA;
- Import in the workspace the CLup project downloaded from the repository.

For what concerns the application server:

- Download Apache TomEE plume 8.0.4 from repository (since we adopted 8.0.3 version so far);
- Configuration of the downloaded folder as new TomEE Home directory, for the already specified Application Server "Apache TomEE 9.0.37" in Run > Edit Configurations… > Application Server.

For what concerns the Database:

- Creation of a new schema in MySQL Workbench called “clup_db”;
- Import of the dump file “dump_db.sql” downloaded from the repository to populate the created schema.

For what concerns Application-Database connectivity:

- Edit of “persistence.xml” file by substituting database connection credentials with personal ones.

For what concerns the client side, the targeted device for the application is an iOS smartphone. Since we did not have the possibility of installing **XCode** on our Windows devices, for which a macOS one is needed, and since such a program is required in order to build the application and to test it on a real or simulated device, we needed to search for fallback solutions.

Therefore, we tried to make use of some emulators: at first, we installed **Xamarine** plugin for Visual Studio, which has revealed to be unsuitable due to the persistent lack of a macOS device which remotely connect to, then we downloaded and installed a **macOS Catalina 10.15 VMDK** to run in Oracle VirtualBox emulator. Also in this case we obtained poor results due to the high performance requirements to properly run the emulator and the need to re-install and configure on it all necessary frameworks.

Finally, we decided to proceed with the last attempt, consisting of making use of **Appetize.io**, an online tool that allows to run iOS mobile apps on browser. Even in this case some difficulties have arisen. We first successfully managed to install the application on the emulated device, but there were problems concerning the connectivity with local server. In fact, application suddenly stopped when we tried to open it.

We overcome this final problem by installing **Ngrok**, a tool that allowed us to inspect and manage client-to-server request. After downloading it, we opened the executable and typed “*http 8080*” on the command line, allowing Ngrok to create a stable URL with which access the server through the emulated device.

N.B. We obtained the executable of the application from the developers team, that still produced it with XCode by changing the attribute *baseURL* of the file *Routes.swift* available under the project tree with “http://<SERVER_IP>:8080/CLup”, where <SERVER_IP> is the URL address generated by Ngrok.

3 Acceptance test

3.1 Functional test cases from user side

Once running CLup mobile app on Appetize.io we performed some functional tests simulating the typical behaviour of a simple user during the use of the several services.

We carried out a research of a supermarket by using the search bar to select the city, choosing a supermarket chain and then the location of one of the proposed stores. From this page we proceeded with some combinations of testing by changing several times the input values:

- Book a visit providing a date, a time, an estimated visit duration, a number of people associated with the reservation, some section of the supermarket to visit;
- Line up with the same provided information (except date and time) but with different values;
- Open the section of the active reservations in the designated section from the home page, showing details of the tickets and the associated QR codes;
- Cancel a reservation from 'Your Reservations' section.

From this analysis we concluded that everything works properly.

3.2 Functional test cases from developer side

We have checked if the corresponding created reservation is properly stored in the database and if it also cancelled after a deletion request by the user. Finally, we have also checked if exceptions are handled properly; for instance, inserting a duration visit that will exceed the closing time of the selected supermarket, setting a data from previous days, or setting a number of people that will exceed the maximum capacity of a supermarket section. All this test cases, regarding the user side, are succeeded successfully.

For what concerns the provided functionalities, we have analysed the main functions which were implemented in the prototype and we have compared them with the requirements stated in the RASD document, verifying that they were met in the practice.

3.3 Test cases from developer side

These tests were carried out by manually running the pool of available dedicated test classes directly on IntelliJ IDEA, making use of JUnit Framework.

Both unit and integration tests were performed on all system components. The former ones are executed avoiding the use of the database, whereas the latter ones involving the database communication as support of data access and storage.

Test classes contains test cases concerning the main functionalities of the different services, in particular *CustomerController*, *RequestHandler*, *StoreStatusHandler* and *VisitManager*.

3.3.1 Unit tests

3.3.1.1 CustomerController

`getCostumerActiveBookings(): OK`

`getCostumerActiveLineups(): OK`

This test cases check, respectively, whether the list of active bookings/lineups of a customer effectively contains only active and not completed yet bookings/lineups.

3.3.1.2 RequestHandler

`lineup(): OK`

`book(): OK`

This test cases check, respectively, whether several line-up/booking requests are correctly forwarded and accepted or refused, basing on their wellness and the status of the customer active reservations. In case of a bad request, all warning messages are correctly printed.

3.3.1.3 StoreStatusHandler

`getChains(): OK`

`getAutonomousStores(): OK`

`getChainStores(): OK`

This test cases check whether, once selected a city or further a certain chain, the lists of the stores provided to user are the correct ones, case-by-case.

`getStoreGeneralInfo(): OK`

This test case checks whether the information about a specific store is shown coherently with respect to the selected one.

3.3.1.4 VisitManager

`newLineupRequest(): OK`

`newBookingRequest(): OK`

This test cases check, respectively, whether each lineup/booking request performed returns the expected state. In particular, in both cases a single request is made, for which a ready state is expected, then another request is made, for a number of people that leads to an excess on the store maximum capacity, for which the expected result is the pending state of the reservation.

3.3.2 Integration tests

3.3.2.1 VisitManagerIntegrationTest

To check if the integration process is completed successfully, there is a method invoked before every test that initializes a new chain of supermarkets, a new store, a new request of line-up and a new request of booking by a new user and then make them persistent in the database.

`validateAccess(): OK`

It checks if VisitManager can handle properly the validation of the accesses to the supermarket by testing the various states in which a line-up request or a booking request could be by changing its status from “Completed”, “Fulfilled” and “Ready”.

`confirmAccess(): OK`

It checks if VisitManager can handle properly the confirmation of the accesses to the supermarket by testing the various states in which a line-up request or a booking request could be by changing its status from “Completed”, “Fulfilled” and “Ready”. It also checks if the status of the supermarket is updated successfully.

`validateExit() : OK`

It checks if VisitManager can handle properly the validation of the exits from the supermarket by testing the various states in which a line-up request or a booking request could be by changing its status from “Completed”, “Fulfilled” and “Ready”.

`confirmExit() : OK`

It checks if VisitManager can handle properly the confirmation of the exits from the supermarket by testing the various states in which a line-up request or a booking request could be by changing its status from “Completed”, “Fulfilled” and “Ready”. It also checks if the status of the supermarket is updated successfully.

`newRequest() : *FAILED*`

This test case performs analogous checks with respect to those defined in the previous section (see “Unit test”) for `newLineupRequest` and `newBookingRequest` test cases. However, this is an Integration test, thus it involves the database to check that data are stored and accessed properly.

Test case does not fail because of a functioning error, but this is caused by the computational speed that, basing on the testing machine, affects the method call outcomes, since Timestamp objects are involved.

In particular, one expects that estimated time of entrance (assertion in row 237) is less than the current time, which actually is, but the time slot between the two computations is so much short that such a difference is not detected by the machine, which considers them as equal.

As a proof of the test case efficiency, we try to include the equality statement in the assertion changing from “before” to “!after”, in this way the test case terminates with a positive outcome.

`checkNewReadyRequest() : OK`

It checks if VisitManager can handle properly the new line-up/booking requests assigned them the right status, even respecting the booking requests priority over line-up ones.

3.3.2.2 StoreStatusHandlerIntegrationTest

To prepare the prototype, there is a method that creates three new supermarkets, including all the information needed, before the execution of the test functions. Then some checks are made to verify if the new instances are correctly persisted in the database.

`getStoreGeneralInfo(): OK`

This test checks if the instance returned by the method *getStoreGeneralInfo(storeId:Integer)* corresponds exactly with the instance related to the first supermarket created.

`getChains() : OK`

This method verifies if the chain effectively is present among all the chains that are created in the database.

`getChainStores(): OK`

This test checks if the stores are correctly present in the database by calling the collection of all stores.

`getAutonomousStores(): OK`

This method tests if the second store, that was not associated to any chain in the setUp method, is effectively present once the method *getAutonomousStores(city:City)* is called.

3.3.2.3 CustomerControllerIntegrationTest

To prepare the environments to run the test cases, a new user istance is created and several bookings and line-ups with different status are associated to it in the @Before method.

`registerApp(): OK`

This method verifies if the created user is correctly inserted in the database table.

`getCustomerActiveBookings():OK`

This method tests if the active bookings created are effectively assigned to the corresponding user once the functions *getCustomerActiveBookings()* is called.

`getCustomerActiveLineups():OK`

This method tests if the active bookings created are effectively assigned to the corresponding user once the functions *getCustomerActiveLineups()* is called.

3.3.2.4 RequestHandlerIntegrationTest

To prepare the environment for these test cases, a new store, a new user and new reservations associated to him are made persistent.

`lineup(): OK`

This method verifies if RequestHandler component is able to manage the exceptions such as:

- A reservation request associated to one customer not registered into the database;
- A reservation request associated to one store not registered into the database;
- The number of people referred to a reservation is equal to zero;
- The number of people referred to a reservation is greater than the maximum capacity;
- A reservation request associated to one customer already lined up;
- A reservation request that exceeds the closing time of the selected store.

Moreover, it checks if a request is accepted when it comes from a customer that has just been into the store.

`book(): OK`

This method verifies if RequestHandler component is able to manage some exceptions such as:

- A user that performs two booking requests that overlap each other with regard of their schedules;
- A reservation request that exceeds the closing time of the selected store;
- The number of people referred to a reservation is greater than the maximum capacity;
- The number of people referred to a reservation is equal to zero;
- A reservation request associated to one customer not registered into the database;
- A reservation request associated to one store not registered into the database.

Moreover, it checks if a new app customer can fulfil a new request.

`cancelRequest(): OK`

It checks if RequestHandler correctly removes a request.

4 Conclusions

4.1 Architecture

The implemented prototype complies with the functional and non-functional requirements stated in the RASD, in particular those specified in the “Requirements achieved” section of the ITD.

For what concerns the adopted architectural styles, the 3-tier structure has been effectively used as a guideline to carry out the implementation. Each layer is independent and well decoupled from the others. BusinessModule and DataModel components belong to well-separated packages, each of them with a specific function:

BusinessModule effectively takes care of guaranteeing the operativity of the application, as well as managing interactions between entities.

Furthermore, DataModel provides methods to specify, represent and manipulate data.

4.2 Code structure

The code is very well structured and also supported by several comments that help for a better understanding of the methods and their behaviours.

The planning of the software took into account several potential unpleasant situations, proposing for each of them an efficient feasible solution.

4.3 Suggestions

However, in our personal opinion, some developed functionalities seem to be potentially improvable, in order to guarantee a better service to the customers. In particular:

- One could allow the user to compute the *estimated time of entrance* every time he wants to. In fact, this can be considered as the most important feature to avoid crowding outside the facilities, and the unpredictability of the effective duration of the visit of the customers makes essential the presence of the possibility to periodically update such a value.

- The computation of the *estimated time of entrance* is performed basing on statistical values; this could cause serious inaccuracies on the estimation because of the large interval of values that a visit duration can assume.
- The maximum value of "*numberOfPeople*" parameter for a reservation could be decreased, since the chosen value of 20 could lead to an excessive gathering during the lining up process, conflicting with the imposed health restrictions, as well as precluding potential accesses to other customers.
- A lower bound to the "desired starting time" may be established for the booking service. Otherwise, it would be disadvantageous for a customer to make a line-up request, since, if all other customers perform booking requests up to reach the maximum capacity, they will always have priority over him and his request might get stuck in pending status. If the application gives the customer the possibility of booking a visit selecting a "desired starting time" from a certain time slot from the current time, the above-described unpleasant situation might be partially avoided.