ITD – Implementation and Test Deliverable

POLITECNICO
MILANO 1863

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

Politecnico di Milano

A.A. 2020/21

Table of Contents

Implementation Document

1 Introduction and scope

This document is aimed to describe the Implementation and Test deliverable (ITD) of CLup software application.

Our implementation of CLup service consists of a web application entirely realized by using JEE platform. The code has been developed following the guidelines previously specified in RASD and DD. However, not every functionality of the software project has been implemented, but only a prototype able to demonstrate the general efficiency of the designed system. All functions defined in this model intend to fulfil the requirements and the goals mentioned in RASD; their implementation and integration have been based, as much as possible, on the architectures and the patterns described in DD, moreover, the integration of components and the testing process have followed the plan defined in that document.

The next sections take care of better described:

- The functions and the services implemented in the software;
- The adopted framework and the external APIs;
- The code structure of the project;
- The integration and the testing processes;
- The installation guide to run the prototype.

For each decision that has been made, advantages and disadvantages are provided, in addition to some justifications about why the choice has been fulfilled, or not.

2 Main Functions/Requirements

In this chapter all the main functions that are implemented in the prototype are presented, considering how they will satisfy the requirements listed in RASD, which are stated below.

[**R1**]: System must assign a unique code to every reservation.

[**R2**]: System should produce a QR code for each reservation based on its unique code.

[**R3**]: System must assign a *ticket number* to every *real-time reservation*.

[**R4**]: System must unlock the sliding door if and only if the reservation code scanning gives a positive outcome.

[**R5**]: System must lock sliding doors right after Client entrance.

[**R6**]: A reservation code scanning must give a positive outcome only if such reservation concerns the supermarket Client is attempting to enter.

[**R7**]: A code scanning of a *real-time reservation* must give a positive outcome only if the store has not reached its maximum capacity.

[**R8**]: A code scanning of a *real-time reservation* must give a positive outcome only if the corresponding reservation number is among those of the *virtual line-up* authorized to enter, since their entrance does not make the store exceed the maximum capacity.

[**R9**]: A code scanning of a *planned reservation* must give a positive outcome only if the actual time is included in the interval between the starting time of its *schedule* and the *maximum tolerated delay*.

[**R10**]: System must keep track of the number of Clients within the store.

[**R11**]: System should provide Client, having a *real-time reservation*, with *expected time of entrance*.

[**R12**]: System could notify User, who made a reservation, that his turn is coming.

[**R13**]: System should allow the *line* to flow anyway, even if one of Clients next in the *queue* did not arrive within the *maximum tolerated delay*.

[**R14**]: System must validate a *real-time reservation* request only if it occurs during the hours of operation of the chosen supermarket.

[**R15**]: System must validate a *planned reservation* request if and only if at least one slot is available in the chosen store for the selected *schedule*.

[**R16**]: System must forbid User from making a reservation if he has got another active one.

[**R17**]: System must propose making a *planned reservation* starting from some hours later the current time.

[**R18**]: System should detect an alternative *schedule* for doing grocery shopping in the chosen store if the selected one is not available.

[**R19**]: System could find the closest stores to the chosen one, in which the selected *schedule* is available.

All the functionalities designed in RASD and DD have been partially developed. The implementation includes the most relevant services, able to prove the devised project is effectively feasible. We focused on defining the essential features of a web application involving a reservation system, such as the methods of login, registration, profile update, sending of a booking request, deletion of a reservation. Furthermore, we expand the implementation by adding some more polished feature to give the software application a

functional logic, therefore methods for the queue management, such as proposing an available schedule, checking the availability of a reservation request, computing the ETE, removing an expired reservation. We have neglected some more specific functions we have certified to not be worth to implemented in the prototype, because of the amount of code compared to the lack of possibilities of demonstrating their efficacy by only using the web browser (i.e. Scanner, TicketGenerator and sliding doors functionalities). We have also overlooked other services we have previously declared to be additional functionalities; for instance, the option of selecting the desired supermarket by means of a map.

All the developed classes are listed in the following, in addition to the related functions and the explanation of which requirements each of these attempts to satisfy.

2.1 ReservationService.java

- *insertPlannedReservation(visitDuration : Integer, ownerId : Integer, SupermarketId : Integer) : Boolean*
  insertRealTimeReservation(visitDuration : Integer, EntranceTime : Time, ownerId : Integer, SupermarketId : Integer) : Boolean

  The aim of these functions is to create a reservation after a user request: they will insert in the related database table the reservation inclusive of the user ID to which it refers and of the supermarket ID. The database will assign to the new reservation an identifier that will be automatically incremented every time a new reservation will be inserted; in this way the uniqueness of the reservation is guaranteed, therefore requirement [R1] is satisfied.
  The second function also assigns to every *real-time reservation* a specific ticket number by retrieving from the selected supermarket the last ticket number which refers to; in this way each *real-time reservation* will have an assigned ticket number as stated in requirement [R3]. This function also computes for the first time the ETE related to the reservation that then will be proposed to the user, who will be able to confirm or refuse it.

- *findReservationByUserId(userId : Integer) : Reservation*

  This function returns the reservation belonging to a user through his ID. This method is used to check whether the user has already an active reservation and to prevent him from doing another one, as stated in requirement [R16].

- *getQRCodeImage(text :String, width : Integer, height : Integer) : byte[]*

  This method is used to create the QR code concerning to the active reservation and to display it as an image through an HTML page. This function is useful to make the reservation identification readable through the scanner device at the entrance of the supermarket, as stated in requirement [R2].

2.2 SupermarketService.java

- *checkAvailability(superId: Integer, start : Time, duration : Integer) : Boolean*

  This method is implemented for checking the maximum capacity in the time interval specified by the user once it has submitted a *planned reservation* request. This method is essential for guaranteeing that the capacity of the selected supermarket will stay under the established threshold, as stated in [R15].

- *getSchedule(supermarketId : Integer) : List<Time>*

  This method returns a list of available temporal slots by retrieving the corresponding supermarket through its identifier. It will return the temporal slots from two hours later the actual time to the closing time as stated in requirement [R17].

- *expiredReservation(type: Char, reservationId: Integer) : Boolean*

  This function reflects the [R9] statement: it checks if either the active reservation of the user has expired, or it has not valid anymore.

2.3 ETEComputationService.java

- *computeETE(reservation : RealTimeReservation) : LocalTime*

  This function calculates the ETE of the *real-time reservation* given as parameter and returns it to the user. This function is essential to provide a correct estimation time the user must wait until his turn is coming. This method fulfils requirement [R11].

3 Source code structure

The implementation structure follows the model of the web applications developed using the JEE platform.

The main subdivision of the code regards the distinction into two different Java projects: “CLupEJB” (EJB project) and “CLupWEB” (Dynamic Web project).

The first project contains Entity EJBs (“src/project.clup.entities” package), Session EJBs (“src/project.clup.services” package), some standard Java classes (“src/project.clup.utilities” package), exceptions (“src/project.clup.exceptions” package), test cases (“src/project.clup.tests” package).

The latter includes Servlets (“src/project.clup.controllers” package) and HTML pages (“WebContent” folder).

3.1 CLupEJB

3.1.1 Entity EJBs

Main function of these components is to embed data to make them persistent and to manipulate them. These elements are represented by Java classes with private visibility, they must provide method to obtain and modify the information they have included (getters and setters).

The entities defined in this project have been depicted in the Class Diagram of the previous documents (RASD and DD), they are: *User*, *Supermarket*, *Reservation* (abstract class), *RealTimeReservation* (extends Reservation), *PlannedReservation* (extends Reservation).

3.1.2 Session EJBs

The main purpose of these elements is to provide a simple interface for the Client who fulfils a request of accessing information comes. Different from the Entity EJBs, they have a limited lifecycle; when their use ends, they will be eliminated. For CLup web application, these Session “Stateless” EJBs have been used: *UserService*, *SupermarketService*, *ReservationService*, Router, *ETEComputationService*. These services implement methods to find objects in the database, to check if some conditions occur, to insert or delete data, to do some computation.

3.1.3 Utilities

Timetable is a simple Java class that has been implemented to create transient new type objects.

3.1.4 Exceptions

These classes take care of declaring constructors for the different type of exceptions which manage possible invalid behaviours that can occur during application execution.

In particular, the exceptions are: *BadReservation*, *BadRetrieval*, *CreateProfile*, *Credentials*, *Timetable*, *UpdateProfile*, *UpdateReservation*, *UpdateSupermarket*.

3.1.5 Tests

The following test cases have been implemented to execute unit and integration testing of components:

Unit:

- *PlannedReservationTest*
- *RealTimeReservationTest*
- *SupermarketTest*
- *TimetableTest*
- *ETEComputationServiceTest*
- *ETEComputationServiceTest2*
- *ReservationServiceTest*
- *SupermarketServiceTest*
- *UserServiceTest*

Integration:

- *SupermarketAndDatabaseIntegrationTest*
- *UserAndDatabaseIntegrationTest*
- *ReservationSupermarketUserAndDatabaseIntegrationTest*

They will be better described in the designated section of this document.

3.2 CLupWEB

3.2.1 Servlets

Servlet technology is used within the Web-container. A servlet (or controller) is charged of managing a Client request, making some elaborations, then sending a response.

Controllers contain two important methods to handle HTTP services: *doGet* and *doPost*. They take as input *HttpServletRequest* and *HttpServletResponse* objects that identify client request and server response. Other important methods have been used: *getParameter* (which gathers a parameter value from the request), *getServletContext* (that returns the context in which the Servlet operates), *getSession* (that returns an instance of the session object), *setAttribute* (which associates a value to a variable) and *getAttribute* (which retrieves a value from a variable previously assigned).

These are the implemented servlets: *CheckLogin*, *CheckRegistration*, *ComputeETE*, *CreatePlannedReservation*, *CreateRealTimeReservation*, *DeleteReservation*, *EditFavouriteSupermarket*, *EditInfo*, *EditPassword*, *GoToBookNow*, *GoToEditFavouriteSupermarket*, *GoToEditInfo*, *GoToEditPassword*, *GoToHomePage*, *GoToMyReservation*, *GoToPlanYourVisit*, *GoToShowProfile*, *ShowETE*, *ShowQRCode*.

3.2.2 HTML pages

Client interface is composed by several dynamic HTML pages able to interact with the user. During a user session these pages can change depending on access conditions or data input as parameter by Client (retrieved by the Servlets).

HTML pages created for this web application are: *Index* (first page), *Homepage*, *Registration*, *BookNow*, *BookNowETE*, *EditFavouriteSupermarket*, *EditInfo*, *EditPassword*, *MyReservation*, *PlanYourVisit*, *QRCode*, *ShowProfile*.

4 Adopted development frameworks

In this section all the development frameworks adopted during the implementation process are presented.

The main adopted framework is **JEE** (Java Enterprise Edition) that is a java-based software framework for development enterprise application. This choice is due to JEE features of providing multiples APIs that facilitate the development process of all the tiers which compose the CLup software architecture previously presented in DD. The programming

language with which the prototype is implemented is Java, this choice was affected by the employment of JEE as software framework. Among all the APIs included in JEE, the following ones were used during the development process are:

- **EJB API** (Enterprise JavaBean API): this API is used to handle and provide the Client with all the functions that are implemented in the Application Tier and, for this reason, it creates a communication between the Client and the Business Logic through containers that manage the EJB instances.
- **JPA** (Java Persistence API): this API is used to create an interface between the Data Layer and the Application Layer by means of the mapping of the database instances and the Java classes. It also provides some functionalities to handle relational data management, such as the relationships between tables defined in the database. It can manage query that are sent to the Data Layer in order to extract useful information.
- **JTA** (Java Transaction API): it is an API adopted to manage transaction, JTA defines standard Java interfaces between a transaction manager and the parties involved in a distributed transaction system.

The choice to not use the Google Maps API, that is presented in the DD document, is due to the fact that is not essential to show the basic and most important functionalities of the software that is actually the main goal of the prototype development.

Another API, not presented in the DD document, used for the implementation, is **Thymeleaf**. It is a Java XML/XHTML/HTML5 template engine that can work both in web (servlet-based) and non-web environments. It is better suited for serving XHTML/HTML5 at the view layer of web applications, but it can process any XML file even in offline environments.

In web applications, Thymeleaf aims to be a complete substitute of JavaServer Pages (JSP) and implements the concept of Natural Templates: template files that can be directly opened in browsers and displayed correctly as web pages.

Eventually, one more external API was used to generating reservations QRCode. To make it possible, Zebra Crossing API was adopted. **ZXing** is a popular API for QR codes processing in Java. Its library has multiple components, focusing on the most relevant ones it was possible to create simple QRCodes to fulfil our needs.

As a middleware between the Presentation Layer and Data Layer, **Apache TomCat** was used to provide a software platform for the execution of web applications and as servlet containers.

The obtained advantages are that one can allow a higher level of service for users, and a higher level of abstraction for programmers. It can also facilitate the maintenance, writing and integration of applications.

5 Performed test

For testing development, we adopted the **JUnit** framework both for unit and integration testing. It allows to verify that returned object are those expected and that proper exceptions are thrown. The defined test cases were manually triggered using exclusively Eclipse JUnit environment. Anyway, it would be possible to adopt some tools for automated testing.

5.1 Unit tests

In the following, tests of all units of the software are described. For what concerns Entity EJBs tests, we focused on analysing the most interesting methods, since *Getters* and *Setters* does not contain sufficiently relevant functions to justify their testing. Instead, for the Services we have implemented tests case for checking the functions which represent the core of the component.

Unit tests must be decoupled from the database, so we cannot use the *EntityManager* for the implementation of Services test. For this purpose, nested classes are defined to be able to override methods for getting information of different objects by means of their ID, avoiding *EntityManager.find* method invocation.

5.1.1 PlannedReservationTest

Attributes:

- `r : PlannedReservation`

Setup: this method is called before each test is executed. A new empty PlannedReservation object is created.

Tear Down: this method is called after each test is executed. Reservation *r* is set to null.

Tests:

`testGetExitTime():` *start time* and *visit duration* are set with a default value. The expected outcome is an *exit time* with a value corresponding to the sum of the two.

5.1.2 RealTimeReservationTest

Attributes:

- `r : RealTimeReservation`

Setup: this method is called before each test is executed. A new empty RealTimeReservation object is created.

Tear Down: this method is called after each test is executed. Reservation *r* is set to null.

Tests:

`testGetExitTimeWithEntranceTime():` *entrance time* and *visit duration* are set with a default value. The expected outcome is an *exit time* with a value corresponding to the sum of the two.

`testGetExitTimeWithETE():` *ETE* and *visit duration* are set with a default value. The expected outcome is an *exit time* with a value corresponding to the sum of the two.

5.1.3 SupermarketTest

Attributes:

- `s : Supermarket`

Setup: this method is called before each test is executed. A new empty Supermarket object is created. Then, four Reservation (two PlannedReservation and two RealTimeReservation) objects are created with default values on their parameters and associated with *s*. One PlannedReservation and one RealTimeReservation are considered as inside the supermarket (*entranceTime* != null) while the other two are respectively scheduled and lined up.

Tear Down: this method is called after each test is executed. Reservations list is cleared.

Tests:

`testAddReservation():` the effectiveness of the reservations-supermarket association is tested by evaluating the size of the *reservations* array of the supermarket. The expected value is the number of created reservations, that is four in this case.

`testRemoveReservation():` two of the reservations are removed from the supermarket, then the size of *reservations* array is evaluated. The expected value is the previously evaluated size minus the number of removed reservations, that is two in this case.

`testGetTimetable():` the number of scheduled planned reservations is evaluated. The expected value is the number of planned reservations associated with the supermarket, that is two in this case.

`testGetVirtualLineUp():` the size of the *virtual line-up* is evaluated. The expected value is the number of real-time reservations waiting for the entrance, that is one in this case.

`testGetDoingGroceryShopping():` the number of entered reservations is evaluated. The expected value is the number of reservations with not null entrance time value, that is two in this case.

5.1.4 TimetableTest

Attributes:

- `timetable : Timetable`
- `reservations : ArrayList<PlannedReservation>`

Setup: this method is called before each test is executed. For valid test cases, a new ArrayList<PlannedReservation> object is created, it is filled with 4 Reservation objects, then a new empty Timetable object is created. The same happens for the setup method in the nested class, but only one Reservation with null *startTime* value is added to the ArrayList.

Tear Down: this method is called after each test is executed. Timetable is set to null.

Tests:

`testComputeSchedules():` *computeSchedules* method of Timetable class is invoked. The expected outcome is a Timetable object with a *schedule* field of a certain size (4 in this case).

`testGetAllScheduledNotScanned():` *getAllSchedulesNotScanned* method of Timetable class is invoked. The expected outcome is a Timetable object with a *schedule* containing a certain number of **not scanned** (so not enter the supermarket yet) PlannedReservation (2 in this case).

`testComputeSchedules() (WhenStartTimeIsNull):` this method checks that *computeSchedules* method of Timetable class fails, since the *startTime* value cannot be null.

5.1.5 ETEComputationServiceTest 1 & 2

Attributes:

- `s : Supermarket`

Setup: this method is called before the test is executed. A new empty Supermarket object is created. Then, its capacity is set and some Reservation containing several information are added.

Tests:

`testComputeETE():` this method checks the efficiency of *computeETE* method, given a RealTimeReservation as input, of ETEComputationService component. The actual outcome of the test case is compared with the values of the considered Reservation ETE that we have estimated.

5.1.6 ReservationServiceTest

Attributes:

- `planned : PlannedReservation`
- `real : RealTimeReservation`

Tests:

`testExpiredReservation():` this method checks the efficiency of *expiredReservation* method, given a Reservation data as input (a RealTime for the first assertion and a Planned for the latter), of ReservationService component. The actual outcomes of the test case are verified to be true and possible exceptions throwing are caught.

5.1.7 SupermarketServiceTest

Attributes:

- `supermarket : Supermarket`

Tests:

`testGetSchedules():` this test case intends to check if a supermarket correctly proposes the right time slot list. The final assertion takes care of verifying the proper size of the list.

`testCheckAvailability():` this test case intends to analyse the method *checkAvailability* of SupermarketService, therefore, if a supermarket can correctly accept a given reservation request. In this case, the final assertion takes care of verifying the result is false.

5.1.8 UserServiceTest

Attributes:

- `id : Integer`
- `username : String`
- `password : String`

Tests:

`testAuthenticateValidUser():` this test case analyses the *checkCredentials* method of UserService. Given username and password of a new User, it checks that information are correctly saved in the object.

`testFavouriteSupermarket():` this test analyses the methods for managing the option FavouriteSupermarket of the User, checking if it is correctly set and deleted from the created object.

5.2 Integration tests

The adopted approach for integration testing follows a kind of strategy comparable to a mixture of "bottom-up" and "top-down" approach.

At first, we have implemented and tested the components that do not depend on the others. In particular, we started from **SupermarketService** and **UserService**, and with their integration and testing ("*UserSupermarketDatabaseIntegrationTest*").

At this point, the implementation has carried on by implementing the **Router**, that is the top component of the system and that interacts with all the others. After Router implementation, we have integrated it with Supermarket and then with User (these subsystems have been also tested in "*RouterUserDatabaseIntegrationTest*" and "*RouterSupermarketDatabaseIntegrationTest*").

At the end, we have concluded integrating the subsystem composed by SupermarketService and Router with **ReservationService** and **ETEComputationService**.

5.2.1 SupermarketDatabaseIntegrationTest and UserDatabaseIntegrationTest

These test classes consist of respectively persisting a Supermarket object and a User object with the database and evaluating some interesting methods of SupermarketService and UserService components.

We have tested *SupermarketService.findAllSupermarket* and *UserService.checkCredentials* (with valid credentials and then with invalid ones).

The assertions take care of verifying the results are the expected ones.

Eventually, the object, previously added to the database, are removed.

5.2.2 UserSupermarketDatabaseIntegrationTest

SupermarketService and UserService components have been tested together in this test class.

Before the test case execution, a User object and a Supermarket object are created and associated through the FavouriteSupermarket relation.

They are persisted with the database and then removed at the end of the test case.

The test case makes use of UserService and SupermarketService to retrieve data and check the outcome of getFavouriteSupermarket method.

5.2.3 RouterSupermarketDatabaseIntegrationTest

This class prepares the first test method *testForwardSupermarketListRequest* by retrieving the list of Supermarket objects stored in the database and creating a new one.

The test case verifies that adding the new Supermarket to the system entails an increment of the Supermarket list size.

The added Supermarket is removed from the database table at the end of the test method.

TestForwardAvailableScheduleRequest is defined in a nested class because it has to be preceded by a setup method different from the one that has been implemented for the previous test. This time it would be interesting to verify that, given the current time and a certain closing time, the proposed timeslots are as many as we expect.

5.2.4 RouterUserDatabaseIntegrationTest

This test class takes care of executing and testing methods from Router component concerning Registration and Authentication.

In *testSignUpRequest* method a User is initialized and persisted with the database, then it checks that the sign-up request, forwarded by Router, returns a not null object (which means that registration has been successfully executed).

In *testSignInRequest* method, Router forwards a login request, specifying the username and password previously provided, then it checks that the authentication is correctly fulfilled.

5.2.5 RouterReservationSupermarketETEDatabaseIntegrationTest

In the setup method of this test class, a Supermarket and a User objects are initialized and persisted with the database.

In this case we want to verify the effectiveness of the real-time reservation creation process. In order to do so, the outcome of the *forwardRealTimeReservationCreationRequest* invocation is evaluated, as well as the fact that not null reservation is associated with the User.

TestForwardPlannedReservationCreationRequest is defined in a nested class because it has to be preceded by a setup method different from the one that has been implemented for the previous test. Also in this case, we want to verify the positiveness of *forwardPlannedReservationCreationRequest* outcome and the effective association between the User and the newly created PlannedReservation.

6 Installation instruction

The installation instructions to follow for being able to run the prototype are listed in this section.

At first, all the prerequisites needed to start the installation phase are presented:

- Having a JVM (Java Virtual Machine) properly installed.
- Having the Eclipse IDE for Enterprise Java Developer properly installed on the personal computer.
- Having an updated version of Oracle JRE and JDK (for example jdk-15.0.1), downloadable from this link:
  https://www.oracle.com/java/technologies/javase-downloads.html
- Having an instance of MySQL Community server 8.0 and MySQLWorkbench installed, downloadable from the following links:
  MySQL Community server 8.0: https://dev.mysql.com/downloads/file/?id=501540
  MySQLWorkbench: https://dev.mysql.com/downloads/file/?id=500617
- Having the extracted Apache TomEE installation zip file located in the "*Program Files\Apache Software Foundations*" folder, downloadable from this link:
  https://www.apache.org/dist/tomee/tomee-8.0.3/apache-tomee-8.0.3-plume.zip
- Having the extracted MySQL Connector installation zip file located in the "*Program File\MySQL\mysql-connector-java-8.0.22"* folder, downloadable from this link:
  https://downloads.mysql.com/archives/get/p/3/file/mysql-connector-java-8.0.22.zip

- Adding MySQL Connector to Apache TomEE installation directory: go to "*Program File**MySQL**mysql-connector-java-8.0.22",* copy "mysql-connector-java-8.0.22.jar" file and paste it into "*Program Files**Apache Software Foundations**apache-tomee-plume-8.0.x**lib"* folder.
- Adding TomEE to Eclipse: open the Eclipse workspace, select Windows > Preferences > Server > Runtime Environments and then click on "Add"; among all the possible options, select "Apache Tomcat v9.0".
- Go to the tab "Servers" by selecting Windows > Select View > Server and then click on "Create new server", then select the server runtime environment that you have created in the previous step and click on "Finish". At this point, the TomEE server is connected to Eclipse and one can interact with it through the Eclipse interface.

After these introductory steps it is possible to proceed with the installation guide, to import the project and to set it properly.

1. Import the CLupEJB and the CLupWEB projects into the Eclipse Workspace.
2. Right click on the CLupWEB project and select "Properties". Navigate to "Deployment Assembly" and verify that the EJB project is included. If it is not, add it to the web project.
3. Right click on the CLupWEB project and select "Properties". Navigate to "Targeted Runtimes" and select "Apache Tomcat v9.0".
4. Server connection settings:
    a. Right click on the CLupWEB project and select "Properties". Navigate to "Project Facets", check JPA box and click on "Further configuration available…". Then click on "Manage Libraries…" button and on "New…" from the newly appeared window.
    b. Call the library "EclipseLink", for instance, click OK and then on "Add External JARs…". Select all .jar files that you find in the repository inside the "EclipseLink.zip" archive, then click on "Apply & Close".
    c. Check EclipseLink library and click on "Add connection…", select MySQL and name the connection "CLupDB" for instance.
    d. Click on "Next" and then on "New Driver Definition" symbol.
    e. Select MySQL JDBC Driver version 5.1, click on "JAR List" tab, remove the current connector (if present) and click on "Add JAR/Zip…". Select "*mysql-connector-java-8.0.22"* file from "*Program Files**Apache Software Foundations**apache-tomee-plume-8.0.x**lib"* folder. You can now click "OK" button.

f. Now select the created driver in “Drivers” menu and fill form field with these strings:
   Database: clup
   URL: jdbc:mysql://localhost:3306/clup
   User name: YOUR_USERNAME
   Password: YOUR_PASSWORD
g. Click now on “Finish” and then on “OK”.

5. If not yet added, add to the classpath of CLupWEB project the .jar files of the ThymeLeaf library, downloadable via this link (but you should find them already in the *“Web Content/WEB-INF/lib”* folder of the project CLupWEB): https://www.thymeleaf.org/download.html
6. If not yet added, add to the classpath of CLupWEB and CLupEJB projects the .jar files of the ZXing library, downloadable via this link (but you should find them already in the *“Web Content/WEB-INF/lib”* folder of the project CLupWEB): https://www.jar-download.com/artifacts/com.nepxion/zxing/1.1.1/source-code
7. Import the CLup schema into MySQL Workbench.
8. Add an xml file called “tomee.xml” to the Server -config folder and then add the following code inside:

```
<?xml version="1.0" encoding="UTF-8"?>
<tomee>
        <Resource id="CLup" type="DataSource">
        JdbcDriver com.mysql.cj.jdbc.Driver
        JdbcUrl jdbc:mysql://localhost:3306/clup
        UserName YOUR_USERNAME
        Password YOUR_PASSWORD
        </Resource>
</tomee>
```

9. If not yet added, go to CLupEJB classpath, click on “Add Library…”, then select JUnit and finally “JUnit 5” as JUnit library version. Click on “Finish” and then on “Apply & Close”.
10. In order to carry out tests, open “persistence.xml” file, go to “Connection” tab and change Transaction type from “Default (JTA)” to “Resource Local”. Then click on “Populate from connection…” and select your previously created Data Resource.

Server time zone error

If Eclipse fails to connect to the database due to server time zone error, follow the below steps:

1. Open “tomee.xml” file from Server -config folder.
2. Append the following string to the current JdbcUrl: “?serverTimezone=UTC”

3. You should now have this:

```
<?xml version="1.0" encoding="UTF-8"?>
<tomee>
        <Resource id="CLup" type="DataSource">
        JdbcDriver com.mysql.cj.jdbc.Driver
        JdbcUrl jdbc:mysql://localhost:3306/clup?serverTimezone=UTC
        UserName YOUR_USERNAME
        Password YOUR_PASSWORD
        </Resource>
</tomee>
```

4. Open “Data Source Explorer” tab (if you do not have it you can find it on Window > Show View).
5. Right click on “CLupDB” and then on “Properties”.
6. Open “Driver Properties” tab and append to “URL” field the following string: “?serverTimezone=UTC&allowPublicKeyRetrieval=true&useSSL=false”
7. Click on “Apply & Close”.
8. Open MySQL Workbench.
9. Open your connection.
10. Go to “Administration” tab and then on “Options file”.
11. On the “General” tab search for “default-time-zone” attribute, check it and set it with the value '+01:00' (with quote marks included).
12. Click on “Apply…”

7 Revision History

- **ITD1.pdf**: delivery document
- **ITD2.pdf:**
    a) Added server connection settings
    b) Added server time zone error
    c) Added resources download links