POLITECNICO
MILANO 1863

Software Engineering 2
A.Y. 2020-2021

CLup – Implementation and Testing Document

Vincenzo Riccio, Giancarlo Sorrentino, Emanuele Triuzzi
https://github.com/SirGian99/RiccioSorrentinoTriuzzi

February 7th, 2021

Deliverable:	ITD
Title:	Integration and Testing Document
Authors:	Vincenzo Riccio, Giancarlo Sorrentino, Emanuele Triuzzi
Version:	1.0
Date:	February 7th, 2021
Download page:	https://github.com/SirGian99/RiccioSorrentinoTriuzzi
Copyright:	Copyright © 2021, Vincenzo Riccio, Giancarlo Sorrentino, Emanuele Triuzzi – All rights reserved

Contents

Chapter 1

Introduction

1.1 Purpose and scope

The purpose of this document is to outline the CLup prototype realized following the design proposed in the Design Document (DD), describing the implementation and the testing activity carried out across the development.

In particular, the document highlights the implemented features, the technology stack used for the development, the code structure and details on the testing phase.

Furthermore, it provides the instructions to prepare a workstation configuration as close as possible to the one used during the software development and testing, in order to perfectly use and test the entire system.

All the references to any software used during this phase are mentioned at the end of the document.

1.2 Definitions, acronyms, abbreviations

Definitions, acronyms, abbreviations	
AMS	Access Management System
TAS	Turn Announcement System
CLup	Also known as the system. It is the software to be developed. From a design-oriented point-of-view, the term is also used to refer to the mobile application, the administrative tool and the server all together
Customer application	Also known as application. It is used to access the functions provided by CLup
Administrative tool	the tool provided to store managers in order to administer stores

Proxy	The physical fallback option for customers that want to use CLup but cannot use the application. It is placed outside the store it belongs to
Turn Announcement System	An external system which informs customers about who has been allowed by CLup to enter the store it belongs to
Access Management System	An external system which regulates physical entrances and exits to the store it belongs to by interacting with CLup
App-customer	A customer who uses CLup functions through the application
Proxy-customer	A customer who uses CLup functions through the proxy
User	Either a customer or a store manager
Long-term customer	With respect to a certain store, a customer who already used CLup to visit it
Current occupancy	Also known as occupancy. It can be referred to the store or one of its sections. For the store, it is the number of people inside it. For the product sections, it is an index of the approximate number of people inside it the system can be aware of
Maximum occupancy	Refers to the store or one of its sections. It is the maximum number of people allowed to be in that area
Virtual queue	Also known as access queue or simply queue. It represents the set of customers who lined up through the app or the proxy
Line-up	With respect to a customer and a store, it is the event of joining the queue
Visit request	A customer's request to visit a store. It can be either a line-up request or a booking request
Line-up request	A request made by the customer to line-up for a store
Booking request	A request made by the customer to book a visit to a store

Visit	The realization of a visit request which takes place when a customer enters the store. After the customer exits the store, we talk about completed visit, otherwise it is a visit in progress
Visit token	A unique token bound to a visit request. It allows the Customer to enter and exit the store
Pending request	A customer's visit request that does not have a visit associated with and is not allowed to enter the store it is associated with
Ready request	A customer's visit request that does not have a visit associated with and is allowed to enter the store it is associated with
Fulfilled request	A customer's visit request that has an associated visit in progress
Completed request	A customer's visit request that has an associated completed visit
Active request	A customer's visit request that is not a completed request (thus it is either a pending, a ready or a fulfilled request)

Table 1.1: Definition, acronyms, abbreviations

1.3 Revision history

1.0 – First version of the document (February 7th, 2021).

1.4 Reference documents

- I&T Assignment A.Y. 2020-2021;
- CLup Requirements Analysis and Specification Document;
- CLup Design Document;
- Teaching material provided by professors Matteo Rossi and Elisabetta Di Nitto.

1.5 Document structure

The reference structure used for the document is an adapted version of the one suggested by professor Matteo Rossi of Politecnico of Milan.

Chapter 1

Chapter 1 is an introduction to the implementation and testing of the software presented in the Requirement Analysis Specification Document and the Design Document.

Chapter 2

Chapter 2 defines the implemented functionalities of the system.

Chapter 3

Chapter 3 focuses the analysis on the adopted programming languages, frameworks and external APIs used to implement the system and to perform the testing.

Chapter 4

Chapter4 describes the source code structure of both the Server and the mobile applications.

Chapter 5

Chapter 5 provides information on how the testing has been performed and its outcome.

Chapter 6

Chapter 6 contains a list of instructions to install the mobile applications, the JavaServer and to configure the database.

Chapter 7

Chapter 7 contains a report on the effort spent by all the members of the group while writing the current document.

Chapter 8

Last chapter contains references to the tools and resources used to implement the system and to write the document.

Chapter 2

Implemented functions

2.1 Overview

The developed prototype focuses on the customers' use cases outlined in the RASD, although simplified. It includes:

- a fully-implemented database schema, including a test instance;
- a server providing part of the designed functionalities;
- an iOS mobile application for the customer;

Thus, the provided functionalities are the following:

- allow customers to line up for a desired store;
- allow customers to book a visit for a desired store;
- allow customers to cancel a request they made;
- register a mobile application;
- retrieve the active requests of a customer;
- retrieve chains and stores of a specified city
- manage entrances of a customer;
- manage exits of a customer;
- automatic state changing for line up requests;
- scheduling of automatic state changing for booking requests;
- check the current occupancy constraints of the store;
- provide a raw estimation of the queue disposal time of a store;
- provide a raw estimation of the estimated time of entrance of a lineup request.

The server prototype does not foresee the functionality described in the DD which changes the access policy of each store taking into consideration which of the two different “zones” the store is in (the yellow and green one). For the sake of simplicity of the prototype, stores are considered always in the “green zone”.

Also, the prototype does not focus on the performance or on the software system attributes identified in the RASD, since it is intended only to demonstrate the main CLup functionalities.

2.2 Requirements achieved

The implemented prototype satisfies most of the functional and nonfunctional requirements presented in the CLup Requirements Analysis and Specification Document (RASD) and the Design Document (DD).

Functional requirements

- The system shall allow managers to monitor entrances (R2);
- The system shall allow managers to monitor exits (R3);
- The system shall authorize accesses to the store (R4);
- The system shall authorize customers to enter if and only if the store would not exceed the maximum number of people allowed inside it (R4.1);
- The system shall provide a way to line-up in the virtual queue of the store (R5);
- The system shall provide a way to exit the queue before entering the store (R6);
- The system shall provide the possibility to book a time interval for visiting the store (R8);
- The system must not allow customers to book a visit in a time interval if, over its duration, bookings by other users already maximize store occupancy (R9);
- When booking a visit, the system shall allow customers to specify what kind of products they intend to buy (R10);
- The system shall provide the possibility to cancel a booked visit before entering the store (R11);
- The system shall inform customers when they are allowed to enter the store (R19);

Non-functional requirements

- The application should be easy to use (NF10);
- CLup shall give priority to booking requests over line-up requests (NF11);
- Customers can remotely line-up in a store’s queue only if they are not in the queue of any store at that moment (NF12);

- Customers can book a visit to a store for a specific time interval only if they have not booked any other visit which overlaps with that time interval (NF13);
- Customers can book a visit to a store for a specific time interval only if it starts after the current queue disposal time of that store (NF14).

Chapter 3

Adopted frameworks

This section better delve into the technology stack outlined in the Design Document. The design decisions omitted in this chapter are retrievable from the Design Document.

3.1 iOS Mobile application

The mobile application logic has been developed in Swift, an powerful and intuitive programming language designed by Apple. Meant as an alternative to Objective-C, and up to 2.6 times faster than it, in recent years it has become the main programming language for iOS and macOS applications. To build the UI, the chosen framework is SwiftUI, which offers a declarative way to build user interfaces with few lines of code. It is developed by Apple itself and thus it is designed for Swift and Apple devices.

Furthermore, SwiftyJSON was used for easy managing of JSONs.

3.2 Java Server

The server has been implemented using Apache TomEE, which combines different Java Enterprise projects used to satisfy the requirements:

- *Apache Tomcat*: HTTP server and Servlet container supporting Java Servlet;
- *Apache OpenEJB*: Open-source Enterprise JavaBeans (EJB) container system. Enterprise JavaBeans is a server-side software that encapsulates business logic of the server application;
- Apache OpenJPA: Open-source Java Persistence API (JPA) 2.1 implementation.

  JPA is a Java EE application programming interface specification that describes the management of relational data in enterprise Java applications. Persistence in this context covers three areas:

  - the API itself, defined in the `javax.persistence` package;
  - the Java Persistence Query Language (JPQL), which queries against entities stored in a relational database. Queries resemble SQL queries in syntax, but operate against entity objects rather than directly with database tables;
  - object/relational metadata.

The interfaces between the presentation layer and the application layer are designed according to the REST (Representational State Transfer) architectural style and are based on the HTTP protocol. Thus, each interaction between those layers is stateless and follows a client-server approach.

Each resource is identified by a URI and the transferred data are represented using the JSON standard.

3.2.1 EclipseLink

Since JPA is only a specification, a Persistence Provider is needed to implement the Object Relational Mapping. The chosen provider for this purpose is Eclipselink.

3.2.2 JAX-RS API

JAX-RS is part of the Java EE specification and is designed to simplify the development of RESTful applications. It consists of a set of interfaces and annotations defined in the javax.ws.rs package.

3.2.3 Apache Maven

As Maven is perfectly integrated in IntelliJ IDEA, it has been used as software project management tool to manage the dependencies and the lifecycle of each module.

Each module, better detailed in the next section, is described by its own pom.xml, which is the Maven project object model (POM) descriptor. It is an XML file that contains information about the project and configuration details used by Maven to build the project.

3.2.4 org.json

Also known as "*JSON-Java*", it is one of the most popular libraries used to deal with JSON (JavaScript Object Notation) in Java. It also includes the capability to convert between JSON and XML, HTTP headers and more. It is used to parse (resp. prepare) the request (resp. response) which is received (resp. sent) by the RESTful servlets.

- *JSONObject*: similar to Java's native Map object, where keys are Strings that cannot be null and values are anything from Boolean, Number, String, JSONObject or JSONArray, or null;
- *JSONArray*: it is an ordered collection of values, resembling Java's native Vector implementation.

The main methods used are:

- `put(key: String, value: Object)`: inserts in the JSONObject the field specified;
- `put(element: Object)`: appends the element to the JSONArray;
- `get(key: String)`: gets the object associated with the supplied key, throws JSONException if the key is not found.

3.2.5 JUnit and Mockito

JUnit5 is the tool chosen in order to perform unit and integration testing. The tool is perfectly integrated with Maven and IntelliJ IDEA and serves as a foundation for launching testing frameworks on the JVM. In particular, JUnit has been used together with the Mockito framework, which allows to perform scaffolding, defining specific stubs whenever the components under test cannot be tested in isolation.

Chapter 4

Code structure

4.1 iOS Mobile application

The mobile application code is organized as follows:

- the **Model** folder contains what is needed to model the data received from the server and to make them available to the entire application;
- the **Views** folder contains the pure frontend part of the mobile application, the SwiftUI views;
- the **Assets.xcassets** folder and the **LaunchScreen.storyboard** file gather the static data which are shown in the user interfaces or at the launch of the app
- **SwiftyJSON.swift** contains the homonym framework used to deal with JSONs;
- **Extensions.swift** and **ViewExtensions.swift** contain our extensions of the classes/structures provided by Swift and SwiftUI;
- **Routes.swift** contains the needed endpoints of the RESTful API offered by the server;
- **CLupApp.swift** is the app main file, according to the SwiftUI lifecycle;
- **SIController.swift** contains the controller of the interactions with the server.

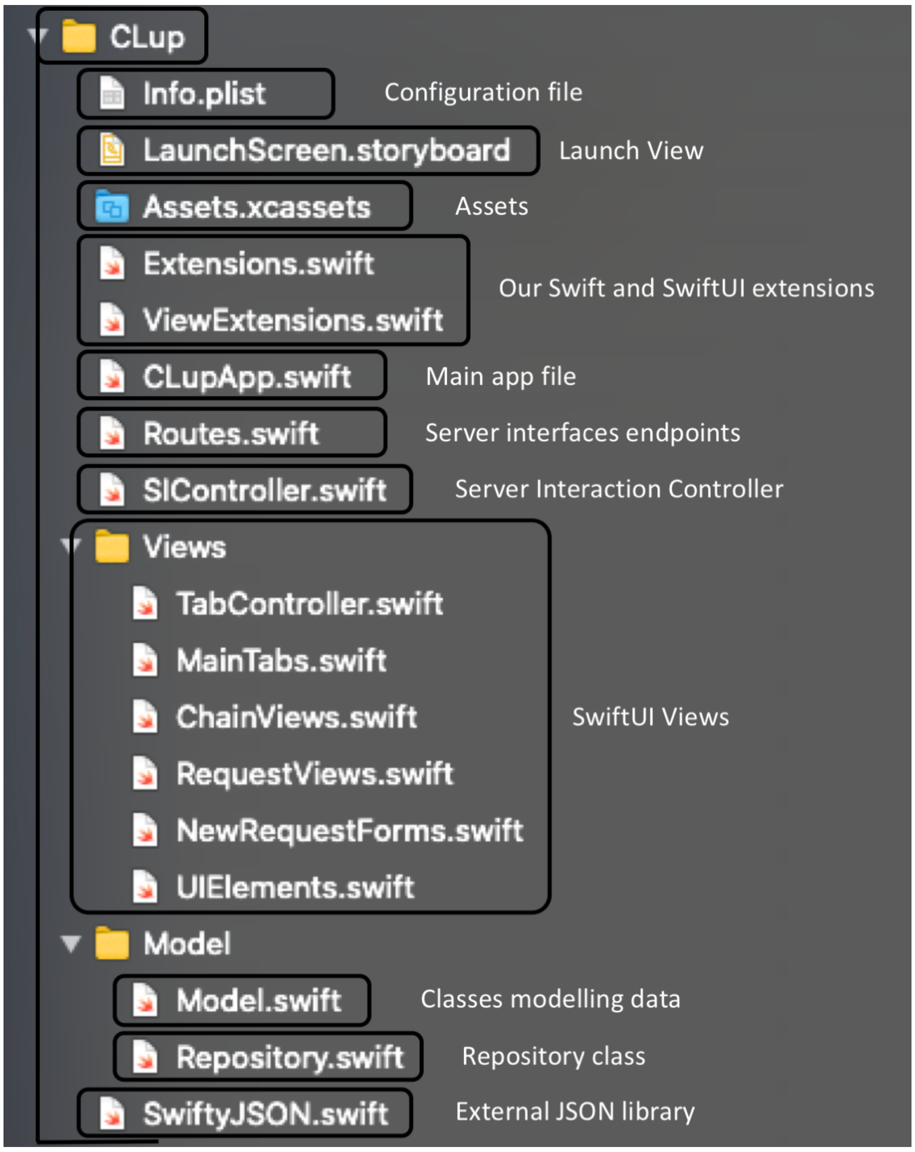


Figure 4.1: Code structure of the mobile application

4.2 Java Server

At the highest level, the source code is organized in three base modules, which implement the internal server components, plus a module for the integration testing:

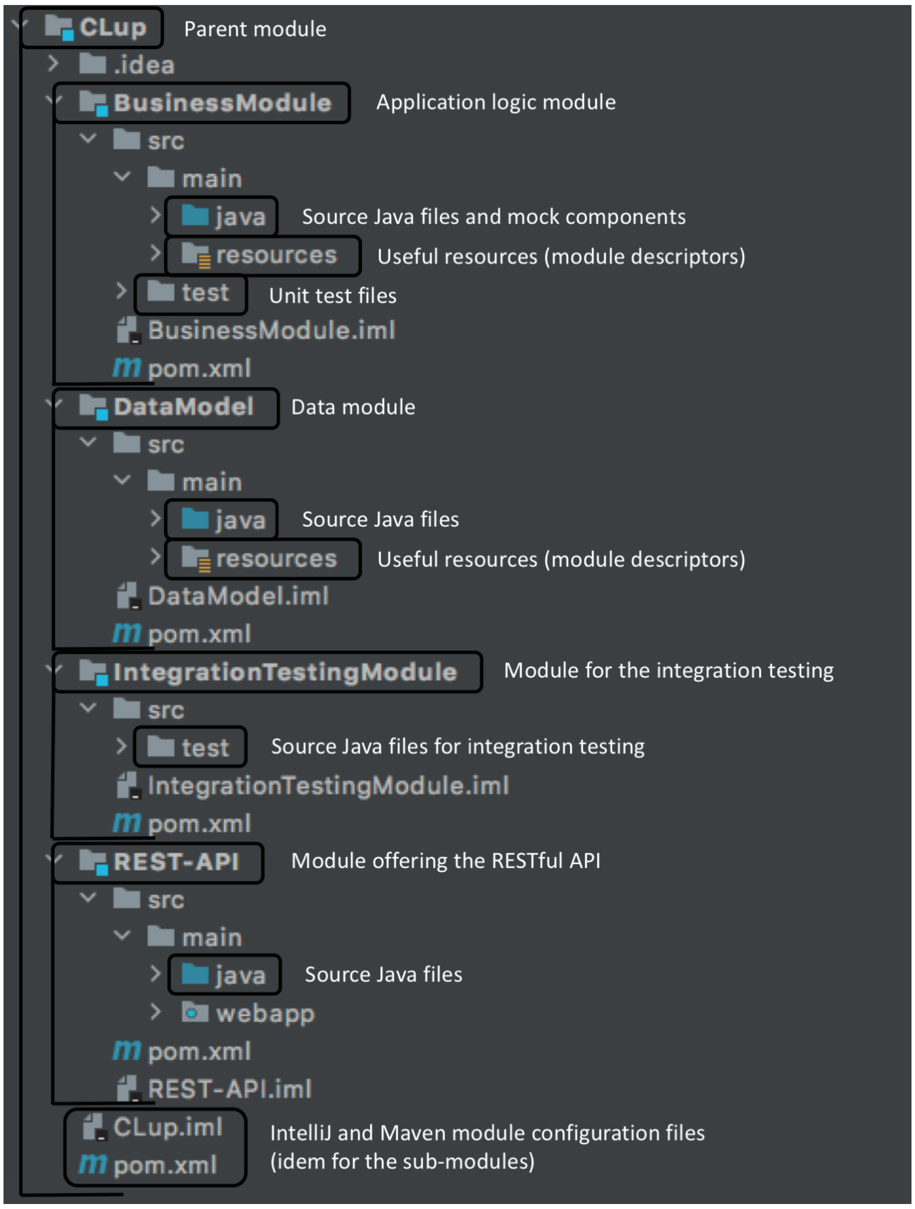
- **BusinessModule**: following the design document guidelines, it includes the following subcomponents as different stateless JavaBeans:
  - **CustomerController**: this component s in charge of retrieving and providing customers with their information, such as their active line-up and booking requests;
  - **RequestHandler**: this component is the one in charge of collecting customers' visit requests, accepting or rejecting them, and of cancelling the same requests whenever the customer who placed them asks for it;

  - **StoreStatusHandler**: this component is the one in charge of providing all the information concerning the stores managed by CLup, such as their opening time, their current occupancy, their product sections and relative occupancies, their current queue disposal time;
  - **VisitManager**: this component is the one in charge of coping with visits and visit requests to the stores managed by CLup. In fact, it regulates the order in which customers are allowed to visit the stores they made visit requests for. The VisitManager also provides an interface to allow entrances and exits to the stores managed by CLup, by checking if, given a visit token and a store, the token is associated with a visit request in ready state for the selected store.

- **DataModel**: it is the module containing the homonym bean and the entities (classes) which map the database tables. It defines the data model of the system and interacts, thanks to JPA, with the DBMS in order to ensure data persistence. All the other components of the system interact with the stateless DataModel bean, which is the only one in charge of dealing with the persistence context. It contains the JPA descriptor (`persistence.xml`).

- **REST-API**: it is the module which, following the design document guidelines, contains the servlets offering the interfaces mentioned in the DD. It contains the classes managed by the JAX-RS framework and which interact with the beans of the BusinessModule to offer the requested service.

- **IntegrationTestingModule**: it is the module used to perform the integration testing of the previous ones. More details are provided in the section concerning the integration testing.



```
CLup                          Parent module
  .idea
  BusinessModule              Application logic module
    src
      main
        java                  Source Java files and mock components
        resources             Useful resources (module descriptors)
      test                    Unit test files
    BusinessModule.iml
    pom.xml
  DataModel                   Data module
    src
      main
        java                  Source Java files
        resources             Useful resources (module descriptors)
    DataModel.iml
    pom.xml
  IntegrationTestingModule    Module for the integration testing
    src
      test                    Source Java files for integration testing
    IntegrationTestingModule.iml
    pom.xml
  REST-API                    Module offering the RESTful API
    src
      main
        java                  Source Java files
        webapp
    pom.xml
    REST-API.iml
  CLup.iml                    IntelliJ and Maven module configuration files
  pom.xml                     (idem for the sub-modules)
```

Figure 4.2: Code structure of the Java Server

Chapter 5

Testing

Following the testing plan outlined in the DD, the testing phase consisted in the unit testing and the integration testing process. At the end of these tests, a deep end-to-end system test of the functions included in the prototype has been performed, simulating the typical user interactions with the mobile application.

5.1 Server Unit test

The purpose of unit tests is to verify the correctness of the algorithms and the code implemented in the component under test. The de-facto standard for executing these kinds of tests on Java is the JUnit framework, which allows to perform repeatable tests, checking for the correct execution of the given test code.

Fundamental characteristic of unit tests is the possibility to test the code in isolation and limiting as much as possible side effect situations. This in order to avoid the unpleasant situation in which the execution of the tests leads the entire component under test in a state of inconsistency.

Unit tests have been performed on the most significant functionalities of the prototype components. In order to achieve the expected results, the Mockito framework has been exploited to run unit tests whenever required. Indeed, Mockito allows to generate mock objects of the needed class, thanks to the method mock(Class), and to simulate the mocked object methods, if needed. The mock extends the class that the component under test requires to correctly work, exposing the same interface of the simulated class. In this way, it is possible to ignore the fact that we are dealing with a mock object. An example of this behaviour can be found in each unit test: since EJBs injection is not performed during the JUnit execution and since all the components interact with the DataModel, this last component has been mocked. Its methods' behavior is simulated whenever needed, thanks to the `when(dataModel.method()).then(doSomething)` Mockito method.

Moreover, in order to manually inject the mocked DataModel bean in each of the different beans under test, specific classes have been designed, which extend the real ones, placed under the package `it.polimi.se2.ricciosorrentinotriuzzi.business.components.mockcomponents` in the BusinessModule.

5.2 Server and Database Integration Test

Integration test is a fundamental test phase in which components interacting with other ones are tested together, in order to check whether their interaction has been designed properly. To do so, the JUnit and Mockito frameworks have been used to test these interactions, mocking absent components when needed, as in the case of the integration test between the RequestHandler, the DataModel and Database.

Tests are performed in the IntegrationModule, a module defined ad-hoc for the integration testing, which depends on both the DataModule and the BusinessModule, that is not included in the CLup deployed artifact and which contains the classes specifying each test. Moreover, a new class that extends the DataModel one has been defined, called "DataModelMock", that facilitates the execution of the integration testing.

Since all the following specified tests involve the database, a correct initialization of its state is required in the setup method of each of them, executed by JUnit before every single test. This is done by starting a new JPA transaction and invoking the dbInit method of the DataModelMock, which initializes the database, emptying all its tables. Then, the database is populated with the instances of the entities involved in each test, generated ad-hoc in the setup method.

Finally, after the execution of each of them, the transaction is rolled back in order to not affect the real data in the database.

5.2.1 I1 - VisitManager and DataModel

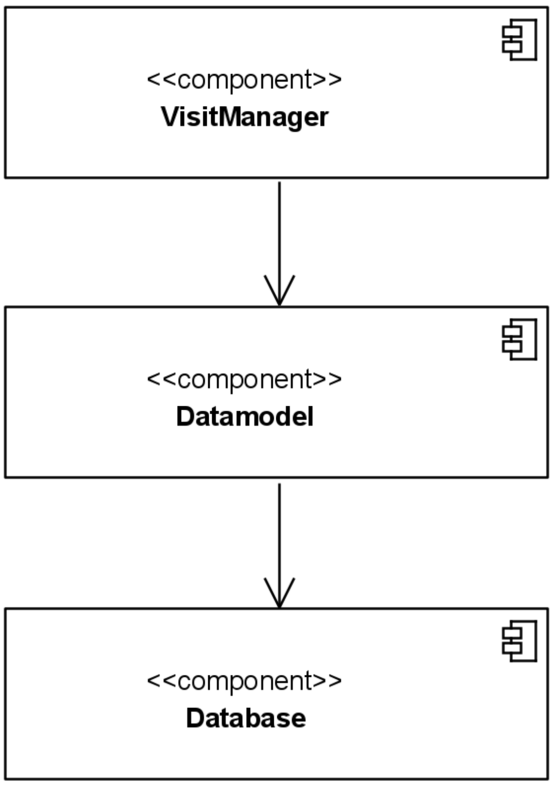


Figure 5.1: VisitManager, DataModel and Database integration test

Test Group ID	I1 - VisitManager and DataModel
Server Subcomponents	VisitManager, DataModel
Description	Group of tests to check the interactions between the VisitManager and the DataModel
Environmental needs	The DataModel and the database are working properly

Table 5.1: Test Group I1 - Visit Manager and DataModel

Validate Access

Test ID	I1.1 - Validate access
Server Subcomponents	VisitManager, DataModel
Input Specification	Invocation of the `validateAccess` method from the AccessControl API, specifying a visit token and a store ID
Output Specification	Check whether the number of people allowed to enter is equal to the one specified in the request if it is in ready state, zero otherwise
Description	The test is focused on verifying the acceptance of a request for accessing its specified store. The VisitManager invokes the `checkReadyRequest` method of the DataModel, which queries the database returns the number of people allowed to enter the store if the request is in ready state, zero otherwise

Table 5.2: Test I1.1 - Validate access

Validate exit

Test ID	I1.2 - Validate exit
Server Subcomponents	VisitManager, DataModel
Input Specification	Invocation of the `validateExit` method from the AccessControl API, specifying a visit token and a store ID
Output Specification	Check whether the number of people allowed to exit is equal to the one specified in the request if it is in fulfilled state, zero otherwise
Description	The test is focused on verifying the acceptance of a request for exiting its specified store. The VisitManager invokes the `validateExit` method of the DataModel, which queries the database returns the number of people allowed to exit the store if the request is in fulfilled state, zero otherwise

Table 5.3: Test I1.2 - Validate exit

Confirm access

Test ID	I1.3 - Confirm access
Server Subcomponents	VisitManager, DataModel
Input Specification	Invocation of the `confirmAccess` method from the AccessControl API, specifying a visit token, the number of people entering the store and the store ID
Output Specification	Check if the store occupancy is correctly increased, if the request status is “fulfilled", its “number of people” field is correctly updated and if the visit starting time is correctly set
Description	The VisitManager invokes the `startVisit` method of the DataModel, which queries the database and, if the request is in ready state, modifies it in “fulfilled", correctly updates the number of people specified in the request if needed and increases the occupancy of the store

Table 5.4: Test I1.3 - Confirm access

Confirm exit

Test ID	I1.4 - Confirm access
Server Subcomponents	VisitManager, DataModel
Input Specification	Invocation of the `confirmExit` method from the AccessControl API, specifying a visit token, the number of people exiting the store and the store ID
Output Specification	Check if the store occupancy is correctly decreased, if the request status is “completed" and if the visit completion time is correctly set
Description	The VisitManager invokes the `endVisit` method of the DataModel, which queries the database and, if the request is in fulfilled state, modifies it in “completed" and decreases the occupancy of the store

Table 5.5: Test I1.4 - Confirm exit

New request

Test ID	I1.5 - New request
Server Subcomponents	VisitManager, DataModel
Input Specification	Invocation of the `newRequest` method from the RequestHandler, which informs that a new visit request has been placed
Output Specification	In case of a Line-up request, it checks that the request is in: • ready state if the customer can immediately enter the store, since the current occupancy would not exceed its maximum; • pending state, otherwise. In case of a Booking request, it performs the same check when the specified desired time of entrance comes
Description	In case of Line-up requests, the visit manager immediately checks if the store would exceed its maximum occupancy if the request was in ready state. If not so, then it invokes the `allowVisitRequest` of the DataModel, that in case of line-ups also updates their estimated time of entrance. The same job is scheduled to be performed when the specified desired time of entrance comes in case of booking requests

Table 5.6: Test I1.5 - New request

Check new ready request

Test ID	I1.6 - Check new ready request
Server Subcomponents	VisitManager, DataModel
Input Specification	Invocation of the `checkNewReadyRequest` method from the RequestHandler when a customer deletes a pending or ready request, and from the VisitManager itself when a customer exits the store
Output Specification	It checks whether all the requests that can enter the store without exceeding its maximum occupancy are set in ready state, following the access policies specified in the RASD and DD
Description	The DataModel is asked by the VisitManager to query the database in order to retrieve all the pending bookings that should be allowed to enter the store. Then the VisitManager determines which of them can actually enter the store, and asks the DataModel to set them in ready state. Then, if there is space available in the store in terms of current occupancy, the same job is performed over pending line-up requests

Table 5.7: Test I1.6 - Check new ready request

5.2.2 I2 - RequestHandler and DataModel

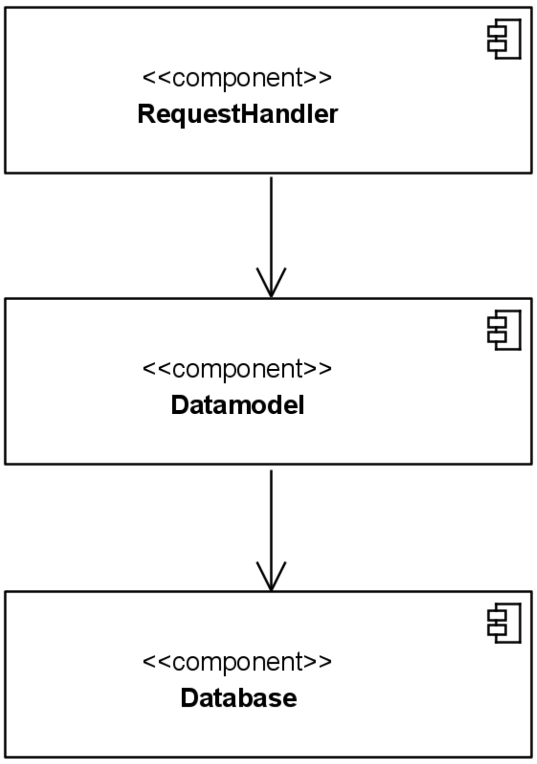


Figure 5.2: RequestHandler, DataModel and Database integration test

Test Group ID	I2 - RequestHandler and DataModel
Server Subcomponents	RequestHandler, DataModel
Description	Group of tests to check the interactions between the RequestHandler and the DataModel
Environmental needs	The DataModel and the database are working properly. The interactions between the RequestHandler and the VisitManager are not considered during the test

Table 5.8: Test Group I2 - RequestHandler and DataModel

Line-up

Test ID	I2.1 - Line-up
Server Subcomponents	RequestHandler, DataModel
Input Specification	Invocation of the `line-up` method from the LineUp interface when a customer requests to line-up for a store
Output Specification	Checks that the request is rejected if: • The number of people specified in the request is not strictly positive or exceeds the store maximum occupancy; • The store is closed; • The customer has already made a line-up request for any of the available stores. Checks that the request is accepted if: • The number of people specified in the request does not exceed the store maximum occupancy, the store is open and the customer is not already in line for any available store.
Description	The number of people specified in the request does not exceed the store maximum occupancy, the store is open and the customer is not already in line for any available store

Table 5.9: Test I2.1 - Line-up

Booking

Test ID	I2.2 - Booking
Server Subcomponents	RequestHandler, DataModel
Input Specification	Invocation of the `book` method from the Booking interface when a customer request to book a visit for a store
Output Specification	The request is rejected if: • The customer is not an app customer; • The customer has already booked a visit (possibly to another store) that overlaps with the specified desired time-interval; • The store is closed during the specified time interval; • The number of people specified in the request is not strictly positive or is greater than the store maximum occupancy; • The desired time-interval begins before the end of the current estimated queue disposal time of the selected store; • The current occupancy of the store is already maximized by others' booking requests in the specified time interval; • The current occupancy of one of the specified store's product sections, if any, is already maximized by others' booking requests in the specified time interval. The request is accepted if: • During the desired time-interval the customer does not have other bookings, the store does not exceed its maximum occupancy, and the store is open.

Description	The RequestHandler asks the DataModel to retrieve the store and customer information from the database. Then, it checks if the customer is an app customer, that the store is open across the specified time interval and if the specified number of people is consistent with the previously mentioned constraints. Then the RequestHandler invokes the DataModel in order to query the Database and retrieve the updated current queue disposal time, and so checks if the desired starting time is past it. Then, the RequestHandler asks the DataModel to retrieve from the Database the bookings already placed by the customer, and if none of them is found, also to retrieve all the bookings placed by other customers for the selected store in the selected time interval. If occupancy constraints are satisfied, then the request is accepted and the DataModel is asked to persist it on the Database.

Table 5.10: Test I2.2 - Booking

5.2.3 I3 - CustomerController and DataModel

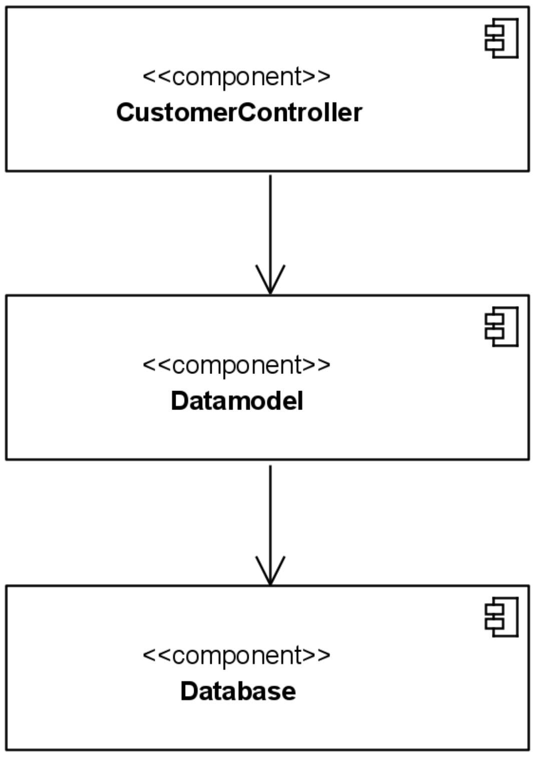


Figure 5.3: VisitManager, DataModel and Database integration test

Test Group ID	I3 - CustomerController and DataModel
Server Subcomponents	CustomerController, DataModel
Description	Testing the interactions between the RequestHandler and the DataModel
Environmental needs	The DataModel and the database are working properly

Table 5.11: Test Group I3 - CustomerController and DataModel

Customer's active line-up requests

Test ID	I3.1 - Customer's active line-up requests
Server Subcomponents	CustomerController, DataModel
Input Specification	Invocation of the method `getCustomerActiveLineups` from the Customer Interface when a customer asks to retrieve its active line-up requests
Output Specification	Checks if each line-up request obtained by querying the database is active
Description	The CustomerController asks the DataModel to query the Database in order to retrieve the active line-up requests of the specified customer

Table 5.12: Test I3.1 - Customer's active line-up requests

Customer's active bookings

Test ID	I3.2 - Customer's active bookings
Server Subcomponents	CustomerController, DataModel
Input Specification	Invocation of the method `getCustomerActiveBookings` from the Customer Interface when a customer asks to retrieve its active booking requests
Output Specification	Checks whether each booking of the customer obtained by querying the database is active
Description	The CustomerController asks the DataModel to query the Database in order to retrieve the active booking requests of the specified customer

Table 5.13: Test I3.2 - Customer's active bookings

5.2.4 I4 - StoreStatusHandler and DataModel

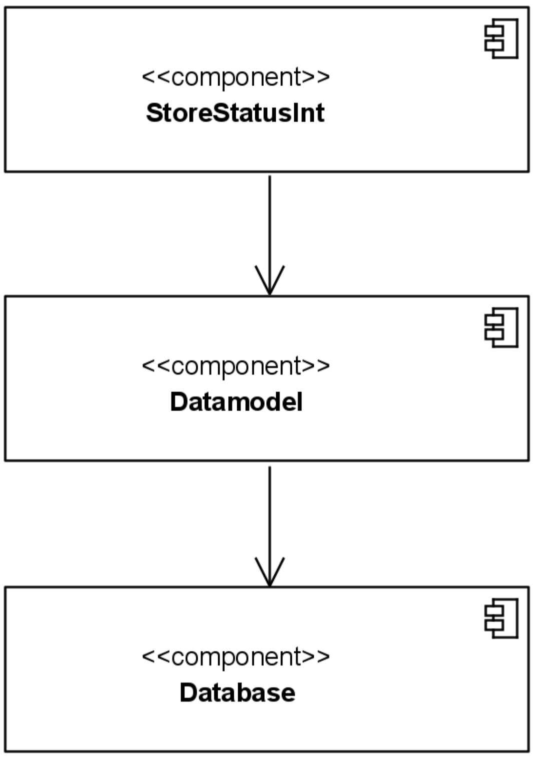


Figure 5.4: StoreStatusHandler, DataModel and Database integration test

Test Group ID	I4 - StoreStatusHandler and DataModel
Server Subcomponents	StoreStatusHandler, DataModel
Description	Testing the interactions between the StoreStatusHandler and the DataModel
Environmental needs	The DataModel and the database are working properly

Table 5.14: Test Group I3 - CustomerController and DataModel

Store general info

Test ID	I4.1 - Store general info
Server Subcomponents	StoreStatusHandler, DataModel
Input Specification	The StoreStatusHandler is asked via the StoreInfoInterface to provide the specified store's information
Output Specification	Checks whether the information obtained from the DataModel is compliant with the test store created and persisted appositely before the test
Description	The StoreStatusHandler asks the DataModel to get retrieve the information about the specified store from the database

Table 5.15: Test I4.1 - Store general info

Get chains and autonomous stores

Test ID	I4.2 - Get chains and autonomous stores
Server Subcomponents	StoreStatusHandler, DataModel
Input Specification	The StoreStatusHandler is asked via the StoreInfoInterface to provide the list of chains and autonomous store, and to restrict the selection to the ones in the specified city, if any
Output Specification	Checks whether the information obtained from the DataModel is compliant with the test stores and the chains created and persisted appositely before the test
Description	The StoreStatusHandler asks the DataModel to get the distinct chains and autonomous stores of the specified city (if any, all of them otherwise) from the database

Table 5.16: Test I4.2 - Get chains and autonomous stores

Get chain's stores

Test ID	I4.3 - Get chain's stores
Server Subcomponents	StoreStatusHandler, DataModel
Input Specification	The StoreStatusHandler is asked via the StoreInfoInterface to provide the list of stores of a given chain, and to restrict the selection to the ones in the specified city, if any
Output Specification	Checks whether the information obtained from the DataModel is compliant with the test stores and the chains created and persisted appositely before the test
Description	The StoreStatusHandler asks the DataModel to get the stores of a specific chain. If a city is specified, the DataModel gets only the stores in that city of the specified chain

Table 5.17: Test I4.3 - Get chain's stores

Chapter 6

Installation instructions

The following instructions are intended to let the reader reach a configuration as close as possible to the one used during the software development and testing. Even if other configurations might work as well, the correct operation of the application is not guaranteed.

As a prerequisite, a Mac running macOS Catalina 10.15.4 or later is required in order to simulate the mobile iOS application.

Through all the instructions, it's assumed that the GitHub repository has already been cloned on the target machine(s).

Check that all the devices used are under the same Local Area Network (otherwise it is needed to configure the software accordingly, which is out of the purpose of this section) and that no firewall is blocking connections between the devices.

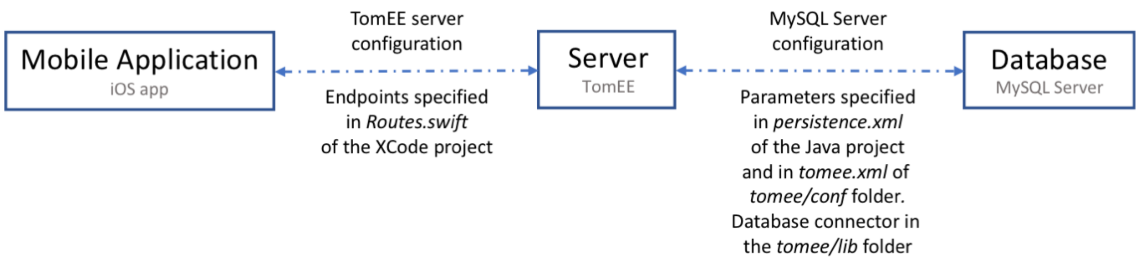

Mobile Application
iOS app
TomEE server configuration
Endpoints specified in *Routes.swift* of the XCode project
Server
TomEE
MySQL Server configuration
Parameters specified in *persistence.xml* of the Java project and in *tomee.xml* of *tomee/conf* folder. Database connector in the *tomee/lib* folder
Database
MySQL Server

Figure 6.1: Configuration preview

6.1 Database

As already stated in the DD, it has been chosen a relational database to persist data. The DBMS used is MySQL, available for free for all the major operating systems. Download and install both MySQL Server and MySQL Workbench. Be sure to install an 8.x version. At the time of writing, version 8.0.23 has some issues running on MacOS, which are not related to this project.

The configuration used during the developing and testing of the software is the following:

- Macbook running macOS Catalina 10.15.7;
- MySQL Server 8.0.22;
- MySQL Workbench 8.0.22.

Through the installation setup of the server, configure it with the following parameters:

- Port number: *3306*;
- Default username: *root*;
- Root password: *EmaGiaVin123*

If MySQL is already installed and/or configured in a different way, it doesn't matter: just edit tomee.xml in tomee/conf and the `persistence.xml` file of the server DataModel module (see the following section). However, they must be changed anyway if the server is hosted by another machine, putting the local IP address of the machine hosting MySQL Server. Then:

- Open MySQL Workbench and connect to the local server;
- Go under the Server menu and choose Data Import;
- Use the option which allows the import from a self-contained file;
- Select the dump provided in Implementation/Database of our GitHub repo;
- Add a new schema `clup_db` as the default target schema;
- Start import by clicking on the homonym button on the bottom right.

6.2 CLup Server

The CLup business tier has been developed using Java Enterprise Edition. IntelliJ IDEA Ultimate is the IDE used during the development, available for free (30-day evaluation period, free for students) for all the major platforms. The configuration used for the development and testing is the following:

- Macbook running macOS Catalina 10.15.7;
- IntelliJ IDEA Ultimate Edition 2020.2.3;
- Oracle OpenJDK 15 (can be downloaded from IntelliJ IDEA);
- Maven 3.6.3 (included in IntelliJ IDEA);
- Apache TomEE plume 8.0.4 (provided in the repo);
- MySQL Connector 8.0.21 (included in the provided tomee/lib folder);

In order to only run the application server, assuming that OpenJDK 15 is already installed and configured, it is only needed to run Apache TomEE. For UNIX-based systems, use the terminal to navigate to the bin folder inside the provided tomee folder and execute `./catalina.sh run`. In order to prepare a suitable workstation for inspecting, running and testing this part of the software, do the following steps:

- Download IntelliJ IDEA Ultimate from the official website and install it;
- Run IntelliJ IDEA and open the project selecting the Implementation/JavaServer/-CLup folder;

- Wait until all the dependencies are automatically downloaded by Maven (take a look on the right part of the bottom bar);
- Go to File > Project Structure > Project (on the side bar). Check that under the section Project SDK it is selected OpenJDK 15 (if not available download and install it by clicking on the relative option of the dropdown menu);
- Configure the application server in IntelliJ IDEA by clicking on Run menu > Edit configuration. Add a TomEE configuration by clicking on the add button (+) and selecting a Local instance of TomEE Server. Click on "Configure" on the upper-right corner and select the provided tomee folder as TomEE Home. Then confirm by clicking on OK. In the deployment tab click on the add button (+) in the "Deploy at the server startup" box and select the CLup artifact (check that the application context is /CLup). Click on OK;
- Run the application by clicking on the play button on the toolbar.

As stated in the instructions for the database installation, *tomee.xml* in *tomee/conf* and the *persistence.xml* file of the DataModel module (available under *src/main/resources* of that module), must reflect the database configuration.

Furthermore, be sure that no other server is running on port 8080, which is the one used by TomEE, otherwise it will throw an error during its execution.

6.3 iOS Customer Mobile application

Since the customer mobile application developed has an iOS device as target, XCode is required to build the application and to test it on a real device or on the simulator included in XCode. An Apple Development Account is needed (it is free and available for every Apple ID). The mobile application has iOS 14.x as target OS.

Thus, a version of XCode which supports iOS 14 is required. The configuration used for developing and testing the application is the following:

- XCode 12.4 (requires macOS Catalina 10.15.4 or later for an Intel-based Mac, or macOS Big Sur 11 or later for an Apple Silicon based Mac);
- iPhone 11 with iOS 14.4, through the simulator;
- iPad Air 2 with iOS 14.4.

XCode can be installed from the Mac App Store. If XCode was never installed before, go to Xcode `Menu > Preferences > Accounts tab`. Then add your Apple Developer Account (basically logging in with your Apple ID) by clicking on the add button on the bottom left. Then open the XCode project file (`CLup.xcodeproj`), go to CLup project (the root element), click on the target element of the left bar and select the `Signing > Capabilities` tab. Under the team section, select your *Personal Team*. Change the bundle identifier if the error "*Failed to register bundle identifier*" appears.

Before running it, the reader must modify the address and the port of the server endpoint according to the server configuration. The attribute `baseURL` of the file `Routes.swift` available under the project tree is what should be edited. If the previous instructions have been followed, overwrite the `baseURL` string with "`http://<SERVER_IP>:8080/CLup`", where `<SERVER_IP>` is the local IP address of the machine hosting the server. If the server is running on the same machine and no physical device is going to be used, simply write localhost instead of `<SERVER_IP>`.

Note that if the TomEE port has been changed, one must overwrite 8080 with the chosen port.

Finally, it only requires to select the target device in the upper toolbar and click on run.

Due to Apple security policy, when testing on a real device, on the first launch of the CLup mobile application it is needed to trust the developer. To do so, go under `Settings > Generals > Device Management > Apple Development > Trust`.

Then restart the app.

6.4 iOS AMS Simulator

In order to test the accesses and exits of customers to a store, the team also developed a test iOS application to simulate the behaviour of the Access Management System of a store. It requires a real device since it uses the device's camera to scan the QR code which identifies the visit request. If not available, one can always simulate it by manually interacting with the RESTful API.

As for the customer mobile application, XCode is required to build the application and to test it on a real device. The AMS Simulator has iOS 13.x as target OS.

Thus, a version of XCode which supports iOS 13 is required. The configuration used for developing and testing the application is the following:

- XCode 12.4 (requires macOS Catalina 10.15.4 or later for an Intel-based Mac, or macOS Big Sur 11 or later for an Apple Silicon based Mac);
- iPad Air 2 with iOS 14.4.

The installation and setup procedure is the same of the iOS customer mobile application.

Chapter 7

Effort spent

All the members of the group worked mainly together, in order to guarantee consistency through the entire document. Each member of the group spent approximately 45 hours doing team working. In addition to team working, individual work has been partitioned as shown in the following tables:

Riccio Vincenzo

Section	**Hours**
Team working	**20**
iOS Customer Mobile Application implementation	17
AMS Simulator implementation	3
Server implementation	8
Database definition	0
Server unit testing	0
Server implementation testing	2
End-to-end testing	5
LaTeX	0

Table 7.1: Effort spent – Riccio Vincenzo

Sorrentino Giancarlo

Section	**Hours**
Team working	**20**
iOS Customer Mobile Application implementation	0
AMS Simulator implementation	0
Server implementation	16
Database definition	2
Server unit testing	7
Server implementation testing	8
End-to-end testing	2
LaTeX	0

Table 7.2: Effort spent – Sorrentino Giancarlo

Triuzzi Emanuele

Section	**Hours**
Team working	**20**
iOS Customer Mobile Application implementation	3
AMS Simulator implementation	0
Server implementation	11
Database definition	4
Server unit testing	4
Server implementation testing	4
End-to-end testing	5
LaTeX	4

Table 7.3: Effort spent – Triuzzi Emanuele

Chapter 8

References

This chapter contains references to the tools and resources used during the writing of the document.

- *GitHub*

  `https://github.com/SirGian99/RiccioSorrentinoTriuzzi`

- *TomEE*

  `https://tomee.apache.org/`

- *IntelliJ IDEA*

  `https://www.jetbrains.com/idea`

- *Apache Maven*

  `https://maven.apache.org/guides/`

- *Apple Swift*

  `https://developer.apple.com/swift/`

- *Apple SwiftUI*

  `https://developer.apple.com/xcode/swiftui/`

- *SwiftyJSON*

  `https://github.com/SwiftyJSON/SwiftyJSON`

- *EclipseLink*

  `https://www.eclipse.org/eclipselink/`

- *JSON Java*

  `https://github.com/stleary/JSON-java`

- *JUnit*

  `https://junit.org/junit5/`

List of Tables

List of Figures