UNIVERSITÀ DI PISA

MSc in Computer Engineering
Information Systems
A.A. 2019/2020

Workgroup Task 1

Designing and Implementing a simple JAVA application
connecting to a relational DB using JPA
and
Feasibility Study on the use of a Key-Value Data Storage

Task Report

Chiara Bonsignori
Marco Del Gamba
Alessio Ercolani
Federico Garzelli

Contents

1 JPA Tutorial

1.1 Introduction

This tutorial explains how to use the Java Persistence API to handle one-to-many and many-to-many relationships in an object-oriented manner.

1.2 One-to-Many Relationship

In a one-to-many relationship, one record in a table can be associated with one or more records in another table. Let us consider an example in which a professor can hold many courses, while one course belongs to one, and only one, professor. In this case, `Professor` is the parent entity, whereas `Course` is the child entity.

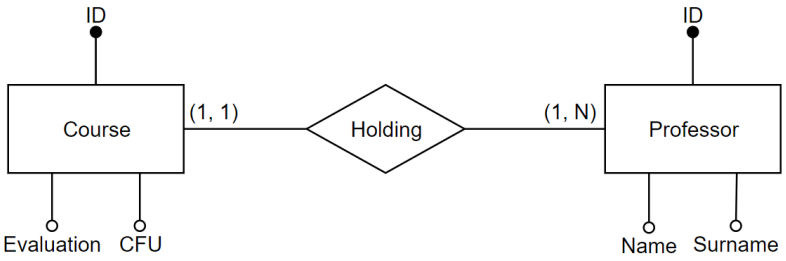

Figure 1.1: Example ER Diagram for One-to-Many Relationship

When it comes to JPA, the one-to-many database association can be represented either through a `@OneToMany` or a `@ManyToOne` annotation. In the first case, moreover, it can be either unidirectional or bidirectional.

One-to-many or many-to-one is a matter of perspective: the `@OneToMany` annotation is used when child entities, `Course` in our example, are mapped as a collection in the parent entity, `Professor`; instead, the `@ManyToOne` annotation consists in having a reference to an object of the parent entity inside the child entity. There are two ways to map the `@OneToMany` association: with a collection or through the `@JoinColumn` annotation.

Unidirectional and bidirectional associations, instead, do not affect the mapping but do make difference on how you can access your data. In a unidirectional association, `Professor` class will have collection of courses, but `Course` will not have an instance of `Professor`. In the bidirectional case, both references are added and this allows to access a professor given a course.

1.2.1 Unidirectional One-to-Many

In the following, the code of `Professor` and `Course` classes is presented. The `@OneToMany` association is mapped with the collection `courses` inside the parent entity `Professor`.

```java
@Entity(name = "Professor")
@Table(name = "professor")
public class Professor {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String surname;

    @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
    private List<Course> courses = new ArrayList<>();

    // constructors, getters and setters removed for brevity
}
```

```java
@Entity(name = "Course")
@Table(name = "course")
public class Course {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;

    private int CFU;
    private int evaluation;

    // constructors, getters and setters removed for brevity
}
```

The problem with this solution is that Hibernate uses a link table between the two entities, which is inefficient, since a third table uses more storage and is not necessary. In addition to that, instead of only one foreign key, we would have two of them, which means we were going to need twice cache than necessary during the indexing operations. Indeed, when adding a new course (`courses.add(new Course(5, 2));`), the `Professor` primary key has to match the corresponding column in the join table, then `courses_id` needs to be mapped into the primary key of the `Course` table. We are not going to present all the CRUD operations here, but we will treat them for the bidirectional case.

Other problems arise when using the other method to map the unidirectional `@OneToMany` relationship, i.e. the `@JoinColumn` annotation.

```java
// Professor class

    @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
    @JoinColumn(name = "professor_id")
    private List<Course> courses = new ArrayList<>();
```

The `@JoinColumn` annotation helps Hibernate to figure out that there is a `professor_id` foreign key column in the `Course` table that defines this association. However, if we insert,

for example, two new courses corresponding to an existing professor, Hibernate executes two insert and two update statements. This is because Hibernate first inserts the child records without the foreign key, because the child entity, `Course`, does not store this information. Afterwards the foreign key column is updated accordingly. Something similar happens during the deletion of a child entity, two statements are executed instead of one: the parent entity state change is executed first, which means that the `professor_id` field of the `Course` entity is set to `null`, then the orphan removal action will delete the child row.

1.2.2 Bidirectional One-to-Many

In the following, we are going to introduce the bidirectional `@OneToMany` relationship, which is the best way to map a one-to-many association, since it relies on a `@ManyToOne` side to propagate all entity state changes.

In the parent entity, i.e. the `Professor` class, we insert the collection `courses` with the `@OneToMany` annotation, which must also include the `mappedBy` attribute. In the child entity, i.e. the `Course` class, we insert a reference to a `Professor`, whose name must match the one specified with `mappedBy`, with the `ManyToOne` and `JoinColumn` annotations.

```
@Entity(name = "Professor")
@Table(name = "professor")
public class Professor {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String surname;

    @OneToMany(
        mappedBy = "professor",
        cascade = CascadeType.ALL, orphanRemoval = true
    )
    private List<Course> courses = new ArrayList<>();

    // constructors, getters and setters removed for brevity
}
```

```
@Entity(name = "Course")
@Table(name = "course")
public class Course {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;

    private int CFU;
    private int evaluation;

    @ManyToOne
    @JoinColumn(name = "professor_id")
    private Professor professor;

```

```
    // constructors, getters and setters removed for brevity
}
```

Implementation of CRUD operations

In the following, implementation of CRUD operations for `Professor` entity are reported.

```
public void createProfessor() {
    Professor professor = new Professor();
    professor.setName("John");
    professor.setSurname("Doe");
    try {
        entityManager = factory.createEntityManager();
        entityManager.getTransaction().begin();
        entityManager.persist(professor);
        entityManager.getTransaction().commit();
        System.out.println("Professor Added");
    } catch (Exception ex) {
        ex.printStackTrace();
    } finally {
        entityManager.close();
    }
}
```

```
public void readProfessor(long professorId) {
    try {
        entityManager = factory.createEntityManager();
        entityManager.getTransaction().begin();
        Professor prof = entityManager.find(Professor.class, professorId);
        System.out.println("Name: " + prof.getName());
        System.out.println("Surname: " + prof.getSurname());
        System.out.println("Professor retrieved");
        entityManager.getTransaction().commit();
    } catch (Exception ex) {
        ex.printStackTrace();
    } finally {
        entityManager.close();
    }
}
```

```
public void updateProfessor(long professorId) {
    try {
        entityManager = factory.createEntityManager();
        entityManager.getTransaction().begin();
        Professor prof = entityManager.find(Professor.class, professorId);
        prof.setName("Jane");
        prof.setSurname("Poe");
        entityManager.getTransaction().commit();
        System.out.println("Professor updated");
    } catch (Exception ex) {
        ex.printStackTrace();
    } finally {
        entityManager.close();
    }
}
```

```java
public void deleteProfessor(long professorId) {
    try {
        entityManager = factory.createEntityManager();
        entityManager.getTransaction().begin();
        Professor professor
                = entityManager.getReference(Professor.class, professorId);
        entityManager.remove(professor);
        entityManager.getTransaction().commit();
        System.out.println("Professor removed");
    } catch (Exception ex) {
        ex.printStackTrace();
    } finally {
        entityManager.close();
    }
}
```

The methods of the `Course` class are analogous to the ones just presented and are not reported for brevity.

In the following we show the operations executed by Hibernate when we add two courses to a professor:

```
Hibernate:

    create table course (
       id bigint not null auto_increment,
        CFU integer not null,
        evaluation integer not null,
        professor_id bigint,
        primary key (id)
    ) engine=InnoDB

Hibernate:

    create table professor (
       id bigint not null auto_increment,
        name varchar(255),
        surname varchar(255),
        primary key (id)
    ) engine=InnoDB

Hibernate:

    alter table course
       add constraint FKqctak3o6xmul2nu2561al3pb5
       foreign key (professor_id)
       references professor (id)

-- inserting a new professor
Hibernate:
    insert
    into
```

```
        professor
        (name, surname)
    values
        (?, ?)

-- inserting the new course
Hibernate:
    insert
    into
        course
        (CFU, evaluation, professor_id)
    values
        (?, ?, ?)

Hibernate:
    insert
    into
        course
        (CFU, evaluation, professor_id)
    values
        (?, ?, ?)
```

In this case, Hibernate generates just one SQL statement for each persisted `Course` entity. The rest of the CRUD operations are successful as well, the output of the console is not shown to keep the tutorial short.

1.2.3 Many-to-One

Bidirectional one-to-many associations are useful, especially when we want to join entities in a JPQL query, but the problem with collections is that we can only use them when the number of child records is rather limited. Therefore, using the `@ManyToOne` annotation on the child side without collections in the parent entity may be a simpler approach.

```java
@Entity(name = "Professor")
@Table(name = "professor")
public class Professor {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String surname;

    // constructors, getters and setters removed for brevity
}
```

```java
@Entity(name = "Course")
@Table(name = "course")
public class Course {

```

```
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;

    private int CFU;
    private int evaluation;

    // LAZY = fetch when needed
    // EAGER = fetch immediately
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "professor_id")
    private Professor professor;

   // constructors, getters and setters removed for brevity
}
```

However, we have to write a JPQL query to get the `Course` entities associated to a `Professor` entity:

```
public void readCourses(long professorId) {
    try {
        entityManager = factory.createEntityManager();
        entityManager.getTransaction().begin();
        List<Course> courses = entityManager.createQuery(
                "select c " +
                "from Course c " +
                "where c.professor.id = :professorId", Course.class)
            .setParameter("professorId", professorId)
            .getResultList();
        entityManager.getTransaction().commit();

        for (Course c : courses) {
            System.out.println("CFU: " + c.getCFU());
            System.out.println("Evaluation: " + c.getEvaluation());
        }
    } catch (Exception ex) {
        ex.printStackTrace();
    } finally {
        entityManager.close();
    }
}
```

It is also easy to add or delete courses entities when we want to. What's nice about using a query is that you can paginate it, so that, if the number of child entities grows with time, the application performance is not going to be affected.

A good practice is using `FetchType.LAZY`, which means a "fetch it when you need it", instead of `FetchType.EAGER`, which basically means "fetch it so you will have it when you need it". The latter is the default for to-one relationships and, in other words, tells Hibernate to get all elements of a relationship when selecting the root entity. For example, when we fetch a `Course` entity from the database, Hibernate will also get the related `Professor` entity, even if the code does not use it. In this case, we can declare that we want professors to be loaded when they are actually needed. This is called lazy loading. On the other hand, `FetchType.EAGER` may be useful for the one-to-many relationships if we know that all of our use cases that fetch a parent entity also need to process the related

child entities, since joining the required tuples and getting all of them in one query is very efficient. However, this is not the usual case.

1.3 Many-to-Many Relationship

In a many-to-many relationship, entities can be related to multiple instances of each other. Hence, a many-to-many association requires a link table that joins the two entities and stores their primary keys. Like the one-to-many, the many-to-many relationship can be either unidirectional or bidirectional: a unidirectional relationship has only an owning side, whereas a bidirectional relationship has also an inverse side. The owning side of a relationship controls the link table and determines how JPA makes updates to the relationships in the database.

The example used in this section consists of a college in which each course is attended by many students and students may take several courses (see ER diagram in Fig. 1.2). Therefore, in this case, `Course` and `Student` entities are related by a many-to-many relationship and `Enrollment` is the additional link table. In the following discussion, the `Student` entity is chosen as the owner of the relationship.

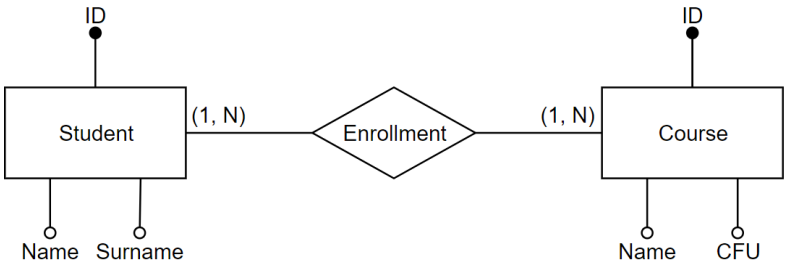

Figure 1.2: Example ER Diagram for Many-to-Many relationship

1.3.1 Unidirectional Many-to-Many

A unidirectional relationship is configured in the owner class. In our example, the `@ManyToMany` and `@JoinTable` annotations must be inserted in the implementation of the `Student` entity. The `@JoinTable` annotation specifies the name of the link table and the foreign keys: the `@JoinColumn` annotation indicates the owner side of the relationship and the `@InverseJoinColumn` refers to the other side. A collection of `Course` instances must also be inserted to store all the courses attended by a student. The class that does not own the relationship, instead, does not require any annotation.

```
@Entity(name = "Student")
@Table(name = "Student")
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long studentID;

    private String name;
    private String surname;

    @ManyToMany
    @JoinTable(name = "Enrollment",
```

```
                joinColumns = @JoinColumn(name = "studentID"),
                inverseJoinColumns = @JoinColumn(name = "courseID"))
    private List<Course> enrolledCourses = new ArrayList<>();

    // constructors and getters and setters removed for brevity
}
```

```
@Entity(name = "Course")
@Table(name = "Course")
class Course {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long courseID;

    private String name;
    private int cfu;

    // constructors and getters and setters removed for brevity
}
```

Implementation of CRUD operations

CREATE, READ and UPDATE operations are implemented in an analogous way to those of one-to-many relationships. Their implementations for a `Student` object are reported below as example.

```
public void createStudent(Student student) {
    System.out.println("Creating a new student");
    try {
        entityManager = factory.createEntityManager();
        entityManager.getTransaction().begin();
        entityManager.persist(student);
        entityManager.getTransaction().commit();
        System.out.println("Student added");
    } catch (Exception ex) {
        ex.printStackTrace();
        System.out.println("A problem occurred in updating a student!");
    } finally {
        entityManager.close();
    }
}

public void readStudent(long studentID) {
    System.out.println("Getting a student");
    try {
        entityManager = factory.createEntityManager();
        entityManager.getTransaction().begin();
        Student student = entityManager.find(Student.class, studentID);

        System.out.println("Name: " + student.getName());
        System.out.println("Surname: " + student.getSurname());
        for(int i = 0; i < student.getEnrolledCourses().size(); i++)
            String course = student.getEnrolledCourses().get(i).getName();
            System.out.println("Course " + (i + 1) + course);

        entityManager.getTransaction().commit();
```

```
31          System.out.println("Student retrieved");
32      } catch (Exception ex) {
33          ex.printStackTrace();
34          System.out.println("A problem occurred in retrieving a student!");
35      } finally {
36          entityManager.close();
37      }
38  }
39
40  public void updateStudent(long studentId, Student newStudent) {
41      System.out.println("Updating a student");
42      try {
43          entityManager = factory.createEntityManager();
44          entityManager.getTransaction().begin();
45          Student student = entityManager.find(Student.class, studentId);
46          student.setName(newStudent.getName());
47          student.setSurname(newStudent.getSurname());
48          entityManager.getTransaction().commit();
49          System.out.println("Student updated");
50      } catch (Exception ex) {
51          ex.printStackTrace();
52          System.out.println("A problem occurred in updating a student!");
53      } finally {
54          entityManager.close();
55      }
56  }
```

As shown in the following snapshot, when a new course is created, only the `Course` table is updated, whereas when a new student with one or more courses is created, both `Student` and `Enrollment` tables are modified. Indeed, a new entry is added to the `Enrollment` table for each course in the list of `enrolledCourses` field of the student (in the example the student is attending one course).

```
-- creating a new course
Hibernate:
    insert
    into
        Course
        (cfu, name)
    values
        (?, ?)

-- creating a new student
Hibernate:
    insert
    into
        Student
        (name, surname)
    values
        (?, ?)
Hibernate:
    insert
    into
        Enrollment
```

```
        (studentID, courseID)
    values
        (?, ?)
```

The DELETE operation instead has different implementations, depending on whether the object to delete is the owner of the relationship or not.

For the owner entity of the relationship, `Student` in the example, the DELETE operation is analogous to the one reported for the one-to-many case. When a student is removed, the associated link records in the `Enrollment` table are safely removed too, as can be seen in the snapshot below, where is reported the translation of the operation in SQL code.

```
public void deleteStudent(long studentId) {
    try {
        entityManager = factory.createEntityManager();
        entityManager.getTransaction().begin();
        Student student;
        student = entityManager.getReference(Student.class, studentId);
        entityManager.remove(student);
        entityManager.getTransaction().commit();
    } catch (Exception ex) {
        ex.printStackTrace();
    } finally {
        entityManager.close();
    }
}
```

```
-- deleting a student
Hibernate:
    select
        student0_.studentID as studentI1_2_0_,
        student0_.name as name2_2_0_,
        student0_.surname as surname3_2_0_
    from
        Student student0_
    where
        student0_.studentID=?
Hibernate:
    delete
    from
        Enrollment
    where
        studentID=?
Hibernate:
    delete
    from
        Student
    where
        studentID=?
```

Deleting an instance of the non-owner entity of the relationship, in this example a `Course` object, is instead not possible. Indeed, there is nothing in the `Course` class that links a course to all the students that are enrolled to it and hence the deletion of a course does not cause the safe removal of the related items in the `Enrollment` table and therefore ends in an error.

It is instead possible to remove an enrollment of a student to a certain course. This operation is implemented with the removal of the course from the list `enrolledCourses` of the student. When translated to SQL code, it corresponds to the deletion of all the entries in `Enrollement` table associated with the student and the successive re-insertion of all records corresponding to other courses attended by the student. Therefore, the deletion of an enrollment of a student is very inefficient.

```java
public void deleteEnrollement(Course course, long studentId) {
    try {
        entityManager = factory.createEntityManager();
        entityManager.getTransaction().begin();
        Student student = entityManager.find(Student.class, studentId);
        student.getEnrolledCourses().remove((Object)course);
        entityManager.getTransaction().commit();
    } catch (Exception ex) {
        ex.printStackTrace();
    } finally {
        entityManager.close();
    }
}
```

```
-- deleting an enrollment
Hibernate:
    select
        student0_.studentID as studentI1_2_0_,
        student0_.name as name2_2_0_,
        student0_.surname as surname3_2_0_
    from
        Student student0_
    where
        student0_.studentID=?
Hibernate:
    select
        enrolledco0_.studentID as studentI1_1_0_,
        enrolledco0_.courseID as courseID2_1_0_,
        course1_.courseID as courseID1_0_1_,
        course1_.cfu as cfu2_0_1_,
        course1_.name as name3_0_1_
    from
        Enrollment enrolledco0_
    inner join
        Course course1_
            on enrolledco0_.courseID=course1_.courseID
    where
        enrolledco0_.studentID=?
```

```
Hibernate:
    delete
    from
        Enrollment
    where
        studentID=?
Hibernate:
    insert
    into
        Enrollment
        (studentID, courseID)
    values
        (?, ?)
```

1.3.2 Bidirectional Many-to-Many

A bidirectional many-to-many relationship must be configured in the implementation of both entities.

The implementation of the `Student` class is equal to the one of the unidirectional case, whereas the `Course` class requires the `@ManyToMany` annotation with the specification of the `mappedBy` attribute and a collection of students attending to a course.

```java
@Entity(name = "Course")
@Table(name = "Course")
public class Course {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long courseID;

    private String name;
    private int cfu;

    @ManyToMany(mappedBy = "enrolledCourses")
    private List<Student> enrollements = new ArrayList<>();

    // constructors and getters and setters removed for brevity
}
```

Implementation of CRUD operations

CREATE, READ and UPDATE operations are implemented in the same way of unidirectional case.

The DELETE operation requires, instead, some additional considerations. As in the unidirectional case, the deletion of an object of the owner class `Student` causes automatically the deletion of the related items in the `Enrollment` link table. In a many-to-many relationship, the ownership is determined by the entity which doesn't contains the `mappedBy` attribute (there is no chance for both sides to be owners).

In the bidirectional case, it is possible to delete an instance of the non-owner class, in this case `Course`. However, records in the link table must be explicitly deleted, as shown in the implementation reported below. This operation is here possible since each `Course` object contains the list of students that are enrolled to it.

```
public void deleteCourse(long courseId) {
    try {
        entityManager = factory.createEntityManager();
        entityManager.getTransaction().begin();
        Course course = entityManager.find(Course.class, courseId);
        entityManager.remove(course);
        for (Student student : course.getEnrollements())
            student.getEnrolledCourses().remove(course);
        entityManager.getTransaction().commit();
    } catch (Exception ex) {
        ex.printStackTrace();
    } finally {
        entityManager.close();
    }
}
```

2 Design and Implementation of a simple JAVA application connecting to a relational DB using JPA

2.1 Design

2.1.1 Informal Requirements

The application manages room reservations for an hotel chain with different branches. A typical scenario consists of a customer calling the hotel phone number in order to book a room with a certain capacity. If he is not yet registered, the receptionist needs to create a new profile for the client. Subsequently, the receptionist checks if there are available rooms during the period the client wants to spend in the hotel. Once the list of bookable rooms is displayed, the receptionist is able to add a new reservation. In addition to that, if the customer demands for it, the receptionist can delete a reservation or update it changing the hotel, the customer, the check-in or the check-out date. When the customer arrives at the hotel, the receptionist must check his reservation in order to give him the key of the assigned room.

Rooms have different capacities and may be unavailable due to maintenance. The receptionist can visualize the list of branches of the hotel chain and the list of rooms and reservations for each branch. Receptionist can also see which rooms are already booked or unavailable and is in charge to set a room as unavailable when some maintenance is needed and to set it again available when restructuring ends.

From his own device, the customer can visualize his upcoming reservations, the branches of the hotel chain and the bookable rooms in an hotel for a date interval, without the intervention of a receptionist. He can also modify the default password inserted by the receptionist during the registration phase. Both customer and receptionist must log-in with their credentials before performing any operation in the application. Once they have completed their actions, they can log out from the system.

2.1.2 Actors and Functionalities

The application has two actors: customer and receptionist.

Functional Requirements

A customer shall be able to:

- login and logout in the application,
- visualize his upcoming reservations,
- visualize the list of branches of the hotel chain,

- visualize the bookable rooms of a specific hotel for the desired length of stay,
- change the password.

A receptionist shall be able to:

- login and logout in the application,
- visualize the list of branches of the hotel chain,
- visualize all the rooms of a hotel,
- visualize the bookable rooms in a hotel for a given date interval,
- visualize the rooms that cannot be booked in a hotel in a given date interval,
- visualize all the reservations for a given hotel booked from a certain date on
- add a reservation,
- update a reservation by modifying one or more fields among hotel, room, check-in date, check-out date and customer,
- delete a reservation,
- set a room as available or not available,
- register a new customer,
- visualize the list of customers of the hotel chain,
- perform check-in, i.e. see the reservation of a customer when he arrives at the hotel,
- perform check-out, i.e. see customer's full name and room when he leaves the hotel.

Non-Functional Requirements

Availability The system shall be highly available during the hotel operating hours:

- receptionists shall be able to access data regarding rooms and reservations in any moment in order to satisfy the incoming requests of the customers,
- clients shall be able to access to their reservations whenever they want.

Consistency Consistency is an important requirement for the system: each receptionist must be ensured to see a consistent view of the data in order to avoid overlapping reservations.

Scalability Unlimited users should be allowed: many users can be connected to the application at the same time and the system must be able to face with high number of requests.

Security Customers and receptionists shall be able to log in to the system. Receptionists, in particular, should have access to additional functionalities. Access to all the functionalities must be protected by the user log in screen that requires username and password.

Performance To achieve quite good performance the system shall respect the following requirements:

- the load time for user interface screens shall take no longer than three seconds,
- the log in information shall be verified in no longer than five seconds,
- commands shall return results in less than five seconds.

Portability The system shall be run on every Windows, Linux or MacOS platform having Java Runtime Environment 1.8 and MySQL server.

2.1.3 Use Cases

Figure 2.1 shows the UML use case diagram of the hotel chain application, which represents receptionist's and customer's interaction with the system.

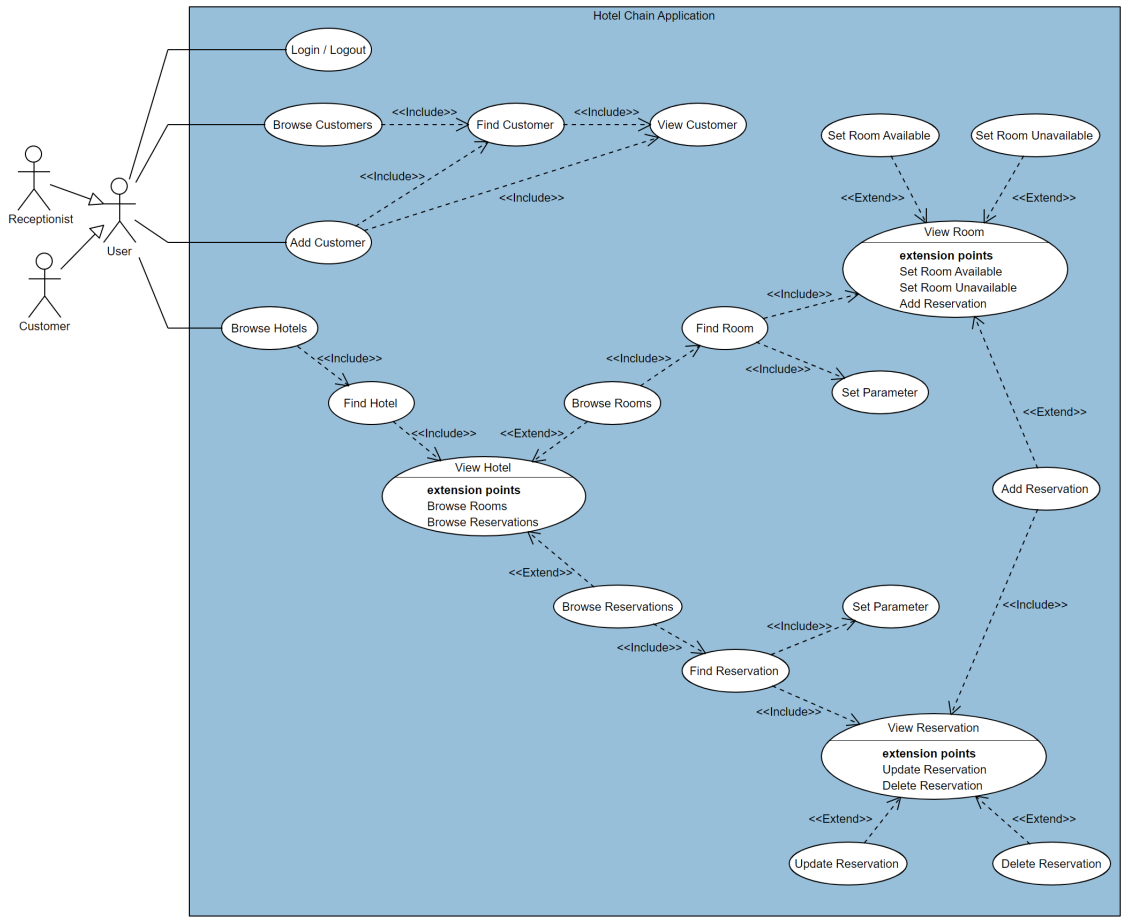

Figure 2.1: Use Case Diagram

2.1.4 Diagram of Analysis Classes

Figure 2.2 shows the UML diagram of analysis classes of the hotel chain application, which highlights the main components of the system and their relationships.

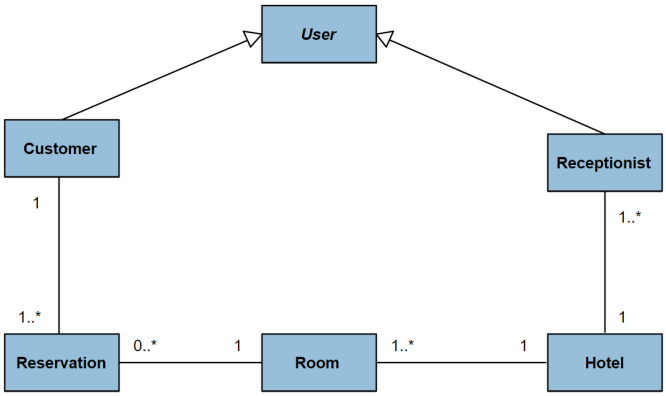

Figure 2.2: Diagram of Analysis Classes

2.1.5 Software Architecture

The client-side part of the system is only composed by the front-end that contains the methods to run the terminal interface.

The server side layer is instead divided into two sub-layers: the middleware, which contains all the modules for the business logic and for the interaction with the database, and back-end, which contains the MySQL database.

A schematic representation of such architecture is shown in Figure 2.3.

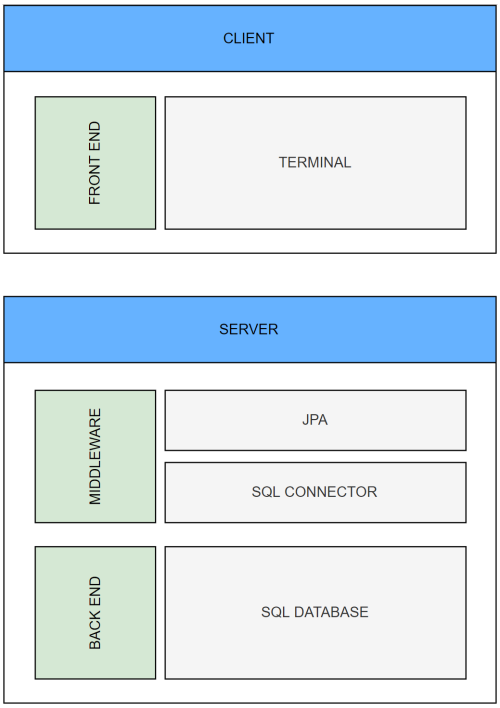

Figure 2.3: Software Architecture

2.2 Implementation

2.2.1 Class Diagram

Figure 2.4 shows the UML class diagram. After that, a description of classes responsibilities is provided.

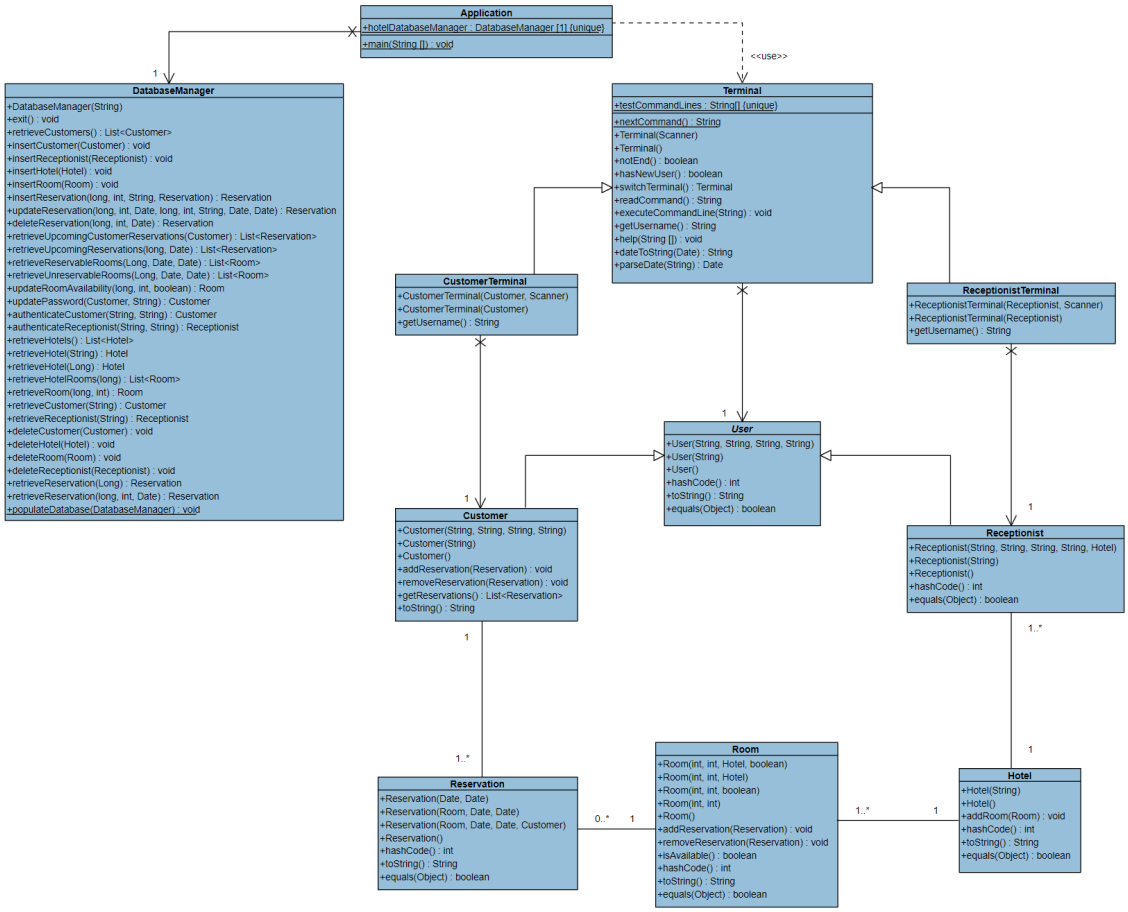

Figure 2.4: Class Diagram

Terminal, Receptionist Terminal, Customer Terminal Class `Terminal` manages the interaction with the user. It receives user's input from keyboard, checks its syntax and invokes commands. Moreover, it shows the user result messages. In particular, `Terminal` executes the commands of a non-logged user. `ReceptionistTerminal` and `CustomerTerminal` extend `Terminal`. Each of them defines and implements either the receptionist's or customer's set of commands. Common functionalities are implemented by `Terminal` class.

User, Receptionist, Customer Class `User` represents the model of an actor of the application. A `User` can be either a `Receptionist` or a `Customer`, which are JPA entities.

Hotel Class `Hotel` represents the model of a hotel. It is a JPA entity. `Hotel` objects are instantiated and provided to the other classes by the `DatabaseManager`.

Room Class `Room` represents the model of a room in a hotel. It is a JPA entity. `Room` objects are instantiated and provided to the other classes by the `DatabaseManager`.

Reservation Class `Reservation` represents the model of the reservation of a room in a hotel during a certain period of time. It is a JPA entity. `Reservation` objects are instantiated by `Terminal` or by `DatabaseManager` respectively with user's input or database information.

DatabaseManager Class `DatabaseManager` manages the interaction with the database. It opens and closes connections and performs queries to provide results to the other classes.

Application Class `Application` instantiates a `DatabaseManager` and creates the `Terminal` needed to interact with the application.

2.2.2 ER Diagram and Tables

The ER diagram of the database is shown in Figure 2.4. Table 2.1 shows the tables and their columns.

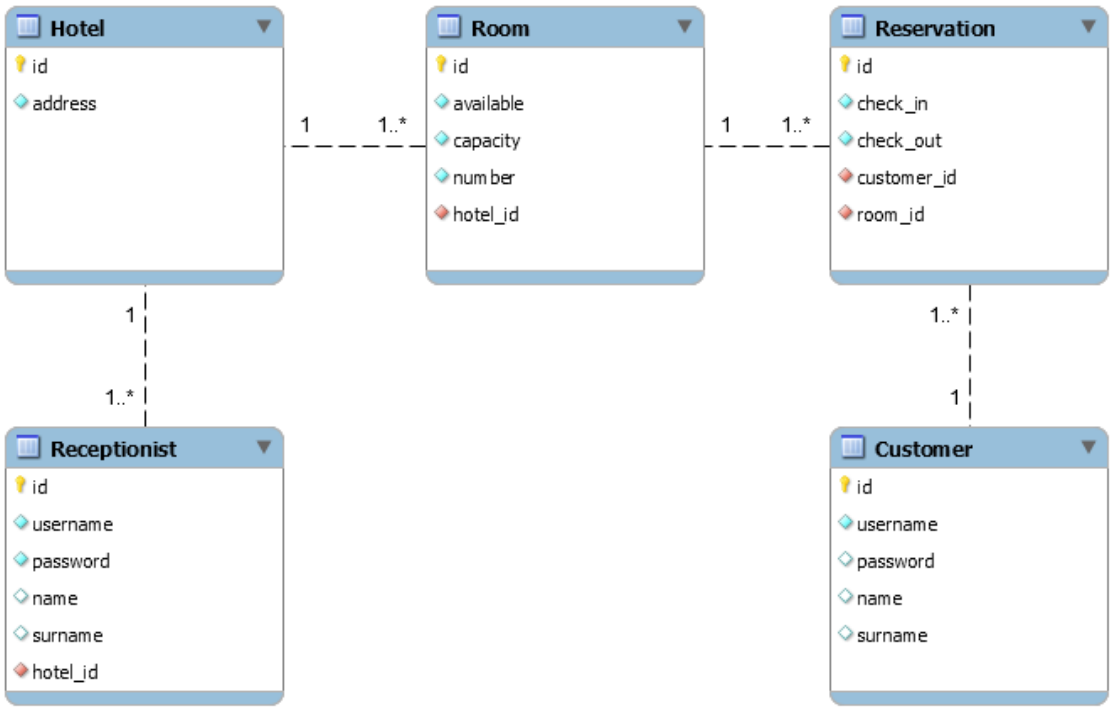

Figure 2.5: ER Diagram

Tables				
Hotel	**Customer**	**Receptionist**	**Room**	**Reservation**
Id	Id	Id	Id	Id
Address	Username	Username	Available	Check-In
	Password	Password	Capacity	Check-Out
	Name	Name	Number	Customer Id
	Surname	Surname	Hotel Id	Room Id
		Hotel Id		

Table 2.1: List of tables

2.2.3 JPA Relationships

In this section we look in detail at the implementation of JPA entities and their relationships.

Customers and Receptionists are particular types of Users. `User` class must contain the `@MappedSuperclass` annotation, as well as the fields that `Customer` and `Receptionist` share, i.e. name, surname, username and password. `Customer` and `Receptionist` classes must extend `User`. This class is shown below. To mark the username as unique we can use the `@Column(unique = true)` annotation.

```
@MappedSuperclass
public abstract class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(unique = true)
    private String username;
    private String password;
    private String name;
    private String surname;

    // constructors, getters and setters omitted for brevity
}
```

There is a one-to-many relationship between `Receptionist` and `Hotel`. To map this relationship, we insert a reference to a `Hotel` with `@ManyToOne` and `@JoinColumn` annotations in the `Receptionist` entity. This class is shown below.

```
@Entity(name = "Receptionist")
@Table(name = "receptionist")
// name queries omitted for brevity
public class Receptionist extends User {
    @ManyToOne
    @JoinColumn(name = "hotel_id", referencedColumnName = "id")
    private Hotel hotel;

    //  constructors, getters and setters omitted for brevity
}
```

On the other hand, in the `Hotel` class, we must insert a collection of Receptionists and the `@OneToMany` annotation. The value of the `mappedBy` attribute represents the name of the reference in the `Receptionist` class. The `Hotel` class is shown below.

```
@Entity(name = "Hotel")
@Table(name = "hotel")
// name queries omitted for brevity
public class Hotel {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private Long id;

    @Column(unique = true)
    private String address;

    @OneToMany(mappedBy = "hotel", cascade = CascadeType.ALL,
               orphanRemoval = true)
    private List<Room> rooms = new ArrayList<>();

    @OneToMany(mappedBy = "hotel", cascade = CascadeType.ALL,
               orphanRemoval = true)
    private List<Receptionist> receptionists = new ArrayList<Receptionist>();

     // constructors, getters, setters and other methods omitted for brevity
}
```

There is a one-to-many relationship between `Hotel` and `Room`. The strategy adopted to map this relationship in the same as before. There is a collection of Rooms in the `Hotel` class with a `@OneToMany` relationship, as shown above. In the `Room` class, we insert a reference to a `Hotel` with `@ManyToOne` and `@JoinColumn` annotations. The `Room` class is shown below. To mark the couple `hotel_id-number` as unique, we need a unique constraint, which is added with a `@UniqueConstraint` annotation.

```
@Entity(name = "Room")
@Table(name = "room",
       uniqueConstraints =  @UniqueConstraint(
          name = "uk_hotel_number",
          columnNames = {
              "hotel_id",
              "number"
          }
       ))
// name queries omitted for brevity
public class Room {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne
    @JoinColumn(name = "hotel_id", referencedColumnName = "id")
    private Hotel hotel;

    @Column(name = "number")
    private int number;

    @Column(name = "capacity")
```

```
    private int capacity;

    @OneToMany(mappedBy = "room", cascade = CascadeType.ALL,
                orphanRemoval = true)
    List<Reservation> reservations = new ArrayList<Reservation>();

    private boolean available;

     // constructors, setters, getters and other methods omitted for brevity
}
```

A `Reservation` involves one `Room` and one `Customer`, therefore we insert a reference to each of them, with `@ManyToOne` and `@JoinColumn` annotations. `Room` and `Customer` classes have hence a collection of Reservations. The code of `Reservation` and `Customer` classes is shown below.

```
@Entity(name = "Reservation")
@Table(name = "reservation",
       uniqueConstraints =  @UniqueConstraint(
          name = "uk_room_checkIn",
          columnNames = {
                "room_id",
                "check_in"
          }
       ))
// named queries omitted for brevity
public class Reservation {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne
    @JoinColumn(name = "room_id", referencedColumnName = "id")
    private Room room;

    @ManyToOne
    @JoinColumn(name = "customer_id", referencedColumnName = "id")
    private Customer customer;

    @Column(name = "check_in")
    @Temporal(TemporalType.DATE)
    private Date checkInDate;

    @Column(name = "check_out")
    @Temporal(TemporalType.DATE)
    private Date checkOutDate;

     // constructors, getters and setters omitted for bevity
}
```

```
@Entity(name = "Customer")
@Table(name = "customer")
// named queries omitted for brevity
public class Customer extends User {
    @OneToMany(mappedBy = "customer", cascade = CascadeType.ALL,
                orphanRemoval = true)
    private List<Reservation> reservations = new ArrayList<>();
```

```
8
9       // constructors, getters, setters and other methods omitted for brevity
10  }
```

2.2.4 CRUD Operations

This section describes the implementation of CRUD operations on the entities of the application.

CREATE Entities are added to the database using the `persist()` method of `EntityManager` interface from Java Persistence API. We report, as an example, the code of the method that creates a new `Room` and adds it to the database. Create methods for the other entities are analogous.

```
1   private void persistObject(Object obj) {
2       entityManager.persist(obj);
3   }
4
5   public void insertRoom(Room room) throws RoomAlreadyPresentException,
6                                            DatabaseManagerException {
7       try {
8           beginTransaction();
9           persistObject(room);
10      } catch (PersistenceException e) {
11          throw new RoomAlreadyPresentException(e.getMessage());
12      } catch (Exception e) {
13          throw new DatabaseManagerException(e.getMessage());
14      } finally {
15          commitTransaction();
16          close();
17      }
18  }
```

READ If we want to retrieve an object using its key, e.g. getting an object from its id, we use the `find()` method of `EntityManager`. Instead, we use named queries to retrieve an object starting from some of its fields. As an example we show, in the following, the methods that retrieve an `Hotel` starting from its id and from its address. We also report the named query used in the second method.

```
1   // class Hotel.java
2
3   @Entity(name = "Hotel")
4   @Table(name = "hotel")
5   @NamedQuery(name = "Hotel.findByAddress",
6               query = "SELECT h FROM Hotel h WHERE h.address = :address")
7   public class Hotel {
8       // ...
9   }
10
11
12  // class DatabaseManager.java
13
14  public Hotel retrieveHotel(Long id) throws HotelNotFoundException,
15                                             DatabaseManagerException {
16      Hotel hotel = null;
```

```
    try {
        beginTransaction();
        hotel = entityManager.find(Hotel.class, id);
        if (hotel == null)
            throw new HotelNotFoundException(id.toString());
        return hotel;
    } catch (HotelNotFoundException e) {
        throw e;
    } catch (Exception e) {
        throw new DatabaseManagerException(e.getMessage());
    } finally {
        commitTransaction();
        close();
    }
}

public Hotel retrieveHotel(String address) throws HotelNotFoundException,
                                                  DatabaseManagerException {
    Hotel hotel = null;
    try {
        beginTransaction();
        hotel = entityManager
                    .createNamedQuery("Hotel.findByAddress", Hotel.class)
                    .setParameter("address", address)
                    .getSingleResult();
        return hotel;
    } catch (NoResultException nr) {
        throw new HotelNotFoundException();
    } catch (Exception ex) {
        throw new DatabaseManagerException(ex.getMessage());
    } finally {
        commitTransaction();
        close();
    }
}
```

UPDATE To update an entity we must first retrieve it from the database (using the `find()` method of `EntityManager` or a named query) and then we can update its fields using the defined setter methods. When the transaction is committed updates are saved in the database. As an example we report the method that updates a `Customer`'s password and the corresponding setter method.

```
// class User.java

@MappedSuperclass
public abstract class User {
    // ...

    public void setPassword(String password) {
        this.password = password;
    }

    // ...
}


// class DatabaseManager.java

```

```
public Customer updatePassword(Customer customer, String newPassword)
                                        throws DatabaseManagerException {
    try {
        beginTransaction();
        Customer ref = entityManager.find(Customer.class, customer.getId());
        ref.setPassword(newPassword);
        return ref;
    } catch (Exception ex) {
        throw new DatabaseManagerException(ex.getMessage());
    } finally {
        commitTransaction();
        close();
    }
}
```

DELETE To delete an object we must first retrieve if from the database (using the `find()` method of `EntityManager` or a named query) and then we can call the `remove()` method of `EntityManager`. The example of the operation that deletes a reservation is shown below.

```
// class Reservation.java

@Entity(name = "Reservation")
@Table(name = "reservation")
@NamedQuery(
        name="Reservation.getByHotelAndRoomAndCheckInDate",
        query="SELECT r FROM Reservation r WHERE r.room.hotel.id = :hotelId
                AND r.room.number = :roomNumber
                AND r.checkInDate = :checkInDate")
public class Reservation {
    // ...
}


// class DatabaseManager.java

public Reservation deleteReservation(long hotelId, int roomNumber,
                    Date checkIn) throws DatabaseManagerException,
                                         ReservationNotFoundException {
    try {
        Reservation reservation;
        beginTransaction();
        try {
            reservation = entityManager
                    .createNamedQuery(
                        "Reservation.getByHotelAndRoomAndCheckInDate",
                        Reservation.class)
                    .setParameter("hotelId", hotelId)
                    .setParameter("roomNumber", roomNumber)
                    .setParameter("checkInDate", checkIn, TemporalType.DATE)
                    .getSingleResult();
        } catch (NoResultException nr) {
            throw new ReservationNotFoundException();
        }

        entityManager.remove(reservation);

        // ...

```

```
40          return reservation;
41      } catch (ReservationNotFoundException e) {
42          throw e;
43      } catch (Exception e) {
44          throw new DatabaseManagerException(e.getMessage());
45      } finally {
46          commitTransaction();
47          close();
48      }
49  }
```

2.2.5 Validation and Test

The system is tested using JUnit. The following functionalities are tested:

- the insertion of a new customer,
- the insertion, the update and the deletion of a new reservation,
- the insertion of a new hotel and a new room,
- the insertion of a new receptionist,
- the read of all bookable and non-bookable rooms in a hotel and the modification of the availability of a room.

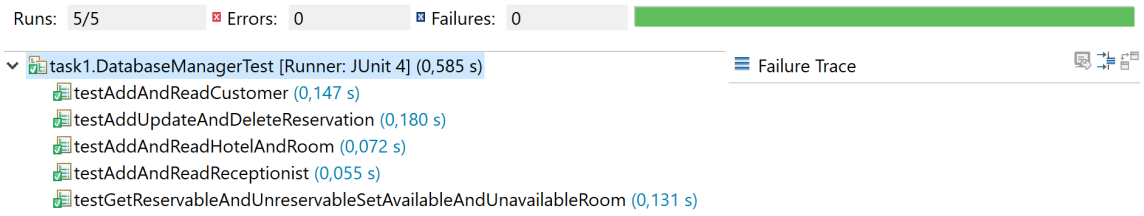

Figure 2.6: JUnit Test

2.3 Manual of Usage

Users interact with the application through a command line interface. Each type of user, namely Receptionist and Customer, has its own set of commands. Each command has a certain syntax, which specifies some options. Options may be inserted in two forms: if only one character is typed, then they have to be preceded by a single dash (`-`); if the user instead wants to type the option's full name, the latter must be preceded by two dashes (`--`). These two forms are completely equivalent. The syntax of a command also specifies whether an option requires an argument or not.

Users have to login to use the application. Before that, they can also type three commands, which are explained in the following. In case of errors, a message is displayed. Errors may be for example syntax errors or the insertion of inconsistent values.

Login Command `login` authenticates a user. The user has to type either the option `-c` or the option `-r`, to login respectively as Customer or Receptionist. She has also to type the username and the password, with the options `-u` and `-p`. For example: `login --customer -u john -p pwd`. Once the user has logged in, she can use the commands described in the following subsections.

```
usage: login -c | -r -u <arg> -p <arg>
 -c,--customer         login as a customer
 -r,--receptionist     login as a receptionist
 -u,--username <arg>   username
 -p,--password <arg>   password
```

Help Command `help` has no options, but interprets its arguments as command names. For each of them, it shows the syntax of the corresponding command. If no argument is specified, it shows the list of possible commands. All types users can use `help`, but each one will see only its set of commands. Examples: `help`, `help login`, `help exit login`.

Exit Command `exit` has no options. It closes the application. Usage: `exit`.

2.3.1 Receptionist Commands

Showing Hotels Command `show-hotels` shows the list of branches of the hotel chain. It has no options. Usage: `show-hotels`.

Showing Rooms Command `show-rooms` shows a list of rooms according to the parameters specified in the options. None of the option is mandatory. Only one option among `-b`, `-n` and `-a` can be specified to select bookable, non bookable or all rooms. By default, bookable rooms are showed. The option `--hotel` requires a hotel identifier as argument and can be used to specify a hotel. By default, the system show rooms of the hotel in which the Receptionist is working. Options `--from` and `--to` can be used to specify the interval of dates during which the rooms are required to be bookable or non bookable. Note that if the Receptionist chooses the option `--all` these two options are ignored. Dates must be inserted in format `yyyy-mm-dd`. By default, the `--from` value is the current date and the `--to` value is equal to the day after. Examples: `show-rooms --from 2019-11-15 --to 2019-11-17`, `show-rooms -a --hotel 3`. Command `show-rooms` without options shows rooms of the hotel in which the Receptionist is working that are bookable today.

```
usage: show-rooms [-b | -n | -a] [-h <arg>] [-f <arg>] [-t <arg>]
 -b,--bookable      show rooms that can be booked (default)
 -n,--notbookable   show rooms that cannot be booked
 -a,--all           show all rooms
 -h,--hotel <arg>   hotel identifier
 -f,--from <arg>    check-in date (format: yyyy-mm-dd) (default: today)
 -t,--to <arg>      check-out date (if not specified is equal to
                    the day after the check-in date)
```

Adding Reservations Command `add-reservation` inserts a new reservation in the database. All of its options require an argument. The Receptionist has to specify an hotel identifier with the `-h` option, a room number with the `-r` option, the username of an existing customer with the `-c` option and a check-in date with the `-f` option. The check-out date can be specified with the `-t` option, but this is not mandatory. If it is not specified, it is considered equal to the day after the check-in date. Dates must be inserted in format `yyyy-mm-dd`. For example: `add-reservation -h 3 -r 201 -c john -f 2019-12-01 -t 2019-12-03`.

```
usage: add-reservation -h <arg> -r <arg> -c <arg> -f <arg> [-t <arg>]
 -h,--hotel <arg>      hotel identifier
 -r,--room <arg>       room number
 -c,--customer <arg>   customer's username
 -f,--from <arg>       check-in date
 -t,--to <arg>         check-out date (if not specified is equal to
                       the day after the check-in date)
```

Showing Reservations Command `show-reservations` shows the reservations for a given hotel. Its options are optional. The hotel can be specified with the `-h` option, which requires as argument an hotel identifier. By default, the system shows reservations for the hotel in which the Receptionist is working. A minimum date for the check-in date may specified, in format `yyyy-mm-dd`, with the `-f` option. By default, this value corresponds to the current date. Examples: `show-reservations`, `show-reservations --hotel 2`.

```
usage: show-reservations [-f <arg>] [-h <arg>]
 -f,--from <arg>    the minimun date (yyyy-mm-dd) for the check-in field
 -h,--hotel <arg>   hotel identifier
```

Updating Reservations Command `update-reservation` modifies a reservation according to the parameters specified in the options. All of this command's options require an argument. In order to identify the reservation to update, the Receptionist must specify a hotel identifier with the `--currenthotel` option, a room number with the `--currentroom` option and a check-in date with the `--currentcheckin` option. Note that these are the only options in the application that do not have a short form. The other options are not mandatory, but the Receptionist must to specify at least one of them. She can specify a new hotel identifier with the `-h` option, a new room number with the `-r` option, a new customer's username with the `-c` option, a new check-in date with the `-f` option, a new check-out date with the `-t` option. Dates, as usual, must be specified in format `yyyy-mm-dd`. If one of these non-mandatory options is omitted, the corresponding value of the reservation remains unchanged. Examples: `update-reservation --currenthotel 3 --currentroom 401 --currentcheckin 2019-11-15 -r 201` changes the room number

from the 401 to the 201; `update-reservation --currenthotel 3 --currentroom 401 --currentcheckin 2019-11-15 -c john -t 2019-11-20` changes the customer and the checkout date.

```
usage: update-reservation --currenthotel <arg> --currentroom <arg>
       --currentcheckin <arg> [-h <arg>] [-r <arg>] [-c <arg>] [-f <arg>]
       [-t <arg>]
    --currenthotel <arg>     current hotel identifier
    --currentroom <arg>      current room number
    --currentcheckin <arg>   current check-in date
 -h,--hotel <arg>            new hotel identifier
 -r,--room <arg>             new room number
 -c,--customer <arg>         new customer's username
 -t,--to <arg>               new check-out date
 -f,--from <arg>             new check-in date
```

Deleting Reservations Command `delete-reservation` deletes a reservation from the database. The Receptionist must specify three options and their arguments. The hotel identifier is specified with the `-h` option, the room number is specified with the `-r` option, the check-in date is specified with the `-d` option, in format `yyyy-mm-dd`. For example: `delete-reservation -h 3 -r 401 -d 2019-11-15`.

```
usage: delete-reservation -h <arg> -r <arg> -d <arg>
 -h,--hotel <arg>   hotel identifier
 -r,--room <arg>    room number
 -d,--date <arg>    check-in date
```

Setting Rooms as Available or Not Available Command `set-room` is used to mark a room as available or not available. The Receptionist must specify either the `-a` option or the `-n` option, to set the room respectively as available or not. The Receptionist has also to specify an hotel identifier with the `-h` option and a room number with the `-r` option. For example: `set-room --notavailable --hotel 3 --room 401`.

```
usage: set-room -a | -n -h <arg>  -r <arg>
 -a,--available      set the room as available
 -n,--notavailable   set the room as not available
 -h,--hotel <arg>    hotel identifier
 -r,--room <arg>     room number
```

Registering Customers Command `register` adds a customer to the database. It has three mandatory options and all of them require an argument. The Receptionist must specify the customer's name with the `-n` option, the customer's surname with the `-s` option and the customer's username with the `-u` option. The customer's password will be set to the default value `pwd`. For example: `register -n John -s Doe -u johnny`.

```
usage: register -n <arg> -s <arg> -u <arg>
 -n,--name <arg>       customer's name
 -s,--surname <arg>    customer's surname
 -u,--username <arg>   customer's username
```

Showing Customers Command `show-customers` shows the list of customers of the hotel chain, ordered by surname. It has no options. Usage: `show-customers`.

Check-In Command `check-in` shows the essential information the Receptionist needs to receive and accommodate customers when they arrive at the hotel, namely the customer's full name and the number of the assigned room. The Receptionist must specify the identifier of the reservation with the `-i` option. For example: `check-in --id 123`.

```
usage: check-in -i <arg>
 -i,--id <arg>   reservation identifier
```

Check-Out Command `check-out` shows the customer's full name and the number of the assigned room. The Receptionist must specify the identifier of the reservation with the `-i` option. For example: `check-out --id 123`.

```
usage: check-out -i <arg>
 -i,--id <arg>   reservation identifier
```

Help Command `help`, as previously explained, is the same for every type of user. See above for details.

Logout Command `logout` terminates the session of the current Receptionist. It has no options nor arguments. After the logout, only non-logged-user commands can be typed. Usage: `logout`.

2.3.2 Customer Commands

Showing Reservations Command `show-reservations` shows the upcoming reservations of the Customer, which means the ones with a check-in date that is greater than or equal to the current date. It has no options. Usage: `show-reservations`.

Showing Hotels Command `show-hotels` shows the list of branches of the hotel chain. It has no options. Usage: `show-hotels`.

Showing Rooms Command `show-rooms` shows a list of bookable rooms according to the parameters specified in the options. The `--hotel` option is mandatory and requires as argument an hotel identifier. The Customer may also specify a check-in date with the `-f` option and a check-out date with the `-t` option, both in format `yyyy-mm-dd`. The default check-in date is the current date, while the default check-out date is equal to the day after the check-in. Examples: `show-rooms --hotel 3`, `show-rooms -h 2 -f 2019-12-20`.

```
usage: show-rooms -h <arg> [-f <arg>] [-t <arg>]
 -h,--hotel <arg>   hotel identifier
 -f,--from <arg>    check-in date (format: yyyy-mm-dd) (default: today)
 -t,--to <arg>      check-out date (if not specified is equal to the
                    check-in date)
```

Changing Password Command `change-password` changes the Customer's password. The Customer must specify as argument of the `-n` option the new password. For example: `change-password -n newpwd`.

```
usage: change-password -n <arg>
 -n,--newpassword <arg>   new password
```

Help Command `help`, as previously explained, is the same for every type of user. See above for details.

Logout Command `logout` terminates the session of the current Customer. It has no options nor arguments. After the logout, only non-logged-user commands can be typed. Usage: `logout`.

3 Key-Value Feasibility Study

3.1 Introduction

Main requirements of the hotel chain application are availability, scalability and consistency. Indeed, when the receptionist is at the phone with a customer, she must be able to add, delete or update a reservation in a fast and efficient way and without having to face inconsistency problems of overlapping reservations. Moreover, many receptionists must be able to access the application without being affected by high load traffic that might verify, for example, during the holiday period, when a lot of people call the hotels to get a reservation. Finally, at check-in time, when customers arrive at the hotel, the receptionist should be able to verify rapidly if they actually have a reservation and accommodate them in the assigned room.

Availability and horizontal scalability require a distributed database. However, deploying a RDBMS over multiple servers is inefficient from the performance point of view, whereas it is possible for a key-value database. Using such kind of database makes the partitioning of data over multiple nodes possible. Scalability is indeed one of the main advantages of key-value databases and also one of the main requirement of the hotel chain application. Since we also want to guarantee high availability, we could replicate partitions on multiple servers. The master-less replication schema is the most suitable one for our application because it increases availability and is more tolerant to server faults.

3.2 Hotel Chain Data Model over a Key-Value Database

Some operations become faster if performed on a key-value database, e.g. checking the customer's or receptionist's credentials would be more efficient. Anyway, RDBMS performance are still acceptable for these simple operations. Moreover, write operations involving entities with foreign keys would require to handle consistency issues at the application-level, thus increasing the complexity. This is the case for example of room and reservation entities, which are also involved in most of the operations of the application.

One of the most frequent activities in our application is searching for bookable rooms in a hotel in a given period of time. This query is implemented in a relational database with a join between `Reservation` and `Room` entities and involves almost all attributes of the two tables. This operation would hence become even more complex if translated into a key-value database because it would require to maintain a structured schema. Moreover, each request would require at least two scans of the key-value database: one to retrieve all rooms of a hotel and another to check if each room is bookable between a given check-in date and a given check-out date. Therefore, translating the hotel chain database into a key-value model is not convenient, as it would still require a schema structure and would also introduce complexity in implementing the query at the application level.

Another frequent and complex operation is the check-in. Every time a customer arrives at the hotel and shows his reservation identifier, the receptionist must check if there ac-

tually exists a reservation with such id and customer, in order to give him the key of the assigned room. In the relational database, the name and the surname of the customer are stored in the `Customer` table, the reservation id is in the `Reservation` table and the room number is stored in the `Room` table. Therefore, the implementation in SQL of the check-in functionality corresponds to a join query over three tables, one of which, `Reservation`, may become very large since it stores all the past reservations of all the hotels.

This functionality can be implemented more efficiently if we introduce a key-value cache. Performance of the check-in operation can increase if the receptionist uses the key-value cache rather than accessing to relational database. This cache stores three values for each reservation using three different keys:

- key `res:$reservation_id:name` for the customer's name,
- key `res:$reservation_id:surname` for customer's surname,
- key `res:$reservation_id:room` for room number.

The reservation id used in the key-value database is the one stored in relational model. Moreover, thanks to the schema-less nature of key-value databases, we could easily add in the future, if needed, more fields, e.g. check-in date or additional customer's requests.

The cache is updated whenever a reservation is added or updated in the relational database. This operation is implemented using a two-phase commit protocol: the relational database is immediately written whereas the cache is updated later by a dedicated thread. In this way the control on the application can return back to receptionist as soon as the relational database is updated and the receptionist does not have to wait the system accesses to both databases. We choose an eventual consistent model because it is very unlikely that a customer arrives at the hotel immediately after booking a room, hence it is not a problem if the cache is updated a little later.

If for some reasons the key-value database is down, the receptionist can still perform the check-in operation using the relational database, so that availability is still guaranteed. We implemented a simple simulation in which read operations are still executed in the SQL database when key-value is down. We decided to save in a log file all records that, for any reasons, cannot be written in the cache.

When the customer leaves the hotel, receptionist performs check-out operation. This operation simply removes from the cache records corresponding to the reservation of the departing customer. In this way we avoid to store in the cache useless information about past reservations, which are however still stored in the relational database.

This key-value cache is suitable for sharding: it can be partitioned and replicated over multiple servers to increase scalability. A good sharding key could be the hotel id, indeed each receptionist needs to know only reservations of the hotel where she works.

3.3 Performance Comparison

Performance regarding the check-in functionality are measured in order to see if the key-value store is actually faster than the MySQL database. The two databases are previously populated with a thousand new reservations to simulate a more realistic scenario.

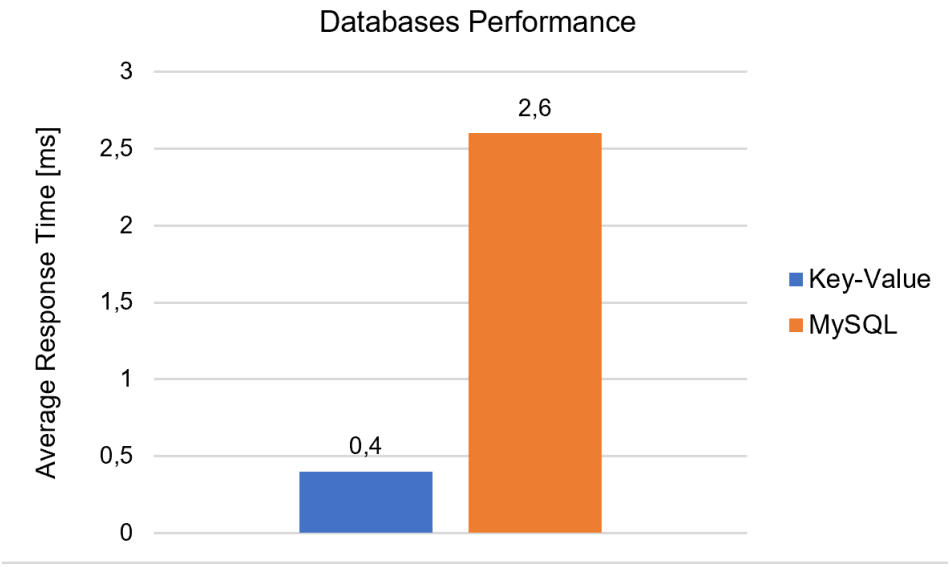

Figure 3.1: Performance Evaluation

The average time spent by MySQL to execute the queries needed by the check-in functionality is around 2.6 milliseconds, instead the same operation takes 0.4 milliseconds on average if performed on the key-value store.