CAR RENTING APPLICATION

TASK 1 - LARGE SCALE AND MULTISTRUCTURED DATABASE

UNIVERSITÀ DI PISA

Eugenia Petrangeli, Daniela Comola, Carmelo Aparo, Leonardo Fontanelli

INDEX

Introduction

The task that we realized is a Car Renting application, where users can register, search for available cars and place reservations with a graphic interface.
Also, we have implemented a feedback functionality, that allows the Customer to leave a mark and a comment to the Renting services.

A customer can register and log in, after the log in he is able to place reservations and delete them and also to leave a feedback.
An employer is able to register and log in as well, but differently from the previous case, he's able to add or delete cars from the available roster. Moreover, in the interface, an Employer has many functionalities: Car Manager, User Table, Feedback Table, Reservation Table.
In the Car Manager he can select a car in the list and delete one of them.
In the User Table he can consult the list of registered Customer to the system.
In the Feedback Table he can view the feedbacks left by the Customer and he's able to filter them by mark.
In the Reservation Table the Employer has the possibility to have a summary of the reservation and it's also able to filter them by the licence plate of the cars.

Requirements

Following are the requirements of the computer system.

The system is able to store the vehicle records. A vehicle record includes license plate, vendor, branch, seat number, kilometers, price and removed status. Seat number are 2, 4, 5 or 6. Branches are Firenze or Pisa. Removed field can be true or false and depends if the car has been removed from the system by an Employer.
The system is able to store the feedback records. A feedback record includes the id of the feedback, the mark, comment, data, the fiscal code of the user.
The system stores the reservation records. A reservation record includes the id of the reservation, license plate, fiscal code, pick-up date and delivery date.
The system stores the user record that includes name, surname, email, nickname, password, fiscal code and status. The status of the user can be employer or customer.

The system provides an employer interface where only him is able to access. In this interface he can view all the existing cars in a table whose columns are: license plate, vendor, seat number, location, kilometers, day price, status. By selecting a row, he/she can delete an existing car and this is permitted only if the car has no pending reservations. In the same interface he has the possibility to add new cars by compiling a form. When a car's license plate that refers to a previously removed car is inserted, we assume that it refers to the same car as before. The values of Kilometers, Location and Price are updated with the new ones but the Vendor and Number of Seats have to remain the same.
In addition, the Employer interface allows to view all the registered customers in a table whose columns are: fiscal code, nickname, name, surname, email.
Another possibility is to show the feedbacks table whose columns are: fiscal code, nickname, date, mark and comment. In this table the employer can filter the feedbacks by specifying a maximum mark.
Finally, he can view a reservations table whose columns are: fiscal code, license plate, pickup date and delivery date. In this table he can search the reservations associated to a specific license plate.

The system provides a log-in/registration interface. In order to register to the system, a user must fill a form indicating Name, Surname, Fiscal Code, Nickname, Email, Password and its status witch can be Customer or Employer. It is assumed that no Customer registers as Employer.
The system provides a customer interface that allows to search the available cars by specifying the renting period, the pickup location and the number of seats. The available cars are shown through a table whose columns are: vendor, seat number, kilometres and day price, then he can rent a car by selecting one of the rows.
In this interface it is also possible to view all the reservations associated to the logged customer in a table whose columns are: license plate, pickup date and delivery date. He can delete one of the pending reservations by selecting the corresponding row. Reservations and cancellations must be carried out at least one day in advance. Customers can return the car on the same day they picked it up.

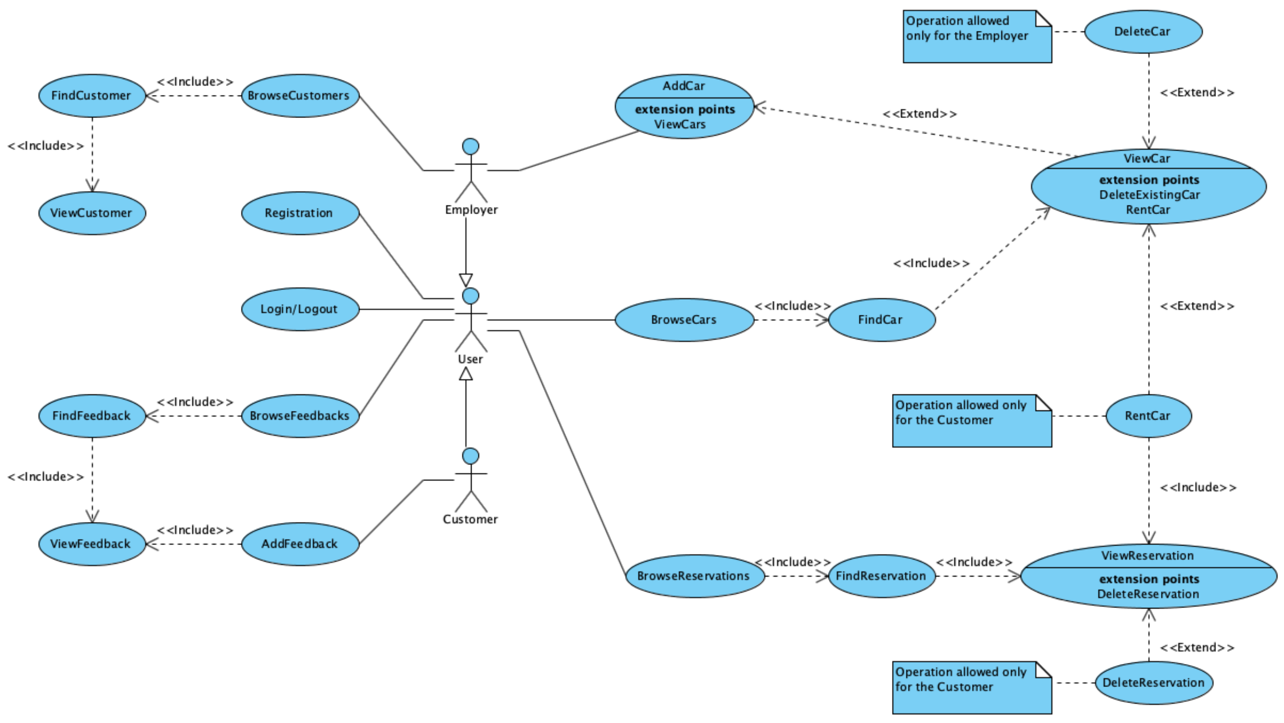
Finally, there is the feedback table where the customer can see the feedbacks left by customers. The columns of the table are: nickname, date, mark and comment. Moreover, he can add a feedback by selecting a mark and writing a comment.

As non functional requirements:
The system is designed in order to be user friendly, it has a graphical interface that is simple and intuitive and guide the user to make the right choice by showing also messages in case of errors.

The system manages the passwords in a secure way because it encrypts the passwords using SHA-1 when the user inserts them, so the passwords are sent to the DB already encrypted. This is done in order to prevent sniffing and to store them already encrypted.
In addition the system uses a specific user to connect to the DB, in this way the application has limited privileges on the DB in particular it can only modify the specific DB dedicated for the application.

Design

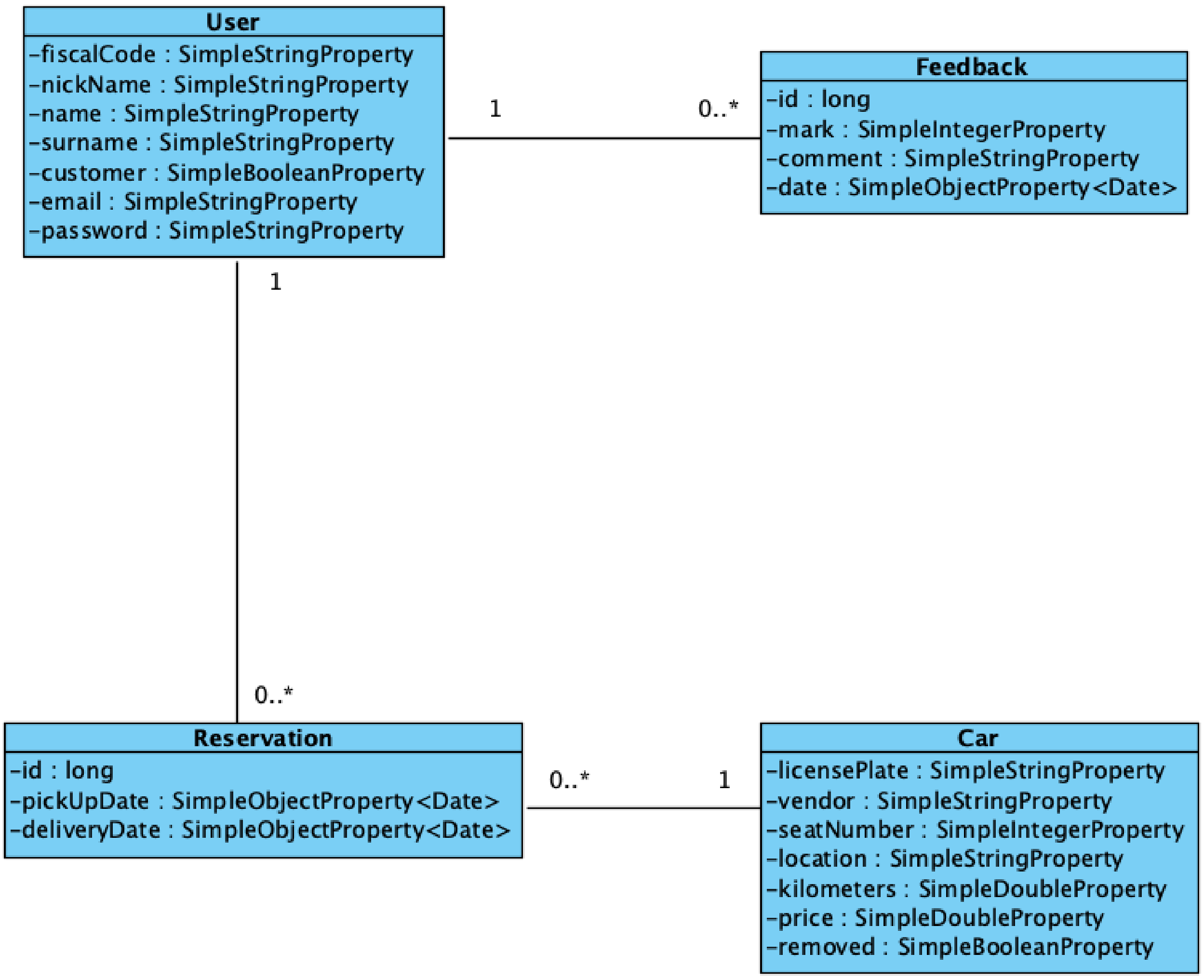
Use case diagram



The main actors of the system are the Employer and the Customer.
Both ones can register to the system, login and logout and browse the feedbacks of the customers.
They can also browse the cars, the Employer can browse through all system's cars, the Customers instead can search through only the available ones in a certain period.
In addition, both the actors can browse the reservations in the system, but the Employer can see all the ones in the system, instead the Customer can see only the ones associated to him.

There are actions that can be computed only from a specific type of actor, in fact the Employer can add or delete a car and he can browse the customers registered. On the other side, the customer can add a feedback.

Analysis classes diagram

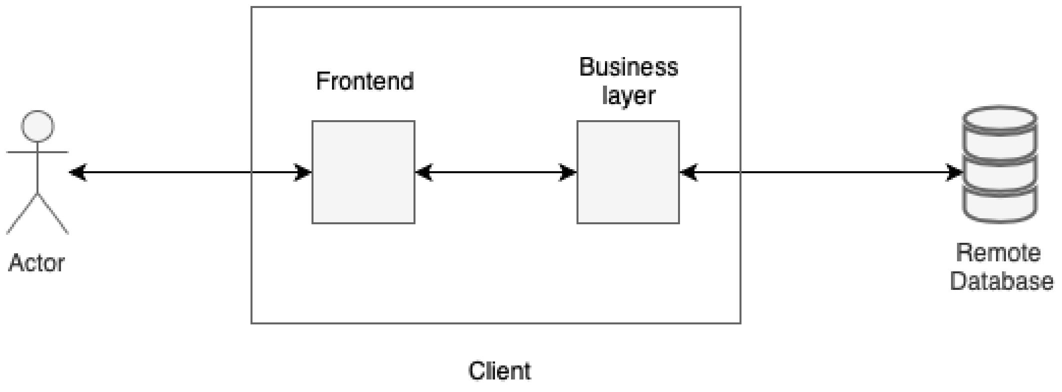


User
-fiscalCode : SimpleStringProperty
-nickName : SimpleStringProperty
-name : SimpleStringProperty
-surname : SimpleStringProperty
-customer : SimpleBooleanProperty
-email : SimpleStringProperty
-password : SimpleStringProperty
Feedback
-id : long
-mark : SimpleIntegerProperty
-comment : SimpleStringProperty
-date : SimpleObjectProperty<Date>
Reservation
-id : long
-pickUpDate : SimpleObjectProperty<Date>
-deliveryDate : SimpleObjectProperty<Date>
Car
-licensePlate : SimpleStringProperty
-vendor : SimpleStringProperty
-seatNumber : SimpleIntegerProperty
-location : SimpleStringProperty
-kilometers : SimpleDoubleProperty
-price : SimpleDoubleProperty
-removed : SimpleBooleanProperty
1
0..*
1
0..*
0..*
1

The User class has a one to many relationship with the Feedback class, because each user can write many feedbacks about his experience with the system but each feedback is associated with only one user.
In addition, the User class has a one to many relationship with the Reservation class, because each user can reserve zero or many cars but each reservation is associated with only one user.
The Car class has a one to many relationship because each car can be reserved by zero or many users (in different periods) but each reservation is associated with only one car.

Software architecture

The system is composed by a client application which is divided into a front-end and a business layer.



The last one is connected to a remote relational database where are stored all the information.
The architecture of the client is implemented in JAVA.
The Frontend is composed by the graphic interfaces for the Customer and the Employer that are realized with JavaFX libraries.
That classes are responsible to take the inputs of the users for all the operations permitted and then they send the data to the business layer. When the user inserts the credentials, the password is encrypted with SHA-1 algorithm.

The business layer is the one that satisfies the requests coming from the interface using JPA or both JPA and Riak to perform queries on the remote database. At the end of the operations, it returns the results to the graphic interface. This layer is composed by RentHandler class, JPAHandleDB class and RiakHandleDB class.

Implementation

JPA implementation

The entities defined in JPA are: *User*, *Car*, *Feedback* and *Reservation*.
For all the relationships among the entities the application doesn't use cascade parameters because each operation on the entities is isolated, so the other entities are not involved.
For example, when a new user is created, there is no need to create also a feedback associated with him/her.
For all the entities the attribute @Id is applied in the getter method so we must apply every other annotation that affects the fields of the class on the related getter methods. If instead we use @Id directly on a class field, we are specifying a direct access to the class fields. We use the first approach because using the getter and setter methods is the standard pattern.
The attribute @Column has been used for all the entities in order to specifies the names of the attributes in the tables in the DB.

```
@Entity
@Table(name="User")
public class User {
   private final SimpleStringProperty fiscalCode;
   private final SimpleStringProperty nickName;
   private final SimpleStringProperty name;
   private final SimpleStringProperty surname;
   private final SimpleBooleanProperty customer;
   private final SimpleStringProperty email;
   private final SimpleStringProperty password;
   private List<Feedback> feedbacks;
   private List<Reservation> reservations;

   public User() {
      fiscalCode = new SimpleStringProperty("");
      nickName = new SimpleStringProperty("");
      feedbacks = new ArrayList<>();
      reservations = new ArrayList<>();
      name = new SimpleStringProperty("");
      surname = new SimpleStringProperty("");
      customer = new SimpleBooleanProperty(true);
      email = new SimpleStringProperty("");
      password = new SimpleStringProperty("");
   }

   public User(String cf, String nm, String n, String c, Boolean cust, String e, String pwd)
   {
      fiscalCode = new SimpleStringProperty(cf);
      nickName = new SimpleStringProperty(nm);
      feedbacks = new ArrayList<>();
      reservations = new ArrayList<>();
      name = new SimpleStringProperty(n);
      surname = new SimpleStringProperty(c);
      customer = new SimpleBooleanProperty(cust);
      email = new SimpleStringProperty(e);
      password = new SimpleStringProperty(pwd);

   }

   public void setFiscalCode(String cf) {fiscalCode.set(cf);}
   public void setNickName(String nm) {nickName.set(nm);}
   public void setName(String n) { name.set(n); }
   public void setSurname(String s) { surname.set(s); }
   public void setEmail(String e) { email.set(e); }
   public void setCustomer(boolean c) { customer.set(c); }
   public void setPassword(String p) { password.set(p); }
   public void setFeedbacks(List<Feedback> feed) {feedbacks = feed;}
   public void setReservations(List<Reservation> res) {reservations = res;}

   @Id
   @Column(name = "FiscalCode", unique = true)
   public String getFiscalCode() { return fiscalCode.get();}

   @Column(name = "NickName", unique = true)
   public String getNickName() { return nickName.get();}
```

```
    @Column(name = "Name")
    public String getName() { return name.get(); }

    @Column(name = "Surname")
    public String getSurname() { return surname.get(); }

    @Column(name = "Email", unique = true)
    public String getEmail() { return email.get(); }

    @Column(name = "Customer")
    public Boolean getCustomer() {return customer.get(); }

    @Column(name = "Password")
    public String getPassword() { return password.get(); }

    @OneToMany(
            mappedBy = "user",
            fetch = FetchType.LAZY,
            cascade = {}
        )
    public List<Feedback> getFeedbacks() { return feedbacks;}
    @OneToMany(
        mappedBy = "user",
        fetch = FetchType.LAZY,
        cascade = {}
      )
    public List<Reservation> getReservations() { return reservations;}

    public SimpleStringProperty nickNameProperty() {
      return this.nickName;
    }

    public SimpleStringProperty fiscalCodeProperty() {
      return this.fiscalCode;
    }

}
```

Each field of the Entity *User* corresponds to a column of the table *User* in the database.
The fields *fiscalCode*, *nickName* and *email* have the *unique = true* attribute since they have to be unique in the user table.
The entities *User* and *Feedback* are associated through a one to many relationship, so each *User* has a list of *Feedback* objects and each *Feedback* has a *User* object. The attribute *mappedBy* specifies that the entity *User* is associated to the field *user* of the entity Feedback.
The fetch parameter for this relationship defined for *User* entity is *FetchType.LAZY*, since the system doesn't need to fetch always all the feedbacks of each user.

The entities *User* and *Reservation* are associated through a one to many relationship, so each *User* has a list of *Reservation* objects and each *Reservation* has a *User* object. The attribute *mappedBy* specifies that the entity *User* is associated to the field *user* of the entity Reservation.
The fetch parameter in the *User* entity is *FetchType.LAZY*, because the system stores all the reservations done by the users, also the ones that are already expired, but only the active ones are shown to the customer, so it is not useful to retrieve every time all the *Reservation* objects associated to a *User*.

```
@Entity
public class Feedback {
   private long id;
   private final SimpleIntegerProperty mark;
   private final SimpleStringProperty comment;
   private final SimpleObjectProperty<Date> date;
   private SimpleObjectProperty<User> user;

   public Feedback() {
      mark = new SimpleIntegerProperty(0);
      comment = new SimpleStringProperty("");
      date = new SimpleObjectProperty<Date> ();
      user = new SimpleObjectProperty<User> ();
   }

   public Feedback(int mark, String comment, Date date ,User user) {
      this.mark = new SimpleIntegerProperty(mark);
      this.comment = new SimpleStringProperty(comment);
      this.date = new SimpleObjectProperty<Date> (date);
      this.user = new SimpleObjectProperty<> (user);
   }

   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   @Column(name = "idFeedback")
   public long getId() {
      return id;
   }

   public void setId(long id) {
      this.id = id;
   }

   @Column(name = "Mark")
   public int getMark() {
      return mark.get();
   }
   public void setMark(int mark) {
      this.mark.set(mark);
   }
```

```
  @Column(name = "Comment")
  public String getComment() {
    return comment.get();
  }
  public void setComment(String comment) {
    this.comment.set(comment);
  }

  @Column(name = "Date")
  public Date getDate() {
    return date.get();
  }
  public void setDate(Date date) {
    this.date.set(date);
  }

  @ManyToOne(
        fetch=FetchType.EAGER,
        cascade = {}
        )
  public User getUser() {
    return user.get();
  }

  public void setUser(User user) {
    this.user.set(user);
  }

  public SimpleStringProperty nickNameProperty() {
    return user.get().nickNameProperty();
  }

  public SimpleStringProperty fiscalCodeProperty() {
    return user.get().fiscalCodeProperty();
  }
}
```

Each field of the Entity *Feedback* corresponds to a column of the table *Feedback* in the database. @GeneratedValue(strategy = GenerationType.IDENTITY) attribute is used to generate an auto incremental counter used for the field *idFeedback*.

The annotation @ManyToOne is used in order to specify that the field *user* is the one used in the *MappedBy* attribute in the *User* entity.
The fetch parameter for the *Feedback* entity is *FetchType.EAGER*, since the application shows the user who wrote the feedback, so it needs to retrieve the *User* object related to the *Feedback* one every time it has to show a feedback.

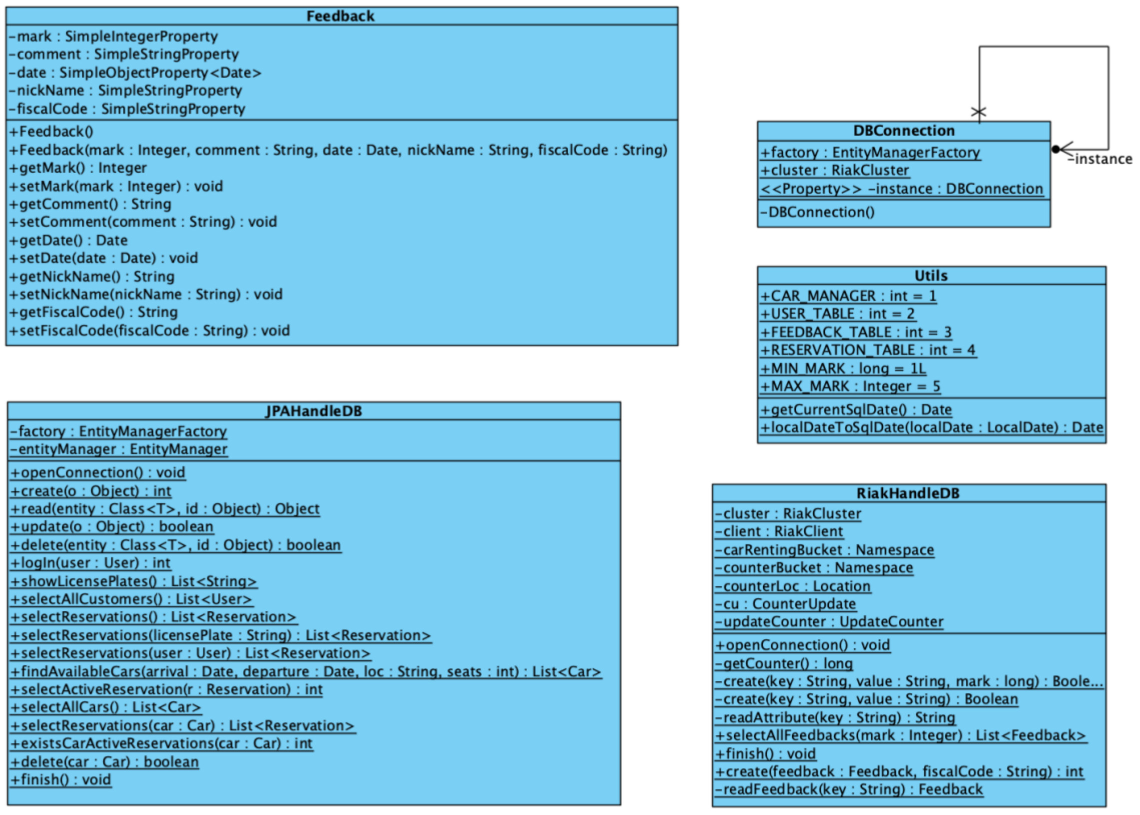
```java
@Entity
@Table(name = "Reservation")
public class Reservation {
   private long id;
   private SimpleObjectProperty<Date> pickUpDate;
   private SimpleObjectProperty<Date> deliveryDate;
   private User user;
   private Car car;

   public Reservation() {
      pickUpDate = new SimpleObjectProperty<Date> ();
      deliveryDate = new SimpleObjectProperty<Date> ();
   }

   public Reservation(LocalDate pickUpDate, LocalDate deliveryDate, User user, Car car)
   {
      this.pickUpDate = new SimpleObjectProperty<Date>
         (Utils.localDateToSqlDate(pickUpDate));
      this.deliveryDate = new SimpleObjectProperty<Date>
         (Utils.localDateToSqlDate(deliveryDate));
      this.user = user;
      this.car = car;
   }

   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   @Column(name = "Id")
   public long getId() {
      return id;
   }

   @Column(name = "PickUpDate")
   public Date getPickUpDate() {
      return pickUpDate.get();
   }

   @Column(name = "DeliveryDate")
   public Date getDeliveryDate() {
      return deliveryDate.get();
   }

   @ManyToOne(fetch = FetchType.EAGER, cascade = {})
   public User getUser() {
      return user;
   }

   @ManyToOne(fetch = FetchType.EAGER, cascade = {})
   public Car getCar() {
      return car;
   }

   public void setId(long id) {
      this.id = id;
   }

   public void setPickUpDate(Date  pickUpDate) {
      this.pickUpDate.set(pickUpDate);
   }

   public void setDeliveryDate(Date  deliveryDate) {
      this.deliveryDate.set(deliveryDate);
   }

   public void setUser(User user) {
      this.user = user;
   }

   public void setCar(Car car) {
      this.car = car;
   }

   public SimpleStringProperty fiscalCodeProperty() {
      return user.fiscalCodeProperty();
   }

   public SimpleStringProperty licensePlateProperty() {
      return car.licensePlateProperty();
   }

   public SimpleStringProperty vendorProperty() {
      return car.vendorProperty();
   }

   public SimpleIntegerProperty seatNumberProperty() {
      return car.seatNumberProperty();
   }

   public SimpleDoubleProperty priceProperty() {
      return car.priceProperty();
   }

   public SimpleDoubleProperty kilometersProperty() {
      return car.kilometersProperty();
   }
}
```

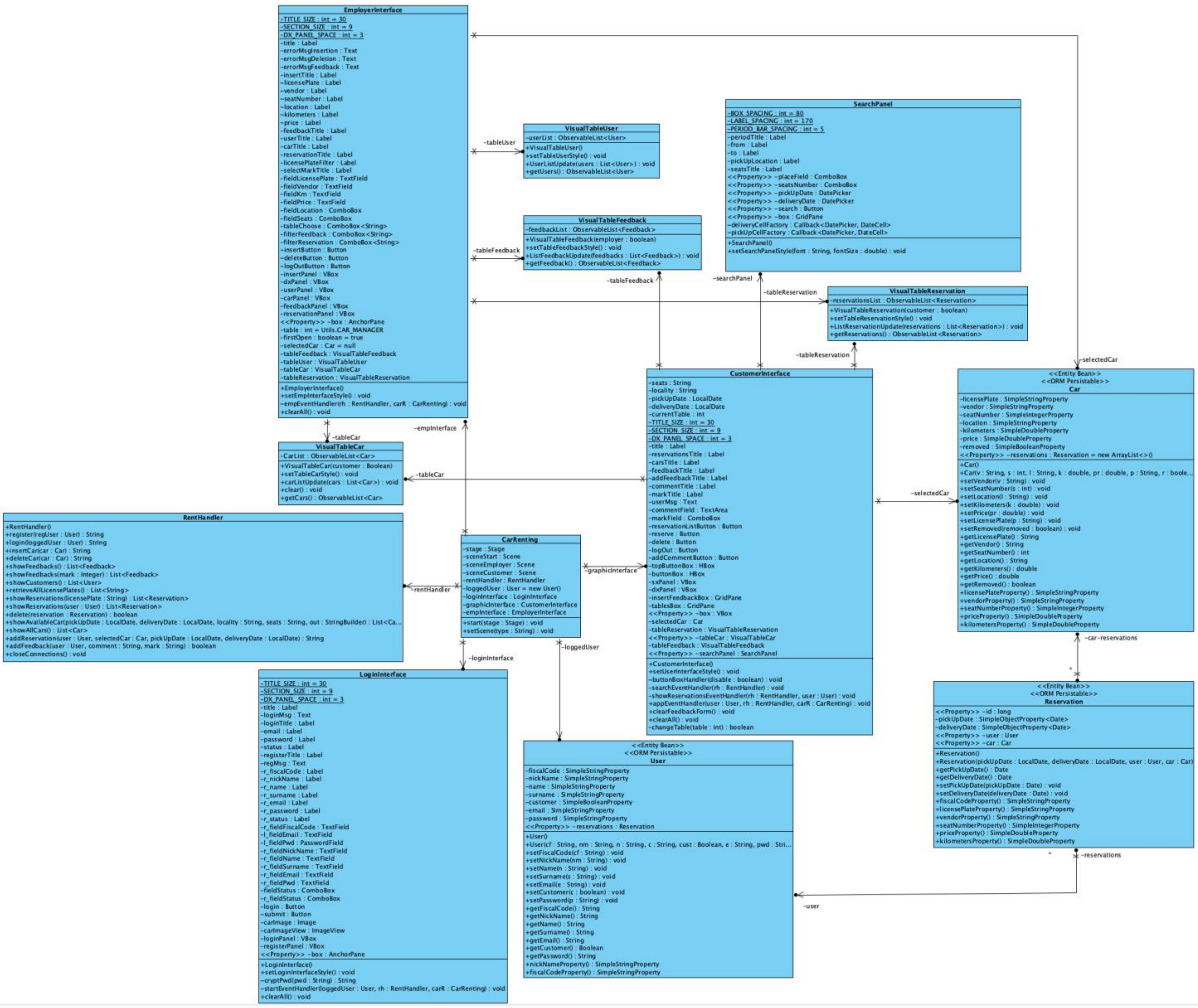
Each field of the Entity *Reservation* corresponds to a column of the table *Reservation* in the database.
@GeneratedValue(strategy = GenerationType.IDENTITY) attribute is used to generate an auto incremental counter used for the field *id*.
The annotations @ManyToOne are used in order to specify that the fields *user* and *car* are the ones used in the *MappedBy* attributes in the *User* and *Car* entities respectively.
The fetch parameter for the relationship with *User* is *FetchType.EAGER*, because when a *Reservation* is retrieved by the Employer, it is useful to have the *User* information to eventually search him.
The fetch parameter for the relationship with *Car* is *FetchType.EAGER*, because a customer can see his active reservations and when the system shows the reservations, it shows also the information about the cars rented.

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

   private final SimpleStringProperty licensePlate;
   private final SimpleStringProperty vendor;
   private final SimpleIntegerProperty seatNumber;
   private final SimpleStringProperty location;
   private final SimpleDoubleProperty kilometers;
   private final SimpleDoubleProperty price;
   private final SimpleBooleanProperty removed;

   private List<Reservation> reservations =    new ArrayList<>();

   public Car() {
      licensePlate = new SimpleStringProperty("");
      vendor = new SimpleStringProperty("");
      seatNumber = new SimpleIntegerProperty(0);
      location = new SimpleStringProperty("");
      kilometers = new SimpleDoubleProperty(0);
      price = new SimpleDoubleProperty(0.0);
      removed = new SimpleBooleanProperty(false);
   }

   public Car(String v, int s, String l, double k, double pr, String p, boolean r) {
      licensePlate = new SimpleStringProperty(p);
      vendor = new SimpleStringProperty(v);
      seatNumber = new SimpleIntegerProperty(s);
      location = new SimpleStringProperty(l);
      kilometers = new SimpleDoubleProperty(k);
      price = new SimpleDoubleProperty(pr);
      removed = new SimpleBooleanProperty(r);
   }

   public void setVendor(String v) {vendor.set(v);}
public void setSeatNumber(int s) {seatNumber.set(s);}
public void setLocation(String l) {location.set(l);}
public void setKilometers(double k) {kilometers.set(k);}
public void setPrice(double pr) {price.set(pr);}
public void setLicensePlate(String p) {licensePlate.set(p);}
public void setRemoved(boolean removed) {this.removed.set(removed);}
public void setReservations(List<Reservation> res) {reservations = res;}

@Id
@Column(name="LicensePlate", unique = true)
public String getLicensePlate() {return licensePlate.get();}
@Column(name="Vendor")
public String getVendor() { return vendor.get();}
@Column(name="SeatNumber")
public int getSeatNumber() { return seatNumber.get();}
@Column(name="Location")
public String getLocation() { return location.get();}
@Column(name="Kilometers")
public double getKilometers() {return kilometers.get();}
@Column(name="Price")
public double getPrice() {return price.get();}
@Column(name="Removed")
public boolean getRemoved() { return removed.get(); }
@OneToMany(
        mappedBy = "car",
        fetch = FetchType.LAZY,
        cascade = {}
        )
public List<Reservation> getReservations() { return reservations; }

public SimpleStringProperty licensePlateProperty() {
    return this.licensePlate;
}

public SimpleStringProperty vendorProperty() {
    return this.vendor;
}

public SimpleIntegerProperty seatNumberProperty() {
    return this.seatNumber;
}

public SimpleDoubleProperty priceProperty() {
    return this.price;
}

public SimpleDoubleProperty kilometersProperty() {
    return this.kilometers;
    }
}
```

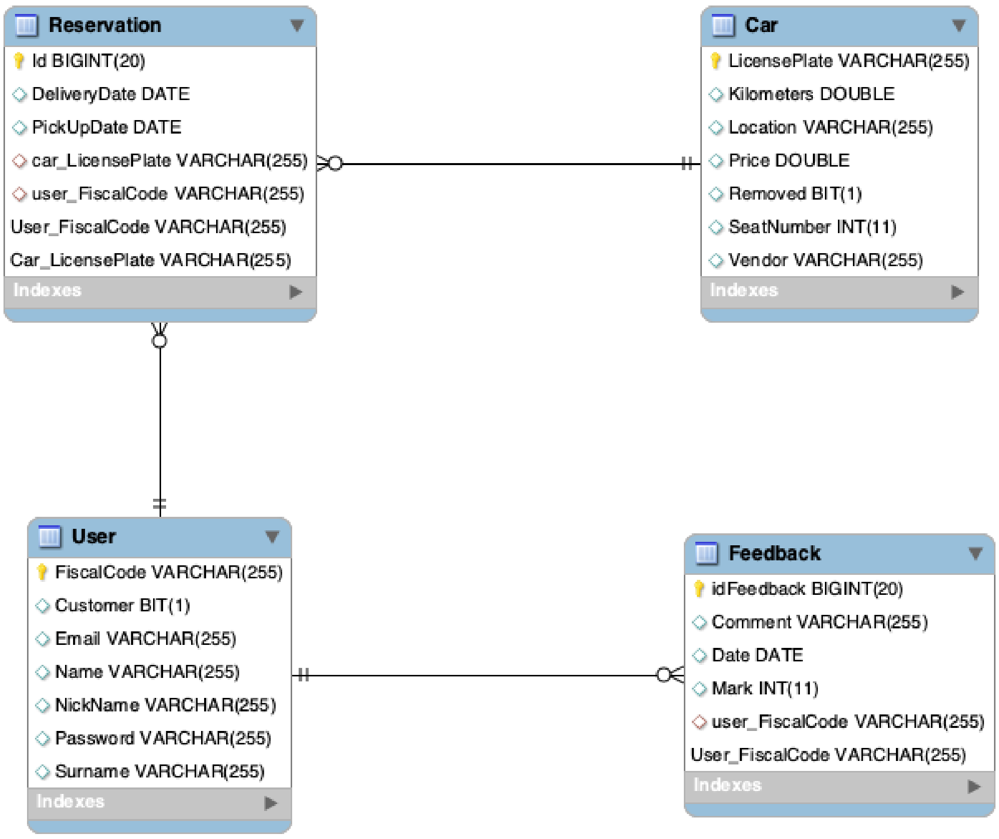
Each field of the Entity *Car* corresponds to a column of the table *Car* in the database.
The entities *Car* and *Reservation* are associated through a one to many relationship, so each *Car* has a list of *Reservation* objects and each *Reservation* has a *Car* object. The attribute *mappedBy* specifies that the entity *Car* is associated to the field *car* of the entity *Reservation*.
The fetch parameter in the *Car* entity is *FetchType.LAZY*, because the *Car*, as the *User* in the previous relationship, has also past reservations and it is not useful to retrieve all the past reservations each time.

Implementation Class Diagram


Feedback
-mark : SimpleIntegerProperty
-comment : SimpleStringProperty
-date : SimpleObjectProperty<Date>
-nickName : SimpleStringProperty
-fiscalCode : SimpleStringProperty
+Feedback()
+Feedback(mark : Integer, comment : String, date : Date, nickName : String, fiscalCode : String)
+getMark() : Integer
+setMark(mark : Integer) : void
+getComment() : String
+setComment(comment : String) : void
+getDate() : Date
+setDate(date : Date) : void
+getNickName() : String
+setNickName(nickName : String) : void
+getFiscalCode() : String
+setFiscalCode(fiscalCode : String) : void
DBConnection
+factory : EntityManagerFactory
+cluster : RiakCluster
<<Property>> -instance : DBConnection
-DBConnection()
-instance
Utils
+CAR_MANAGER : int = 1
+USER_TABLE : int = 2
+FEEDBACK_TABLE : int = 3
+RESERVATION_TABLE : int = 4
+MIN_MARK : long = 1L
+MAX_MARK : Integer = 5
+getCurrentSqlDate() : Date
+localDateToSqlDate(localDate : LocalDate) : Date
JPAHandleDB
-factory : EntityManagerFactory
-entityManager : EntityManager
+openConnection() : void
+create(o : Object) : int
+read(entity : Class<T>, id : Object) : Object
+update(o : Object) : boolean
+delete(entity : Class<T>, id : Object) : boolean
+logIn(user : User) : int
+showLicensePlates() : List<String>
+selectAllCustomers() : List<User>
+selectReservations() : List<Reservation>
+selectReservations(licensePlate : String) : List<Reservation>
+selectReservations(user : User) : List<Reservation>
+findAvailableCars(arrival : Date, departure : Date, loc : String, seats : int) : List<Car>
+selectActiveReservation(r : Reservation) : int
+selectAllCars() : List<Car>
+selectReservations(car : Car) : List<Reservation>
+existsCarActiveReservations(car : Car) : int
+delete(car : Car) : boolean
+finish() : void
RiakHandleDB
-cluster : RiakCluster
-client : RiakClient
-carRentingBucket : Namespace
-counterBucket : Namespace
-counterLoc : Location
-cu : CounterUpdate
-updateCounter : UpdateCounter
+openConnection() : void
-getCounter() : long
-create(key : String, value : String, mark : long) : Boole...
-create(key : String, value : String) : Boolean
-readAttribute(key : String) : String
+selectAllFeedbacks(mark : Integer) : List<Feedback>
+finish() : void
+create(feedback : Feedback, fiscalCode : String) : int
-readFeedback(key : String) : Feedback


EmployerInterface
VisualTableUser
VisualTableFeedback
SearchPanel
VisualTableReservation
CustomerInterface
<<Entity Bean>> <<ORM Persistable>> Car
VisualTableCar
RentHandler
CarRenting
LoginInterface
<<Entity Bean>> <<ORM Persistable>> User
<<Entity Bean>> <<ORM Persistable>> Reservation
-tableUser
-tableFeedback
-searchPanel
-tableReservation
-selectedCar
-tableCar
-empInterface
-rentHandler
-graphicInterface
-loginInterface
-loggedUser
-car
-reservations
-user

Database implementation

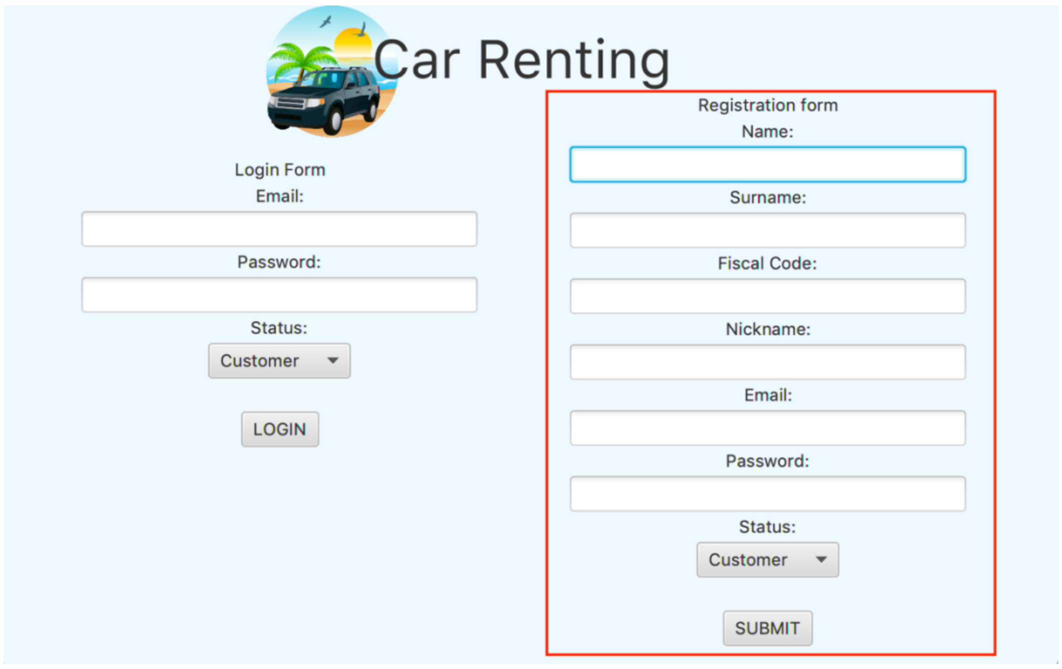


The database is realized by using a MySQL database and the operations on it are performed by using Java Persistence API (JPA).

User manual

Registration/Login Interface Manual

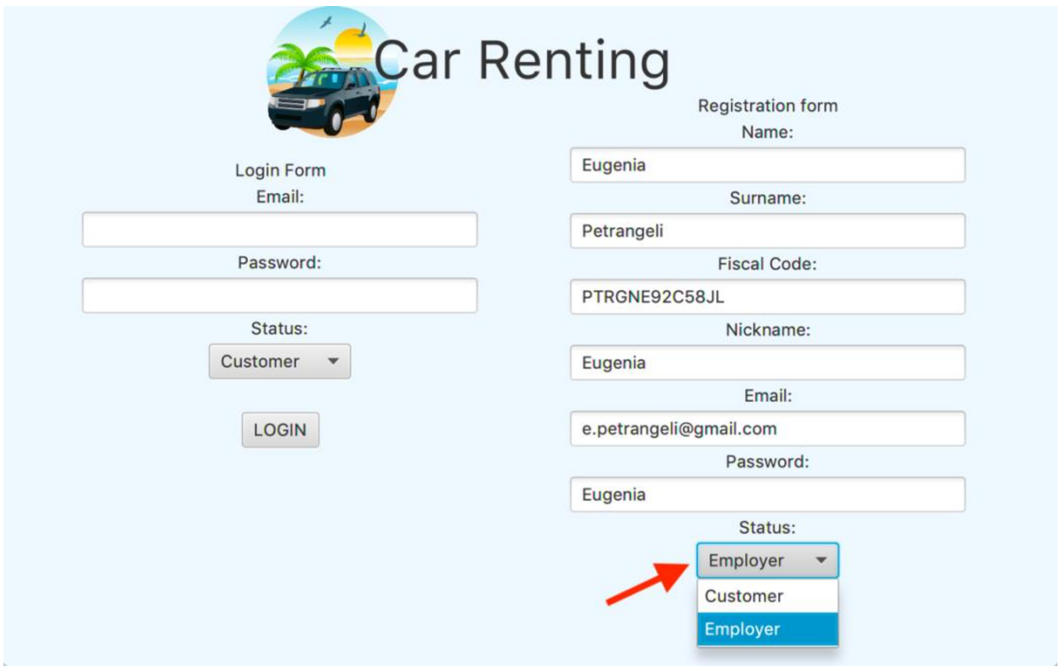
A user can register himself using the form on the right:


Car Renting

Login Form
Email:
Password:
Status: Customer
LOGIN

Registration form
Name:
Surname:
Fiscal Code:
Nickname:
Email:
Password:
Status: Customer
SUBMIT

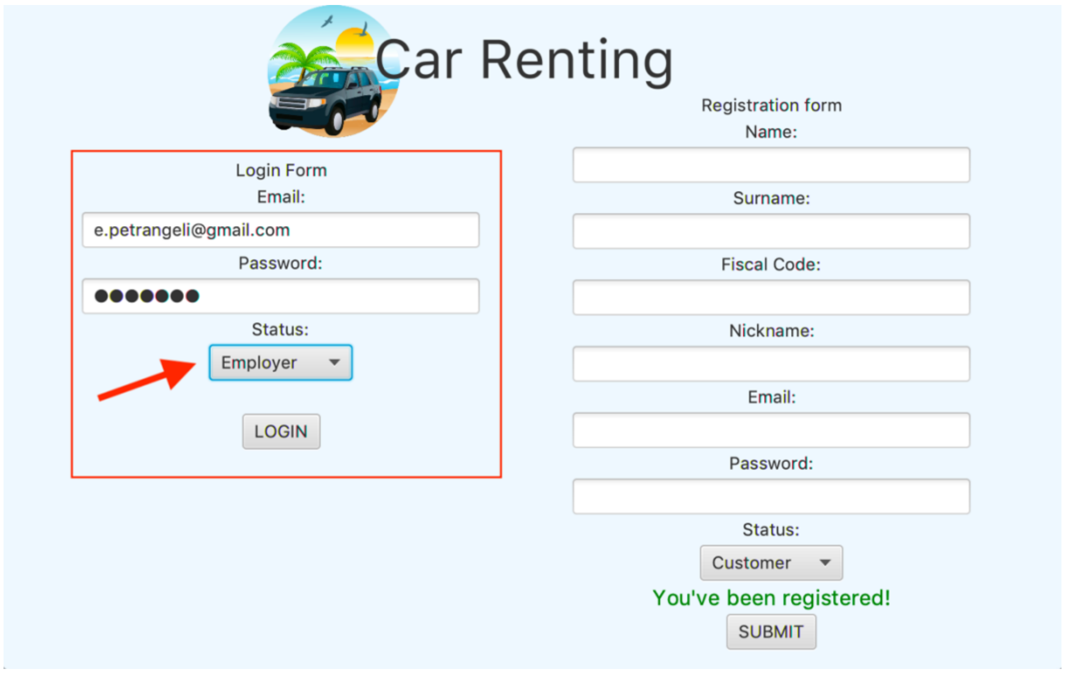
Name, surname, fiscal code, nickname, email, password and the status (Customer or Employer) are required. After entering the previous fields, simply click on the "SUBMIT" button.
The user cannot enter a nickname or email that has already been used by someone else.
The status is selected from the Status box has shown in the following picture:


Car Renting

Login Form
Email:
Password:
Status: Customer
LOGIN

Registration form
Name: Eugenia
Surname: Petrangeli
Fiscal Code: PTRGNE92C58JL
Nickname: Eugenia
Email: e.petrangeli@gmail.com
Password: Eugenia
Status: Employer
Customer
Employer

A user can login using the form on the left:


Car Renting

Login Form

Email: e.petrangeli@gmail.com

Password: ●●●●●●●

Status: Employer

LOGIN

Registration form

Name:

Surname:

Fiscal Code:

Nickname:

Email:

Password:

Status: Customer

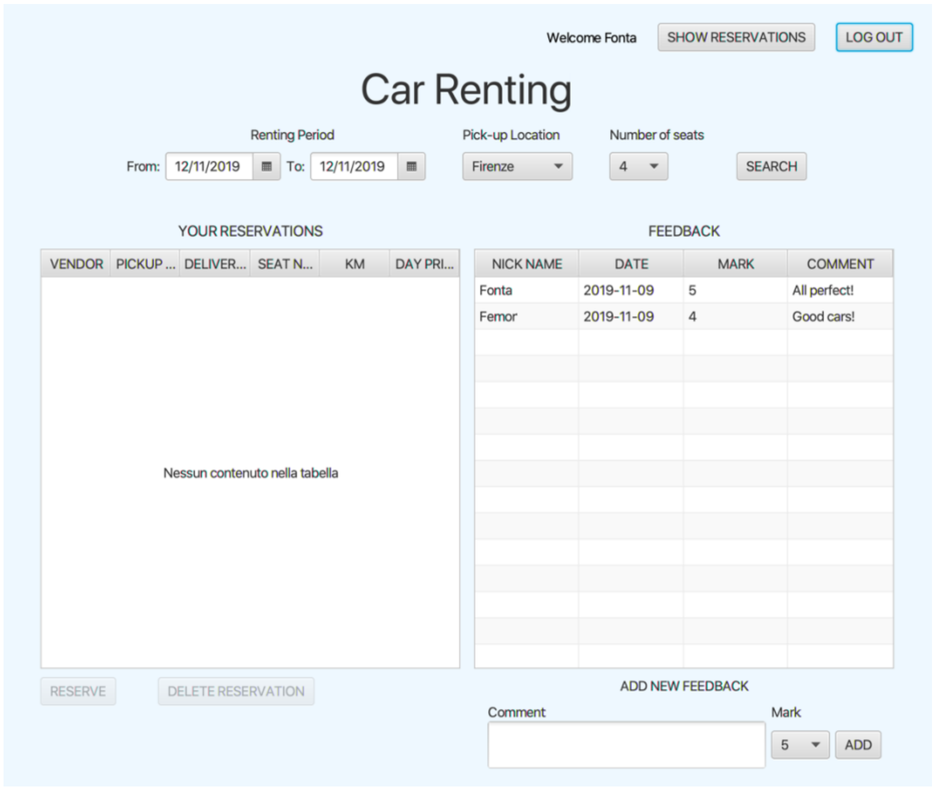
You've been registered!

SUBMIT

Email, password and status entered during registration are required. After entering the previous fields, simply click on the "LOGIN" button.
Here too it is possible to select the status from the box shown in the previous figure.
After logging in, the Customer Interface or Employer Interface will be displayed depending on the selected Status

Customer Interface Manual

Our application provides a specific interface, that allows customers to consult the list of the available cars in a certain period, location, and seats number. Consequently, he can reserve a car, delete the reservation and finally, to leave a feedback to the company's services.
Here's the layout of the application in the Customer Interface:


Welcome Fonta SHOW RESERVATIONS LOG OUT

Car Renting

Renting Period — From: 12/11/2019 To: 12/11/2019

Pick-up Location: Firenze

Number of seats: 4

SEARCH

YOUR RESERVATIONS

VENDOR	PICKUP ...	DELIVER...	SEAT N...	KM	DAY PRI...

Nessun contenuto nella tabella

RESERVE DELETE RESERVATION

FEEDBACK

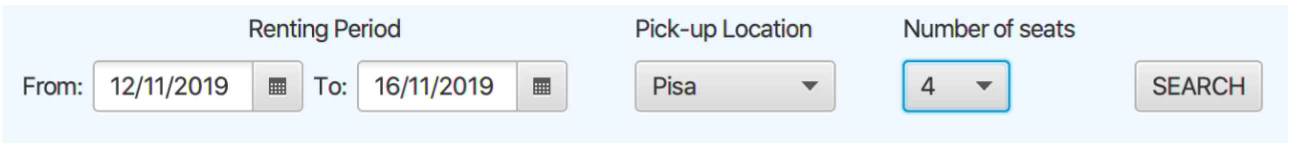
NICK NAME	DATE	MARK	COMMENT
Fonta	2019-11-09	5	All perfect!
Femor	2019-11-09	4	Good cars!

ADD NEW FEEDBACK

Comment

Mark: 5 ADD

The image above shows how the interface presents itself to the user after the login.
Going deeply into the functioning: filling the form, indicating the desired Renting Period, the Pick-Up Location and the Number of seats of the car, will be possible to see the available cars.
Let’s suppose, for example, to search for a car in Pisa with 4 seats for the period from the 12th to the 16th of November.

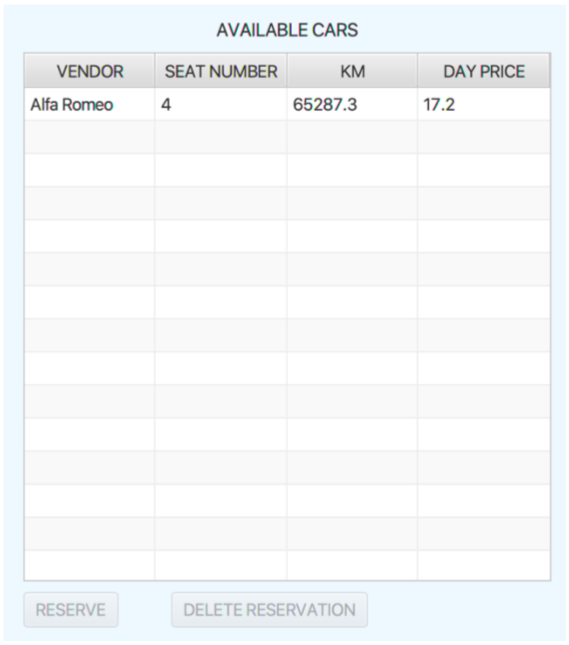

Renting Period — From: 12/11/2019 To: 16/11/2019

Pick-up Location: Pisa

Number of seats: 4

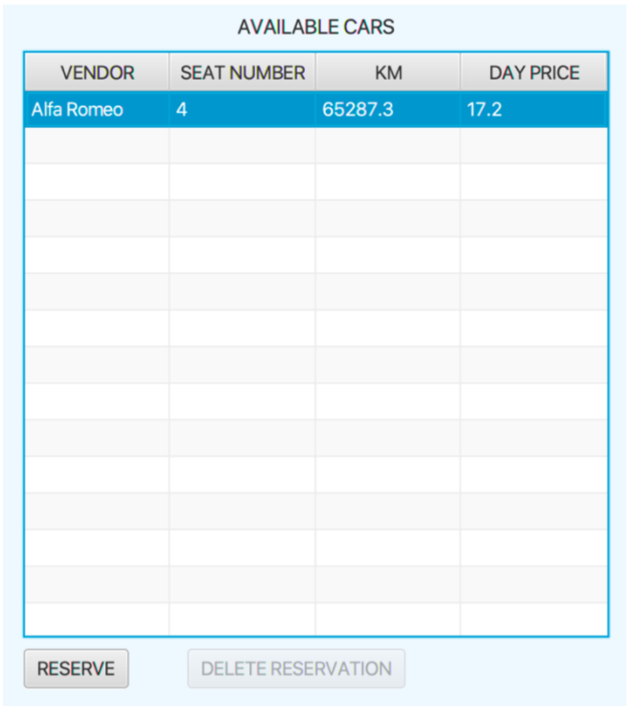
SEARCH

By clicking on the “Search” button, on the left side of the interface, will be shown the list of the available car for the desired parameters.
Following there is the resulting table for our research:


AVAILABLE CARS

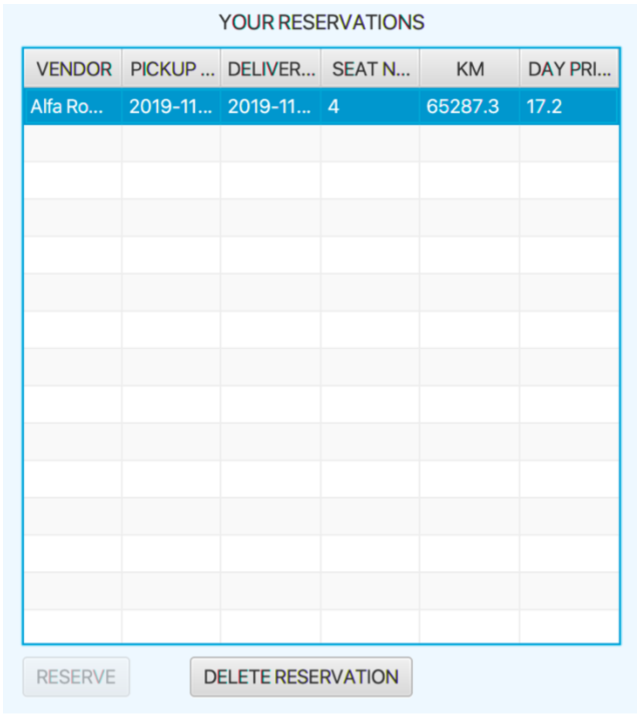
VENDOR	SEAT NUMBER	KM	DAY PRICE
Alfa Romeo	4	65287.3	17.2

As we can see, there is an available car, which is an Alfa Romeo. Now by selecting the corresponding row in the table, the "Reserve" button below will be activated.


AVAILABLE CARS

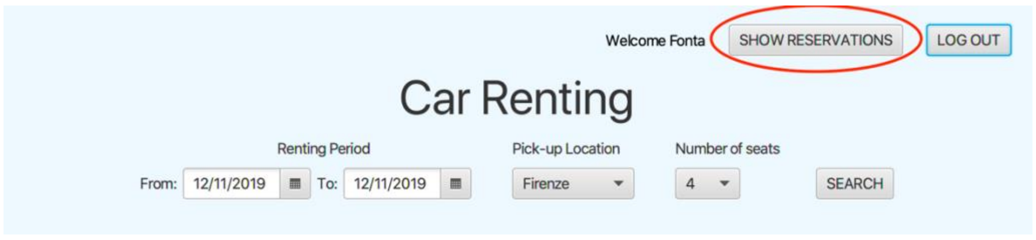
VENDOR	SEAT NUMBER	KM	DAY PRICE
Alfa Romeo	4	65287.3	17.2

Clicking on the Reserve button, the reservation will be confirmed and the "Available Cars" table will be replaced with the "Reservation" table, where will be shown the active reservation of the user.
There is, also, the possibility to delete a reservation. This is possible by selecting the row in the table of the reservation and the "Delete Reservation" button will be activated.


YOUR RESERVATIONS

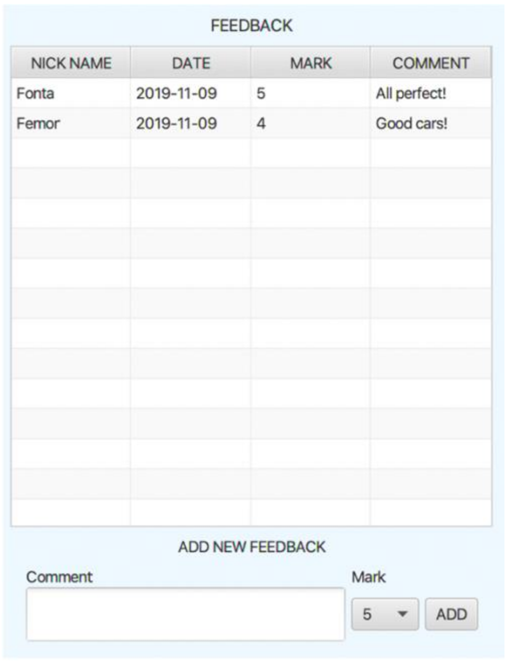
VENDOR	PICKUP ...	DELIVER...	SEAT N...	KM	DAY PRI...
Alfa Ro...	2019-11...	2019-11...	4	65287.3	17.2

By clicking on the button, the reservation will be deleted.
In the upper part of the interface we can find the “Show Reservation” button in order to see the table with the list of the active reservation of the user, as indicated in the figure below.



Another functionality for the customer regards the feedbacks. He can leave a new feedback, with a comment and a mark (from 1 to 5) and consult the others left from the customers of the company.
In the figure below there is the “Feedback Table”, the system for every entry stores the Nickname, date, mark and the comment.

It is possible to logout at any time by clicking the “LOG OUT” button.


FEEDBACK

NICK NAME	DATE	MARK	COMMENT
Fonta	2019-11-09	5	All perfect!
Femor	2019-11-09	4	Good cars!

ADD NEW FEEDBACK

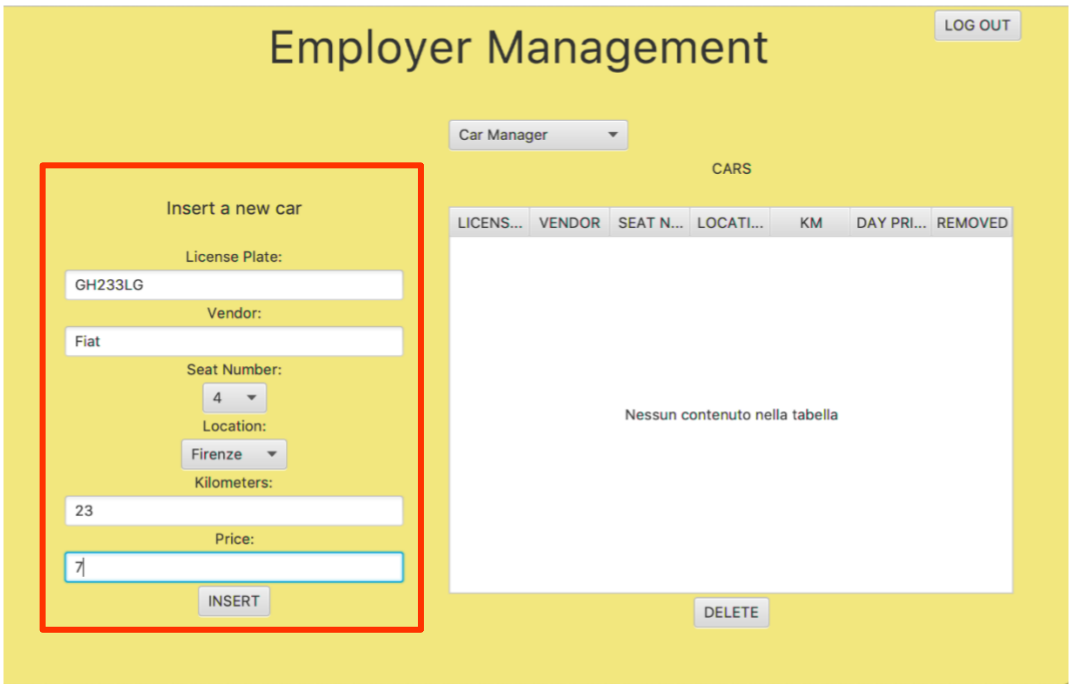
Comment

Mark: 5

Employer Interface Manual

The application provides a graphic interface for Car Renting Staff use only, where they can monitor and handle any kind of data related to the car renting activity, as the list of cars available in the system, a summary of the received reservations, registered customers and feedback received for the service.
In particular, once logged in, an employer can perform a set of actions:

- Adding a new car, using the form on the left of the interface


Employer Management

Insert a new car

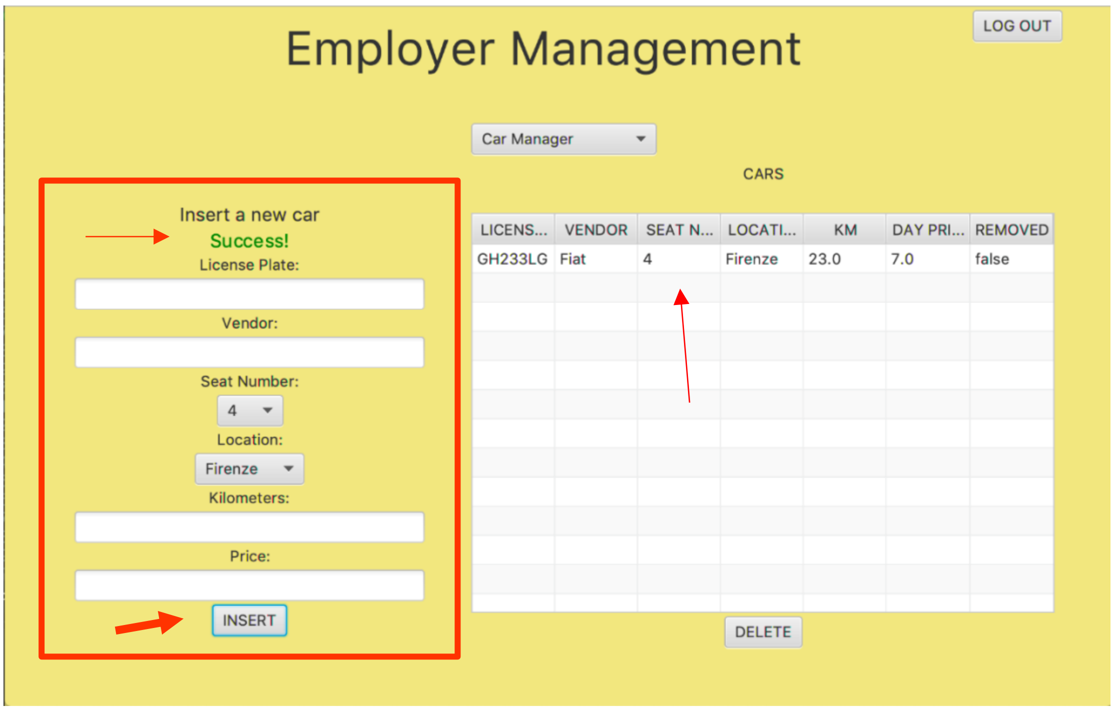
License Plate:	GH233LG
Vendor:	Fiat
Seat Number:	4
Location:	Firenze
Kilometers:	23
Price:	7

CARS

LICENS...	VENDOR	SEAT N...	LOCATI...	KM	DAY PRI...	REMOVED

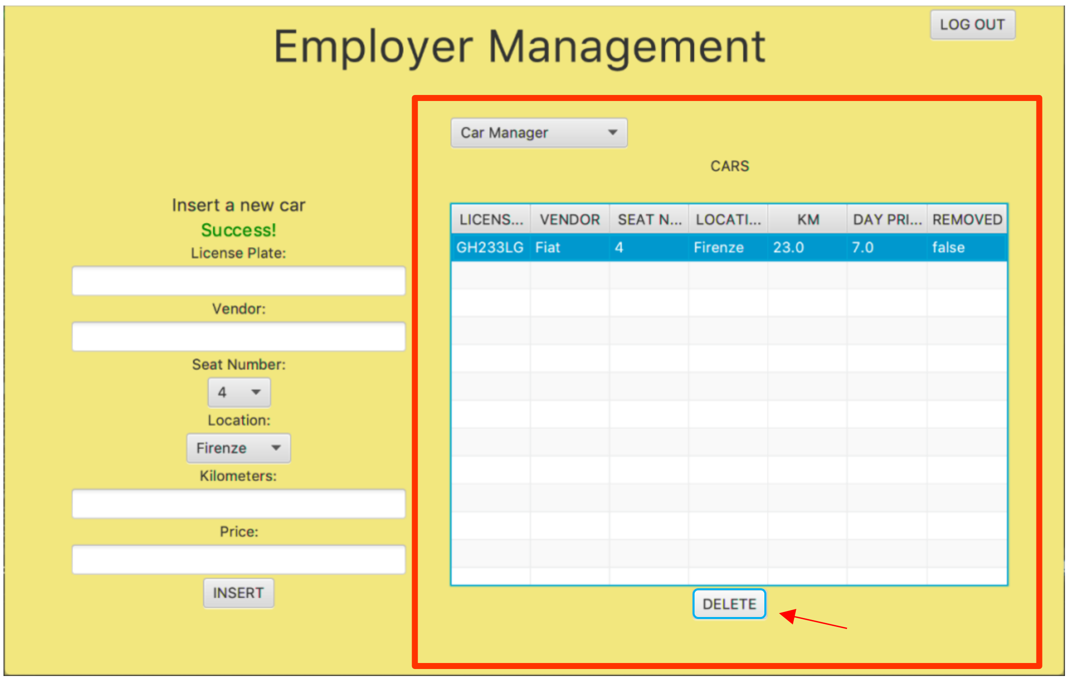
Nessun contenuto nella tabella

License Plate, Vendor, Seat Number, Location, Kilometers, Price are required. Once entered all the previous fields, simply click the “INSERT” button and the new car will be stored.



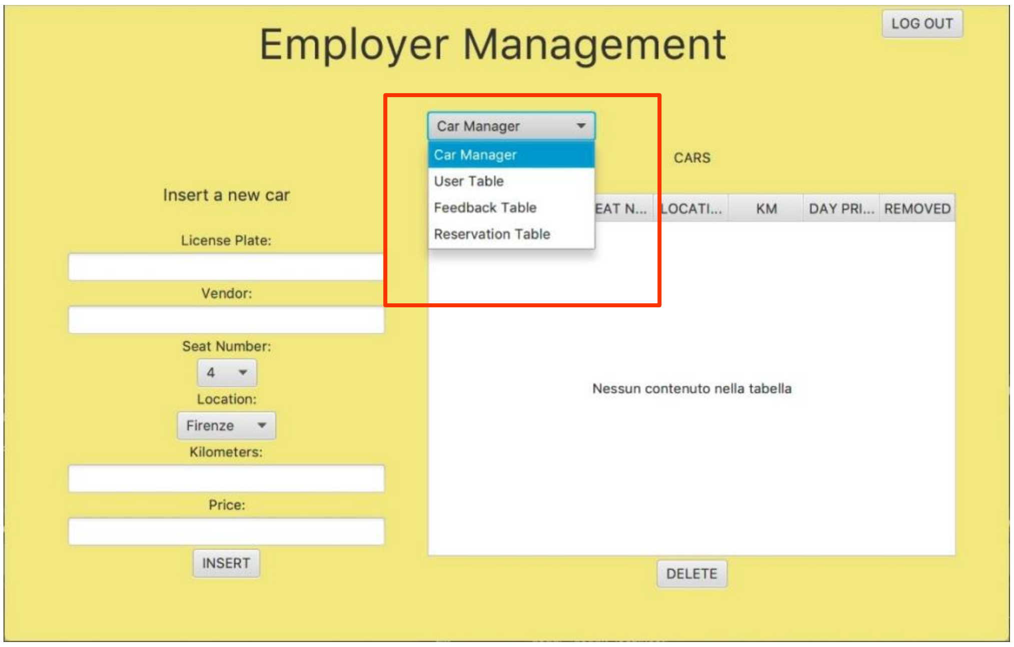
LICENS...	VENDOR	SEAT N...	LOCATI...	KM	DAY PRI...	REMOVED
GH233LG	Fiat	4	Firenze	23.0	7.0	false

On the right side of the interface there's a panel showing a list with all the cars available in the branches. Every time a new car is inserted, when the Car Manager is selected, the table is updated and the added car is showed in the list, too.
It is also possible to delete a car from the roster, simply by selecting the removing car in the table and clicking on the "DELETE" button.



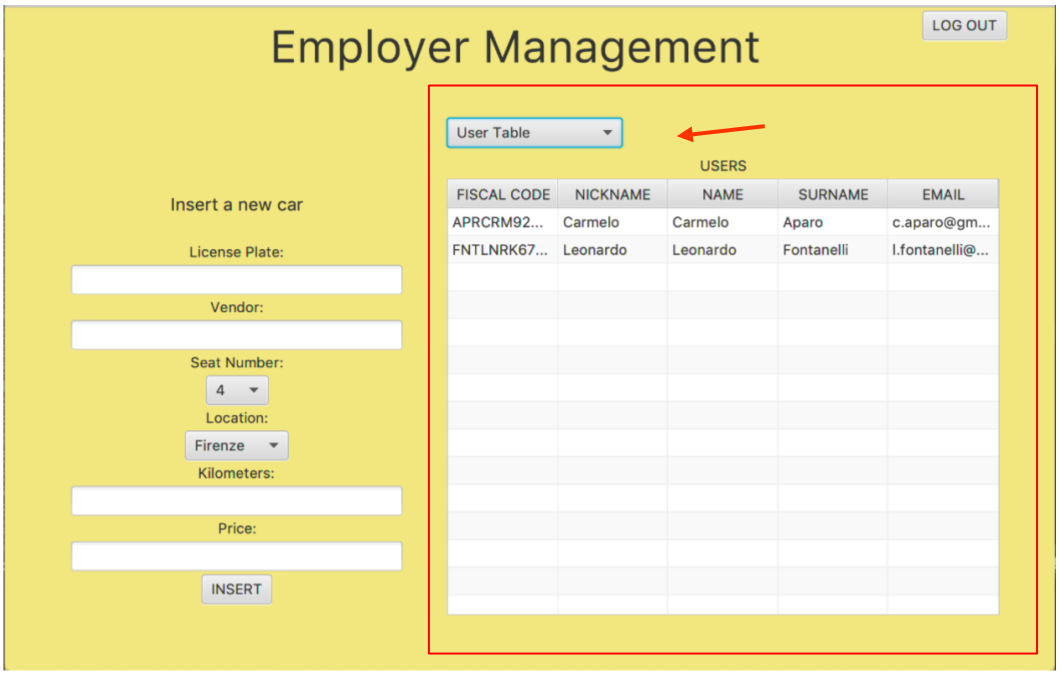
LICENS...	VENDOR	SEAT N...	LOCATI...	KM	DAY PRI...	REMOVED
GH233LG	Fiat	4	Firenze	23.0	7.0	false

An employer can visualize different type of data (Cars, Customers, Feedbacks, Reservations), depending on the information he would like to retrieve, simply choosing the desired view (table) from a "drop down menu":



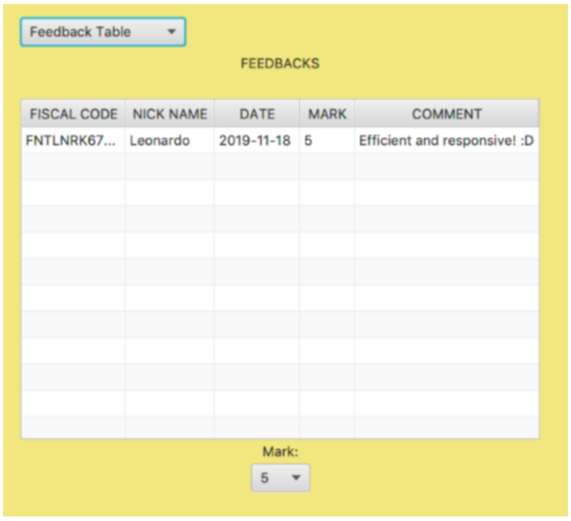
In particular:

- choosing "User Table" an Employer can see all the registered customers



FISCAL CODE	NICKNAME	NAME	SURNAME	EMAIL
APRCRM92...	Carmelo	Carmelo	Aparo	c.aparo@gm...
FNTLNRK67...	Leonardo	Leonardo	Fontanelli	l.fontanelli@...

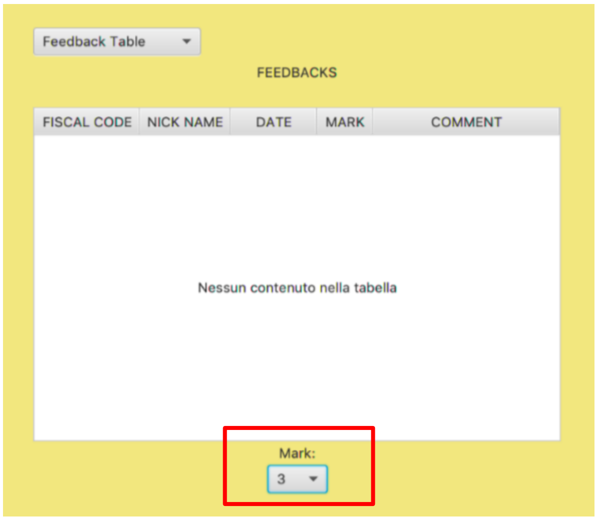
- choosing "Feedback Table" an Employer can see all the received service's feedbacks from the customers, or filter them by selecting a MARK through a "drop down menu":


Feedback Table

FEEDBACKS

FISCAL CODE	NICK NAME	DATE	MARK	COMMENT
FNTLNRK67...	Leonardo	2019-11-18	5	Efficient and responsive! :D

Mark: 5


Feedback Table

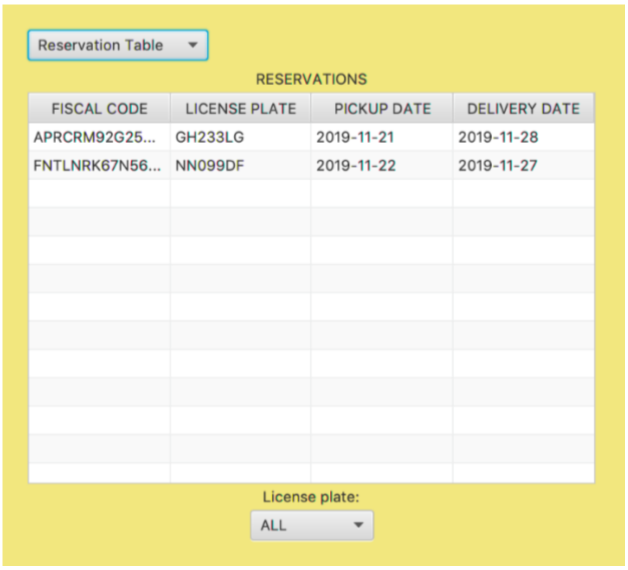
FEEDBACKS

FISCAL CODE	NICK NAME	DATE	MARK	COMMENT

Nessun contenuto nella tabella

Mark: 3

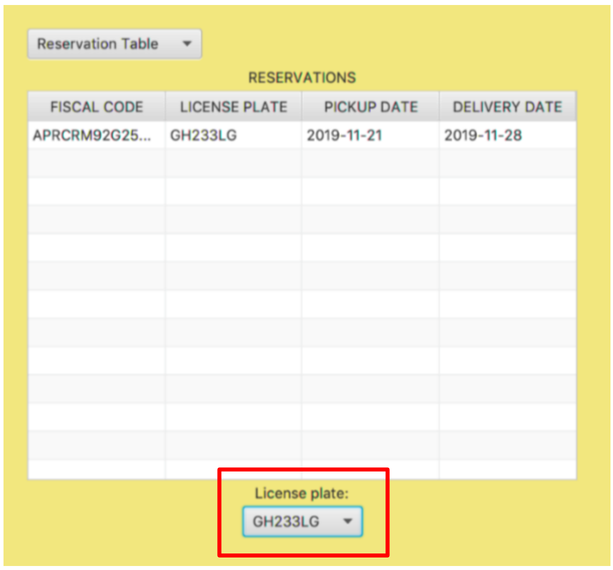
- choosing “Reservation Table” is possible to consult ALL the registered reservations or filter them by “LICENSE PLATE”, selecting one between the available cars showed by the “drop down menu”, so that is possible to retrieve only the ones for a certain car.


Reservation Table

RESERVATIONS

FISCAL CODE	LICENSE PLATE	PICKUP DATE	DELIVERY DATE
APRCRM92G25...	GH233LG	2019-11-21	2019-11-28
FNTLNRK67N56...	NN099DF	2019-11-22	2019-11-27

License plate: ALL

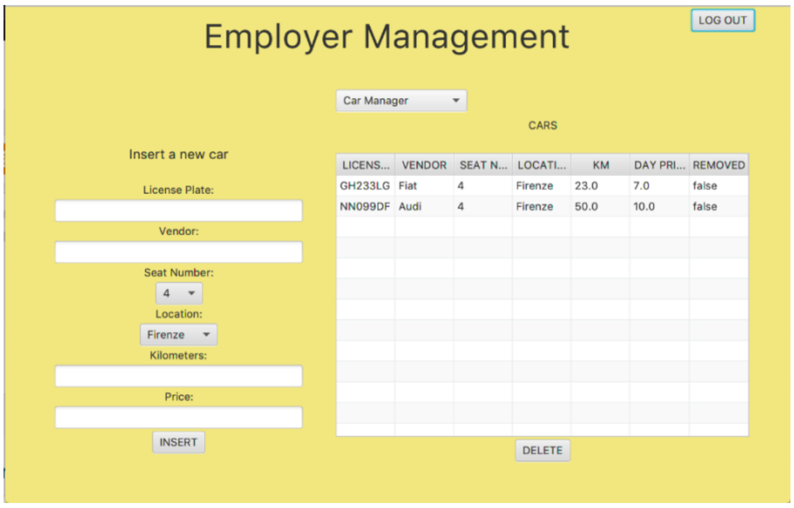

Reservation Table

RESERVATIONS

FISCAL CODE	LICENSE PLATE	PICKUP DATE	DELIVERY DATE
APRCRM92G25...	GH233LG	2019-11-21	2019-11-28

License plate: GH233LG

- choosing “Car Manager” an employer can come back to the view of all the cars available.
- He/She can LogOut in any moment, simply clicking the “LOG OUT” button.



LICENS...	VENDOR	SEAT N...	LOCATI...	KM	DAY PRI...	REMOVED
GH233LG	Fiat	4	Firenze	23.0	7.0	false
NN099DF	Audi	4	Firenze	50.0	10.0	false

Feasibility study

Introduction

Our Car Renting Company is a startup company that aims to expand itself in terms of number of customers and offered services.
At this time it hasn't a so huge amount of data to handle so a classic relational database seems to be enough to deal with them, but with their increase the system has to guarantee the best service as possible, maintaining good performances in terms of responsiveness and reliability for the customers side, scalability and simplicity in managing the system for the Employers side.
So, we want to analyze the main features of a key-value database in order to evaluate the feasibility of introducing this in our system and improve the previous aspects.

Some of the goals of a key-value database are:

1. Simplicity
2. Speed
3. Scalability

The key-value's schema-less structure makes it very simple. There's no need to a-priori define attributes and type for each of them.
This is very interesting for our application because in the case that the Car Renting company would update something in the future, with a relational database it could be necessary to change or add fields in the tables, changing their schemas. With a key-value database instead, we can directly work on code with no need to modify database structure.
However, the fact that there's no definition of the types, can be considered a disadvantage in some cases given that it could happen that a data, escaping from a control, could then be "wrong" for the field to which it refers (i.e. assigning a string to a “price” field that should be a double type).
There's another bad thing for this schema-less structure:
The key-value database does not check the uniqueness of the fields so more attention is needed inserting new values, in order to avoid duplicates. The relational one instead, provides more functionality from this point of view, “automatizing” this check using the "unique" attribute on the interested fields.

By the way, paying attention on these aspects, we could benefit from others key value features, improving performances.
For example, this model is particularly suitable when there are portions of the dataset that are frequently requested. Key-value can be useful in order to create a sort of cache to store only those data that are more needed, instead of the whole tables, so that the reading operations could potentially be faster.
In our application, for example, there are some data that are always needed, every time it is started: the Feedback table shown in the Customer Interface requires that feedback data are fetched at each customer's login and every time a filter is applied on their mark.
In the same way, in the Employer Interface, the whole informations about cars, users, reservations and feedback are frequently shown so a key value database could speed up their recovery, for the reasons previously explained.

Another important aspect for our application is to guarantee scalability to the system.
A key-value model allows to easily scale up in a horizontal manner if the load of the system would increase. It is possible to use different approaches for the servers architecture and each one could be more or less appreciated depending on the operations performed (growing demand for reading or writing operations).
In our case, the amount of reading operations is higher than the one of writing. This would lead to consider the master-slave as the best solution for a server architecture. Nevertheless, if the failure of the master occurs, we would lose the ability to make writing operations, so we could register reservations no more. In this way we would lose the main function of our application. Therefore, the best solution could be to implement a message mechanism between the slaves in such a way as to be aware of a possible failure of the master.

Key value offers also the possibility to have data partitioning (or sharding) and this could be useful if we would balance the workload through the different branches.

Table translation

The biggest limitation of a key value model is that it supports only very simple operations on it, no complex queries are allowed because all data are retrieved only by keys. So, first of all, before applying this model to our system, is very important to analyze which kind of operations are more frequently performed on the data and eventually to think about the best way to construct a key in order to easily retrieve them.

In particular, in our case, we are dealing with four types of objects: User, Feedback, Car and Reservation.

User Table

The User Table as the following attributes:

- *fiscalCode*
- *nickName*
- *name*
- *surname*
- *customer*
- *email*
- *password*

A possible translation from MySQL table to the KV database could be arranged exploiting the fiscalCode attribute, which is the primary key.
Hence, a possible structure of the key could be:

user:$FiscalCode:nickName = $value
user:$FiscalCode:name = $value
user:$FiscalCode:surname = $value
user:$FiscalCode:customer = $value
user:$FiscalCode:email = $value
user:$FiscalCode:password = $value

However, if the DB uses a lexicographic order for key storage, is useful to add the information about the status of the user at the beginning of the key. In this way the search of the user is faster because is not necessary to browse all the keys in the DB.
A possible organization could be the following one:

customer:$FiscalCode:nickName = $value
customer:$FiscalCode:name = $value
customer:$FiscalCode:surname = $value
customer:$FiscalCode:email = $value
customer:$FiscalCode:password = $value

employer:$FiscalCode:nickName = $value
employer:$FiscalCode:name = $value
employer:$FiscalCode:surname = $value
employer:$FiscalCode:customer = $value
employer:$FiscalCode:email = $value
employer:$FiscalCode:password = $value

However, this structure would not allow to perform some necessary operations, as searching a user by email, password and status, as in the login phase, because these fields are not part of the key and in the login the fiscalCode is not available.
In fact, the login operation requires a sql query with a "where clause" that is not possible to obtain with the simpler key value, but it could be realized using indexes.
The index should be built on the interested values: email and password.

The big limitation in using key value database is given by the number of relationship that the user entity has. Building a bucket for an entity could be useful if in the same bucket, together with all its fields, are stored some other information of all the related entities. In such a way, data could be easily and quickly retrieved while in a relational database they should be retrieved using JOIN operations. By the way, when there are many relationships to maintain, the amount of related data could become huge and harder to manage.
In this case User Table has two of them:

- one-to-many with the feedback entity
- one-to-many with the reservation entity

In addition, Reservation entity has another relation with Car entity, so also the latter one should be inserted into the bucket, increasing the amount of data to store even more.

For these reasons we decide to not implement this in Key Value model.

Feedback Table

The Feedback Table has the following attributes:

- *Feedback ID*
- *Mark*
- *Comment*
- *Date*

In this case we have to consider that there's a relationship between each feedback and a user. So, the FiscalCode of the user must be included in the key. A possible translation from MySQL table to the KV database could be arranged exploiting the Feedback ID attribute, which is the primary key and it was an auto-incremental integer on MySQL.
Hence, a possible structure of the key could be:

feedb:$ID:$FiscalCode:mark = $value
feedb:$ID:$FiscalCode:comment = $value
feedb:$ID:$FiscalCode:date = $value
feedb:$ID:$FiscalCode:nickname = $value

Even in this case the $ID attribute should be an incremental integer as well.

Due to the relationship between feedback and user, the key is composed by the term mentioned before, so if we would like to take some specific feedback fields, we should know both feedbackID and user FiscalCode. This last attribute is known for a logged user because these data are stored in the app when customer log in.

By adding the nickname value, we can easily resolve the previous relationship, so we don't need anymore the join operations with the user table.
So, when the application needs to show feedbacks data, including this user attribute, it will retrieve all the data easily.
In addition, for feedback's attributes an eventual consistency is enough, because even if the customer nickname would be updated in the user table (in the relational database), it is not important to update it immediately also in the feedback bucket.

In case of large-scale database, implementing the Feedback bucket in this way, it could be easily distributed because of the schema-less structure of this and because it is independent from the rest of the dataset.

However, again, with the simpler implementation of the key value model, we will have problems performing filter operation on feedbacks. Indeed, in our application we need to be able to extract feedbacks by mark and this is not possible using only the previous keys.
In a lexicographic ordered database, we can improve the research by adding the mark in the key. In this way is not necessary browse all the keys when we want the feedbacks associated to a specific mark. In the other hand this could be a limitation because in order to retrieve one of them we have to know its marks.

So, we can overcome this limitation by building indexes on the mark value and in this way retrieving data would become faster.

Car Table

The car Table has the following column:

- *Licence Plate*
- *Kilometers*
- *Location*
- *Price*
- *Removed*
- *SeatNumber*
- *Vendor*

Regarding a possible translation from MySQL to the KV database, the KV for this table could be arranged with the respect to the License Plate attribute, which is the primary key.
Hence, a possible structure of the key could be:

car:$LicencePlate:Kilometers = $value
car:$LicencePlate:Location = $value
car:$LicencePlate:Price = $value
car:$LicencePlate:Removed = $value
car:$LicencePlate:SeatNumber = $value
car:$LicencePlate:Vendor = $value

In this way we could be able to retrieve in a very easy way every attribute for an entry of the Car table. Hence, some simple query, like in our case, can be compatible with the translation of the table from MySQL to KV database. However, as we can see from the following section, the problem arises with the relation with the Reservation table (as we saw in the user case) and the query related to search of available cars.

Reservation Table

The Reservation Table has the following column:

- *Reservation id*
- *DeliveryDate*
- *PickUpDate*
- *car_LicencePlate*
- *user_FiscalCode*

As we can see from the previous list, we have 2 relations with 2 other tables: Car and User. This creates problem dealing with the KV translation. In addition, we perform a query in order to retrieve the list of cars that have not been reserved in a certain period, in a desired location, for a certain number of seats.
The mentioned query is the following:
“SELECT c FROM Car c WHERE c.location = :location AND c.seatNumber = :seatNumber AND c.removed = false AND c.licensePlate NOT IN (SELECT r.car FROM Reservation r WHERE (r.pickUpDate BETWEEN :pickUpDate AND :deliveryDate) OR (r.deliveryDate BETWEEN :pickUpDate AND :deliveryDate) OR (pickUpDate < :pickUpDate AND deliveryDate > :deliveryDate))”.

So, the previous query makes very difficult the translation from MySQL to Key-Value database, both for the Car and Reservation tables.

In conclusion we decided to implement only the Feedback in key value because Car, User and Reservation are not suitable for this kind of database.

Database Implementation

Riak is a distributed NoSQL key-value data store that offers high availability, fault tolerance, operational simplicity, and scalability.
This Database offers many kinds of functionalities in order to handle more complex search in a very efficient way.
In our specific case, we used the Counters and Secondary Indexes functionalities.
Regarding the Counter functionality, they are a bucket-level data type that can be used by themselves, associated with a bucket/key pair. A counter's value can only be a positive integer, negative integer, or zero. So, the counter type will be associated with the specified key and will be incremented every time the same key type will be added in the database. This will help the user to don't create conflicts in data and to keep key ordered.
The second kind of functionality that has been used in our case is the Secondary Index. This enable the programmer to tag objects stored in Riak, at write time, with one or more queryable values. Those values can then be used to find multiple objects in Riak. Once tagged, you could find all objects in a Riak's bucket sharing that tag.
Secondary indexes allows two types of secondary attributes: integers and strings (binaries), allows querying by exact match or range on one index.

Thanks to the help of those Riak's services, we've translated the Feedback table, that has the following fields: Mark, Comment, Date and Fiscal Code.
The counter functionality has been helpful for distinguish the attributes of the table and to build the key to retrieve them. In our case the key was built has: "feedb:" + counter: + fiscalCode. The counter has an important role because we allow customers to leave more than one feedback, hence gave the possibility to distinguish every feedback, even if it has been left by the same user.
For example, the entries in the bucket for two different feedbacks left from two different customers could be:
"feedb:1:FNTLRD91M11C415Y:mark",
"feedb:1:FNTLRD91M11C415Y:comment",
"feedb:1:FNTLRD91M11C415Y:date"

"feedb:2:FNTFRC94S14C415X:mark",
"feedb:2:FNTFRC94S14C415X:comment",
"feedb:2:FNTFRC94S14C415X:date".

As already said, we used the secondary index functionality in order to retrieve a key with a tag that represent the attribute used to filter the keys.
In our application, more precisely, in the Employer Interface, we gave the possibility to filter the comments by mark. Consequently, we used the Secondary Index with the purpose of filtering the feedbacks by using the value of the mark as tag for the index.

More in deep, the application needs to research feedbacks that have a mark equal or less the selected value, we used the range functionalities of the secondary index. This gave us the possibility to retrieve the keys of the feedback that has the mark from 1 up to the desired one.

In JPA implementation when the system fetches a feedback, Hibernate returns also the associated user since there is a relationship between them.
With key value, this does not happen, so once the system retrieves all the feedback keys, the users' fiscal codes can be extracted and used to search for their nickname between user fields.

Therefore, a double access in reading is necessary in this case but being key value theoretically faster than Mysql, this should not affect performance and the solution thus adopted is still very simple to implement.

JPA TUTORIAL

Introduction

In this tutorial we will see how to use one to many and many to many relationships in JPA.
We will start considering two common concepts: cascading and fetching and then we will see each relationship in detail.

Cascading

When two entities have a relationship, the existence of one of them can depend on the existence of the other one.
So, there is the possibility that we are interested to execute the same operation on both the entities at the same time. For example, if we consider the relationship between the entity *Person* and the entity *Address*, the *Address* has no sense to exist if there isn't the *Person*. So, we can be interested to always delete the *Address* if we delete the *Person*.
The way to achieve this is to use cascading.
Cascading means that when we perform some action on the target entity, the same action will be applied to the associated entity.

JPA Cascade Types

All JPA-specific cascade operations are represented by the javax.persistence.CascadeType enum containing entries:

- *ALL*: it propagates all the operations from a parent to a child entity;
- *PERSIST*: it propagates only the persist operation, so when we save the *Person* entity, the *Address* entity will be also saved;
- *MERGE*: it propagates only the merge operation, so when we update the *Person* entity, the *Address* entity will be also updated;
- *REMOVE*: it propagates only the remove operation, so when we delete the *Person* entity, the *Address* entity will be also deleted;
- *REFRESH*: it propagates only the refresh operation, so when we re-read the value of the *Person* instance from the database, the value of the *Address* instance will be also re-read from the database;

- *DETACH*: it propagates only the detach operation, so when we remove the *Person* entity from the persistent context, the *Address* entity will be also removed from the persistent context.

Fetching

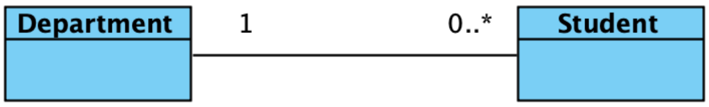
The *FetchType* attribute defines how to get the entities from the database.
There are two fetch types:

- *FetchType.EAGER*:
  it tells Hibernate to get all the elements of a relationship when selecting the root entity.
  For example, if we have a one to many relationship between the entities *Department* and *Student*, if we get the entity *Department*, Hibernate will get also all the *Student* entities associated to the *Department*.;

- *FetchType.LAZY*:
  it tells Hibernate to only fetch the related entities from the database.
  For example, if we have a one to many relationship between the entities *Department* and *Student*, if we get the entity *Department*, Hibernate will get only the *Department* entity without all the *Student* entities associated to it.

One to many relationship

Suppose that we have a one to many relationship between the entities *Department* and *Student*.



As we can see from the picture an entity *Department* is associated to many entities *Student*, instead each entity *Student* is associated to a single *Department*.

Annotations

To map a one to many relationship we can use two annotations:

- *@OneToMany* for the entity associated to many entities;
- *@ManyToOne* for the entities associated to a single entity.

So, we can use the following code to map a *Department* to many *Students*:

```
@Entity
@Table(name="Departments")
public class Department {
        @Id
        @Column(name="Id")
        private int id;
        @Column(name="Name")
        private String name;
```

```
          @OneToMany(
                              mappedBy = "department",
                              fetch=FetchType.EAGER,
                              cascade = {CascadeType.PERSIST, CascadeType.MERGE}
                              )
          private List<Student> students = new ArrayList<>();

          // Constructors, getters and setters removed for brevity
          public void addStudent(Student student) {
                    students.add(student);
                    student.setDepartement(this);
          }

          public void removeStudent(Student student) {
                    students.remove(student);
                    student.setDepartement(null);
          }
}
```

The entity contains a list of Student entities and before the list there is the annotation *@OneToMany* which contains three attributes:

- mappedBy = "department" tells Hibernate that this entity is associated to the field "department" of the entity Student;
- fetch=FetchType.EAGER tells Hibernate to get all the Student entities in the list when it gets a Department entity;
- cascade = {CascadeType.PERSIST, CascadeType.MERGE} tells Hibernate to propagate the persist and merge operations.

Instead to map each *Student* to a single *Department* we can use the following code:

```
@Entity
@Table(name="Students")
public class Student {
          @Id
          @Column(name="Id")
          int id;
          @Column(name="Name")
          String name;
          @Column(name="Surname")
          String surname;
          @ManyToOne(fetch=FetchType.EAGER)
          @JoinColumn(name="DepartmentId")
          Department department;
          // Constructors, getters and setters removed for brevity
}
```

In this case the entity *Student* has the object *Department* to which is associated.
To map the relationship, in this case there is the annotation *@ManyToOne* before the *Department* entity with the attribute fetch equal to *FetchType.EAGER*.

The annotation *@JoinColumn* tells Hibernate the name of the column that will be used as join column.

Operations

HandleDB class

This is an example of a class to manage the operations on the database:

```
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class HandleDB {
	private EntityManagerFactory factory;
	private EntityManager entityManager;

	public void setup() {
		factory = Persistence.createEntityManagerFactory("JPA_Test");
	}

	public void exit() {
		factory.close();
	}
```

Create

```
	public void create(Object o) {
		System.out.println("Add a new object");
		try {
			entityManager = factory.createEntityManager();
			entityManager.getTransaction().begin();
			entityManager.persist(o);
			entityManager.getTransaction().commit();
			System.out.println("New object added");
		} catch (Exception ex) {
			System.out.println("Exception during persist: " + ex.getMessage());
		} finally {
			entityManager.close();
		}
	}
```

Read

```
	public <T> Object read(Class<T> entity, Object id) {
		System.out.println("Getting a new object");
		Object o = null;
```

```
        try {
            entityManager = factory.createEntityManager();
            o = entityManager.find(entity, id);
        } catch (Exception ex) {
            System.out.println("Exception during read: " + ex.getMessage());
        } finally {
            entityManager.close();
        }
        return o;
    }
```

Update

```
    public void update(Object o) {
        System.out.println("Update an object");
        try {
            entityManager = factory.createEntityManager();
            entityManager.getTransaction().begin();
            entityManager.merge(o);
            entityManager.getTransaction().commit();
        } catch(Exception ex) {
            System.out.println("Exception during update: " + ex.getMessage());
        } finally {
            entityManager.close();
        }
    }
```

Delete

```
    public void deleteStudent(Student student) {
        System.out.println("Delete an object");
        try {
            entityManager = factory.createEntityManager();
            entityManager.getTransaction().begin();

            // You have to delete the reference of the children to the owner entity
            Department department = student.getDepartment();
            department.getStudents().remove(student);
            entityManager.merge(department);

            Student s = entityManager.getReference(Student.class, student.getId());
            entityManager.remove(s);
            entityManager.getTransaction().commit();
        } catch(Exception ex) {
            System.out.println("Exception during delete: " + ex.getMessage());
        } finally {
            entityManager.close();
        }
    }
```

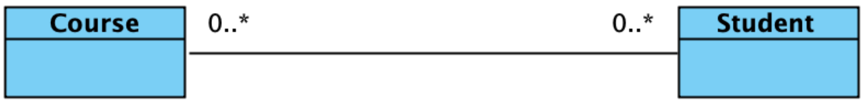
```
public void deleteDepartment(Department department) {
		System.out.println("Delete an object");
		try {
			entityManager = factory.createEntityManager();
			entityManager.getTransaction().begin();

			// You have to delete the children from the owner entity
			List<Student> students = department.getStudents();
			for(int i = 0; i < students.size(); i++) {
				Student student = students.get(i);
				student.setDepartement(null);
				entityManager.merge(student);
			}

			Department d = entityManager.getReference(Department.class,
				department.getId());
			entityManager.remove(d);
			entityManager.getTransaction().commit();
		} catch(Exception ex) {
			System.out.println("Exception during delete: " + ex.getMessage());
		} finally {
			entityManager.close();
		}
}
```

Since we don't use the *CascadeType.REMOVE* option, when we delete an entity, we have to remove the references of the associated entities in order to not have inconsistencies.
So, before removing a *Student*, we have to remove the *Student* from the list of *Student* entities of the *Department* associated.

If we use the *CascadeType.REMOVE* option, we can simply call the function remove on a *Department* entity and Hibernate will delete all the *Student* entities associated to it.

Many to many relationship

Suppose that we have a many to many relationship between the entities *Course* and *Student*.



As we can see from the picture an entity *Course* is associated to many entities *Student* and each entity *Student* is associated to many entities *Course*.

Annotations

To map a one to many relationship, we can use the annotation @ManyToMany on both the entities.

So, we can use the following code for the two entities:

```
@Entity
@Table(name="Courses")
public class Course {
        @Id
        @Column(name="Id")
        int id;
        @Column(name="Name")
        String name;
        @ManyToMany(mappedBy = "courses", cascade = CascadeType.MERGE, fetch=FetchType.EAGER)
        List<Student> students = new ArrayList<>();

        // Constructors, setters and getters removed for brevity

public void addStudent(Student student) {
                students.add(student);
                student.getCourses().add(this);
        }

        public void removeStudent(Student student) {
                students.remove(student);
                student.getCourses().remove(this);
        }
}

@Entity
@Table(name="Students")
public class Student {
        @Id
        @Column(name="Id")
        int id;
        @Column(name="Name")
        String name;
        @Column(name="Surname")
        String surname;
        @ManyToMany(fetch=FetchType.EAGER)
        List<Course> courses = new ArrayList<>();

        // Constructors, setters and getters removed for brevity
        public void addCourse(Course course) {
                courses.add(course);
```

```
		course.getStudents().add(this);
	}

	public void removeCourse(Course course) {
		courses.remove(course);
		course.getStudents().remove(this);
	}
}
```

Both the entities have a list that contains the entities to which it is associated.
Before the list there is the annotation @ManyToMany used by Hibernate to map the relationship and the attributes specified within the annotations are the same of the one to many relationship.

Operations

We can use the same HandleDB class also for this relationship, in fact the create, read and update operations are the same as before.

The only operation that is different is the delete, since when we want to remove an entity, we need to remove the reference of this entity on all the other entities.
For example, if we want to remove a *Course*, we need to remove it from the list of each *Student* that follows the Course.
The following one is a possible code to delete a *Course* and a *Student*:

```
	public void deleteCourse(Course course) {
		System.out.println("Delete an object");
		try {
			entityManager = factory.createEntityManager();
			entityManager.getTransaction().begin();

			// You have to remove the course from the list of courses of the students
			List<Student> students = course.getStudents();
			for(int i = 0; i < students.size(); i++) {
				Student student = students.get(i);
				student.getCourses().remove(course);
				entityManager.merge(student);
			}

			Course c = entityManager.getReference(Course.class, course.getId());
			entityManager.remove(c);
			entityManager.getTransaction().commit();
		} catch(Exception ex) {
			System.out.println("Exception during delete: " + ex.getMessage());
		} finally {
			entityManager.close();
		}
	}
```

```
public void deleteStudent(Student student) {
        System.out.println("Delete an object");
        try {
                entityManager = factory.createEntityManager();
                entityManager.getTransaction().begin();

                // You have to remove the student from the list of students of the courses
                List<Course> courses = student.getCourses();
                for(int i = 0; i < courses.size(); i++) {
                        Course course = courses.get(i);
                        course.getStudents().remove(student);
                        entityManager.merge(course);
                }

                Student s = entityManager.getReference(Student.class, student.getId());
                entityManager.remove(s);
                entityManager.getTransaction().commit();
        } catch(Exception ex) {
                System.out.println("Exception during delete: " + ex.getMessage());
        } finally {
                entityManager.close();
        }
}
```

Also in this case, if we use the *CascadeType.REMOVE* option, we can simply call the function remove on a *Course* entity and Hibernate will delete all the *Student* entities associated to it.