myFood

Large Scale and Multi-Structured Data Bases – Workgroup Task 1

University of Pisa

Academic Year 2019-2020

Barsellotti Luca

Gholami Sina

Pacini Federico

Paolini Emilio

1 CONTENTS

2 ANALYSIS PHASE

2.1 DESCRIPTION

myFood is an application designed for customers and restaurant managers. Both of them can register to the application and then use its services.

A customer can access to the restaurants list, view a restaurant menu, make an order by choosing foods from the menu and add them to a cart. Then, she/he will confirm the order. A customer can browse orders her/his orders to check the details.

Restaurant Manager can create her/his own menu, adding, removing and modifying foods. She/he can check statistics of her/his own restaurant and view the orders made by the customers.

2.2 MAIN ACTORS

There are two main actors: **customers** and **restaurant managers**. At first, both of them must **register** or **login** to the application. The information required to register as customer are:

- Name
- Birth date
- Email
- Address
- Phone
- Bank account
- password

The information needed to register as restaurant manager are:

- Restaurant name
- Description
- Email
- Address
- Phone
- Bank account
- password

The customer has three scenarios:

1- **Selecting Restaurant Scenario**, where the system shows the list of registered restaurants and where the customer can select one of them.
2- **Showing Menu Scenario**, where the system shows the list of foods available in the menu of a selected restaurant and the customer can add or remove food from his/her cart.
3- **My Orders Scenario**, where the system shows the list of his/her past orders and the customer can check the details of an order by selecting it.

The restaurant manager has three scenarios:

1- **Main Menu Scenario**, where the system shows the list of the foods available in the restaurant to the restaurant manager and where the restaurant manager can add, remove and edit the chosen food in the menu.

2- **Statistics Scenario**, where the system shows the total sold pieces, total earnings, total customers and average earning per customer calculated on stored information.
3- **Orders Scenario**, where the system shows the list of orders submitted by customers in that restaurant.

2.3 REQUIREMENTS

2.3.1.1 Functional Requirements

General user requirements:

- If the user is already registered in the application, then she/he must have the capability of inserting her/his email, password and press a Login button for login. Otherwise, there must be a Register Button for registration.
- After pressing the Register button, the system must show to the user a set of fields to insert information. There must be a button to confirm the registration and proceed in the application.
- There must be a Logout button that allows the user to exit from her/his account.

Customer requirements:

- There must be a section where the customer shall view the restaurant's menu through the application and there must be also a search field that allows the customer to search a specified restaurant.
- After selecting a restaurant, the system must show to the customer a list of food available in the selected restaurant (restaurant menu) and a cart. The customer shall have the possibility to add foods from the menu to the cart or remove food from the cart. There must be a Next button that allows the customer to submit the order and a Back button to return to the list of restaurants.
- There must be a section where the customer can look at all the orders that he have made and, clicking on one of them, he shall look at the order bill.

Restaurant manager requirements:

- There must be a section where the restaurant manager can edit their menu, with a list of the food that can be updated or deleted and two fields that allows him to insert the name and the price of a new food by clicking on an Insert button.
- There must be a section where the restaurant manager shall look at all orders related to his restaurant and, clicking on one of them, she/he shall check the order bill.
- There must be a section where the restaurant manager shall look at her/his restaurant statistics, like total sold pieces, total earnings, average earnings per customer and total customers.

2.3.1.2 Non-Functional Requirements

Non-functional requirements are:

- **Usability**: the operations required to achieve a specific task must be easy to understand for a general user.
- **Readability**: the code must be readable and modulated because in this way it's easier to add new features and to resolve eventual errors.
- **Security**: the software must be used only as intended.
- **Performance:** the software must have good performance and limited response time.
- **Scalability**: the software must be able to handle a high transactional volume.

2.4 USE CASE DIAGRAM

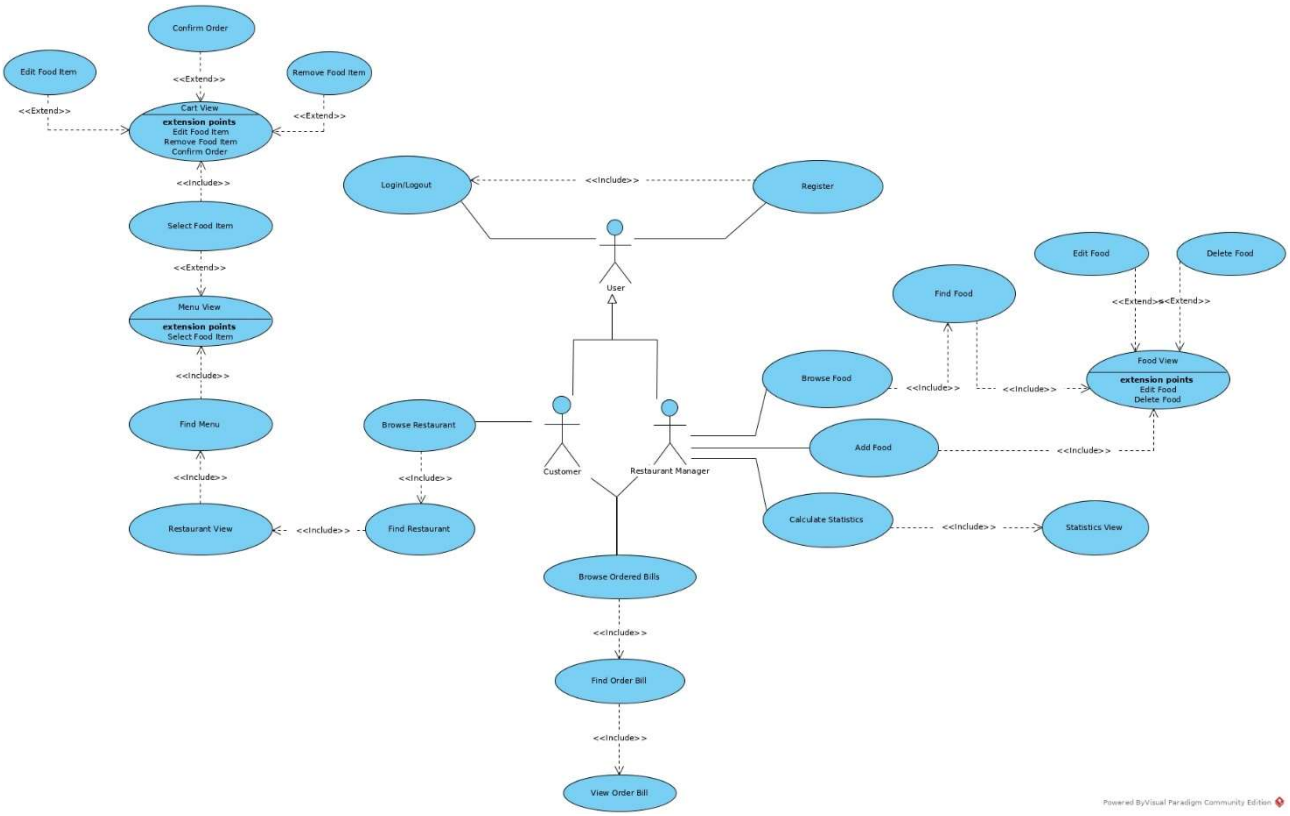

Figure 1: UML Diagram for Use Case.

3 DESIGN PHASE

3.1 DYNAMIC VIEW

The aim of the Dynamic View is to figure out how the application will approximately work. It's based on Scenarios which represent the various application cases where the flow of application will be explained using pseudo-code and mockup, which are snapshot of

3.1.1 Customer Dynamic View

First Screen Scenario

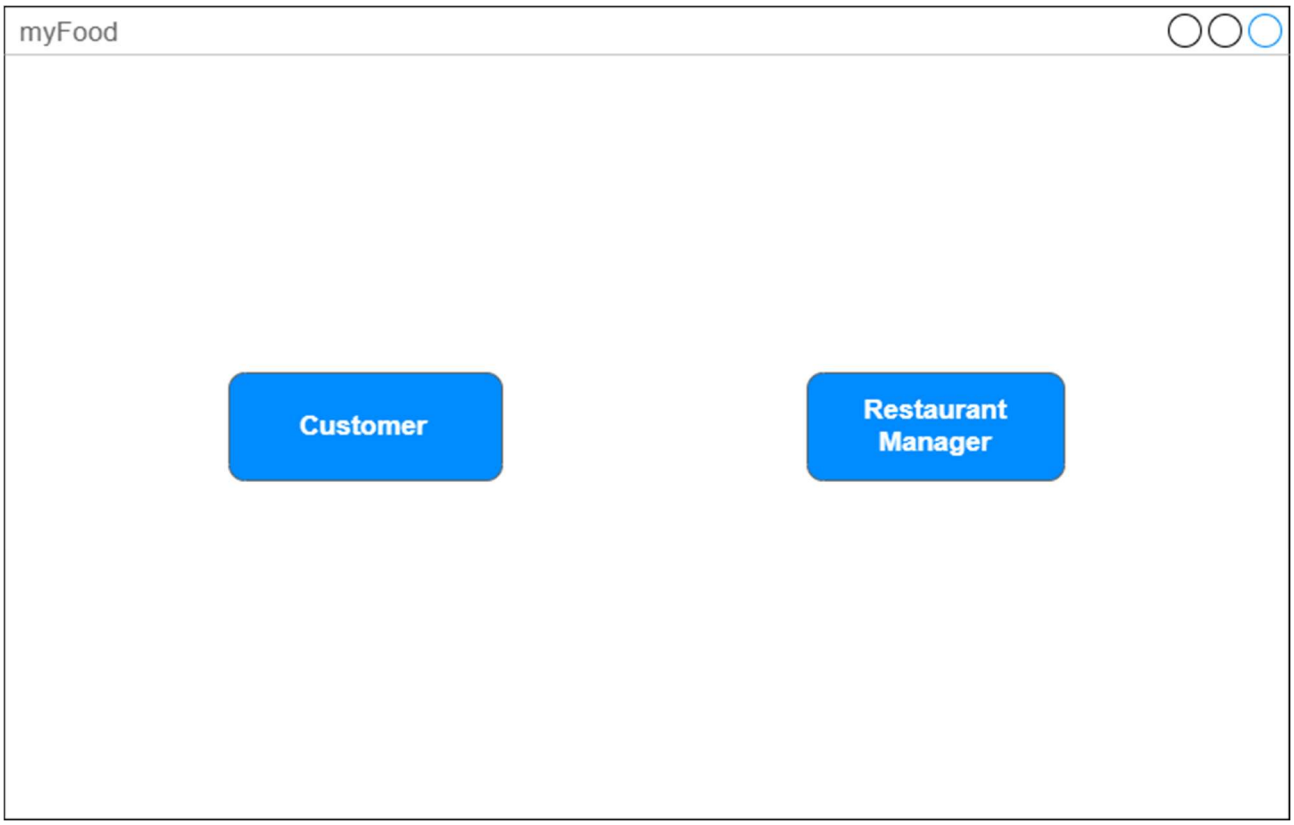

Figure 2: First screen scenario mock-up.

1. The system shows two buttons: *customer, restaurant manager*
2. IF the user clicks on the *customer*
   2.1. GOTO *Customer Login Scenario*
3. IF the user clicks on the *restaurant manager*
   3.1. GOTO *Restaurant Manager Scenario*

Customer Login Scenario

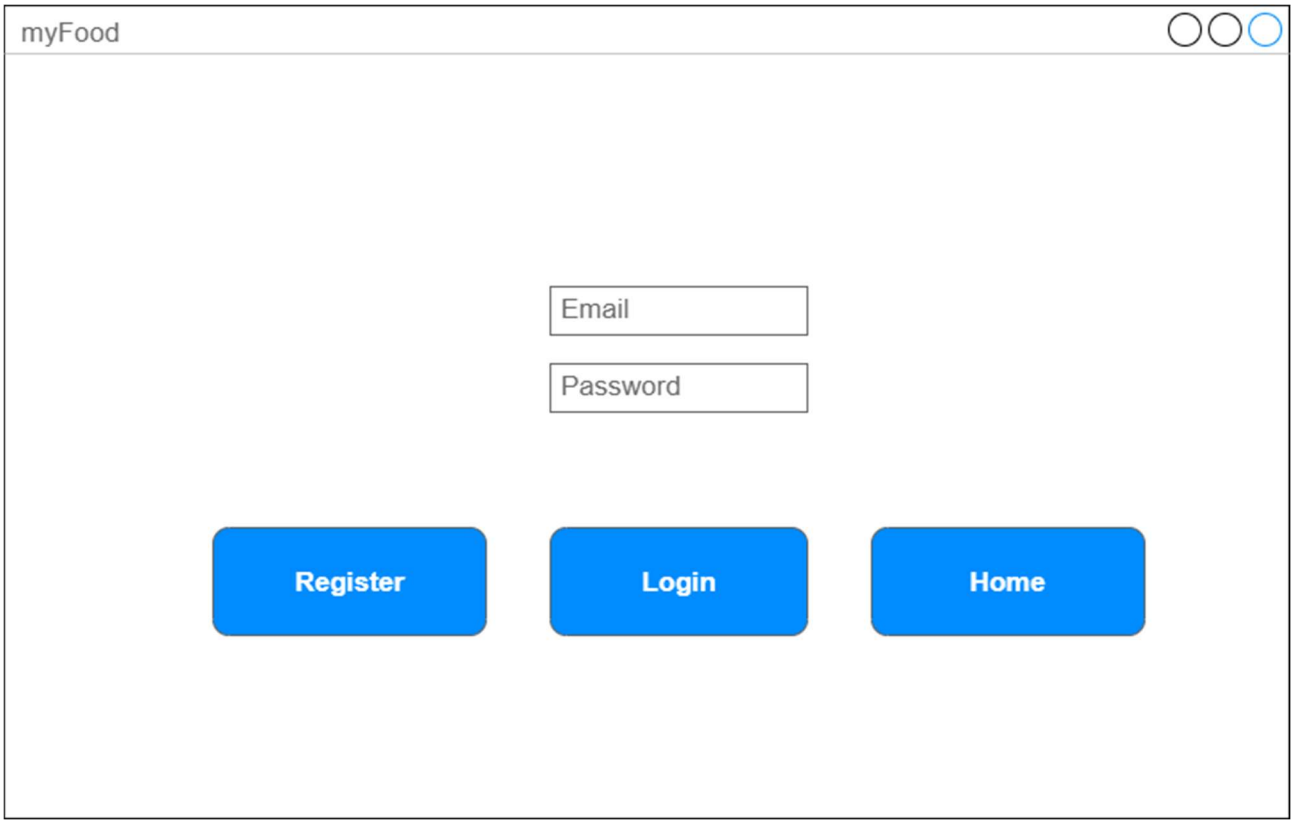

Figure 3: Customer login scenario mock-up.

1. The system shows the following items: *email* text field, password text field
2. The system shows three buttons: *register, login, home*
3. The customer inserts email and password
4. IF the customer clicks on *login button*
   4.1. IF credentials are right
      4.1.1. *GOTO Selecting Restaurant Scenario*
   4.2. ELSE
      4.2.1. The system shows an error message
5. IF the customer clicks on *register button*
   5.1. *GOTO Customer Registration Scenario*
6. IF the customer clicks on *home button*
   6.1. GOTO *First Screen Scenario*

Customer Registration Scenario

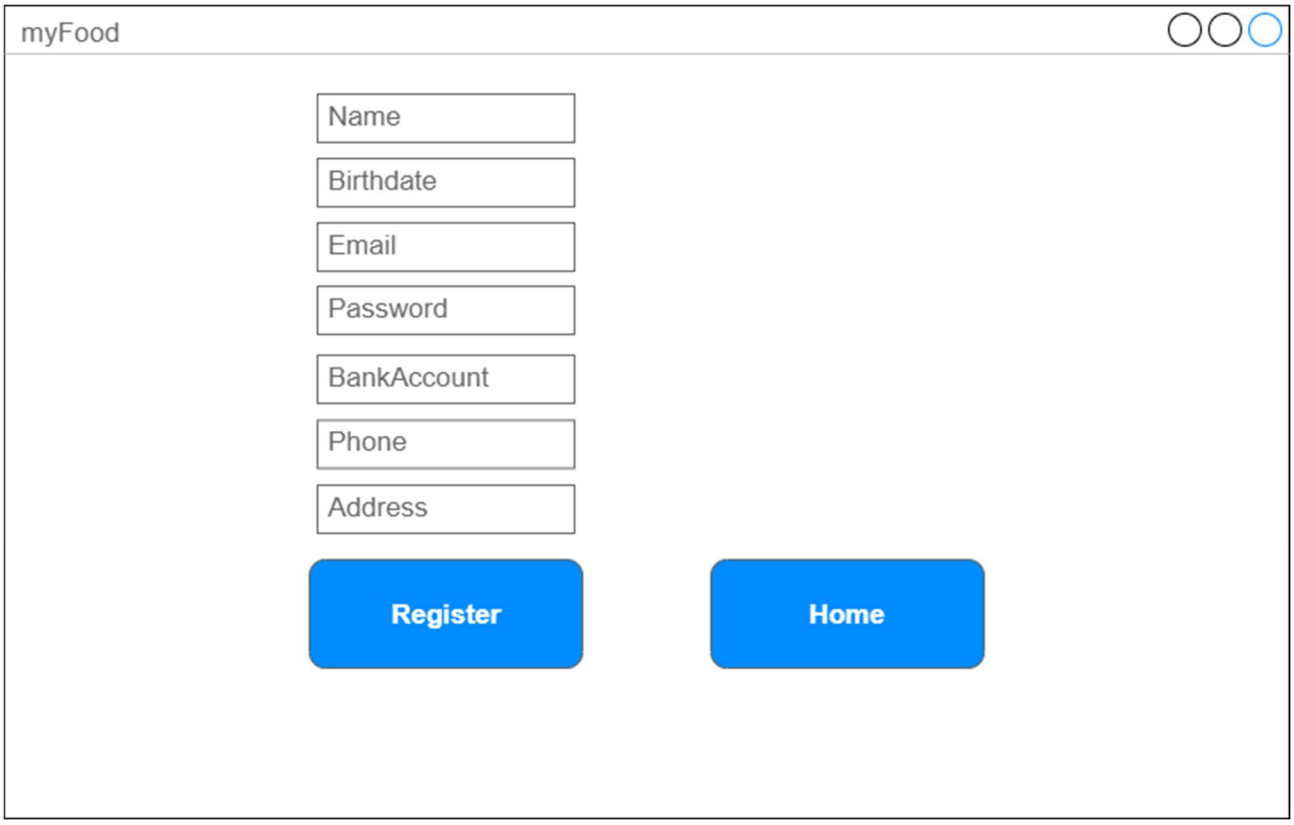

Figure 4: Customer registration scenario mock-up.

1. The system loads the following items: *name* text field, *birthdate* text field, *email* text field, *password* text field, *bank account* text field, phone text field, address text field
2. The system shows three buttons: *register*, *home*
3. The customer inserts *name*, *birthdate*, *email*, *password*, *bank account, phone, address*
4. IF the customer clicks *register button*
   4.1. IF email is not already used by another customer and other fields are not empty the system registers the new customer
      4.1.1. *GOTO Selecting Restaurant Scenario*
   4.2. ELSE
      4.2.1. The system shows an error message
5. IF the customer clicks *home button*
   5.1. *GOTO First Screen Scenario*

Selecting Restaurant Scenario

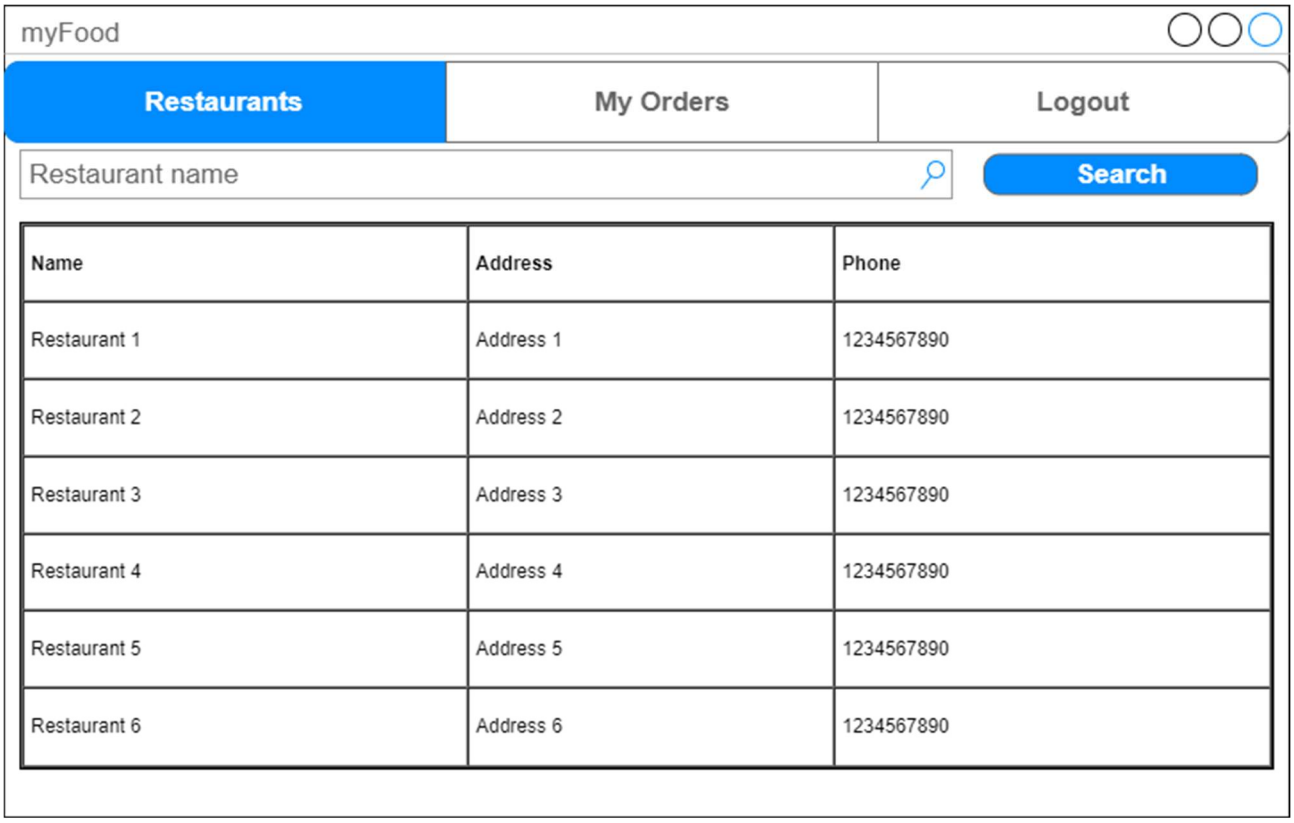

Figure 5: Selecting restaurant scenario mock-up.

1. The system shows a top bar containing three buttons: *Restaurants, My orders, Logout*. The restaurant button is clicked by default.
2. The system loads the following items: *restaurant name text field, search button*
3. IF the customer clicks *search button*
   3.1. The System shows the possible restaurants matching the inserted restaurant name
4. FOR EACH restaurant inside the application
   4.1. The system loads a button containing: *name*, *address*, *phone*
5. IF the customer moves the mouse over a restaurant button
   5.1. The system shows the restaurant description
6. IF the customer clicks on one restaurant
   6.1. GOTO *Showing Menu Scenario*
7. IF the customer clicks the *My Orders* Button
   7.1. *GOTO My Orders Scenario*
8. IF the customer clicks the *Logout* button
   8.1. *GOTO Customer Login Scenario*

Showing Menu Scenario

Figure 6: Showing menu scenario mock-up.

1. The system shows an empty cart
2. The restaurant name text field becomes not editable, with the name of the selected restaurant in it. The *search button* becomes a *back button*
3. IF the customer clicks the *back button*
   3.1. GOTO *Selecting Restaurant Scenario*
4. FOR EACH food in Menu
   4.1. The system loads a button containing: *name*, *the unitary price*
5. The system shows a disabled next button,
6. IF the customer clicks on a food
   6.1. The system adds one unit of the food to the cart
   6.2. The system shows next to the food a – button
7. IF the cart is not empty
   7.1. The system enables the *next button*
8. IF the customer clicks – button
   8.1 The system removes one unit of the food
   8.2 IF the quantity of the food is 0
      8.2.1 The system removes the food from the cart
9. IF the customer clicks next button
   9.1. The system shows the *OrderBill* in a popup
   9.2. The system shows a *confirm button*
   9.3. IF the customer clicks the *confirm button*
      9.3.1. The system inserts the order in the database
      9.3.2. *GOTO Selecting Restaurant Scenario*

My Orders Scenario

Figure 7: My orders scenario mock-up.

1. The system shows a top bar containing three buttons: *Restaurants, My orders, Logout*. *My orders button* is clicked by default.
2. IF the customer clicks Restaurants button
   2.1. GOTO *Selecting Restaurant Scenario*
3. FOR EACH customer's order in the application
   3.1. The system shows a button containing: *the restaurant name*, *total price*, *date*
4. IF the customer clicks on one order
   4.1. The system loads a popup *bill*
   4.2. FOR EACH *Food* inside that *order*
      4.2.1. The system shows name, quantity, total price inside the popup
   4.3. The system shows the total price of the bill
   4.4. The system shows an *exit* button in the popup
   4.5. IF the customer clicks *exit* button
      4.5.1. The popup closes
5. IF the customer clicks *logout* button
   5.1 *GOTO Customer Login Scenario*

3.1.2 Restaurant Manager Dynamic View

First Screen Scenario

1. The system shows two buttons: *customer, restaurant manager*
2. IF the user clicks on the *customer*
   2.1. GOTO *Customer Login Scenario*
3. IF the user clicks on the *restaurant manager*
   3.1. GOTO *Restaurant Manager Scenario*

Restaurant Login Scenario

1. The system shows the following items: *email* text field, password text field
2. The system shows three buttons: *register, login, home*
3. The restaurant manager inserts email and password
4. IF the restaurant manager clicks on *login button*
   4.1. IF credentials are right
      4.1.1. *GOTO Main Menu Scenario*
   4.2. ELSE
      4.2.1. The system shows an error message
5. 5) IF the restaurant manager clicks on *register button*
   5.1. *GOTO Customer Registration Scenario*
6. IF the restaurant manager clicks on *home button*
   6.1 GOTO *First Screen Scenario*

Restaurant Registration Scenario

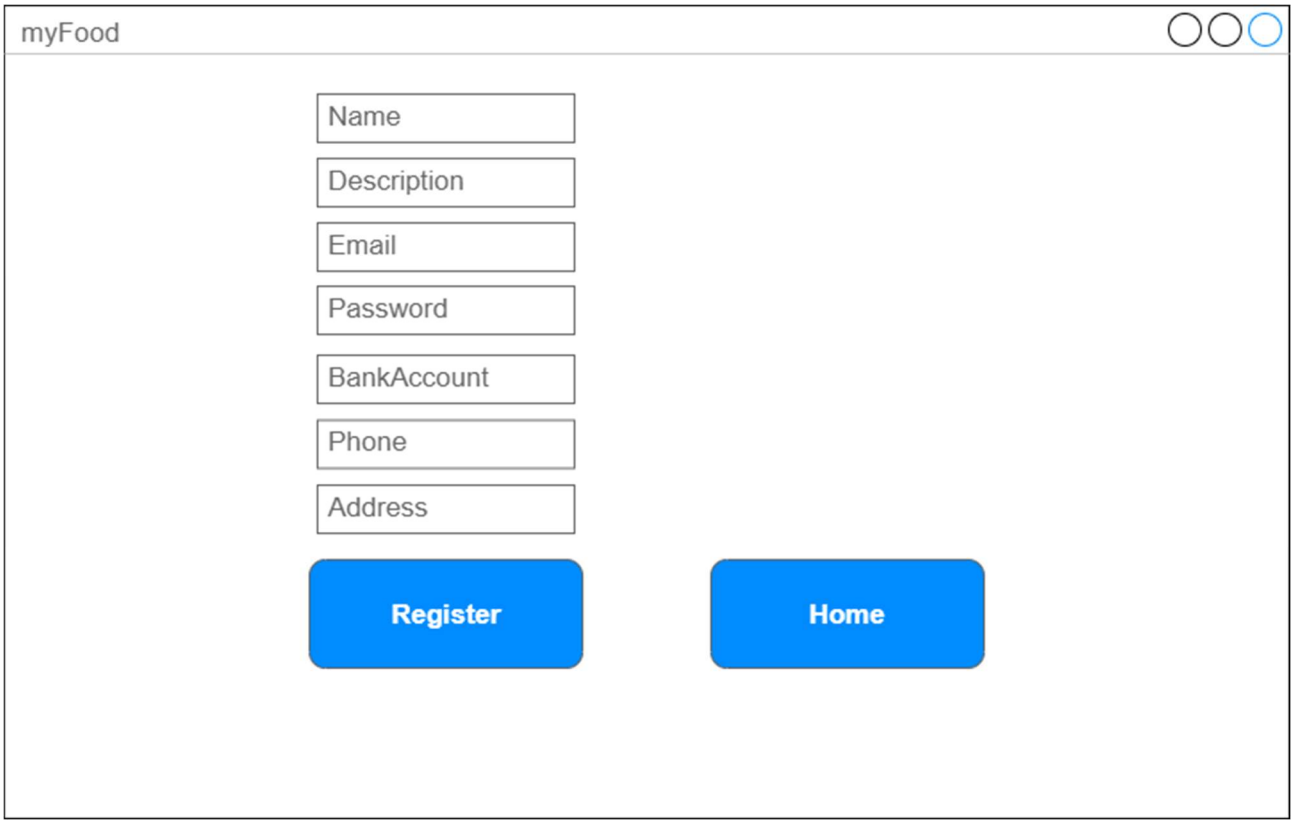

Figure 8: Restaurant manager registration scenario mock-up.

1. The system loads the following items: *name* text field, *description* text field, *email* text field, *password* text field, *bank account* text field, phone text field, address text field
2. The system shows two buttons: *register*, *home*
3. The restaurant manager inserts *name*, *description*, *email*, *password*, *bank account, phone, address*
4. IF the restaurant manager clicks *register button*
   4.1. IF email is not already used by another restaurant manager and other fields are not empty the system registers the new restaurant manager
      4.1.1. *GOTO Main Menu Scenario*
   4.2. ELSE
      4.2.1. The system shows an error message
5. IF the restaurant manager clicks *home button*
   5.1. *GOTO First Screen Scenario*

Main Menu Scenario

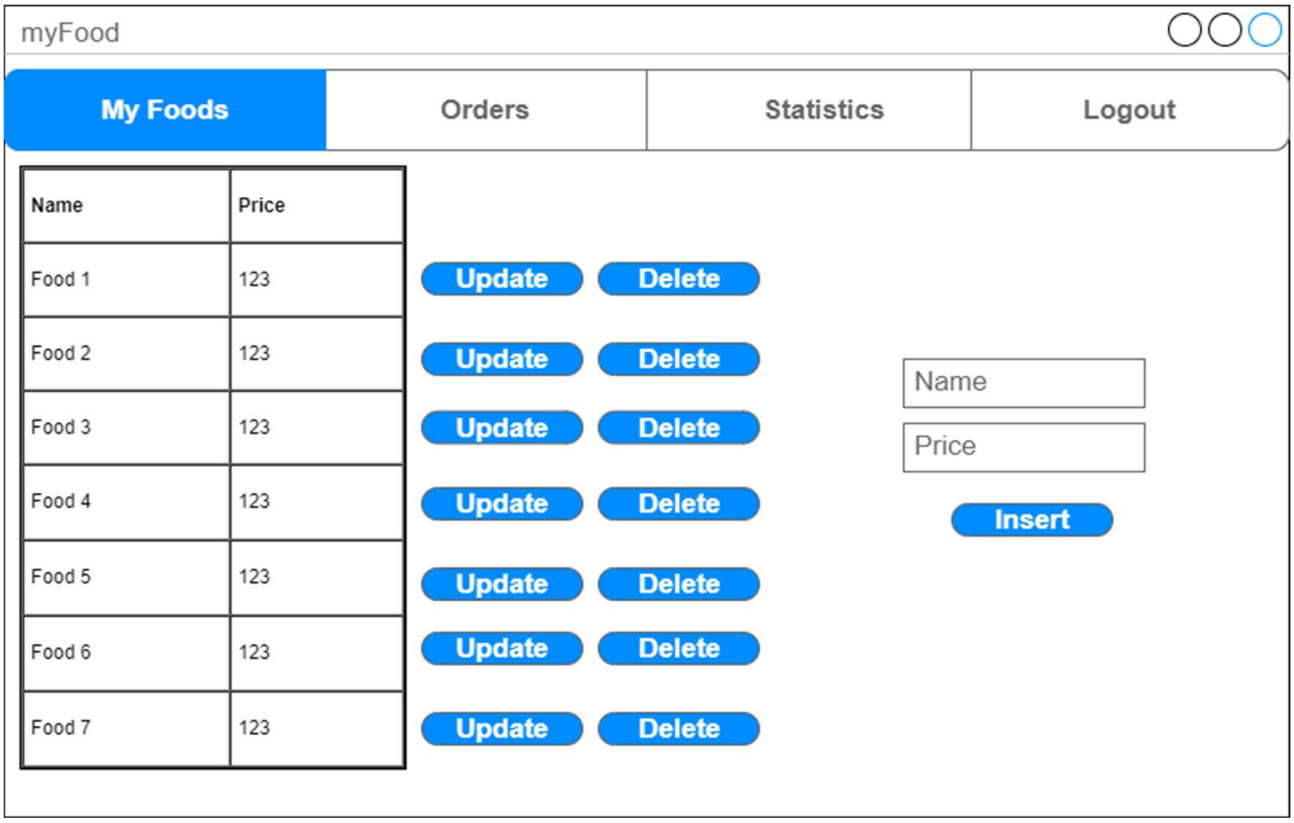

Figure 9: Main menu scenario mock-up.

1. The system shows a top bar containing four buttons: *My Foods, Orders, Statistics, Logout*. The My Foods button is clicked by default.
2. If the Restaurant Manager clicks on *Orders Button*
   2.1. GOTO Orders Scenario
3. IF the restaurant Manager clicks on Statistics Button
   3.1. GOTO *Statistics Scenario*
4. If the Restaurant Manager clicks on *Logout button*
   4.1. GOTO *Restaurant Manager Login Scenario*
5. The system loads the following items: Name text field, Price text field, *insert button*
6. IF the restaurant manager clicks the *insert button*
   6.1. IF the Name text field is not empty and the Price is a number
      6.1.1. The system inserts the food in the database
7. *FOR EACH Food* in the restaurant menu
   7.1. The system loads *name* text field, *price* text field, *update* button, *delete* button
   7.2. IF the restaurant manager clicks the update button
      7.2.1. IF the name text field is not empty and the price text field is a number
         7.2.1.1. The system updates the food in the database
   7.3. IF the restaurant manager clicks the delete button
      7.3.1. The system removes the food from the database
8. IF the Restaurant Manager clicks on the *logout button*
   8.1. GOTO Restaurant Login Scenario

Statistics Scenario

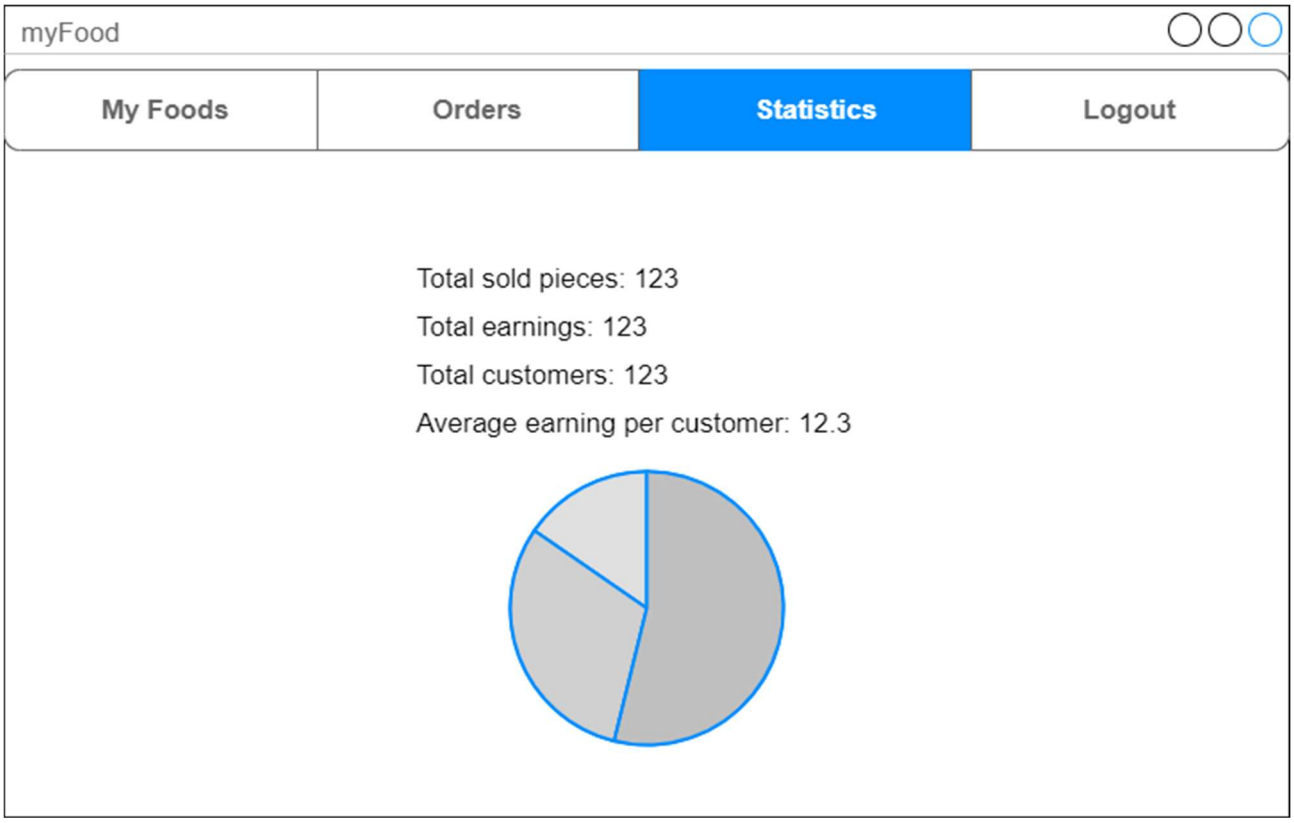

Figure 10: Statistics scenario mock-up.

1. The system shows a top bar containing four buttons: *My Foods, Orders, Statistics, Logout*. The Statistics button is clicked by default.
2. If the Restaurant Manager clicks on *Orders Button*
   2.1. GOTO Orders Scenario
3. IF the Restaurant Manager clicks on the *logout button*
   3.1. GOTO Restaurant Login Scenario
4. IF the Restaurant Manager clicks the *My Foods Button*
   4.1. GOTO *My Foods Scenario*
5. The system displays four values: *total sold pieces*, *total earnings*, *total customers*, *average earning per customer*
6. The system shows also a Pie Chart, showing the popularity of each food

Orders Scenario

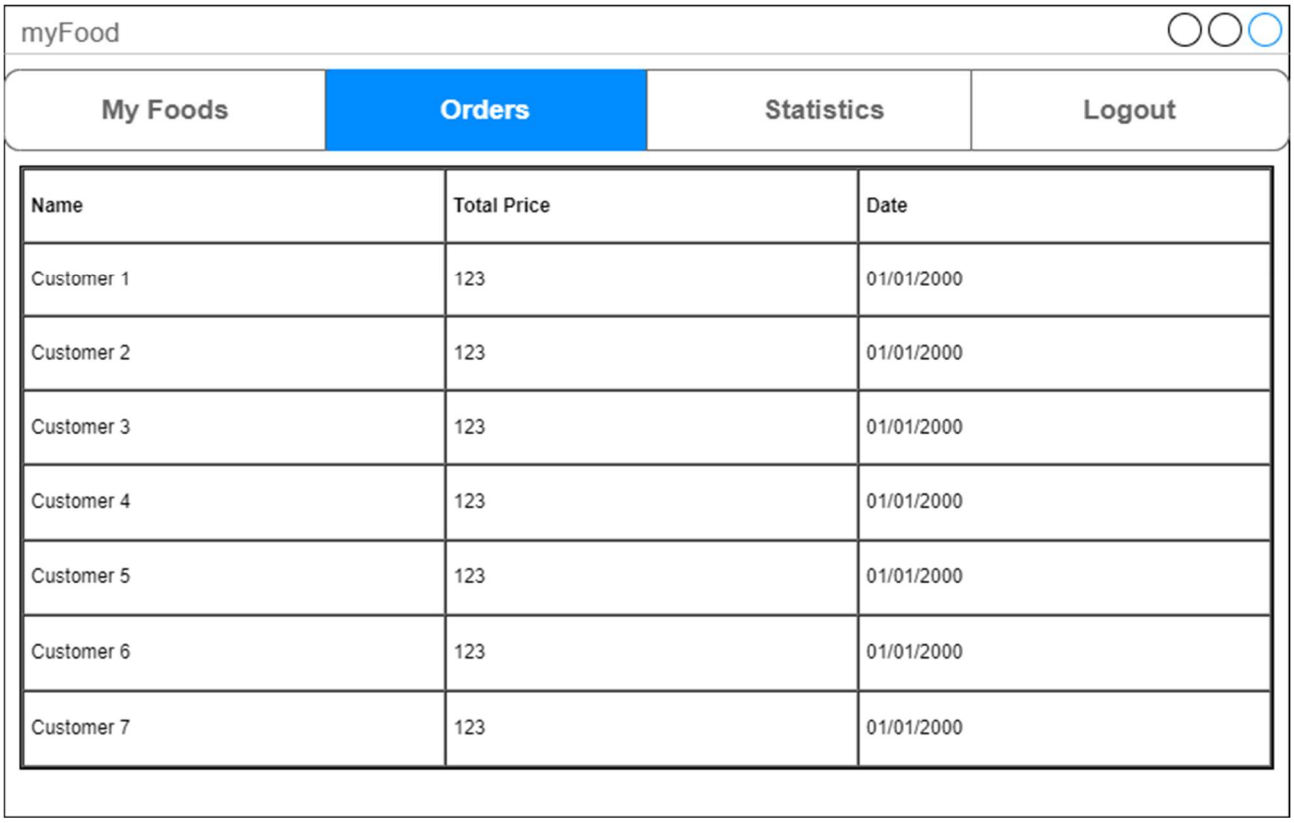

Figure 11: Orders scenario mock-up.

1. The system shows a top bar containing four buttons: *My Foods, Orders, Statistics, Logout*. The Orders button is clicked by default.
2. If the Restaurant Manager clicks on *Orders Button*
   2.1. GOTO Orders Scenario
3. IF the Restaurant Manager clicks on the *logout button*
   3.1. GOTO Restaurant Login Scenario
4. IF the Restaurant Manager clicks the *My Foods Button*
   4.1. GOTO *My Foods Scenario*
5. *FOR EACH* *OrderBill* of the restaurant in the database
   5.1. The system loads a button containing: *customer name*, *total price*, *date*
6. IF the Restaurant Manager clicks on an *order*
   6.1. The system loads a popup *bill*
   6.2. *FOR EACH* *Food* inside that *order*
      6.2.1. The system loads *name*, *quantity*, *total price* inside the popup
   6.3. The system shows the total price of the bill
   6.4. The system shows an *exit* button in the popup
   6.5. IF the customer clicks *exit* button
      6.5.1. The popup closes

3.2 Analysis Class Diagram

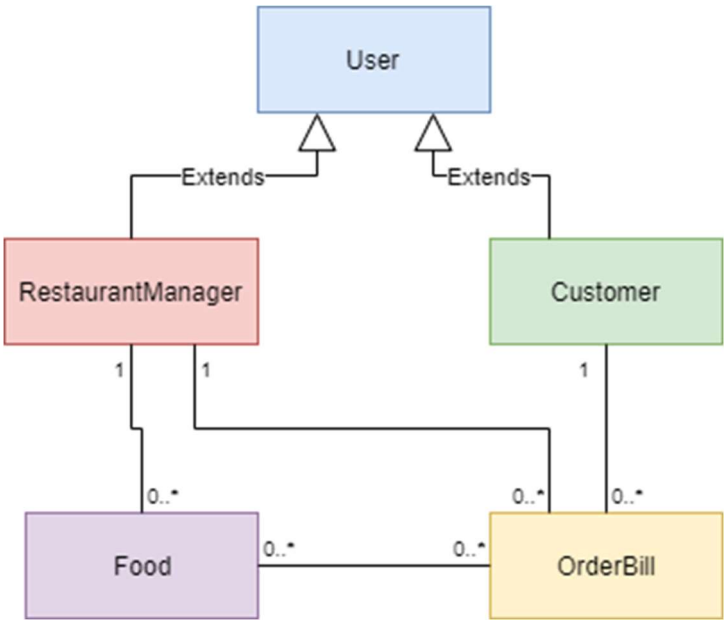

Figure 12: UML Diagram for Analysis Class.

A **Restaurant Manager** can insert more than one **Food**. A **Food** can be inserted only by one **Restaurant Manager**.

A **Customer** can be related to more than one **Order Bill**. An **Order Bill** can be related to only one **Customer**.

A **Restaurant Manager** can be related to more than one **Order Bill**. An **Order Bill** can be related to only one **Restaurant Manager**.

A **Food** can be part of an **Order Bill**. An **Order Bill** may have more than one **Food**.

3.3 Designed Entity-Relationship Diagram

This is the entity relationship diagram which is created by JPA right after the first database request.

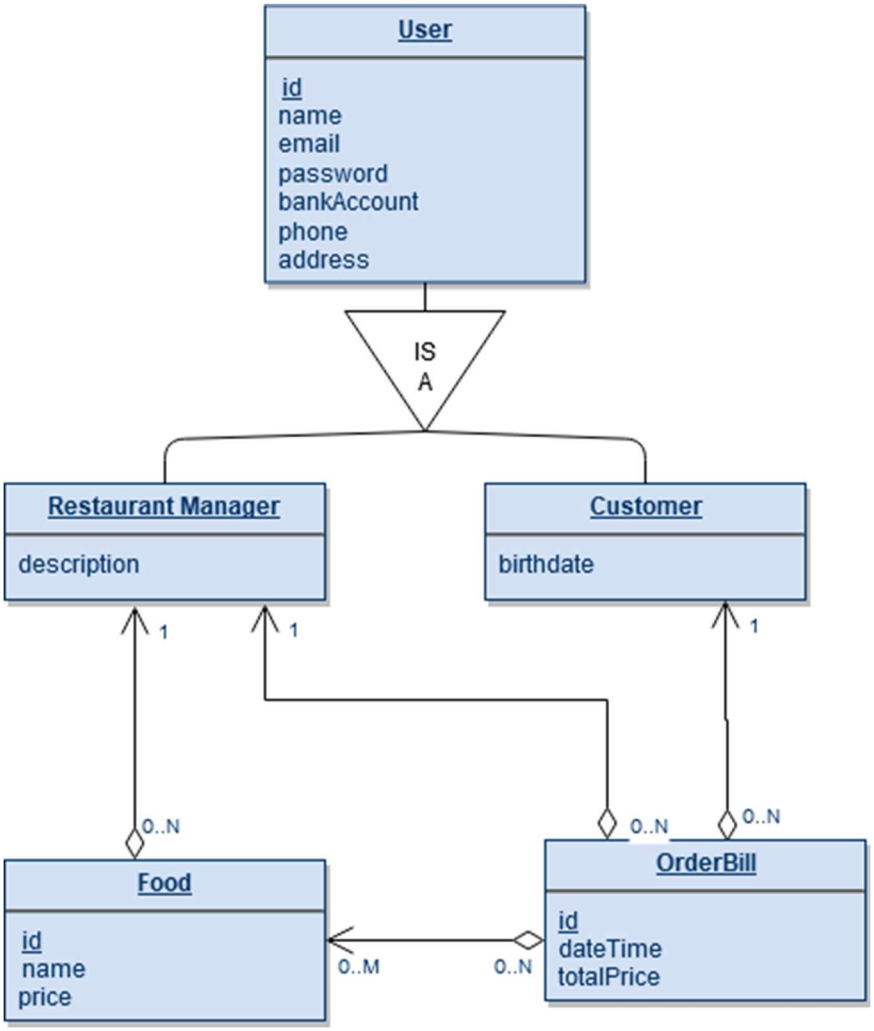

Figure 13, Entity Relationship Diagram

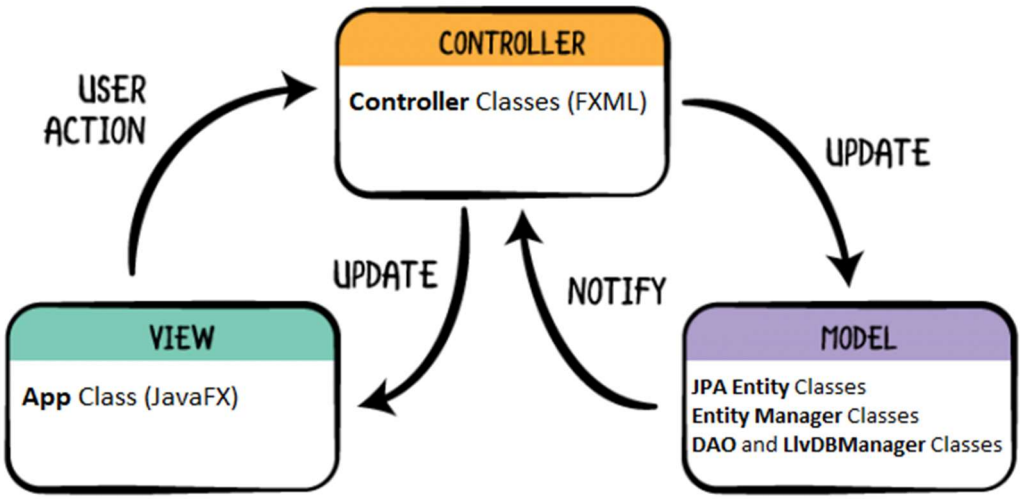

Figure 14: MVC Design Pattern schema.

We decided to apply the MVC (Model-View-Controller) Design Pattern. It splits the application's classes in three categories:

- The **Model** is where data resides. In our case, there are the JPA Entities (with the Entity Manager classes) and the LevelDB implementation.
- The **View** is the "face" of the application. It doesn't depend by the implementation's logic.
- The **Controller** is the communication part that links the View and the Model. In our case it is represented by the FXML implementation.

4 IMPLEMENTATION PHASE

4.1 UML CLASS DIAGRAM

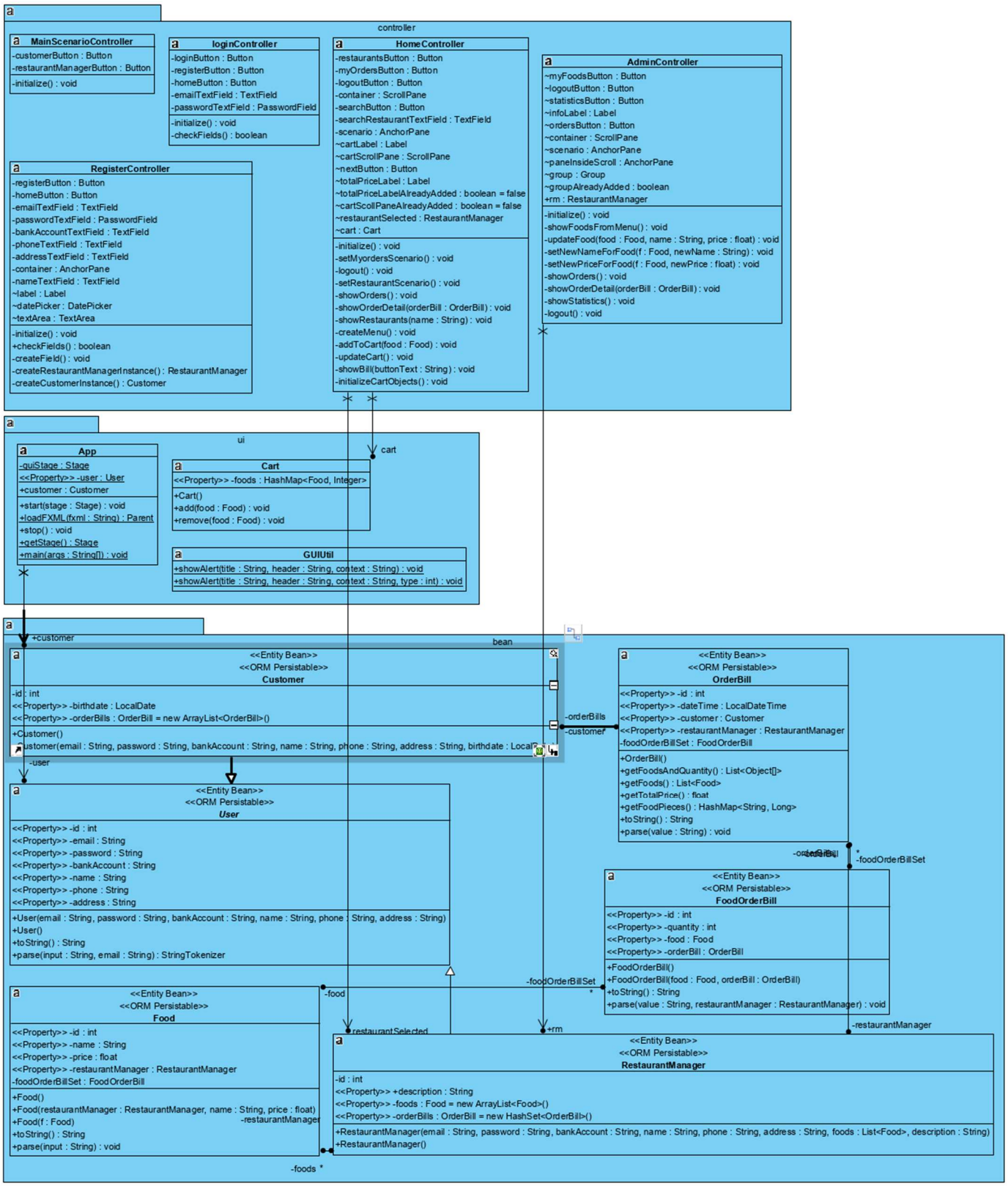

Figure 15: UML Class Diagram for bean, controller and user-interface classes.

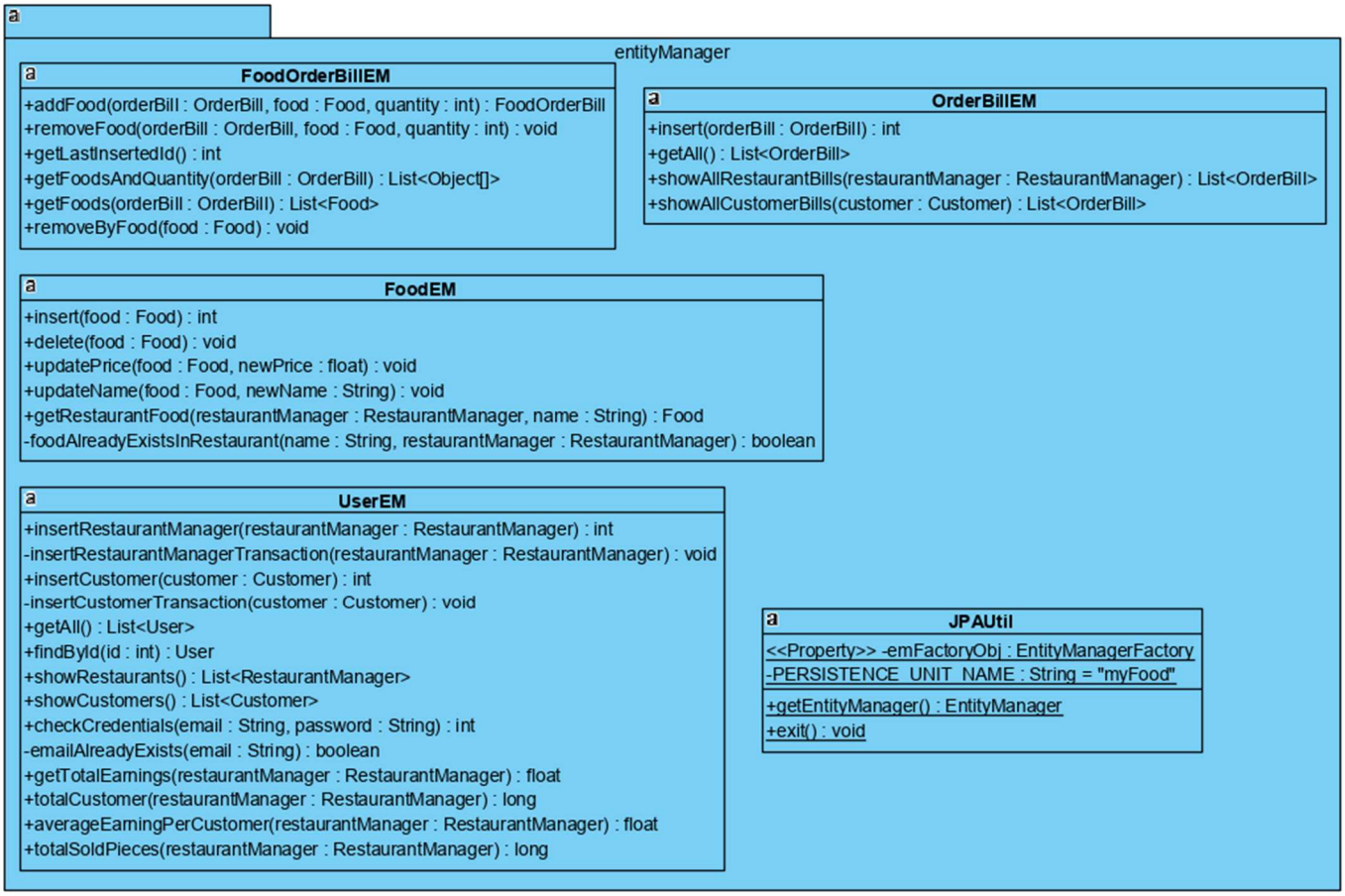

Figure 16: UML Class Diagram for EntityManager classes.

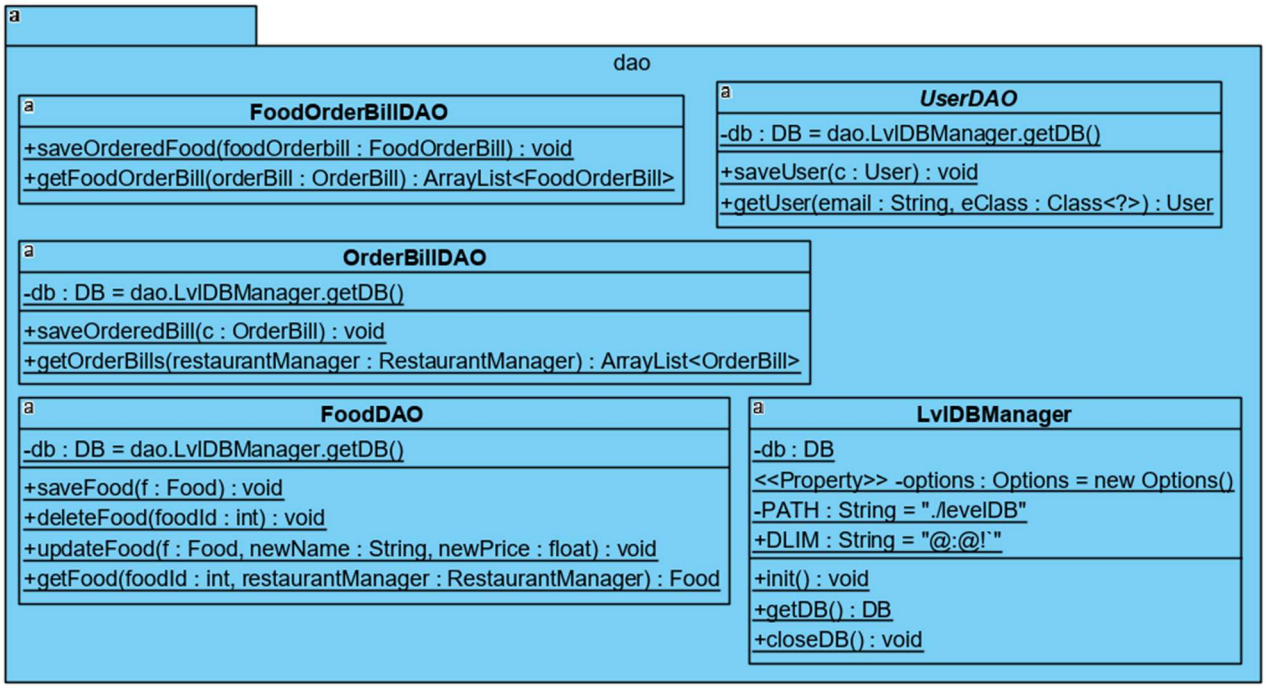

Figure 17: UML Class Diagram for Database Access Objects classes.

Controller classes:

- **MainScenarioController**: This class contains the elements and the methods to load the initial scenario, where the user can choose if he is a Customer or a Restaurant Manager.
- **LoginController**: This class contains the elements and the methods to load the login scenario, for both Customer and Restaurant Manager.
- **RegisterController**: This class contains the elements and the methods to load the registration scenario, for both Customer and Restaurant Manager.
- **HomeController**: This class contains the elements and the methods to load all the logged Customer scenarios.
- **AdminController**: This class contains the elements and the methods to load all the logged Restaurant Manager scenarios.

UI classes:

- **App**: This is the class that is launched at the beginning. It contains the elements to launch FXML scenarios.
- **Cart**: This class represents the temporary cart. A Cart is an object that doesn't persist in the database, so it's stored only for the application session.
- **GUIUtil**: This class contains methods that are frequently used in the GUI.

Bean classes:

- **User**: This class represents a generic User. It contains all the attributes that appear both in Customer and in Restaurant Manager entities. It is implemented by the Single Table JPA approach.
- **Customer**: This class represents the Customer persistence object. It contains the "birthdate" attribute, that doesn't appear in the Restaurant Manager class and it's specific for the Customer, and an OrderBill list, that represents the OneToMany relationship with OrderBill.
- **RestaurantManager**: This class represents the RestaurantManager persistence object. It contains the "description" attribute, that doesn't appear in the Customer class and it's specific for the RestaurantManager, an OrderBill list, that represent the OneToMany relationship with OrderBill and a Food list, that represents the OneToMany relationship with Food.
- **Food**: This class represents the Food persistence object. It contains a RestaurantManager attribute that represents the ManyToOne relationship with RestaurantManager and a FoodOrderBill list, that represents the ManyToMany relationship with OrderBill.
- **OrderBill**: This class represents the OrderBill persistence object. It contains a Customer attribute that represents the ManyToOne relationship with Customer, a RestaurantManager attribute that represents the ManyToOne relationship with RestaurantManager and a FoodOrderBill list that represents the ManyToMany relationship with Food.
- **FoodOrderBill**: This class is used to implement the ManyToMany relationship between OrderBill and Food. We created a new entity because this relationship had an attribute.

EntityManager classes:

- **UserEM**: This class contains all the methods to do operations on Customer and RestaurantManager objects in the database.
- **FoodEM**: This class contains all the methods to do operations on Food objects in the database.

- **OrderBillEM**: This class contains all the methods to do operations on OrderBill objects in the database.
- **FoodOrderBillEM**: This class contains all the methods to do operations on FoodOrderBill objects in the database.
- **JPAUtil**: This class contains the EntityManagerFactory to connect the application to the database (called "myFood").

DAO classes are discussed in Chapter **Errore. L'origine riferimento non è stata trovata.**

4.2 JPA ENTITIES

4.2.1 User

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@Table(name = "User")

public abstract class User {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name="user_id",updatable = false,nullable = false)
    private int id;

    private String email;
    private String password;
    private String bankAccount;
    private String name;
    private String phone;
    private String address;
}
```

To handle the generalization between Customer, RestaurantManager and User we applied the Single Table approach: Customers have the "`birthdate`" attribute, that the Restaurant Managers don't have, but Restaurant Managers have the "description" attribute, that the Customers don't have. So, using the "`SINGLE_TABLE`" inheritance type, we are going to have a single table called User, that contains both "birthdate" and "`description`" and that has a discriminator attribute (which is going to be called "DTYPE"). For Customers, "`description`" will be NULL. For Restaurant Managers, "`birthdate`" will be NULL. Both attributes will be declared respectively into Customer and RestaurantManager classes.We choose this approach because there are only few differences between Customer and RestaurantManager, so it's more efficient to have a single table for both of them. Other possible approaches are described by Thorben Janssen [2].

It's important to notice that there aren't relationships implemented involving the User entity. This entity is just an abstract way to handle the information that are present both in Customer and RestaurantManager objects.

4.2.2 Customer

```
@Entity
public class Customer extends User{
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
```

```
    @Column(name="customer_id",updatable = false,nullable = false)
    private int id;

    @OneToMany(mappedBy = "customer", cascade = CascadeType.ALL,fetch =FetchType.EAGER)
    private List<OrderBill> orderBills = new ArrayList<OrderBill>();

    private LocalDate birthdate;
}
```

The OneToMany attribute is relative to the list of OrderBill items related to a Customer. This relation is mapped by the attribute "`customer`" in the OrderBill entity.

There is also the "`birthdate`" attribute discussed in chapter 4.2.1.

4.2.3 Restaurant Manager

```
@Entity
public class RestaurantManager extends User {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name="restaurantManager_id",updatable = false,nullable = false)
    private int id;

    @OneToMany(mappedBy = "restaurantManager", cascade = CascadeType.ALL,fetch =
FetchType.EAGER)
    private List<Food> foods = new ArrayList<Food>();

    @OneToMany(mappedBy = "restaurantManager", cascade = CascadeType.ALL,fetch =
FetchType.EAGER)
    @Fetch(value = FetchMode.SUBSELECT)
    private Set<OrderBill> orderBills = new HashSet<OrderBill>();

    public String description;
}
```

RestaurantManager entity has a OneToMany relationship with Food that represents the menu inside a restaurant and a OneToMany relationship with OrderBill that represents the list of bills related to the restaurant. Both of them are mapped by "`restaurantManager`" attributes. There is also the "`description`" attribute discussed in chapter 4.2.1.

4.2.4 Food

```
@Entity
public class Food {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name="food_id",updatable = false,nullable = false)
    private int id;

    private String name;
    private float price;
    @ManyToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "fk_restaurantManager")
    private RestaurantManager restaurantManager;

    @OneToMany(mappedBy = "food",cascade = CascadeType.ALL)
    private Set<FoodOrderBill> foodOrderBillSet;
}
```

The ManyToOne annotation indicates to which RestaurantManager the Food is related. The OneToMany annotation, about the relation with FoodOrderBill, allows us to get all the orders where a Food appears, and it's mapped by the attribute "`food`" in the FoodOrderBill entity.

4.2.5 Order Bill

```
@Entity
public class OrderBill {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name="orderBill_id",updatable = false,nullable = false)
    private int id;

    private LocalDateTime dateTime;
    @ManyToOne
    @JoinColumn(name = "fk_customer")
    private Customer customer;

    @ManyToOne
    @JoinColumn(name = "fk_restaurantManager")
    private RestaurantManager restaurantManager;

    @OneToMany(mappedBy = "orderBill",cascade = CascadeType.ALL)
    private Set<FoodOrderBill> foodOrderBillSet;
}
```

The two ManyToOne annotations indicate the RestaurantManager and the Customer related to an order. The OneToMany annotation allows us to get all the food included in an order. This relation is mapped by the "`orderBill`" attribute in the FoodOrderBill entity.

4.2.6 Food Order Bill

```
@Entity
class FoodOrderBill {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name="foodOrderBill_id",updatable = false,nullable = false)
    private int id;

    @ManyToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "food_id")
    private Food food;

    @ManyToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "orderBill_id")
    private OrderBill orderBill;

    @Column(name = "quantity")
    private int quantity;
}
```

We implemented the relation between a Food and an OrderBill with a new entity because this relation is a ManyToMany with an attribute ("`quantity`") associated to the relation. So, to do that, we have used two OneToMany relationships. A different possibility

to do that is the Composite Key approach, as we can see in the Attila Fejer article [1], but in this case it won't work since that the composite key will have been FoodId-OrderBillId so we wouldn't be able to insert and OrderBill with *N istances of the same food* since, after the first insertion, the DBMS will have provided and error of *duplicate-key*.

4.3 Examples of CRUD Operations

4.3.1 Create

When we are going to create a new object and persist it into the database, we use an Entity Manager object to begin a transaction, persist the object and then close the transaction. In the following example there is the code to insert a new Restaurant Manager inside the database. This method is one of the UserEM's methods.

```
private void insertRestaurantManagerTransaction(RestaurantManager restaurantManager) {
   EntityManager entityManager = JPAUtil.getEntityManager();
   try {
      entityManager.getTransaction().begin();
      entityManager.persist(restaurantManager);
      entityManager.getTransaction().commit();
      System.out.println("A new restaurantManager has been added to the database   ");

   } catch (Exception ex) {
      ex.printStackTrace();
      System.out.println("Problem in inserting a new RestaurantManager!");
   } finally {
      entityManager.close();
   }
}
```

4.3.2 Read

When we are going to retrieve a list of objects, we use a method that has a list of these objects as return type. Then, we execute a query on an Entity Manager object and we insert the results (returned by `getResultList()` method) inside a List object that is going to be returned at the end of the method. In the following example, we are getting all the restaurants inside the database (we need to check if the retrieved objects are instances of RestaurantManager because we used the single table approach to handle the user generalization, chapter 4.2.1).

```
public List<RestaurantManager> showRestaurants() {
   EntityManager entityManager = JPAUtil.getEntityManager();
   List<User> users = entityManager.createQuery("from User ").getResultList();
   System.out.println("Size of users " + users.size());
   ArrayList<RestaurantManager> restaurantManagers = new
ArrayList<RestaurantManager>();
   for (User u : users) {
      if (u instanceof RestaurantManager)
         restaurantManagers.add((RestaurantManager) u);
   }
   entityManager.close();
   return restaurantManagers;
}
```

The only difference that is required when we are going to retrieve only one object instead of a list, is the called method on the query. Instead of `getResultList()`, we have to use the `getSingleResult()` method. In the following example we are going to retrieve a User object by using his ID. Both of the examples are located in the UserEM class.

```
public User findById(int id) {
   EntityManager entityManager = JPAUtil.getEntityManager();
   User result = entityManager.createNamedQuery("User.findById",
User.class).setParameter("id", id)
         .getSingleResult();
   entityManager.close();
   return result;
}
```

4.3.3 Update

When we are going to update an attribute of an object, we need to use an EntityManager and a Query objects, begin a transaction, create the query (using `createQuery(String)` method), call `executeUpdate()` method and then commit. In the following example we are going to update the name of a Food. The `getIdentifier(object)` method is used to get the food's ID and then use it in the query. The `executeUpdate()` method returns the old rows that have been replaced.

```
public void updateName(Food food, String newName) {
   EntityManager entityManager = JPAUtil.getEntityManager();
   try {
      entityManager = JPAUtil.getEmFactoryObj().createEntityManager();
      entityManager.getTransaction().begin();
      int foodId = (Integer)
JPAUtil.getEmFactoryObj().getPersistenceUnitUtil().getIdentifier(food);
      System.out.println("New name -> " + newName);
      Query query = entityManager.createQuery("UPDATE Food f SET f.name = :name WHERE
f.id = :id ");
      query.setParameter("name", newName);
      query.setParameter("id", foodId);
      int rowsDeleted = query.executeUpdate();
      System.out.println("entities update: " + rowsDeleted);
      entityManager.getTransaction().commit();

      System.out.println("Food updated!");
   } catch (Exception ex) {
      ex.printStackTrace();
      System.out.println("Problem in updating the food!");

   } finally {
      entityManager.close();
   }

}
```

4.3.4 Delete

The delete operation is very similar to the update operation. The list of required steps is the same: begin a transaction using an Entity Manager object, create a Query object by giving a String as input parameter that contains the query, call the executeUpdate()

method and then commit the transaction. In the following example we are going to delete a food.

```
public void delete(Food food) {
   EntityManager entityManager = JPAUtil.getEntityManager();
   try {
      entityManager.getTransaction().begin();
      System.out.println("Deleting food if-> " + food.getId());
      Query query = entityManager.createQuery("DELETE FROM Food f WHERE f.id = :id ");
      query.setParameter("id", food.getId());
      int rowsDeleted = query.executeUpdate();
      System.out.println("entities deleted: " + rowsDeleted);
      entityManager.getTransaction().commit();
      System.out.println("Food removed!");
   } catch (Exception ex) {
      ex.printStackTrace();
      System.out.println("Problem in deleting the food!");

   } finally {
      entityManager.close();
   }
}
```

4.4 IMPLEMENTED ENTITY-RELATIONSHIP DIAGRAM

This is the entity relationship diagram which is created by JPA.

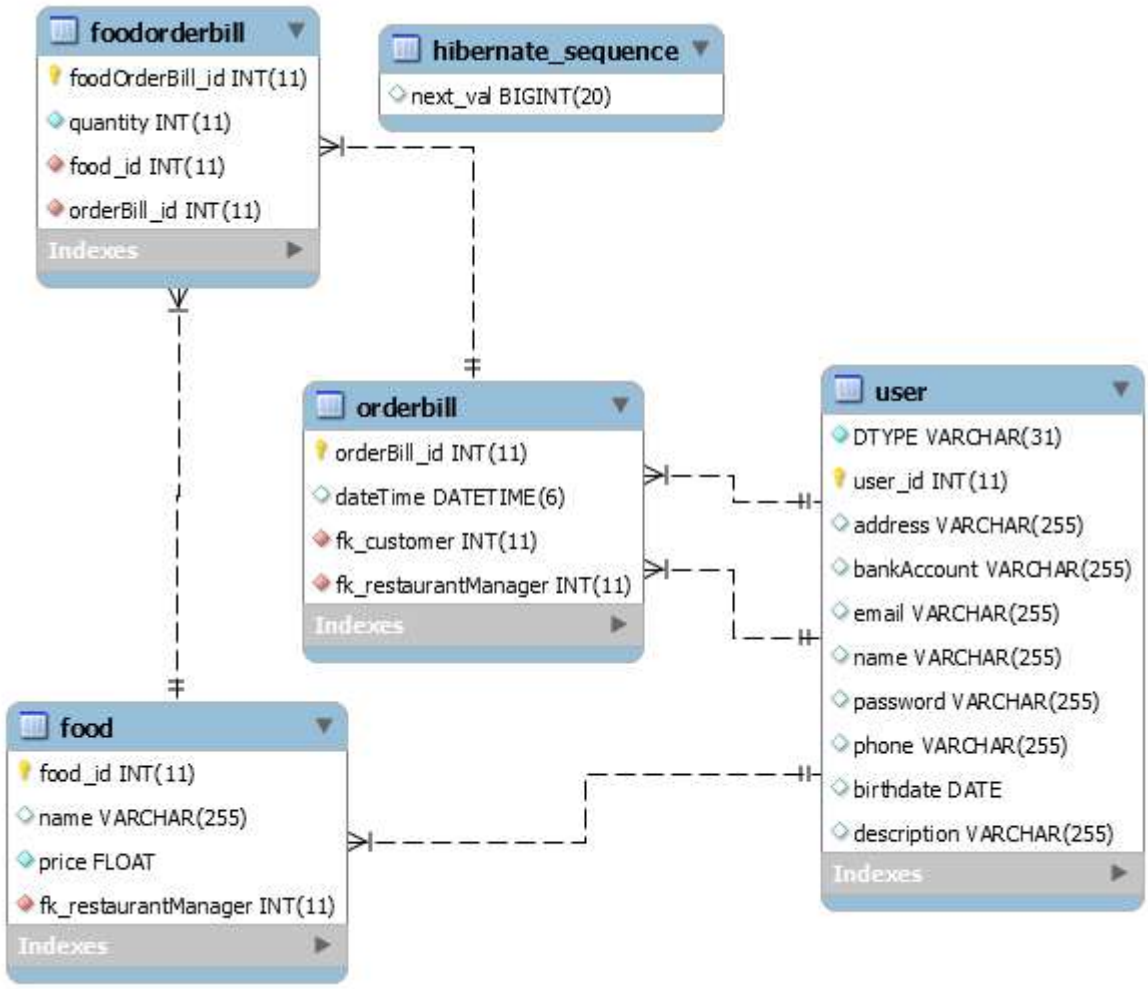

Figure 18: Database structure represented by the entity-relationship model of the implementation.

User:

- user_id: unique ID value that identifies a user.
- address: where the Customer lives (to deliver the food) or where the Restaurant is located.
- bankAccount: identifier of the bank account of the user. Is used to pay (for Customer) or to receive money (for Restaurant Manager).
- email: user's email address.
- name: name and surname for the Customer or restaurant name for Restaurant Manager.
- password: string used to access the user account.
- phone: user's phone number.
- birthdate: this value is null for Restaurant Managers. For Customer, it contains the date of the birthday.
- description: this value is null for Customers. For Restaurant Manager, it contains a text that describes shortly the restaurant.

Food:

- food_id: unique ID value that identifies a food.
- name: a string that contains the food name.
- price: actual price of a piece of this food, inserted by the restaurant manager.
- fk_restaurantManager: ID of the restaurant manager that has the restaurant with this food in the menu.

OrderBill:

- orderBill_id: unique ID value that identifies an order bill.
- dateTime: date and time of the order.
- fk_customer: customer's ID.
- fk_restaurantManager: restaurant manager's ID.

Food/OrderBill:

- foodOrderBill_id: unique ID value that identifies the relation between a food and a order bill.
- quantity: number of pieces of this food inside an order bill.
- food_id: food's ID.
- orderBill_id: order bill's ID.

5 KEY-VALUE FEASIBILITY STUDY

5.1 PREAMBLE

Key-value databases have three main features:

- Simplicity
- Speed
- Scalability

Unlike our relational databases, key-value databases have no tables, so no feature associated with tables. They are quite suitable for simple data models following the agile methodology, because you can easily modify the database code to add, remove or edit an attribute. Hence, the flexibility for changing database allows us to use it as database prototype. Based on our relational database (Figure 18) each table has its own id, so converting our model into the key-value would be continent.

However, we did not implement key-value for every IO operation. Generally, the statistical analysis part of an application requires huge amount of data. Since showing statistics in our application is time consuming for restaurant manager user, we just exploit key-value data base to improve response time of this section. The key-value database records all entities in our application, because the "STATISTICS" section depends on each one of them. In addition, it is a tried and true method to have data consistency between different databases (in our situation key-value database and MySQL). By the way, we need to make sure retrieving data from key-value data base has better performance than retrieving data from MySQL by using JPA. According to our benchmarks, using key-value database is much faster than MySQL as the data number increases.

# Orders	Test	1	2	3	4	5	6	7	8
1	Key-Value	60	59	58	54	66	60	53	65
	MySQL	26	29	28	30	37	29	34	32
10	Key-Value	324	307	321	376	316	323	328	319
	MySQL	157	159	159	235	159	169	183	162
100	Key-Value	599	546	568	569	551	595	555	539
	MySQL	1419	1570	1524	1443	1453	1535	1534	1445
1000	Key-Value	3936	4209	4005	5642	3946	5472	7282	5874
	MySQL	19473	19280	20180	19537	20010	19106	21652	19223

# Orders	Test	9	10	11	12	13	14	15	Mean
1	Key-Value	51	58	59	59	58	61	66	59
	MySQL	29	32	31	31	29	30	29	30
10	Key-Value	348	327	320	307	305	316	369	327
	MySQL	157	159	162	155	179	153	167	168
100	Key-Value	548	516	515	1119	524	581	582	594
	MySQL	1522	1426	1456	1458	1373	1437	1465	1471
1000	Key-Value	5708	6972	7355	4263	5472	6870	5282	5486
	MySQL	19798	20126	22725	19542	19360	19366	19419	19920

Table 1: Key-value Vs MySQL response time in milliseconds

The above experiment has been done by using intel 7th generation core i7 HQ CPU for laptop. Only one restaurant manager and one customer are registered in the application. The order (made by customer) contains the same type and number of foods in each test. It means we just increase the number of orders from 1 to 10, 100 and 1000. The time is calculated by using the method `System.currentTimeMillis()` in java. Note that the values in Table 1 are the response time of opening the "STATISTICS" section for the first time right after the application run. Obviously, the response time would decrease in next clicks Thus, the application was run for 15times in each test. Our key-value database is LevelDB.

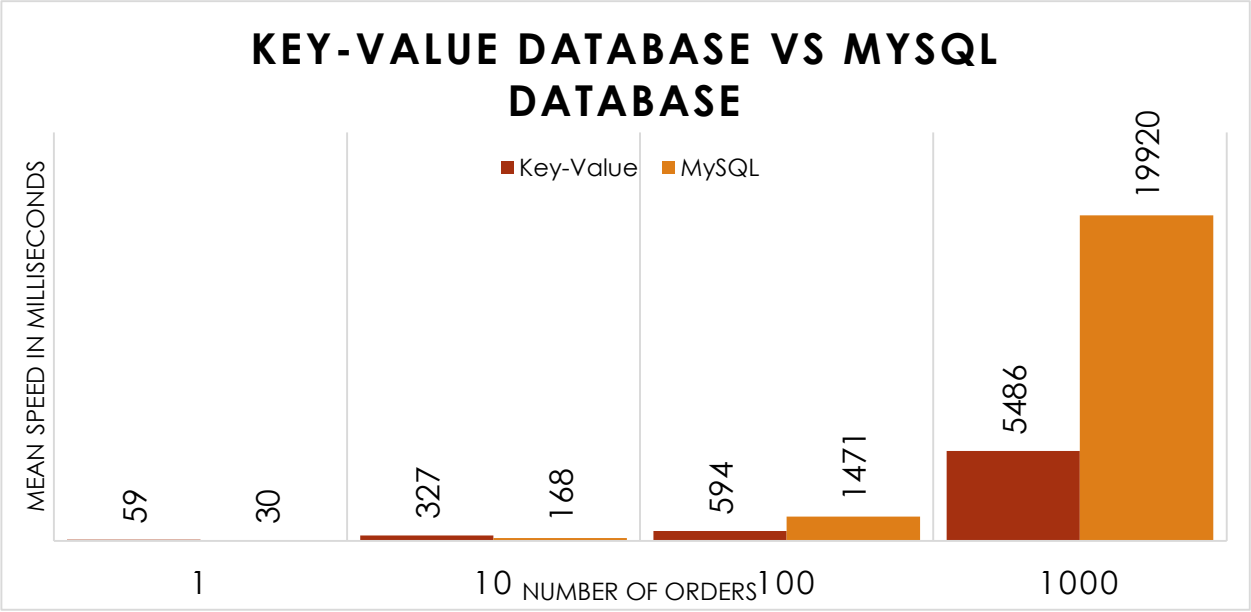

Figure 19: Histogram of the Mean response time

5.2 IMPLEMENTATION

For each of our entities we consider unique keys. Here is the table of the key we are going to store in LevelDB.

Entity	Format
User	user email
Food	"food_"+INTEGER
OrderBill	"ob_"+INTEGER
FoodOrderBill	"fob_"+INTEGER

Table 2, Entity key format

Since each user must provide unique email address for registration, we consider email address as a key for User entity. For other entities, we concatenate the auto generate MySQL id with a short string identifier. For instance, the id of the first inserted food in LevelDB would be "Food_1". You can see the string identifiers in the

Table 2.

Beside the keys, to store values in LevelDB we override `toString()` method for each Entity. the `toString` method returns an object attribute values as a string. We also create a `parse(String input)` method in order to parse the stored value of a key. We will explain these two methods in details.

In Figure 20, we can see the implementation of `toString` method for Food entity. A food object has restaurant manager, name and price attributes. Since we consider the email of the user as the key, we just store the email of the restaurant manager.

```java
@Override
public String toString() {
    return this.restaurantManager.getEmail() + LvlDBManager.DLIM + this.name + LvlDBManager.DLIM + this.price;
}
```

Figure 20, toString method of Food entity

In Figure 21, we parse the stored string base on a key. First, we split the string value based on a separator then we use each separated part to initializing object attributes.

```java
// parse the stored value in key-value level database
public void parse(String input) {
    String[] splits = input.split(LvlDBManager.DLIM);
    if (this.restaurantManager == null)
        this.restaurantManager = (RestaurantManager) UserDAO.getUser(splits[0], RestaurantManager.class);
    this.name = splits[1];
    this.price = Float.parseFloat(splits[2]);

}
```

Figure 21, parse method of Food entity

For User we did the same procedure. Attributes are concatenated with each other by "LvLDBManager.DLIM" in order to be interpretable in parsing stage.

```java
public String toString() {
    return this.password + LvlDBManager.DLIM + this.bankAccount + LvlDBManager.DLIM + this.name + LvlDBManager.DLIM
            + this.phone + LvlDBManager.DLIM + this.address;
}
```

Figure 22, toString method of User entity

Since, our application has two different Users (RestaurantManager and Customer), we exploited the concepts of inheritance in order to maintain simplicity in our code. So, we just implemented `parse` and `toString` methods for User class. Remember that both Customer and RestaurantManager are extending User Class. Here is the `parse` method for User entity:

```java
// parse the stored value in key-value level database
public StringTokenizer parse(String input, String email) {
    this.email = email;
    StringTokenizer st = new StringTokenizer(input, LvlDBManager.DLIM);
    this.password = st.nextToken();
    this.bankAccount = st.nextToken();
    this.name = st.nextToken();
    this.phone = st.nextToken();
    this.address = st.nextToken();
    return st;
}
```

Figure 23, parse method of User entity

OrderBill entity has both restaurant manager and customer ids, so we will save their emails for retrieving data.

```
public String toString() {
    return this.restaurantManager.getEmail() + LvlDBManager.DLIM + this.customer.getEmail() + LvlDBManager.DLIM
            + this.dateTime;
}
```

Figure 24, toString method of OrderBill entity

For parsing, we split the stored value according to the separator and then retrieve data from LevelDB. First, we split the data to parse the stored value. Then, we set attributes value based on separated strings.

```
// parse the stored value in key-value level database
public void parse(String value) {
    String[] splits = value.split(LvlDBManager.DLIM);
    if (this.restaurantManager == null) {
        this.restaurantManager = (RestaurantManager) UserDAO.getUser(splits[0], RestaurantManager.class);
    }
    if (this.customer == null)
        this.customer = (Customer) UserDAO.getUser(splits[1], Customer.class);
    this.dateTime = LocalDateTime.parse(splits[2]);
}
```

Figure 25, parse method of orderBill entity

FoodOrderBill has a Food, OrderBill and quantity. Hence, we use Food and OrderBill id to connect FoodOrderBill to a Food and OrderBill

```
public String toString() {
    // TODO Auto-generated method stub
    return this.food.getId() + LvlDBManager.DLIM + this.getOrderBill().getId() + LvlDBManager.DLIM
            + this.getQuantity() + LvlDBManager.DLIM;
}
```

Figure 26, toString method of FoodOrderBill entity

Retrieving data for FoodOrderBill entity is same as others. We use DAO classes for retrieving values of aggregation relations such as FoodOrderBill and Food. We will explain about DAO classes.

```
public void parse(String value, RestaurantManager restaurantManager) {
    String[] tokens = value.split(LvlDBManager.DLIM);
    this.food = FoodDAO.getFood(Integer.parseInt(tokens[0]), restaurantManager);
    this.quantity = Integer.parseInt(tokens[2]);

}
```

Figure 27, parse method of FoodOrderBill entity

LvlDBManager.DLIM is a separator element we used in key-value data format. Generally, the values are separated by ":", but in our application we split them by "@:@!". Since it is

likely to have ":" in a string such as description of a restaurant, we consider more complicated separator.

```
public class LvlDBManager {
    private static DB db;
    private static Options options = new Options();
    private static final String PATH = "./levelDB";
    public static final String DLIM = "@:@!`";
```

Figure 28, LvlDBManager instance fields

As we mentioned before, we used DAO classes for storing and retrieving data. DAO classes are: FoodDAO, UserDAO, FoodOrderBillDAO and OrderBillDAO classes. They all have saveEntity and getEntity methods. For example, here is the saveUser and getUser methods of UserDAO

```
public abstract class UserDAO {
    private static DB db = LvlDBManager.getDB();

    // ==================================add a new customer to data base
    public static void saveUser(User c) {
        db.put(bytes("user_" + c.getEmail()), bytes(c.toString()));
    }

    // ================================
    public static User getUser(String email, Class<?> eClass) {
        User u = null;
        String value = asString(db.get(bytes("user_" + email)));
        if (eClass == RestaurantManager.class) {
            u = new RestaurantManager();
            u.parse(value, email);
        } else {
            u = new Customer();
            u.parse(value, email);
        }
        return u;
    }
```

Figure 29, UserDAO class

The FoodDAO class has deleteFood and updateFood methods, because the restaurant manager is able to delete and update foods in his/her restaurant. The Figure 30 represents the details about these two methods. For deleting, we consider WriteBatch object to have secure removing. We also exploit threading to increase the key-value performance. However, the thread approach does not affect our comparison experiment in Figure 19.

```
// ==============delete a user from data base
public static synchronized void deleteFood(int foodId) {
    Thread t = new Thread(new Runnable() {
        @Override
        public void run() {
            WriteBatch batch = null;
            batch = db.createWriteBatch();
            batch.delete(bytes("food_" + foodId));
            db.write(batch);
        }
    });
    t.start();

}

// ====================update food
public static synchronized void updateFood(Food f, String newName, float newPrice) {
    Thread t = new Thread(new Runnable() {
        @Override
        public void run() {
            f.setName(newName);
            f.setPrice(newPrice);
            f.toString();
            db.delete(bytes("food_" + f.getId()));
            db.put(bytes("food_" + f.getId()), bytes(f.toString()));
        }
    });

    t.start();
}
```

Figure 30, update and delete Food methods in FoodDAO

6 USER MANUAL

6.1 GENERAL USER

6.1.1 Choose the role

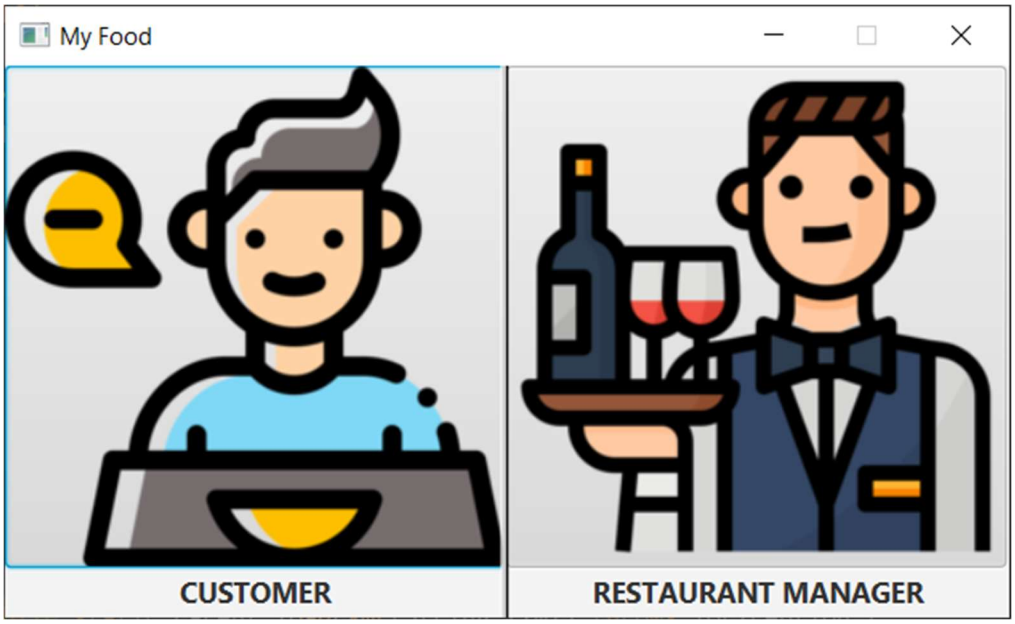

Figure 31: First scenario, where the user can choose his role.

If the user is a customer, he has to click on the left button, otherwise, if he is a restaurant manager, he has to click on the right button. Then, the login window will be loaded for both of them.

6.1.2 Login

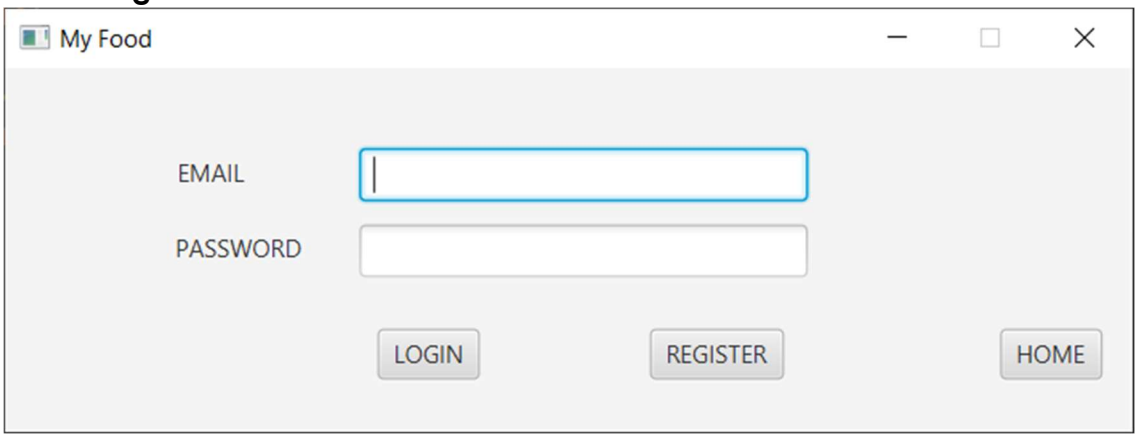

Figure 32: Customer login scenario.

If the user is already registered in the application, he can insert his email address inside the "EMAIL" field and his password inside the "PASSWORD" field, then he has to click on the "LOGIN" button.

Otherwise, if the user is not registered in the application, he can click on the "REGISTERED" button to access to the registration screen.

The "HOME" button allows the user to return to the screen where he can choose his role.

6.1.3 Registration

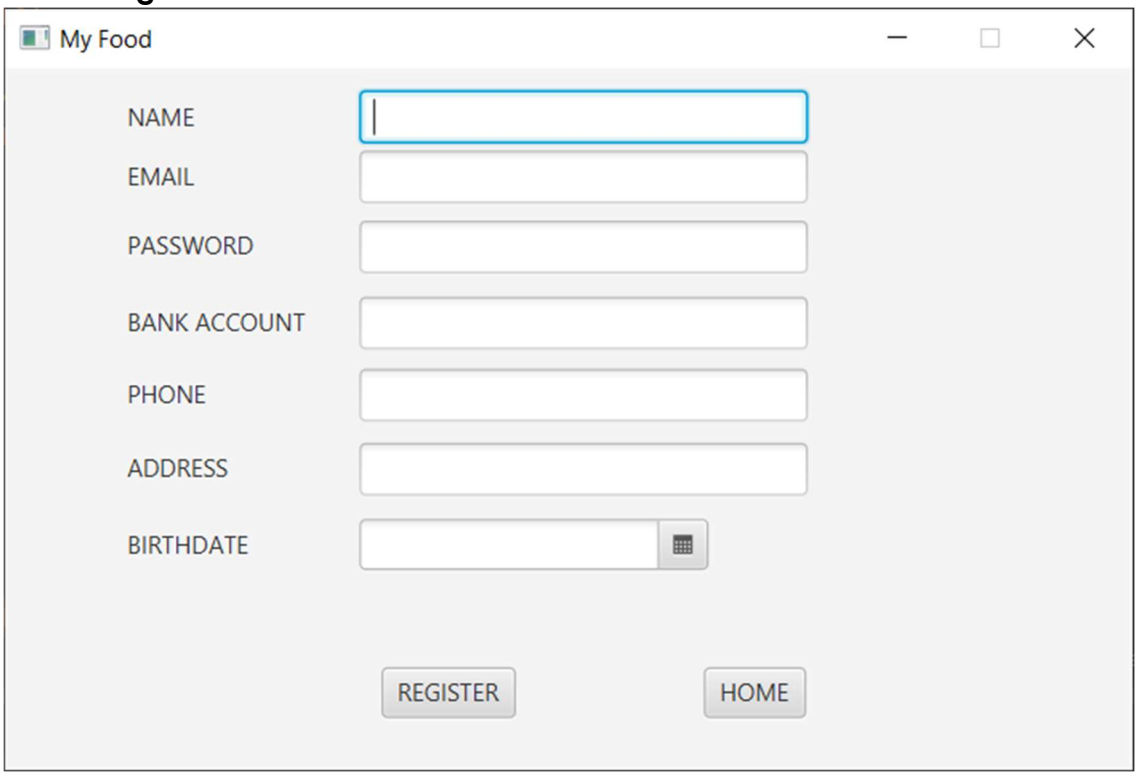

Figure 33: Customer registration screen.

Here, the user can insert his information to register to the application. He has to insert his name if he is a customer or restaurant's name if he is a restaurant manager in the "NAME" field, his email address in the "EMAIL" field, his password in the "PASSWORD" field, his bank account (used to pay the bill or to receive money) in the "BANK ACCOUNT" field, his phone number in the "PHONE" field, his address in the "ADDRESS" field and his birthdate by using the date picker at the right of the "BIRTHDATE" field if he is a customer or the restaurant's description in the "DESCRIPTION" field if he is a restaurant manager. Then, the user has to click on the "REGISTER" button.

The "HOME" button allows the user to return to the screen where he can choose his role.

6.2 CUSTOMER

6.2.1 Restaurant list

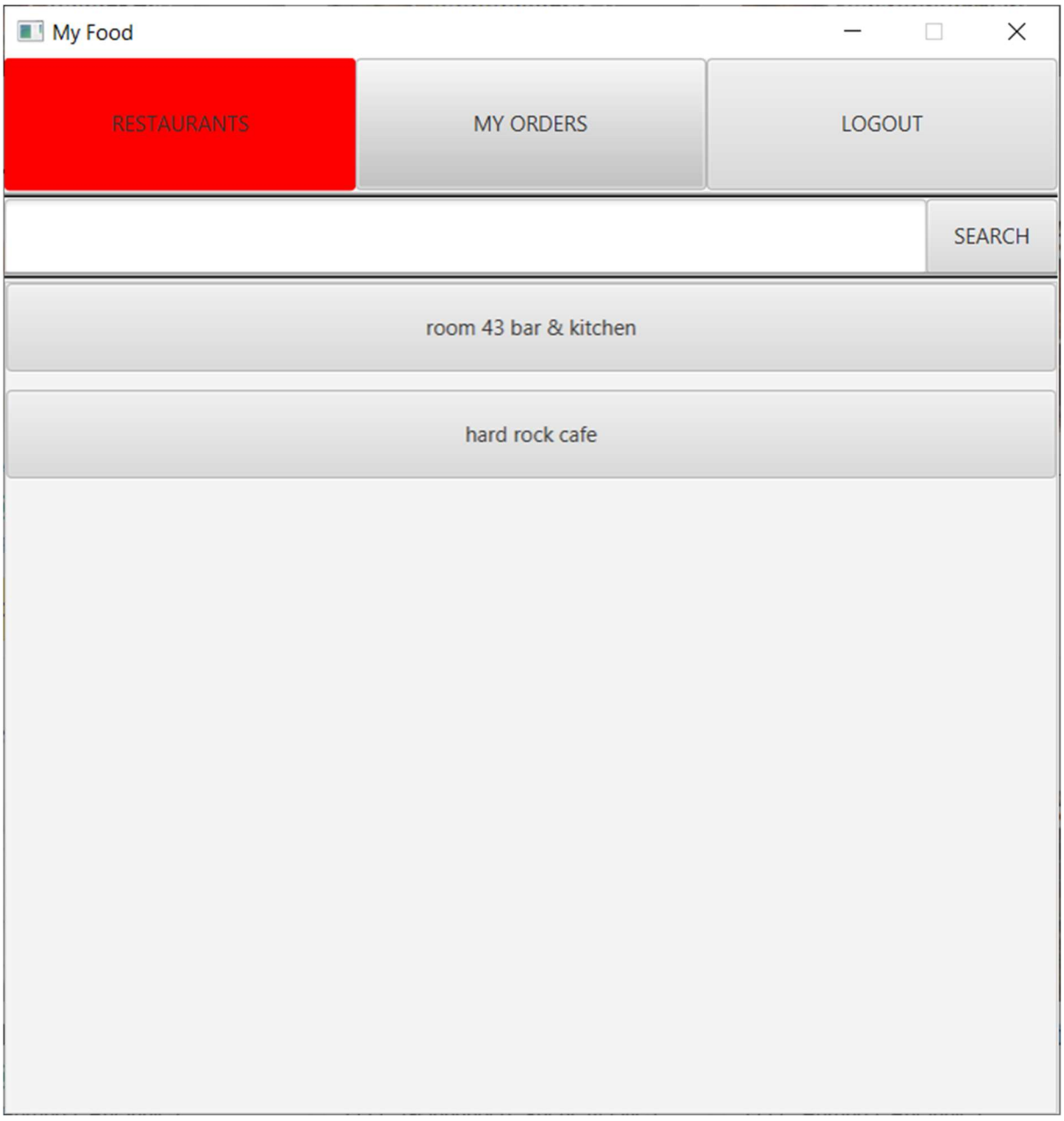

Figure 34: Restaurants' list screen.

This window can be accessed after the login or after clicking on the "RESTAURANTS" button in the top bar. Initially, a list of all the restaurants inside the application is loaded. The customer can write in the field under the top bar a string and click on the "SEARCH" button to execute the research. Then, a list containing the found restaurants will replace the actual list.

If the customer clicks on a restaurant, then the restaurant menu will be loaded. If the customer clicks on the "MY ORDERS" button in the top bar, a window containing the list of customer's orders will replace the actual window. If the customer clicks on the "LOGOUT" button, then he will return to the first window where he can choose his role (chapter 6.1.1).

6.2.2 Restaurant menu

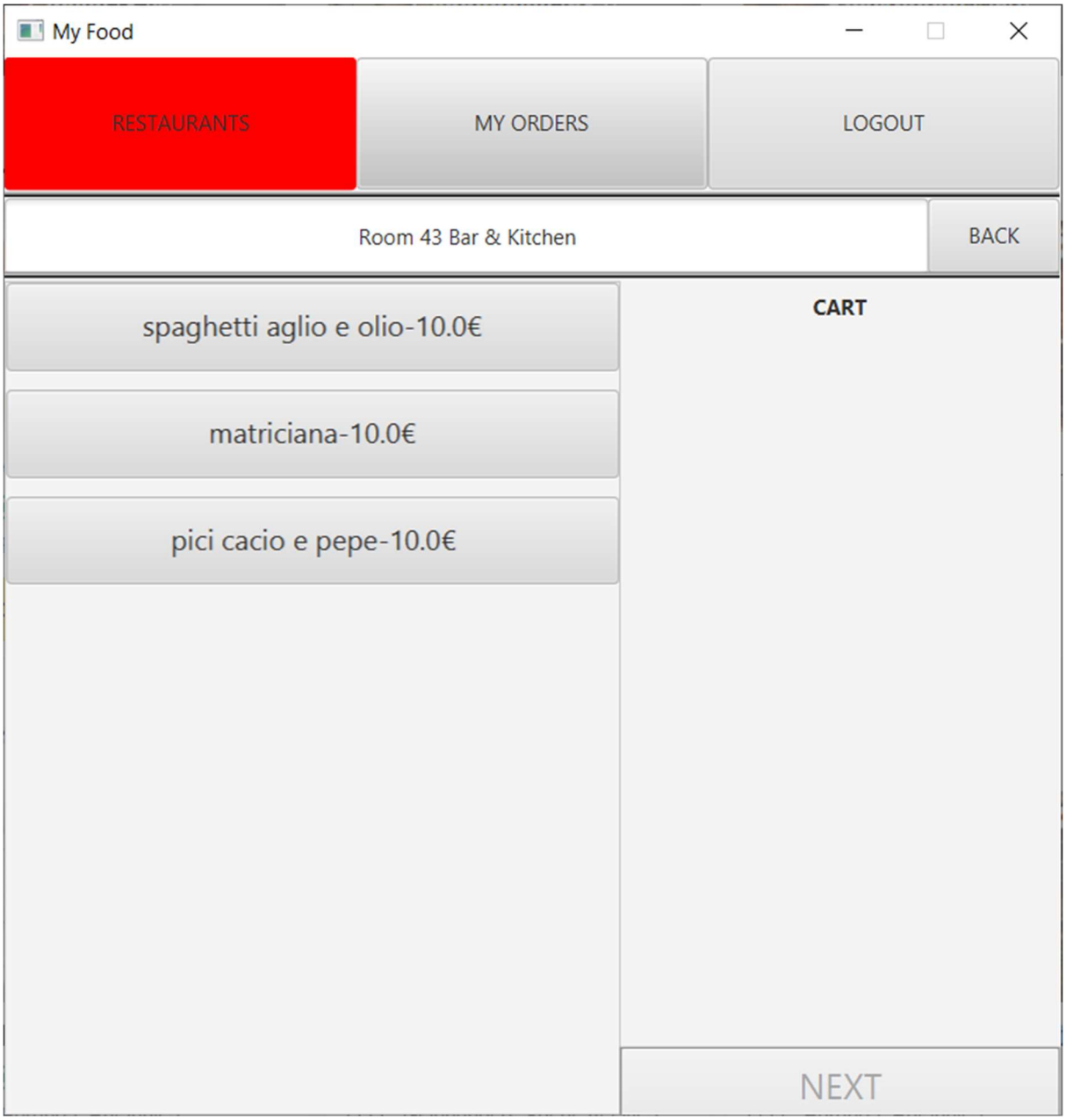

Figure 35: Restaurant menu' screen, with an empty cart.

In this window, the customer can see the restaurant's menu, add foods to the cart and then submit the order. In the left section there is the list of foods available in the restaurant. Initially, the "NEXT" button, that submits the order when the user clicks on it, will be disabled. If the customer clicks on a food, it will be added to the cart, shown in the left list.

If the customer clicks on the "BACK" button, then the window with the list of restaurants available in the application will be loaded (chapter 6.2.1).

If the customer clicks on the "MY ORDERS" button in the top bar, a window containing the list of customer's orders will replace the actual window. If the customer clicks on the "LOGOUT" button, then he will return to the first window where he can choose his role (chapter 6.1.1).

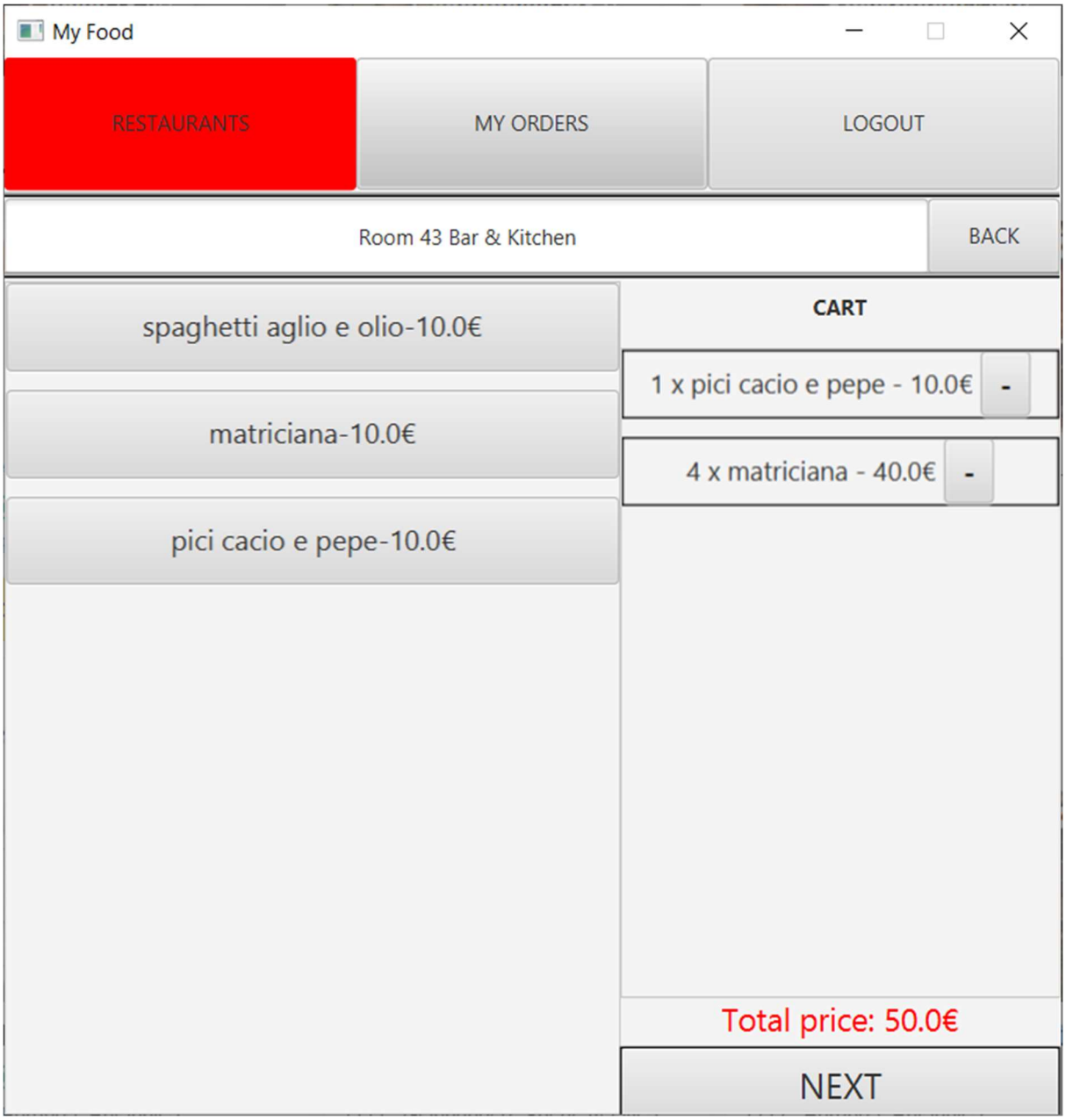

Figure 36: Restaurant menu's screen, with some food in the cart.

To add more pieces of the same food, the customer has to click again on that food in the left list. At the right of each food in the cart list there is a "-" button that allows the customer to remove that food from the cart by clicking on it. When there is almost a food in the cart, the "NEXT" button becomes available. If the customer clicks on it, then the bill will be shown in a pop-up.

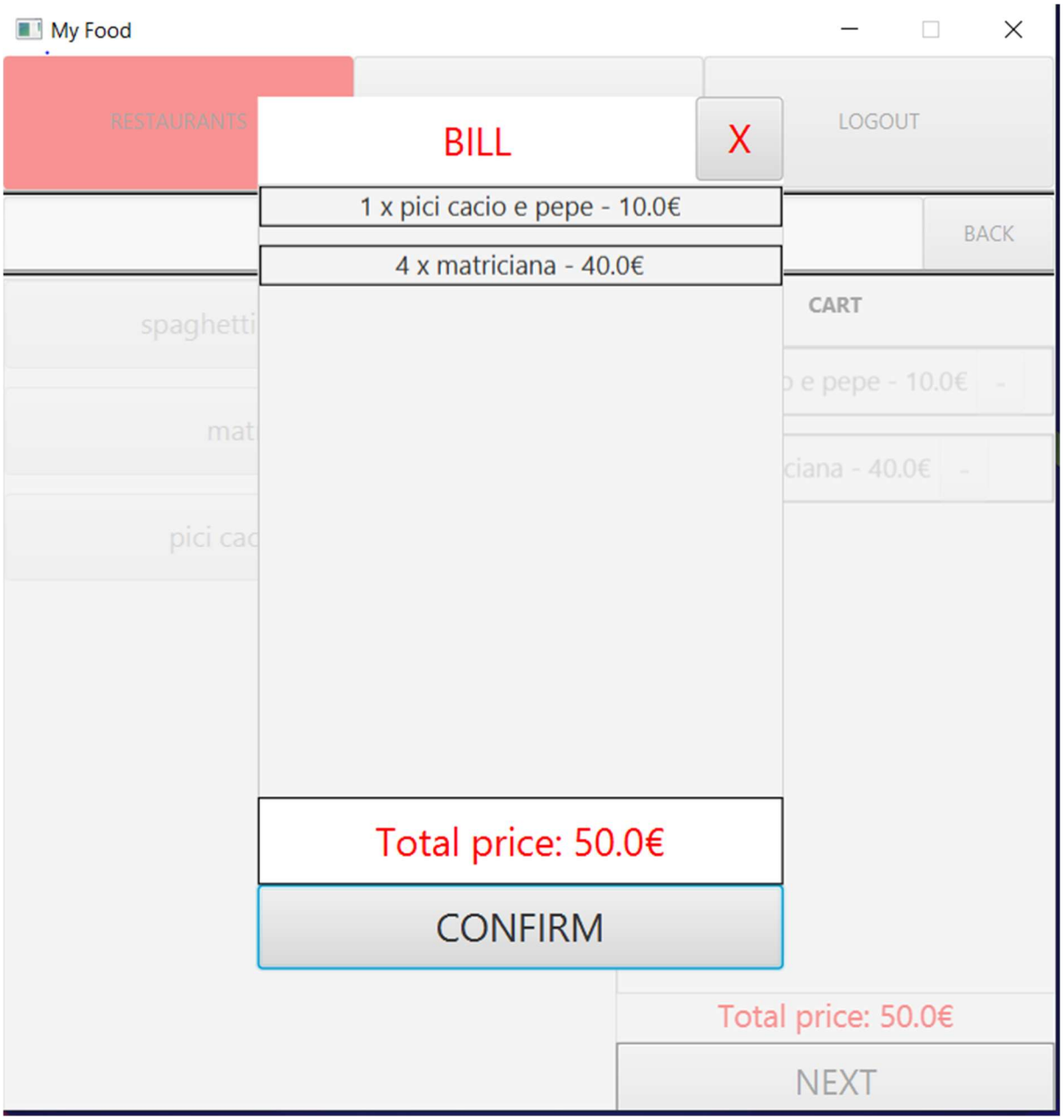

Figure 37: Bill pop-up shown to confirm the order.

If the customer clicks on the "CONFIRM" button, then the system will submit the order. If the customer wants to edit the order before the submitting, he has to click on the "X" button that will close the bill pop-up.

6.2.3 Orders list

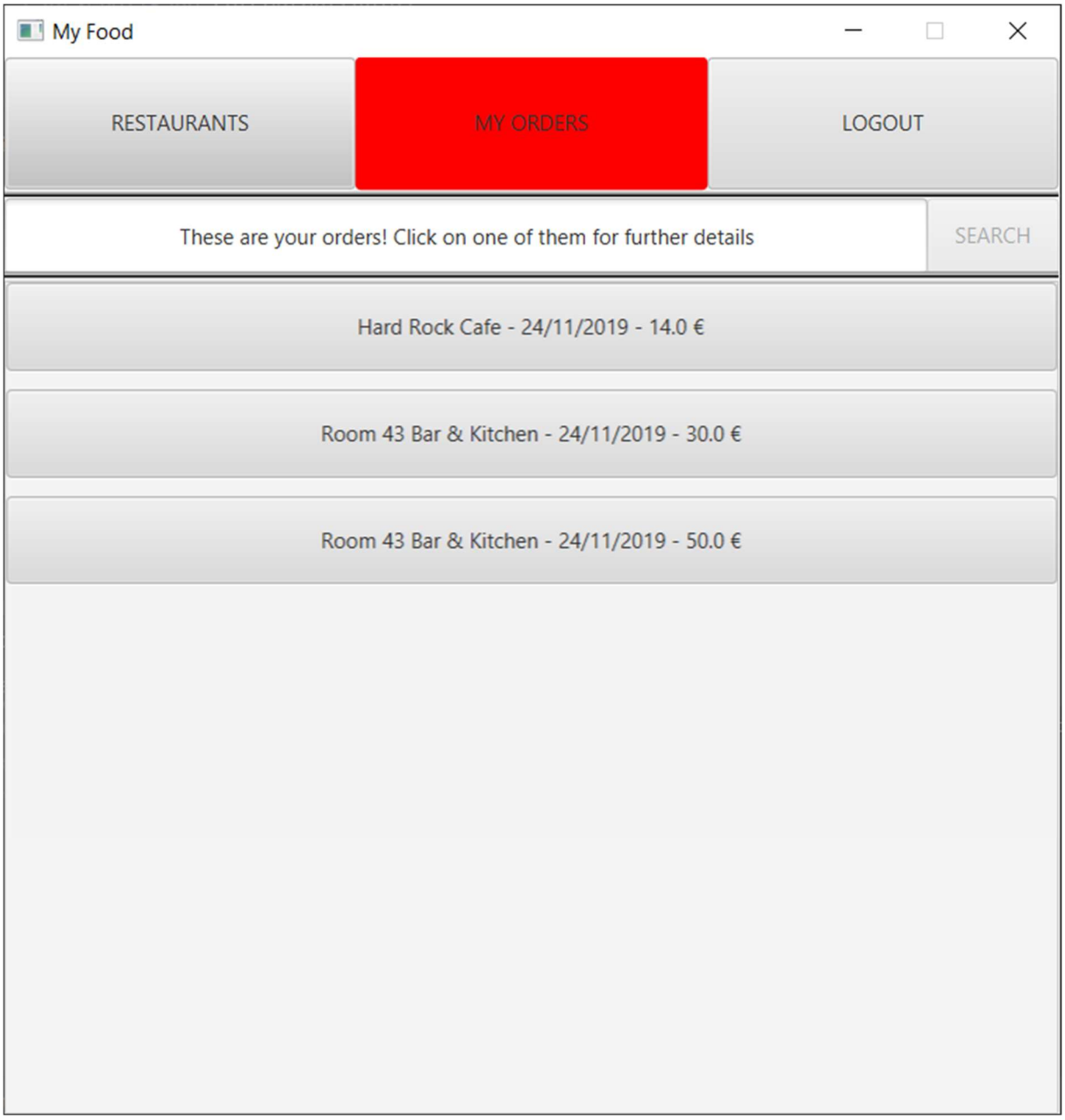

Figure 38: Customer's orders screen.

This window is accessed by clicking on the "MY ORDERS" button in the top bar. A list of the customer's orders will be loaded. If the customer clicks on an order, then a pop-up containing the order details will be shown.

If the customer clicks on the "RESTAURANTS" button in the top bar, a window containing the list of available restaurants (chapter 6.2.1) will replace the actual window. If the customer clicks on the "LOGOUT" button, then he will return to the first window where he can choose his role (chapter 6.1.1).

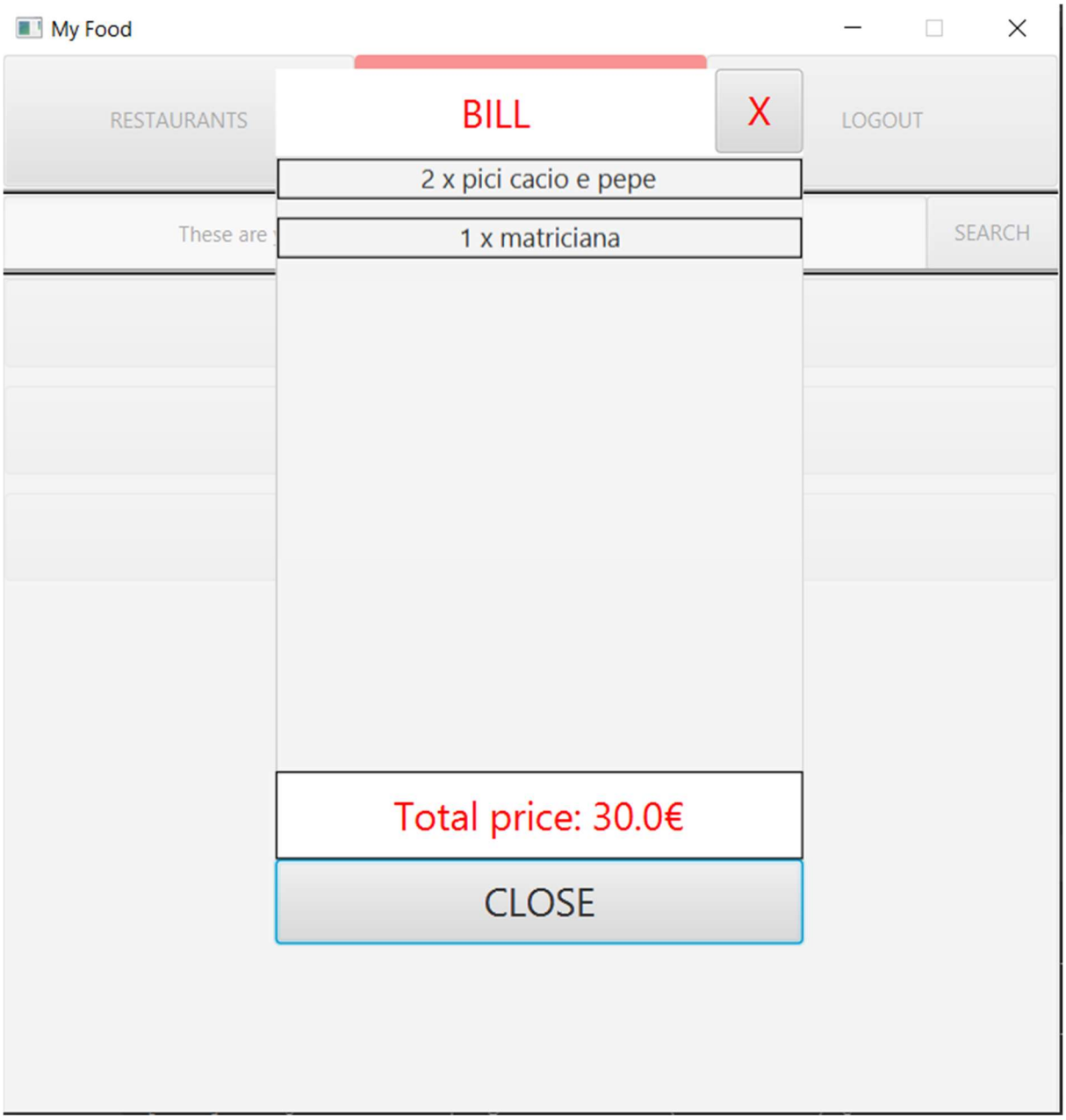

Figure 39: Bill pop-up shown after clicking on a specific order in the list.

When the bill pop-up is shown, the customer can only close it by clicking on the "CLOSE" button or on the "X" button.

6.3 RESTAURANT MANAGER

6.3.1 Manage restaurant menu

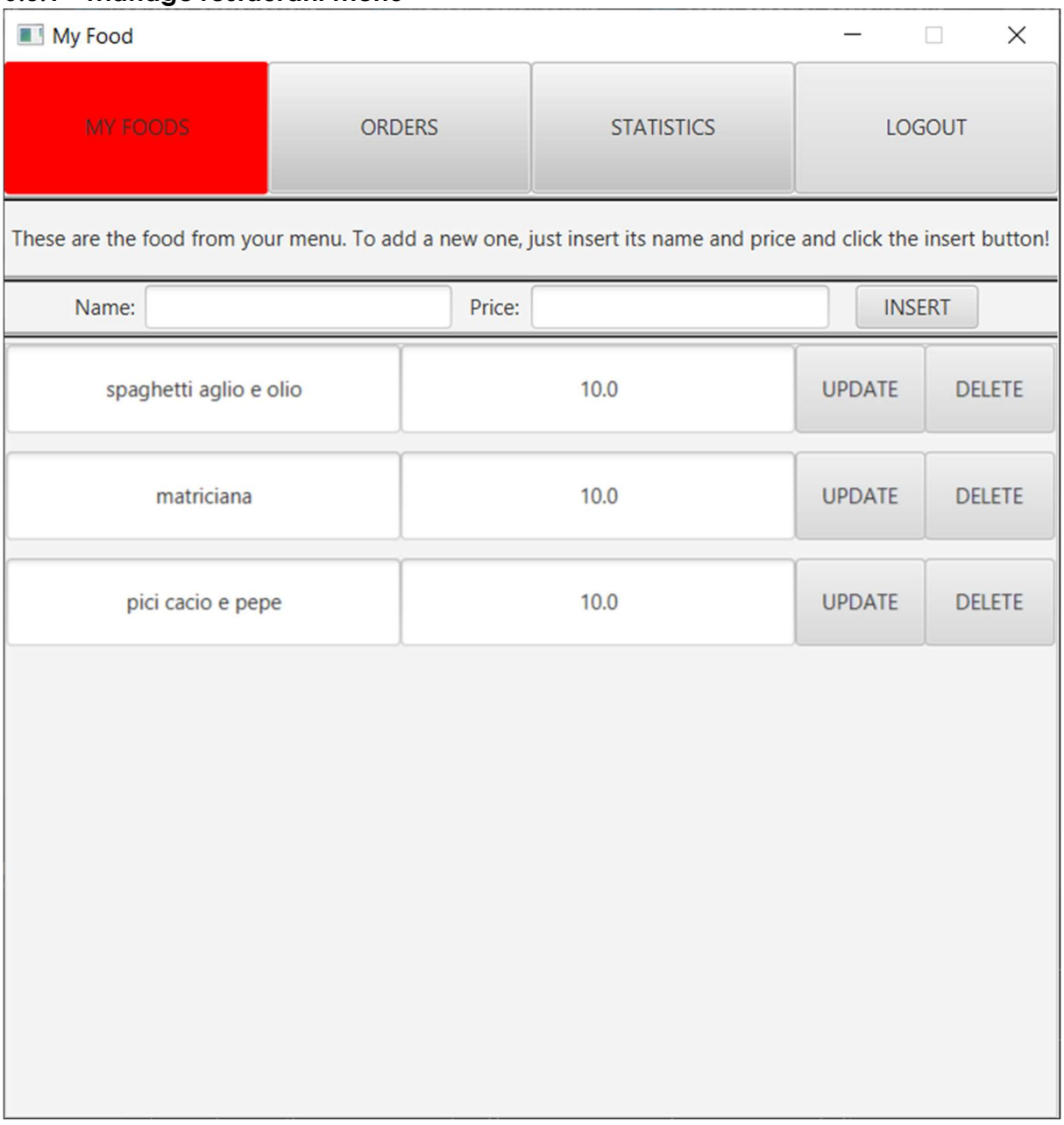

Figure 40: Window where a restaurant manager can manage his restaurant's menu.

This window can be accessed after the login or after clicking on the "MY FOODS" button in the top bar. The system loads a list that contains all the available foods in the menu. For each food are shown two fields containing name and price and two buttons: "UPDATE", that allows the restaurant manager to update the name and the price of the food by picking the values from the two fields and "DELETE" that allows the restaurant manager to remove that food from his menu. At the top of the list there are two field, "Name" and "Price", that allows the restaurant manager to insert a new food in the menu by clicking on the "INSERT" button.

If the restaurant manager clicks on the "ORDERS" button in the top bar, a window containing the list of his orders will replace the actual window. If the restaurant manager

clicks on the "STATISTICS" button in the top bar, a window containing his restaurant's statistics will replace the actual window. If the restaurant manager clicks on the "LOGOUT" button, then he will return to the first window where he can choose his role (chapter 6.1.1).

6.3.2 Orders list

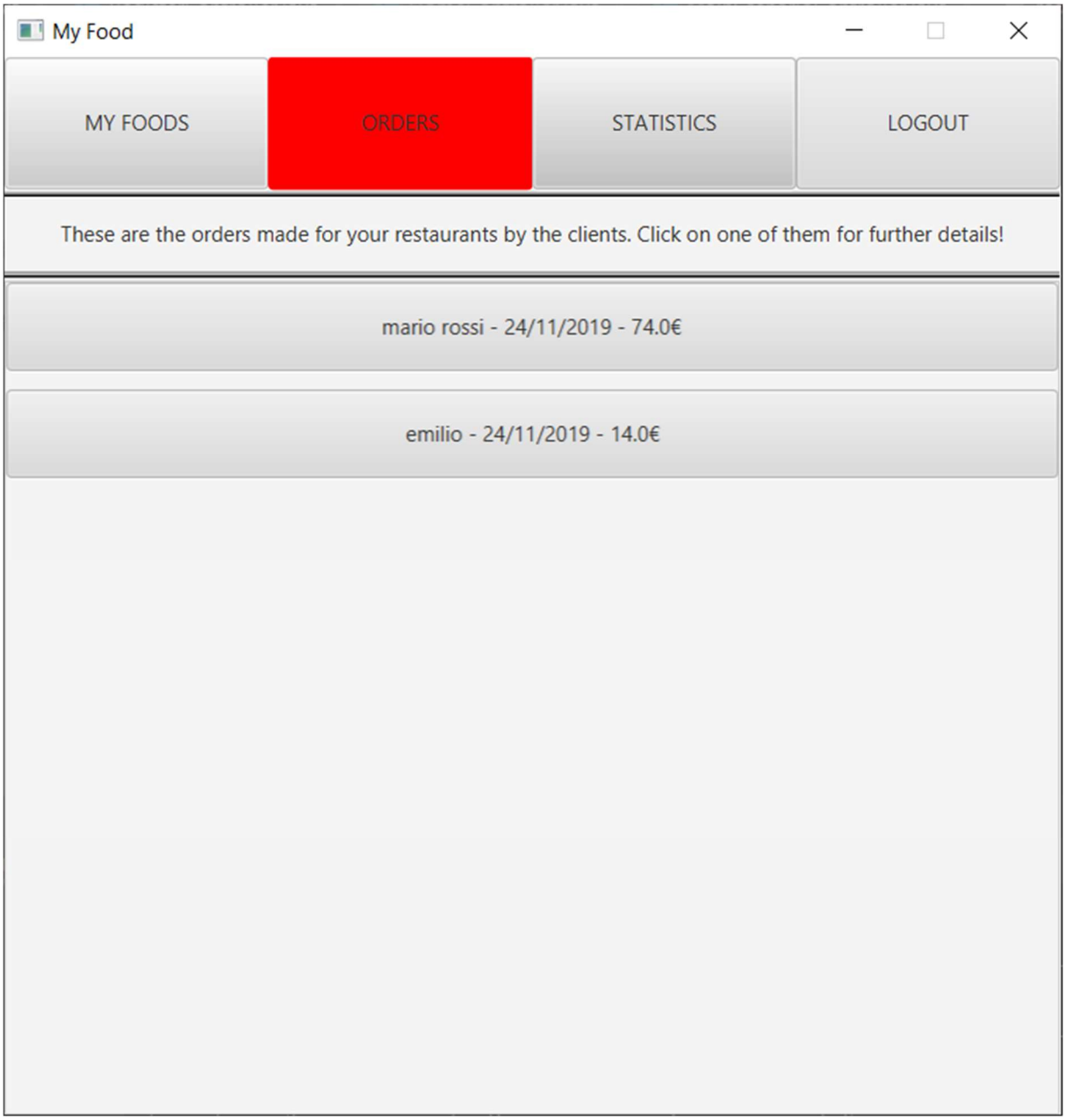

Figure 41: Restaurant's order list.

This window is accessed by clicking on the "ORDERS" button in the top-bar. A list of the restaurant's orders will be loaded. If the restaurant manager clicks on an order, then a pop-up containing the order details will be shown.

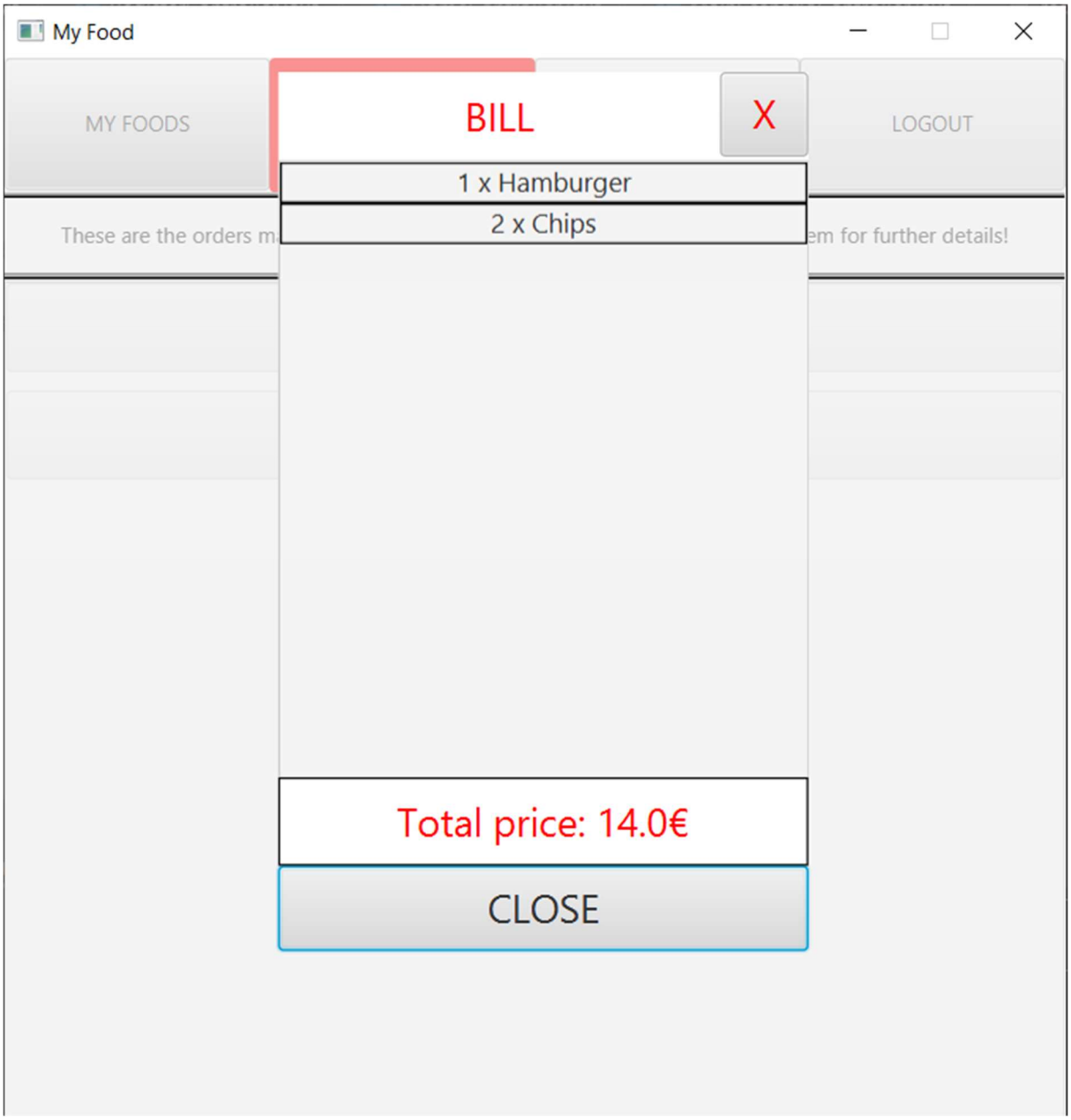

Figure 42: Bill pop-up shown after clicking on a specific order in the list.

When the bill pop-up is shown, the restaurant manager can only close it by clicking on the "CLOSE" button or on the "X" button.

If the restaurant manager clicks on the "MY FOODS" button in the top bar, a window containing his restaurant's menu (chapter 6.3.1) will replace the actual window. If the restaurant manager clicks on the "STATISTICS" button in the top bar, a window containing his restaurant's statistics will replace the actual window. If the restaurant manager clicks on the "LOGOUT" button, then he will return to the first window where he can choose his role (chapter 6.1.1).

6.3.3 Restaurant's statistics

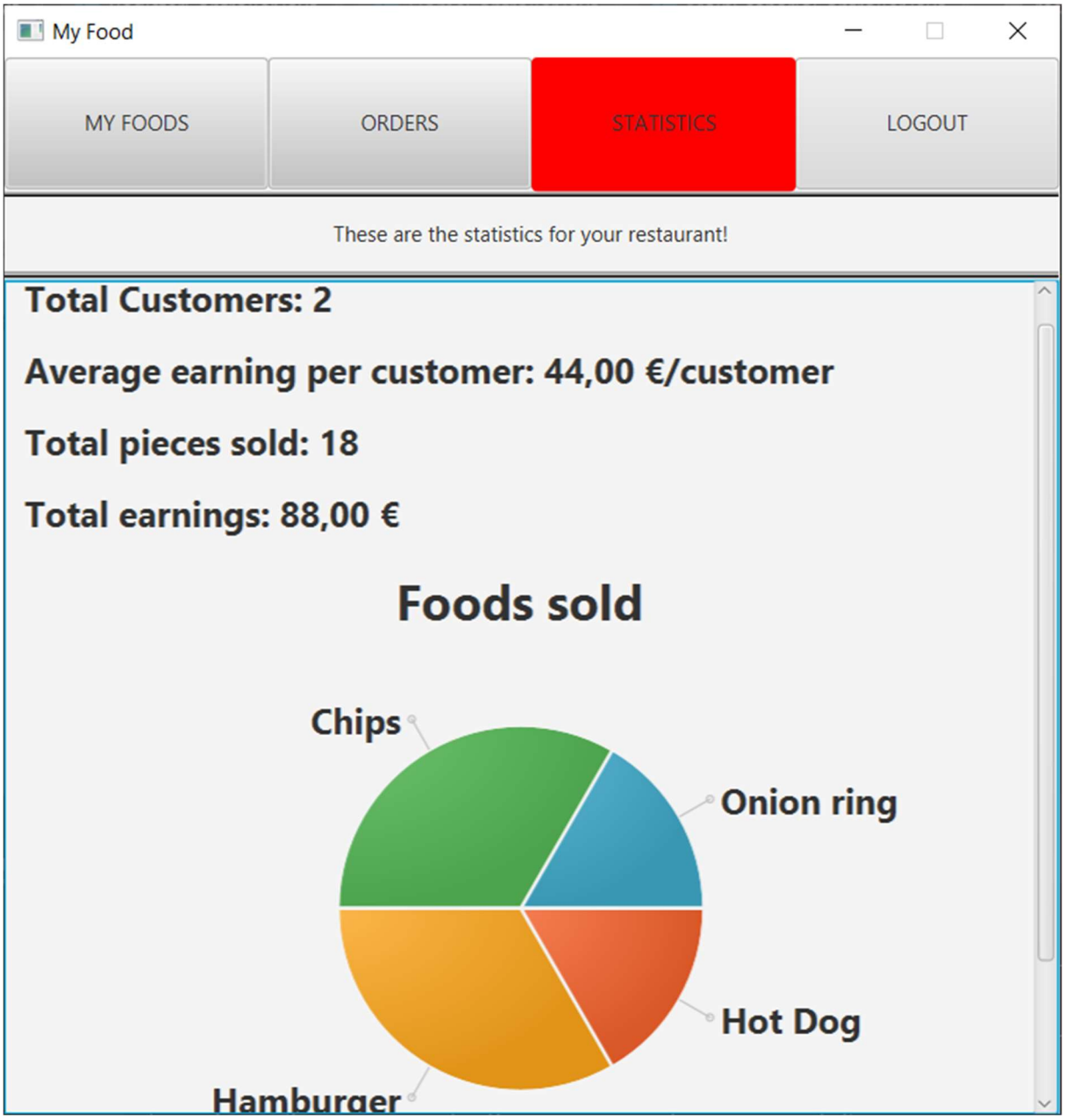

Figure 43: Restaurant's statistics screen.

This window is accessed by clicking on the "STATISTICS" button in the top bar. There are shown four values: total customers, average earning per customer, total pieces sold and total earnings. There is also a pie chart that shows which foods are more requested by customers.

If the restaurant manager clicks on the "MY FOODS" button in the top bar, a window containing his restaurant's menu (chapter 6.3.1) will replace the actual window. If the restaurant manager clicks on the "ORDERS" button in the top bar, a window containing the list of his orders (chapter 6.3.2) will replace the actual window. If the restaurant manager clicks on the "LOGOUT" button, then he will return to the first window where he can choose his role (chapter 6.1.1).

7 REFERENCES

[1] Fejer A., "*Many-To-Many Relationships in JPA*", Baeldung, 2019

[2] Janssen T., "*Inheritance Strategies with JPA and Hibernate – The Complete Guide*", Thoughts on Java, 2017