



# UNIVERSITÀ DI PISA

Computer Engineering

Large Scale and Multi-Structured Databases

## WORKGROUP TASK 1

An application for restaurants' reservation  
management in Java

---

*STUDENTS:*

Cima Lorenzo  
Ferrante Nicola  
Pampaloni Simone  
Scatena Niccolò

Academic Year 2019/2020

## **Abstract**

This project has been developed as a workgroup during the classes of Large Scale and Multi-Structured Databases. The aim of this workgroup is to give a demonstration of the students' capabilities to handle the development of a simple JAVA application connecting to a relational DB using an implementation of Java Persistence API (JPA), starting from requirements and arriving to implementation.

# Contents

<b>1</b>	<b>Specifications</b>	<b>3</b>
<b>2</b>	<b>Analysis</b>	<b>4</b>
2.1	Requirements . . . . .	4
2.1.1	Functional requirements . . . . .	4
2.1.2	Non-functional requirements . . . . .	4
2.2	Use cases . . . . .	5
2.3	Analysis Classes . . . . .	6
2.4	System architecture . . . . .	6
<b>3</b>	<b>Implementation</b>	<b>9</b>
3.1	Technologies used . . . . .	9
3.1.1	JAVA: Object Oriented Programming Language . . . . .	9
3.1.2	MySQL: Relational Database Management System . . . . .	9
3.1.3	Java Persistence API: Object to Relational Mapping (Hibernate implementation) . . . . .	9
3.2	Implementation . . . . .	9
3.2.1	Entities . . . . .	10
3.2.2	Client . . . . .	12
3.2.3	Common Utilities . . . . .	14
3.2.4	Server . . . . .	16
3.2.5	Database . . . . .	18
<b>4</b>	<b>Feasibility Study On The Use Of A Key Value Database</b>	<b>20</b>
4.1	Relational Data Model Analysis . . . . .	20
4.1.1	Volume Table . . . . .	20
4.1.2	Possible Operations . . . . .	20
4.1.3	Analysis Results . . . . .	22
4.1.4	Data Model Translation . . . . .	23
4.2	Key-Value DataBase Implementation . . . . .	24
4.2.1	Technologies used: LevelDB . . . . .	24
4.2.2	UML Implementation Diagram . . . . .	24
<b>5</b>	<b>User manual</b>	<b>27</b>
5.1	Introduction . . . . .	27
5.2	Login and Registration . . . . .	27

5.2.1	Registration . . . . .	28
5.3	Customer interface . . . . .	28
5.3.1	Search restaurants . . . . .	29
5.3.2	Book a table . . . . .	30
5.3.3	Delete a reservation . . . . .	32
5.4	Owner interface . . . . .	32
5.4.1	Modify restaurant informations . . . . .	32
5.4.2	Manage reservations . . . . .	33
5.5	Command Line Interface . . . . .	34
5.5.1	Command line options . . . . .	35
5.6	Configuration file . . . . .	36
5.7	Build standalone JARs . . . . .	37

# 1 | Specifications

The product described in this paper is a **restaurant table booking application**. The main objective of the application is to allow restaurant owners to make their restaurant known to a large number of people and allow customers to find and book a table, quickly and easily, at their favorite restaurants.

This application can be used both from customers and restaurant owners.

**Customers** can use this application to see the list of available restaurants, book a table in one of these restaurants with the possibility to choose date, time and number of seats and see the list of bookings made (with the possibility of deleting them).

Moreover **restaurant owners** can insert all the details of their restaurant within the application: name, type of cooked food, the city and the street in which it is located, a general description of the restaurant and when they are open. In addition, the restaurant owner can see a list of bookings made in his restaurant by customers, so that he can easily manage them.

## 2 | Analysis

### 2.1 Requirements

#### 2.1.1 Functional requirements

- Registration process: new users must register in the system by declaring username, password and if they are customers or restaurant owners.
- The system shall handle login process: with a username and password, a user identifies himself/herself within the system, so the system is able to manage all the data concerning him/her.
- The system shall provide appropriate viewers for the users to see list of restaurants available and/or list of reservations of customers/restaurant owners.
- A customer should be able to book a table of a restaurant choosing from a list, selecting date, hour and number of seats.
- The system shall handle the reservation process, managing also the residual available seats in each restaurant (in order to avoid to take a reservation when the restaurant is full).
- A restaurant owner should be able to add, and if necessary modify, all the details of his/her restaurant in the list provided to customers by the application.

#### 2.1.2 Non-functional requirements

**Availability** The application must be always online, in order to be used at any time by users.

**Usability** The application must be easy to use, with a simple and intuitive user interface in order to have an early spreading among users.

**Portability** The application must be portable, so that any user can use it, independently from the system that he/she dispose.

**Data Persistency** The application must use a relational database to achieve data persistency.



**Add Restaurant** (Require Login) When creating a "restaurant owner" type user, a restaurant is created.

**Delete Restaurant** (Require Login) An owner can delete a restaurant, after this operation the owner account will not be deleted, but it will be a Customer account.<sup>1</sup>

**Edit Restaurant** (Require Login) a restaurant owner is able to modify his/her restaurant.

**Add Reservation** (Require Login) Customers can make a reservation at one of the available restaurants.

**Delete Reservation** (Require Login) Customers can delete a reservation made in the past.

**Edit Reservation** (Require Login) Customers can modify a reservation made in the past.

## 2.3 Analysis Classes

In our application we can distinguish three analysis classes. The name of each class is self explanatory. As we can see from the diagram in Figure 2.2 these classes are associated in the following way:

- Users can be divided in Owners and Customers.
- Each Owner can have one and only one Restaurant.
- Each Customer can have none or many Reservations.
- Each Restaurant must have one and only one Owner associated.
- Each Restaurant can have none or many Reservations.
- Each Reservation can be associated to one and only one Customer.
- Each Reservation can be associated to one and only one Restaurant.

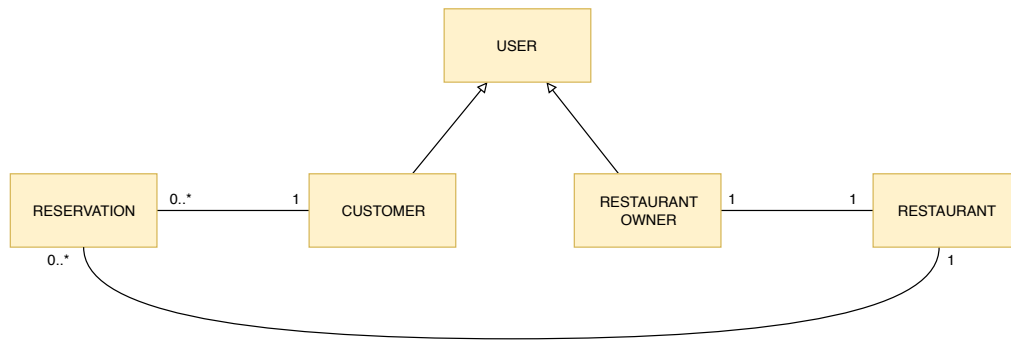
## 2.4 System architecture

The paradigm choosen for our application is Client-Server. Each user will connect to a remote server in order to use the application. The design model is a three-tier application (Presentation, Middleware, Back-end), with a thin client.

---

<sup>1</sup>This functionality is only implemented into the command line interface version of the program



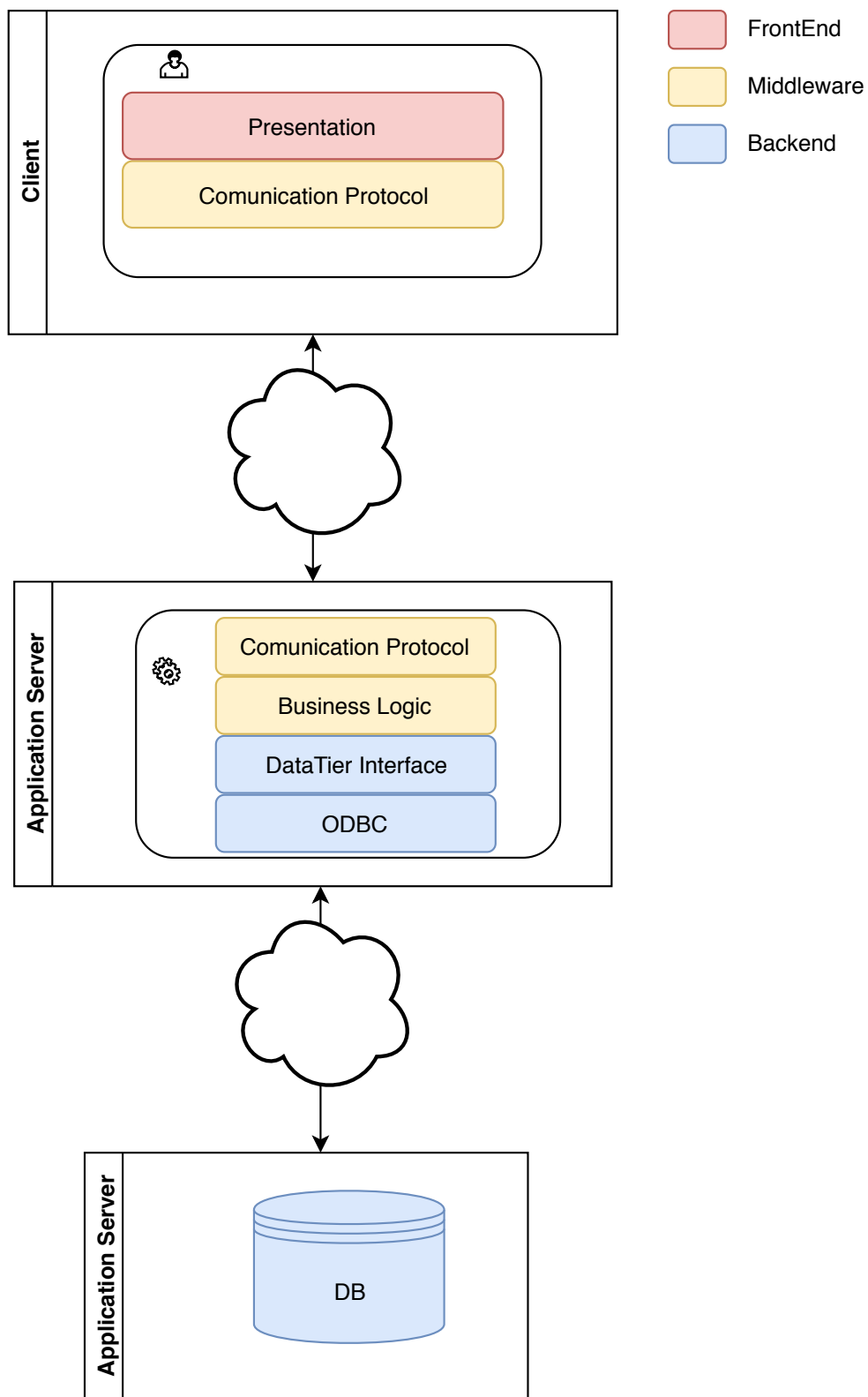


**Figure 2.2:** Analysis classes diagram.

The entire application will be developed in Java, following an object oriented model. The DB used will be an SQL one, more precisely we will use MySQL.

**Client** The client part of the application is composed by two main components, a front end which manage the user interaction and a middleware that will handle the requests from the user and the response from the server side.

**Server** The server part is, similarly to the client, divided in two parts: the middleware to handle the requests coming from the client side and a database manager, the back-end part, which will perform queries on the database.



**Figure 2.3:** System architecture diagram.

## 3 | Implementation

### 3.1 Technologies used

#### 3.1.1 JAVA: Object Oriented Programming Language

The use of Java as programming language guarantees an easy development, thanks to the large amount of APIs and libraries. Furthermore the object oriented model give us an ease in maintenance and a intrinsic modularity of the application. The Java Runtime Environment version used is JAVASE 11.

#### 3.1.2 MySQL: Relational Database Management System

MySQL is one of the most used relational database management systems and it's open-source, it offers a variety of drivers and connectors, including obviously the one for Java programming language. The version used is MySQL 8.0.18.

#### 3.1.3 Java Persistence API: Object to Relational Mapping (Hibernate implementation)

This API allows simplify the Object to Relational Mapping of the business classes, offering an interface which is not dependent from the technology of the relational database. The version used is HIBERNATE 5.4.7.FINAL.

### 3.2 Implementation

In this section we describe the implementation of the main classes of the application. These classes are in part derived from the analysis classes previously mentioned and the remaining are implemented to meet the application requirements. The application is split in three main packages `ristogo.client` which contain the front end, `ristogo.common` which contains some common utilities for server and client and `ristogo.server` which contain the server part of the application.

### 3.2.1 Entities

We preferred to define a super class `Entity` from which to derive the child classes `User`, `Restaurant` and `Reservation`.

```
1 import javax.persistence.*;
2
3 @MappedSuperclass
4 public abstract class Entity_
5 {
6     @Id
7     @GeneratedValue(strategy=GenerationType.IDENTITY)
8     @Column(name="id", nullable=false)
9     protected int id;
10
11     //... constructors, getters, setters ...
12 }
```

Listing 3.1: `Entity_.java`

Then we define the three classes that represent the tables of our database. The class structure is shown below.

```
1 import java.util.*;
2 import javax.persistence.*;
3 import org.hibernate.annotations.*;
4
5 @javax.persistence.Entity
6 @Table(name="users")
7 public class User_ extends Entity_
8 {
9     @Column(name="username", length=32,
10         nullable=false, unique=true)
11     protected String username;
12
13     @Column(name="password", length=32, nullable=false)
14     protected String password;
15
16     @OneToMany(mappedBy="user", fetch=FetchType.LAZY)
17     @LazyCollection(LazyCollectionOption.EXTRA)
18     @Where(clause="date >= CURRENT_DATE()")
19     protected List<Reservation_> activeReservations = new
20         ↳ ArrayList<>();
21
22     @OneToOne(mappedBy="owner", fetch=FetchType.LAZY)
23     @LazyCollection(LazyCollectionOption.EXTRA)
24     protected Restaurant_ restaurant;
25
26     //... constructors, getters, setters ...
27 }
```

Listing 3.2: `User_.java`

```

1 import java.util.*;
2 import javax.persistence.*;
3 import org.hibernate.annotations.*;
4
5 @javax.persistence.Entity
6 @Table(name="restaurants")
7 @DynamicUpdate
8 public class Restaurant_ extends Entity_
9 {
10     @OneToOne(fetch=FetchType.EAGER)
11     @JoinColumn(name="ownerId", nullable=false)
12     protected User_ owner;
13
14     @Column(name="name", length=45, nullable=false)
15     protected String name;
16
17     @Enumerated(EnumType.STRING)
18     @Column(name="genre", nullable=true)
19     protected Genre genre;
20
21     @Enumerated(EnumType.STRING)
22     @Column(name="price", nullable=true)
23     protected Price price;
24
25     @Column(name="city", length=32, nullable=true)
26     protected String city;
27
28     @Column(name="address", length=32, nullable=true)
29     protected String address;
30
31     @Column(name="description", nullable=true)
32     protected String description;
33
34     @Column(name="seats", nullable=false)
35     protected int seats;
36
37     @Enumerated(EnumType.STRING)
38     @Column(name="openingHours", nullable=false)
39     protected OpeningHours openingHours;
40
41     @OneToMany(mappedBy="restaurant", fetch=FetchType.LAZY)
42     @LazyCollection(LazyCollectionOption.EXTRA)
43     @Where(clause="date >= CURRENT_DATE()")
44     protected List<Reservation_> activeReservations = new
45         ↪ ArrayList<>();
46
47     //... constructors, getters, setters, methods ...
48 }

```

Listing 3.3: Restaurant\_.java

```

1 import java.time.LocalDate;
2 import javax.persistence.*;
3
4 @javax.persistence.Entity
5 @Table(name = "reservations")
6 public class Reservation_ extends Entity_
7 {
8     @ManyToOne(fetch=FetchType.EAGER)
9     @JoinColumn(name="userId", nullable=false)
10    protected User_ user;
11
12    @ManyToOne(fetch=FetchType.EAGER)
13    @JoinColumn(name="restaurantId", nullable=false)
14    protected Restaurant_ restaurant;
15
16    @Column(name = "date", nullable=false)
17    protected LocalDate date;
18
19    @Enumerated(EnumType.STRING)
20    @Column(name = "time", nullable=false)
21    protected ReservationTime time;
22
23    @Column(name = "seats", nullable=false)
24    protected int seats;
25
26    //... constructors, getters, setters, methods ...
27 }

```

Listing 3.4: Reservation\_.java

### 3.2.2 Client

The customer can interact easily with the application thanks to a graphical user interface, implemented with JAVAFX library, or he can use the command line interface version (CLI). When the customer issues a command the middleware will handle this **Request Message**, it will serialize it in an XML document and send it to the remote server application, which will produce and send back a **Response Message**. After the response is received from the client the middleware will manage to forward to the upper layer (the GUI) the data just retrieved in a proper format. In Figure 3.1 we can see the UML implementation diagram of the Client side of the application.

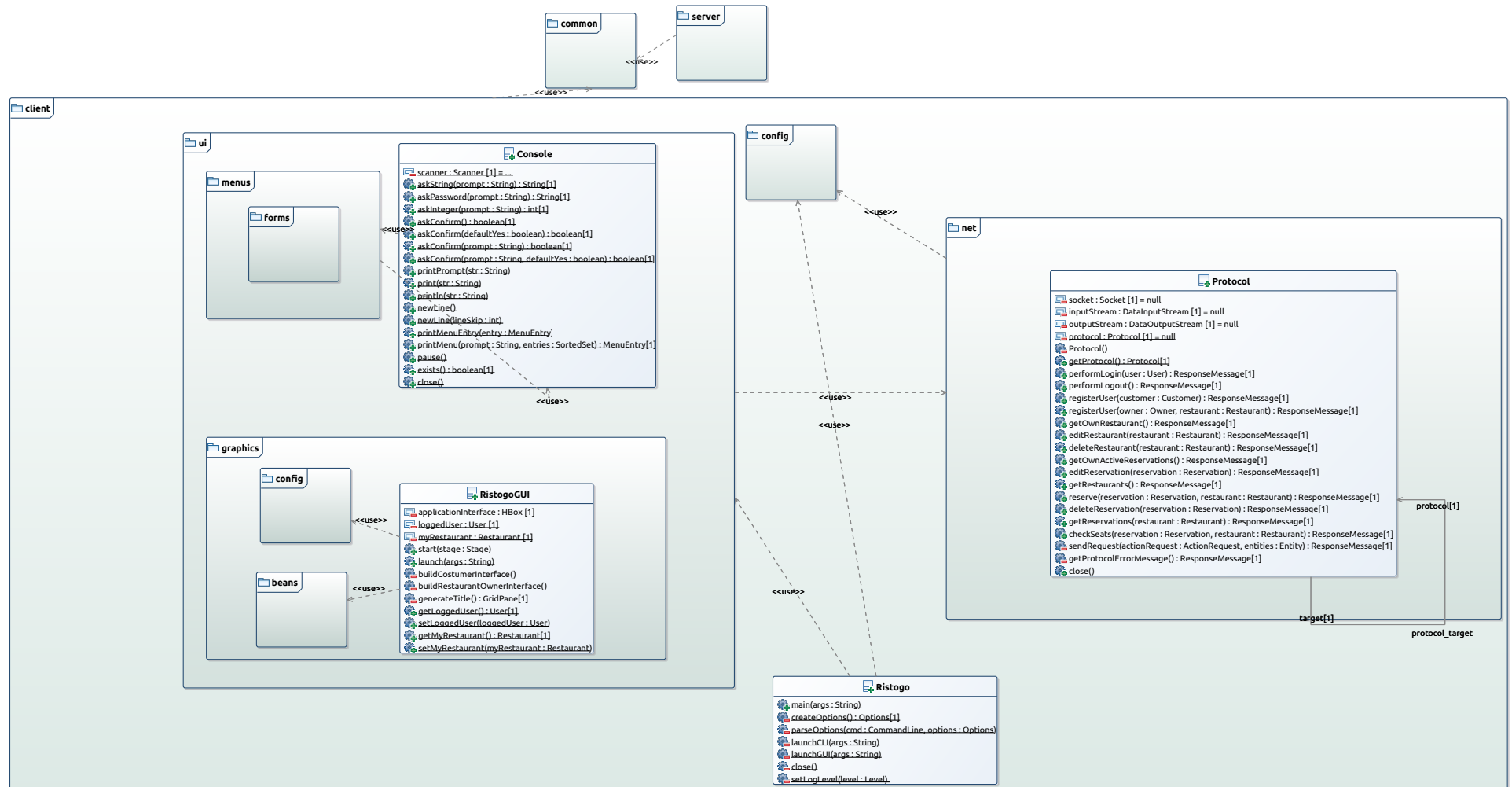


Figure 3.1: ristogo.client UML Implementation Diagram.

### 3.2.3 Common Utilities

This package contains the implementation of serializable classes which can be exchanged among the client and the server in XML format. There are entities classes that represents a simplification of the analysis classes.

Furthermore there are the **RequestMessage** and **ResponseMessage** which are respectively representation of requests from the clients and responses from the server.

These classes are composed by some fields which identifies the type of the request and a set of **Entity** objects that are exchanged among the two parties to fulfil the requests.

The diagram in Figure 3.2 shows the implementation of these classes and their interconnections.



**Figure 3.2:** ristogo.common UML Implementation Diagram.

### 3.2.4 Server

The server is a stateful one, which means that maintain information on the connection. It is composed by a `ThreadPool` which handles the requests from the clients and processes them. The connection persists until the client's logout. After that a connection is established a thread will handle the requests coming from that connection, it will parse that request and issue the correct query to the database, via JPA. After the query is performed the result will be sent back to the client. The diagram in Figure 3.3 shows the implementation of the server's classes.

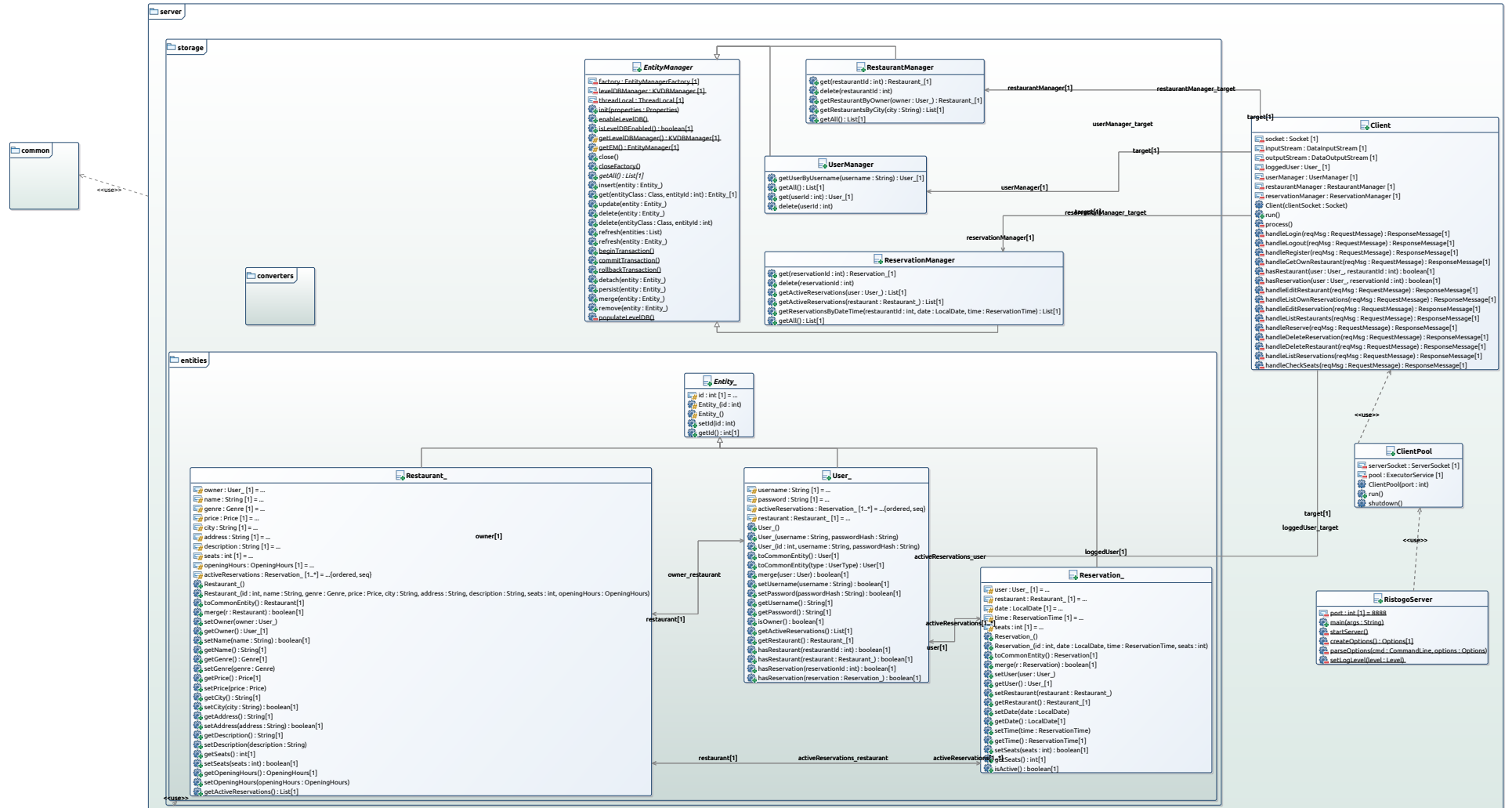
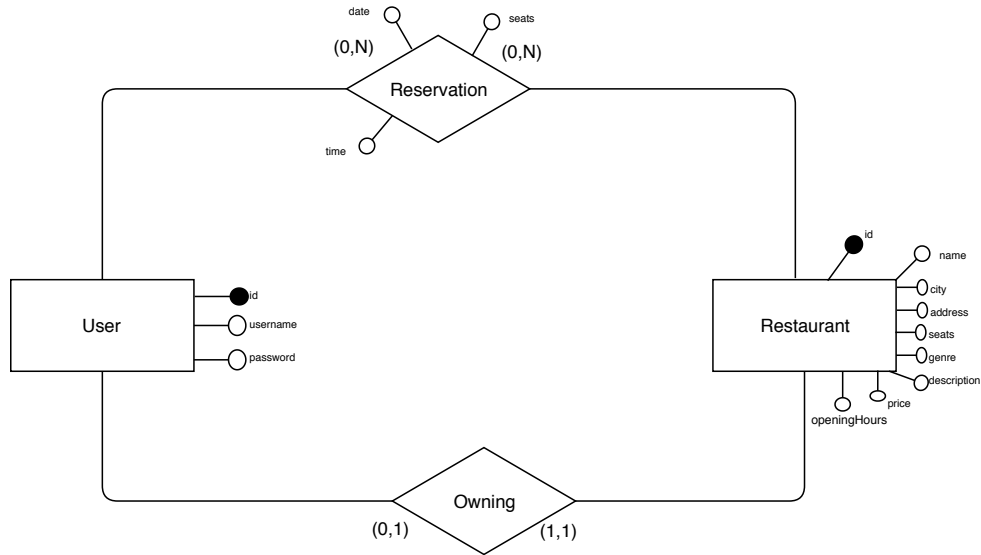


Figure 3.3: ristogo.server UML Implementation Diagram.

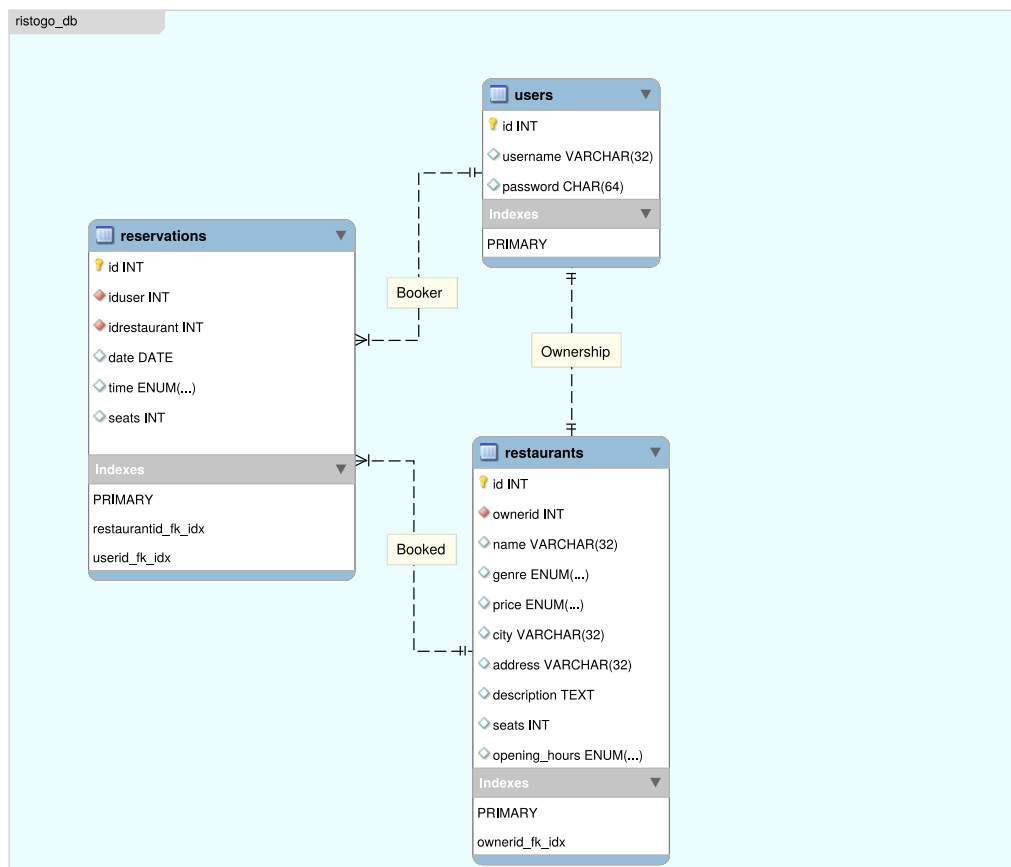
### 3.2.5 Database

The diagram in Figure 3.4 is the logical representation of the DB used by the application.



**Figure 3.4:** ER diagram.

The diagram has been implemented using three tables: as we can see in Figure 3.5.



**Figure 3.5:** DB schema diagram.

## 4 | Feasibility Study On The Use Of A Key Value Database

This study has the objective of determine first of all the possibility and also the advantages or drawbacks, in using a key-value data model in our project. The reference data model is the one designed for the application implemented during this workgroup.

### 4.1 Relational Data Model Analysis

The current data model is the one described in the Figure 3.4.

#### 4.1.1 Volume Table

The first step towards a fair analysis is dimensioning the problem. To do so we have to create a volume table.

Name	E/R	Instances	Justification
User	E	50 000	Initial Assumption
Owning	R	200	We suppose that 2% of users is a restaurant owner
Restaurant	E	200	Every restaurant owner has only 1 restaurant (Cardinality (1,1))
Reservation	R	1 500 000	On average a user has 30 reservations per year, and that an owner in his restaurants has on average 20 reservation per day (7300 per year).

#### 4.1.2 Possible Operations

After dimensioning the problem we provide a set of CRUD operations that can be performed on the database. Each operations has a frequency defined as the number of times the operation is performed per day.

##### Login

- Description: Find a user in the system.
- Frequency: 35 000 per week.

### **Register**

- Description: Insert a new user.
- Frequency: 100 per week.

### **List of Restaurants**

- Description: A customer request the list of all the available restaurants.
- Frequency: 175 000 per week (5 times per login session on average).

### **List of Reservations per User**

- Description: Find all the reservation of a given user.
- Frequency: 175 000 per week (Considering after a login and after each booking we ask for a new list of reservation this operation is performed 5 times per week for each user).

### **List of Reservations per Restaurant**

- Description: Find all the reservations for a given restaurant.
- Frequency: 140 000 per week (every owner will check multiple times the list of reservation (10 times per day) at his restaurant we assume on average 70 per week for each user).

### **Modify a Restaurant**

- Description: Update the informations of a given restaurant.
- Frequency: 25 per week (we assume that these informations are not so likely to change frequently).

### **Book a table in a Restaurant**

- Description: Insert a reservation.
- Frequency: 105 000 per week.

### **Check available seats in a Restaurant**

- Description: Find the number of seats available in a restaurant.
- Frequency: 210 000 per week (considering that a customer will perform on average 3 reservation per week, may happen that some restaurant are full or a customer just want to know if there are seats available, so there will be slightly more check then reservations).

### **Delete a Reservation**

- Description: Remove a reservation from the database.
- Frequency: 525 per week (0,5% of weekly reservations).

### **Add a Restaurant**

- Description: Add a restaurant in the system.
- Frequency: 3 per week.

### **4.1.3 Analysis Results**

The most important thing that we can see from these analysis is that the database is used mostly for read operations. The prominent operations are:

- List of Restaurant (175 000 access/week)
- List of Reservation per User (175 000 access/week)
- List of Reservation per Restaurant (140 000 access/week)
- Check available seats in a Restaurant (210 000 access/week)

Starting from this situation we must consider the complexity in terms of join operations that must be performed for each operation.

#### **List of Restaurants**

Read all the instances in restaurants. It is not necessary any join operation.

#### **List of Reservation per User**

In this operation we need a join between Restaurant and Reservation tables. We need the name of each Restaurant where the User with given userID have a Reservation.

#### **List of Reservation per Restaurant**

In this operation we need a join between User and Reservation tables. We need the name of each user that have a Reservation in the Restaurant with given restaurantID.



## Check available seats in a Restaurant

In this operation we need a join between Restaurant and Reservation table. We need the number of remaining seats in a given restaurant for the given date and hour. The first operation, even if it's a frequently used read

operation, it doesn't require any join. Instead, the last three operations are frequently used read operations that need a join. They are the ones that most will achieve advantages in terms of performances using a Key-Value Database among the above listed ones. In particular, the key-value database will be used as a cache of the relational database for the aforementioned read operations.

### 4.1.4 Data Model Translation

In this step we have to create a key value data model that fits our needs. In other words we need a key-value data model that can be derived from the join of three tables involved in the previously mentioned operations.

Starting from the ER model in Figure 3.4 we can translate our join table as follow:

**reservations:<userId>:<restaurantId>:<date>:<time>:<attribute name>=value**

Attribute name can be:

- id
- seats
- user\_username
- restaurant\_name

The introduction of the fields <date> and <time> in the key give us an advantage when we need to check for available seats. In fact when we need this operation we just have to iterate on the reservations with given date and time without the need to check for that.

## 4.2 Key-Value DataBase Implementation

### 4.2.1 Technologies used: LevelDB

To implement our key-value database we used LEVELDB, a library written at Google that provides a fast key-value persistent data storage engine.

More precisely we used the Java porting of LEVELDB packaged as `org.iq80.leveldb`, which have full code available on GITHUB.

```
1 <dependency>
2   <groupId>org.iq80.leveldb</groupId>
3   <artifactId>leveldb</artifactId>
4   <version>0.12</version>
5 </dependency>
```

**Listing 4.1:** Maven dependency for LevelDB.

This database can be used only via its API, that provides a tiny set of basic operations (put, get, delete) that we can exploit to implement more complex functions.

Furthermore it provides snapshots, support for atomic operations and an iterator to allow us to flow among the entries and examine them one at a time.

### 4.2.2 UML Implementation Diagram

In Figure 4.1 is shown the diagram where we can see the classes that have been implemented to introduce the Key Value database in the application.

The classes have been packaged inside `ristogo.server.storage.kvdb`.

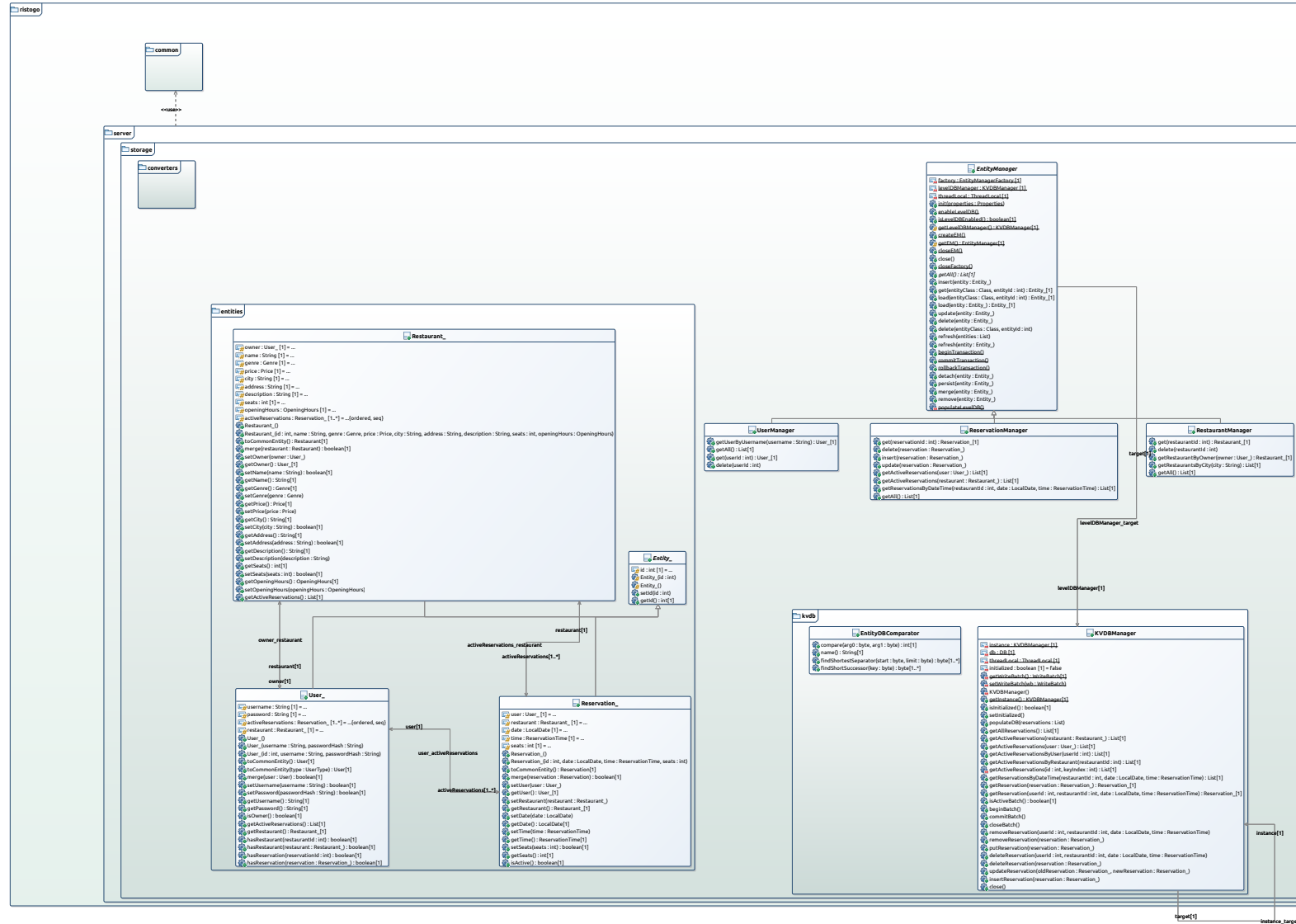


Figure 4.1: ristogo.server.storage.kvdb UML Implementation Diagram.

The behaviour of this datastore is as described previously.

When the server application starts it builds the key value database starting from the data present in the relational database. This operation is performed just if the key-value database (from now shortened as kvdb) is not present in the system, then the application will manage the synchronization among the two databases, keeping the consistency.

When a request arrives at the server, if this request implies a read operation the **EntityManager** involved will read from the kvdb.

If instead the request implies a write operation it is handled in a write-through fashion. The **EntityManager** involved will write into the kvdb and into the relational database.

## 5 | User manual

### 5.1 Introduction

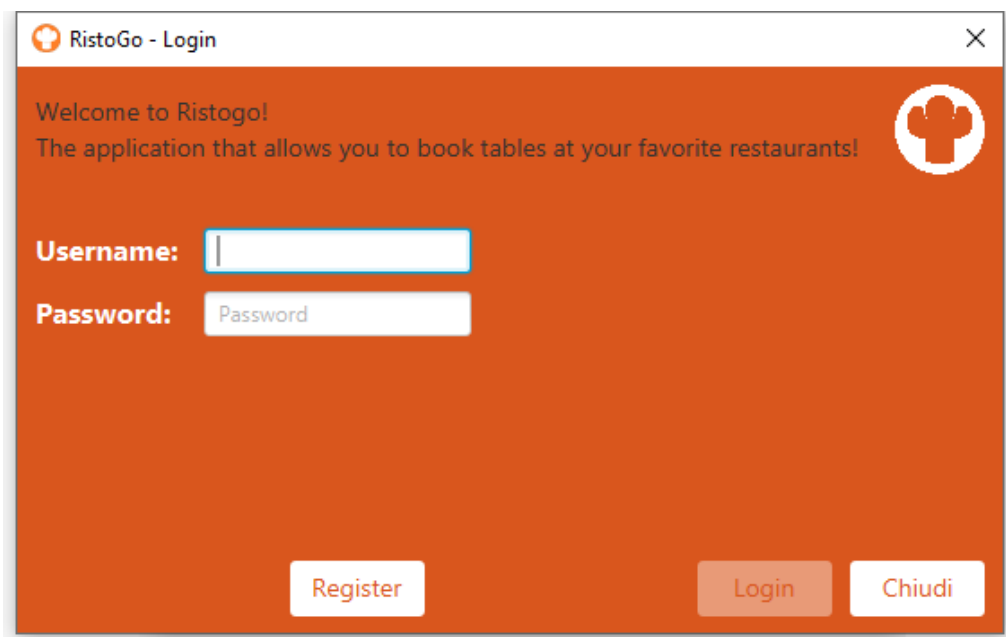
Ristogo is an application that can be used by customers and restaurateurs.

Customers can book a table and modify later their reservations.

Restaurateurs can see the list of daily reservations to organize the work.

The application can be launched with a Graphical User Interface (default) or with a Command Line Interface. This manual mainly speaks about the operations that can be done using the GUI.

### 5.2 Login and Registration



**Figure 5.1:** Login interface.

When the application starts it opens the main page (Figure 5.1), on which an old user can insert username and password to login, clicking on the button “Login”.

Otherwise, if the user is a new one, he must do the registration procedure to use the application.

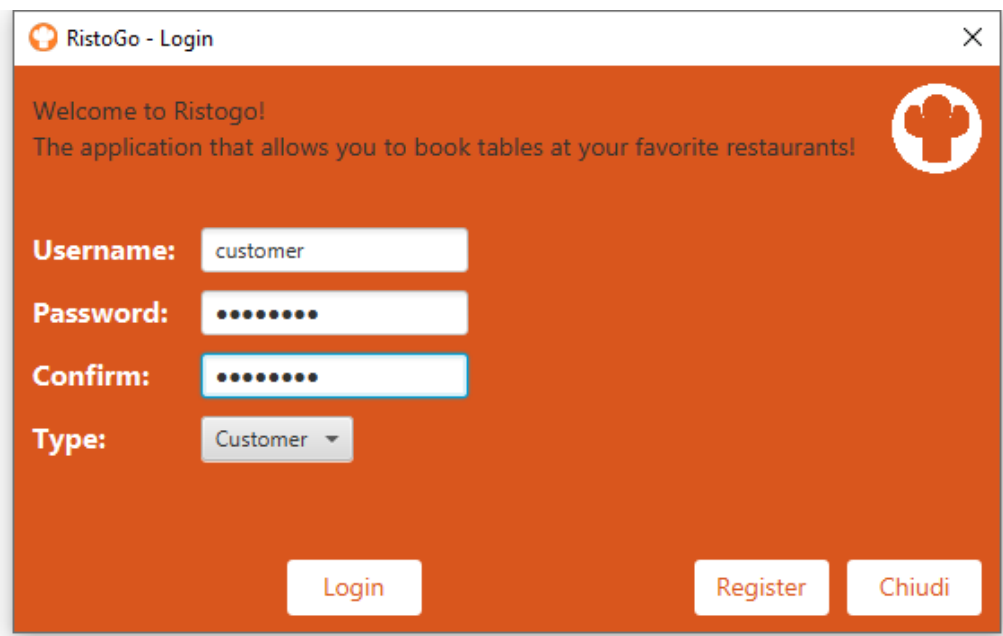
If you want to close the application, click on the button “Close”.

### 5.2.1 Registration

If you are a new user, click on “Register” on the main page to open the registration form (Figure 5.2).

Then, insert your username and your password (the password must contain at least 8 characters). Insert another time your password to confirm and, in the section “type of user”, select **Customer** if you want to use the application for book tables or select **Owner** if you want to insert your restaurant in the application to manage its reservations (Figure 5.3).

If you went in the registration page wrongly, click on “Login” to return to the login page.

The image shows a web browser window titled "RistoGo - Login". The page has an orange background. At the top, it says "Welcome to Ristogo!" and "The application that allows you to book tables at your favorite restaurants!". There is a logo on the right. Below the text, there are four input fields: "Username:" with the value "customer", "Password:" with masked characters, "Confirm:" with masked characters, and "Type:" with a dropdown menu showing "Customer". At the bottom, there are three buttons: "Login", "Register", and "Chiudi".

**Figure 5.2:** Registration of a Customer user.

When you filled out the form, click on “Register”. Now if you don’t receive error messages your account will be create. So you can start to use the application for the first time.

## 5.3 Customer interface

When you are logged with a customer account in the main page you can see all the restaurants managed by this application and all your reservations (Figure 5.4).

**RistoGo - Login**

Welcome to Ristogo!  
The application that allows you to book tables at your favorite restaurants!

**Username:**

**Password:**

**Confirm:**

**Type:**

**Figure 5.3:** Registration of a Owner user.

**RistoGo**

Welcome **customer**

**Book a table**

To select a restaurant click on the table on the side

**Name of Restaurant:**

**Date of Reservation:**

**Booking Time:**

**Seats:**

**List of Restaurant: you can find restaurants searching by city**

insert a name of a city

Name	Type	Price	City	Address
The Pirate Cabin	Italian	High	Donoratico	Del Ciproso,
Pizza to Luca	Pizza	Low	Pisa	Milano, 14
Tavern of Cason	Italian	Middle	Cascina	Torino, 78
HomeHam	SteakHouse	Middle	Pisa	Diotisalvi, 1
To Mexican	Mexican	Middle	Pisa	Giacomo Leopardi

**My Reservation**

Name	Date	Hour	Seats
RistoPizza	2019-12-01	Dinner	2

**Figure 5.4:** Customer interface.

### 5.3.1 Search restaurants

If you want to see what are the managed restaurants into a desired city, insert the name of the city in the text field near the button “Find” and then click it (Figure 5.5).

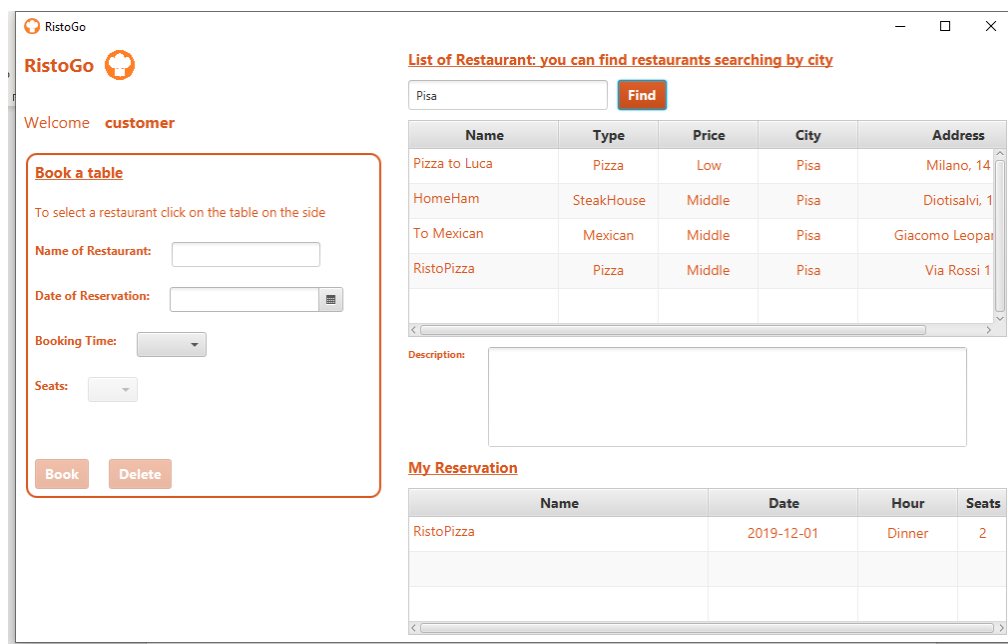


Figure 5.5: Search restaurants by city.

### 5.3.2 Book a table

To book a table, select your desired restaurant from the list. Its description will appear in the correspondent side. Now in the section “Book a table” you can find the name of the selected restaurant (Figure 5.6).

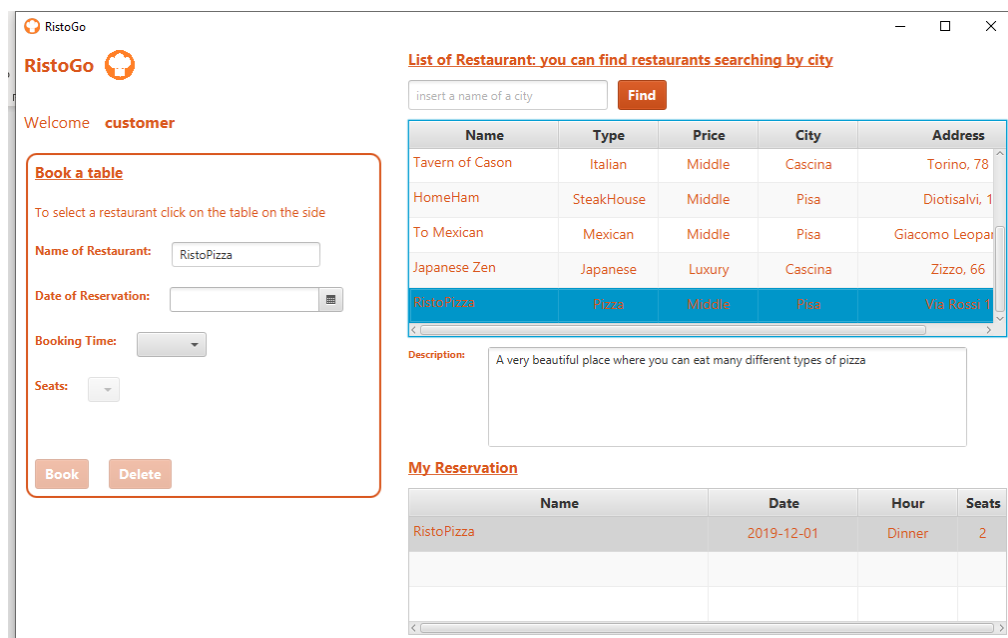


Figure 5.6: Select a restaurant to book a table.

Select the date, the booking time (Lunch or Dinner) and check if there is the number of seats available that you need. If there are, select that and



click on “Book” to confirm (Figure 5.7).

**RistoGo**  
Welcome **customer**

**Book a table**  
To select a restaurant click on the table on the side

Name of Restaurant:

Date of Reservation:

Booking Time:

Seats:

**Book** **Delete**

**List of Restaurant: you can find restaurants searching by city**  
insert a name of a city **Find**

Name	Type	Price	City	Address
Tavern of Cason	Italian	Middle	Cascina	Torino, 78
HomeHam	SteakHouse	Middle	Pisa	Diotisalvi, 1
To Mexican	Mexican	Middle	Pisa	Giacomo Leopardi, 66
Japanese Zen	Japanese	Luxury	Cascina	Zizzo, 66
RistoPizza	Pizza	Middle	Pisa	Via Rossi 1

Description: A very beautiful place where you can eat many different types of pizza

**My Reservation**

Name	Date	Hour	Seats
RistoPizza	2019-12-01	Dinner	2

**Figure 5.7:** Fill the form to book a table.

If there are no errors, you can find your new reservation into the section “My Reservation” (Figure 5.8).

**RistoGo**  
Welcome **customer**

**Book a table**  
To select a restaurant click on the table on the side

Name of Restaurant:

Date of Reservation:

Booking Time:

Seats:

**Book** **Delete**

**List of Restaurant: you can find restaurants searching by city**  
insert a name of a city **Find**

Name	Type	Price	City	Address
The Pirate Cabin	Italian	High	Donoratico	Del Cipresso, 14
Pizza to Luca	Pizza	Low	Pisa	Milano, 14
Tavern of Cason	Italian	Middle	Cascina	Torino, 78
HomeHam	SteakHouse	Middle	Pisa	Diotisalvi, 1
To Mexican	Mexican	Middle	Pisa	Giacomo Leopardi, 66

Description:

**My Reservation**

Name	Date	Hour	Seats
RistoPizza	2019-12-01	Dinner	2
RistoPizza	2019-12-21	Dinner	10

**Figure 5.8:** Added reservation shows in the “My Reservation” table.

### 5.3.3 Delete a reservation

If you want to delete a reservation, select it on the section “My Reservation” (Figure 5.9).

The screenshot shows the RistoGo web application interface. On the left, there is a sidebar with the RistoGo logo and a 'Welcome customer' message. The main content area is divided into two sections. The top section, titled 'Book a table', contains a form for booking a table. The form has fields for 'Name of Restaurant' (RistoPizza), 'Date of Reservation' (21/12/2019), 'Booking Time' (Dinner), and 'Seats' (10). There are 'Save' and 'Delete' buttons at the bottom of the form. The bottom section, titled 'My Reservation', contains a table with reservation details. The table has columns for Name, Date, Hour, and Seats. The first row shows a reservation for RistoPizza on 2019-12-01 at Dinner for 2 seats. The second row shows a reservation for RistoPizza on 2019-12-21 at Dinner for 10 seats. The second row is highlighted in blue.

Name	Type	Price	City	Address
The Pirate Cabin	Italian	High	Donoratico	Del Ciproso,
Pizza to Luca	Pizza	Low	Pisa	Milano, 14
Tavern of Cason	Italian	Middle	Cascina	Torino, 78
HomeHam	SteakHouse	Middle	Pisa	Diotisalvi, 1
To Mexican	Mexican	Middle	Pisa	Giacomo Leopardi,

Name	Date	Hour	Seats
RistoPizza	2019-12-01	Dinner	2
RistoPizza	2019-12-21	Dinner	10

Figure 5.9: Delete a reservation.

Then, click on the button “Delete” situated on the section “Book a table” (for this operation the section is empty). If there are no errors, you can see that the reservation no longer appears in the section “My reservation” (Figure 5.10).

## 5.4 Owner interface

When you do the registration as an owner, the application will create automatically a restaurant with the name equal to your selected username (for example, a new user named “User1” has a default restaurant’s name which is “User1’s Restaurant”).

Figure 5.11 shows the Owner interface.

### 5.4.1 Modify restaurant informations

After the first login you can change your restaurant’s informations, filling the form on the section “Modify your restaurant”. Then click on the button “Commit” (Figure 5.12).

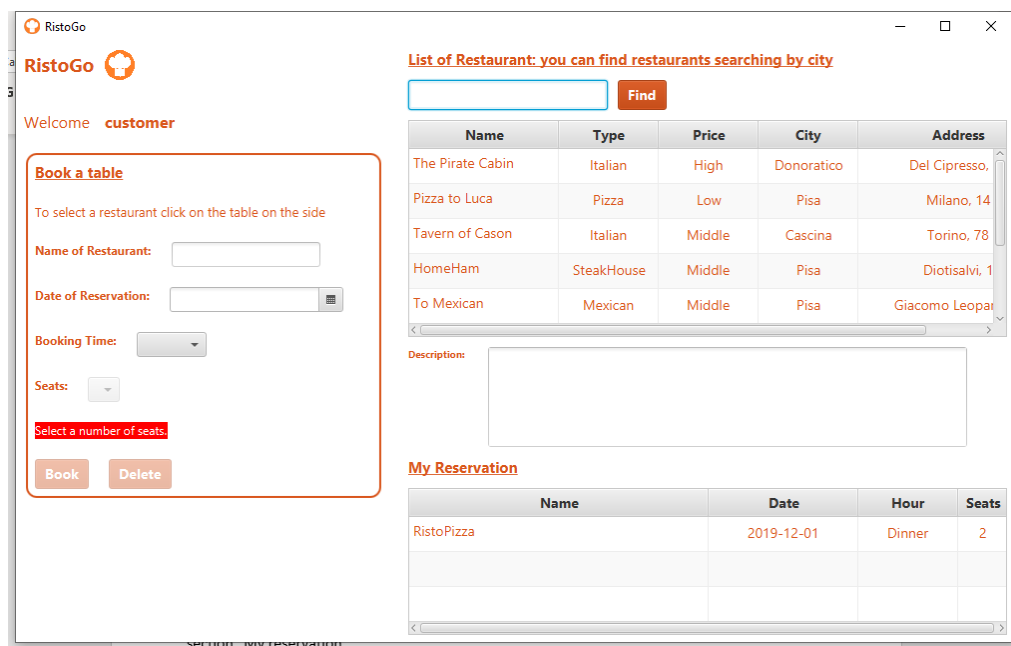


Figure 5.10: Reservation deleted.

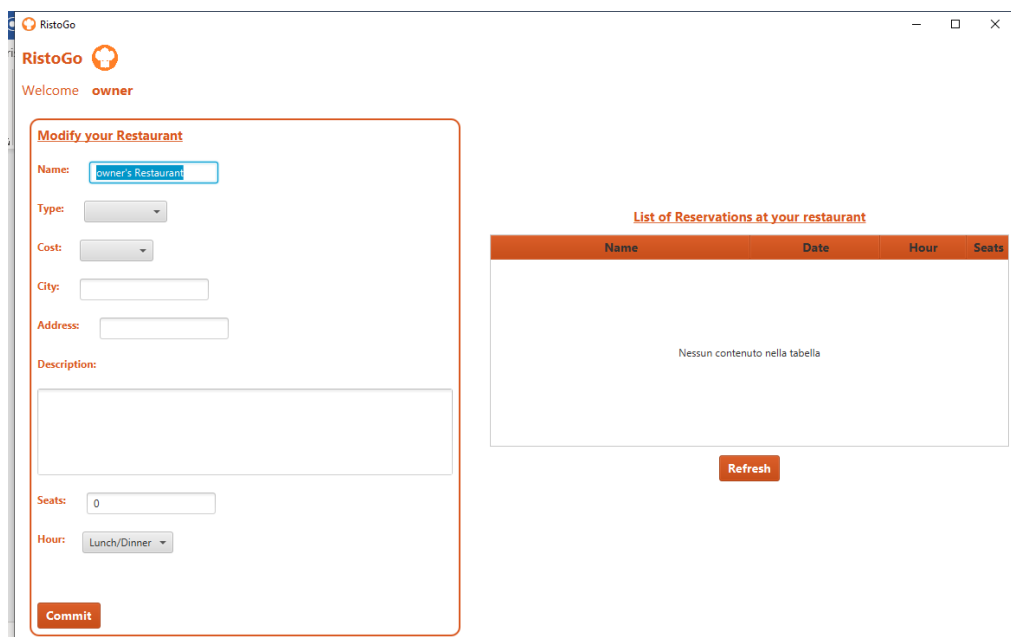


Figure 5.11: Owner interface.

## 5.4.2 Manage reservations

In the section “List of reservations” you can see all your active reservations. The ones of past days are not shown. The list is not updated automatically. To see the changes, you have to click on button “Refresh” (Figure 5.13).

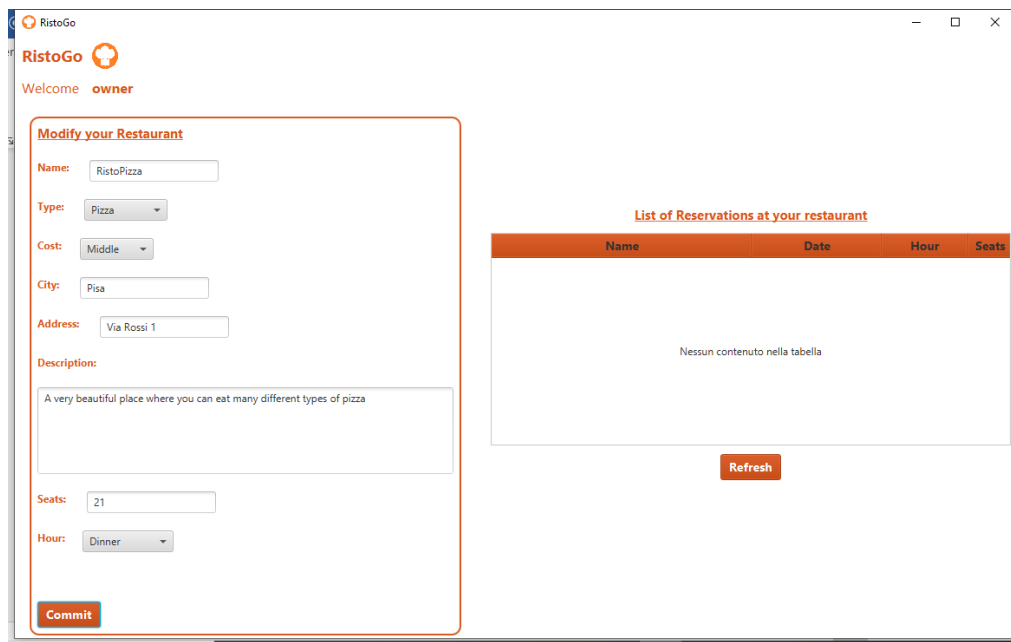


Figure 5.12: Modify restaurant's informations.

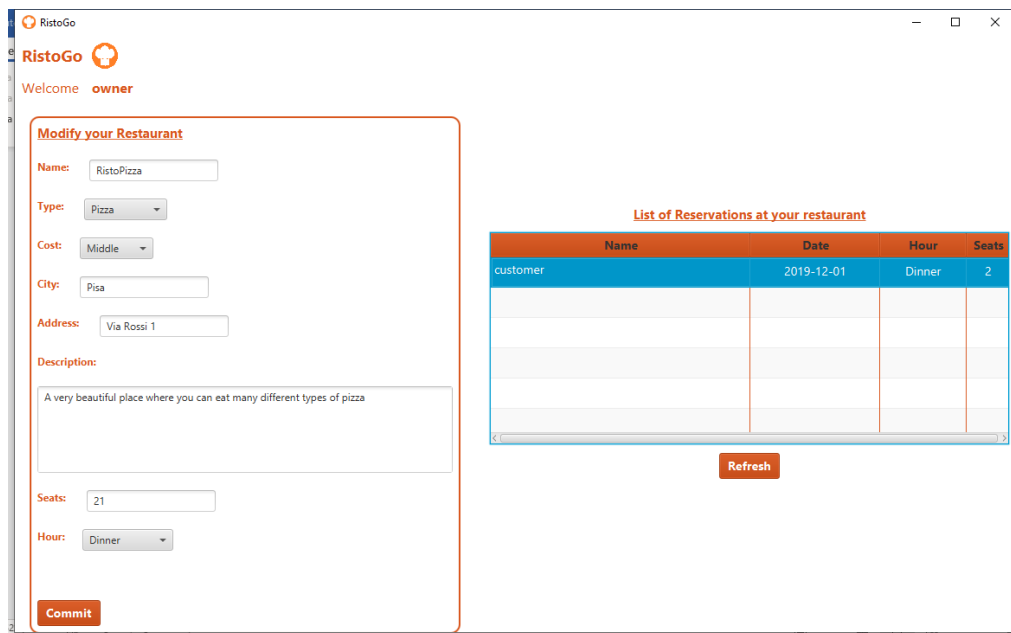


Figure 5.13: Reservation list.

## 5.5 Command Line Interface

The application can be launched also without the GUI, with options selected by command line.

To reach this purpose, launch the application with the option `-c` (or `--cli`). Then follow the instructions written into the console to use the application.

WELCOME TO RISTOGO!

Select an action:

- 1) Log-In
- 2) Register
- 0) Exit

Enter number:

## 5.5.1 Command line options

### Client

To see all the available options, launch the application with the option `-h` (`--help`):

```
usage: ristogo [-h | --help] [-c | --cli] [-g | --gui] [-l
               <LEVEL> | --log-level <LEVEL>]
  -c,--cli           force load the Command Line
                    Interface.
  -g,--gui           force load the Graphical User
                    Interface.
  -h,--help          print this message.
  -l,--log-level <LEVEL> set log level.
```

LOG LEVELS:

ALL: print all logs.  
FINEST: print all tracing logs.  
FINER: print most tracing logs.  
FINE: print some tracing logs.  
CONFIG: print all config logs.  
INFO: print all informational logs.  
WARNING: print all warnings and errors. (default)  
SEVERE: print only errors.  
OFF: disable all logs.

### Server

```
usage: ristogoserver [-h | --help] [-H <HOST> | --host
                        <HOST>] [--dbport <PORT>] [-u <USER>
                        | --user <USER>] [-p <PASS> | --pass
                        <PASS>] [-L | --leveldb] [-P <PORT>
                        | --port <PORT>] [-l <LEVEL> |
                        --log-level <LEVEL>]
  -dbport <arg>      Specify MySQL database port
                    (default: 3306).
```

<code>-h,--help</code>	print this message.
<code>-H,--host &lt;PORT&gt;</code>	Specify MySQL database hostname (default: localhost).
<code>-L,--leveldb</code>	Enable LevelDB database.
<code>-l,--log-level &lt;LEVEL&gt;</code>	set log level.
<code>-p,--pass &lt;PASS&gt;</code>	Specify MySQL database password (default: <empty>).
<code>-P,--port &lt;PORT&gt;</code>	set listening port (default: 8888).
<code>-u,--user &lt;USER&gt;</code>	Specify MySQL database username (default: root).

#### LOG LEVELS:

**ALL:** print all logs.  
**FINEST:** print all tracing logs.  
**FINER:** print most tracing logs.  
**FINE:** print some tracing logs.  
**CONFIG:** print all config logs.  
**INFO:** print all informational logs.  
**WARNING:** print all warnings and errors. (default)  
**SEVERE:** print only errors.  
**OFF:** disable all logs.

## 5.6 Configuration file

The client application can also be configured with a configuration file. The configuration file must be put in the same directory of the application with the name `config.xml`.

```

1 <Configuration>
2   <serverIp>127.0.0.1</serverIp>
3   <serverPort>8888</serverPort>
4   <interfaceMode>AUTO</interfaceMode>
5   <fontName>Open Sans</fontName>
6   <fontSize>14</fontSize>
7   <bgColorName>FFFFFF</bgColorName>
8   <fgColorName>D9561D</fgColorName>
9   <numberOfRowsDisplayable>7</numberOfRowsDisplayable>
10  <logLevel>WARNING</logLevel>
11 </Configuration>

```

**Listing 5.1:** `config.xml`

**serverIp** Specifies the IP of the server (default: 127.0.0.1).

**serverPort** Specifies the port of the server (default: 8888).

**interfaceMode** It can be set to: `FORCE_CLI` to force the application to load the CLI; `FORCE_GUI` to force the application to load the GUI;

**AUTO**<sup>1</sup> to let the application automatically choose between the GUI and the CLI (default: **AUTO**).

**fontName** Specifies the name of the font to use in the GUI (default: Open Sans).

**fontSize** Specifies the name of the font to use in the GUI (default: 14).

**bgColorName** Specifies the background color to use in the GUI (default: FFFFFFFF).

**fgColorName** Specifies the foreground color to use in the GUI (default: D9561D).

**numberOfRowsDisplayable** Specifies the number of rows to display in tables in the GUI (default: 7).

**logLevel** Specifies the log level. Acceptable values are the same of the `--log-level` option (default: **WARNING**).

## 5.7 Build standalone JARs

To build standalone JAR archives for both the client and the server, you can use Maven:

```
cd RistogoCommon/
mvn package
mvn install:install-file \
  -Dfile=target/RistogoCommon-1.0.0.jar \
  -DgroupId=RistogoCommon \
  -DartifactId=RistogoCommon \
  -Dversion=1.0.0 \
  -Dpackaging=jar \
  -DgeneratePom=true
cd ../

cd RistogoServer/
mvn package
cd ../

cd Ristogo/
mvn package
cd ../
```

---

<sup>1</sup>This functionality is based on the presence of a console attached to the standard input/output. On Unixes systems, that always attach a console to the application (even if launched by a double-click), this functionality may not work. Use the `--gui` option (or set the config option to **FORCE\_GUI** to load the GUI).

Then, the two standalone JARs will be available in the following directories:

**Server** RistogoServer/target/RistogoServer-1.0.0.jar.

**Client** Ristogo/target/Ristogo-1.0.0.jar.