Programmation répartie et Services Web

Group Project

CURRERI Dario

DI FINA Domenico

DOMINGO Emanuele

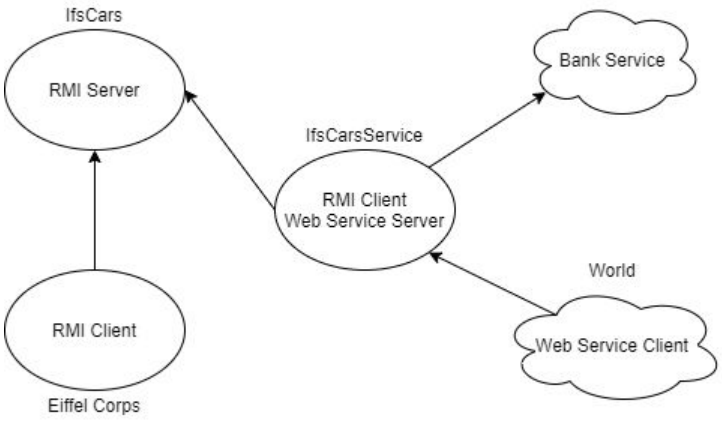
GRISTINA Salvatore Antonino

29 november 2020

Source code available on **GitHub**

1. Introduction

1.1 Main structure



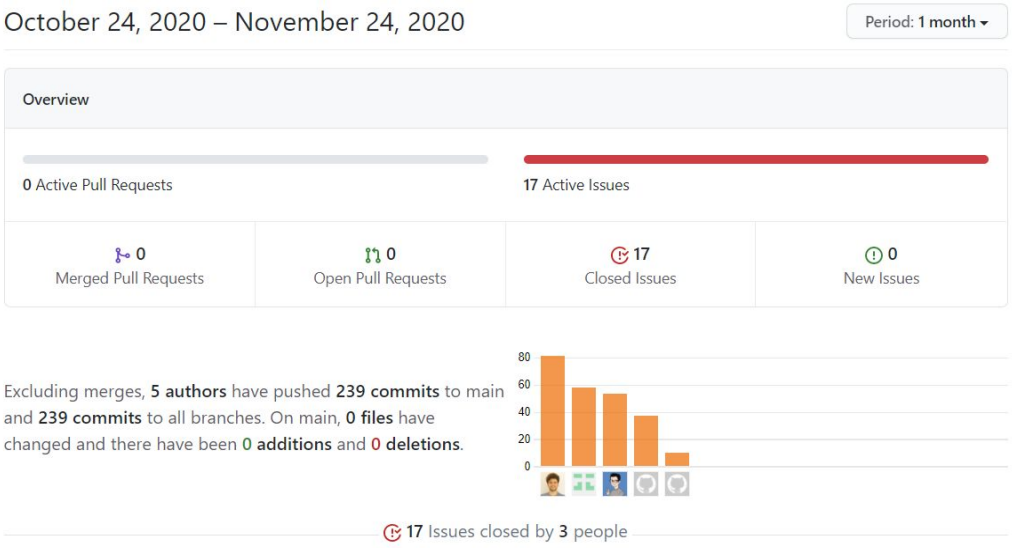
Our project is structured in 5 sub-projects:

- **IfsCars**: the RMI Server that offers the vehicles from the company IfsCars in a special channel for *EiffelCorp's* employees.
- **EiffelCorp**: the RMI Client of *IfsCars* that makes possible to rent vehicles for the employees of the company.
- **IfsCarsService**: the Dynamic Web Project that offers to the world the possibility to buy vehicles rented at least once, via web service. This project is also an RMI Client for *IfsCars*, to get the information about the vehicles and a Web Service Client for *BankWebService* to perform the payment of the purchased vehicle.
- **IfsCarsServiceClient**: is the client of the web service that allows you to see and buy vehicles.
- **BankWebService**: is a Dynamic Web Project that checks the availability of funds for the purchase, making the payment in more currencies.

Due to the fact some single classes are required in more projects, we decided to create a 'common' package that includes this type of classes, shared between all the projects. This package is de facto the ancestor of a personal library needed for all projects in order to communicate with each other.

1.2 Coding approach

One of the requirements of the project was to collaborate within a group of 4 people. We decided to use the Git protocol to share code between the members of the group, enable versioning and manage the team remotely. We also used the “Issue” feature of Github to request code to another team member or fix bugs.



1.2.1 Interface-based programming

As explained **before**, interfaces are the basis of the project's structure: so most of the references in the project, except the gui’s ones, are references to interfaces in order to abstract as much as possible. The use of interfaces also achieves loose coupling, guarantee of reliability, and so on.

1.2.2 Passing *byValue*

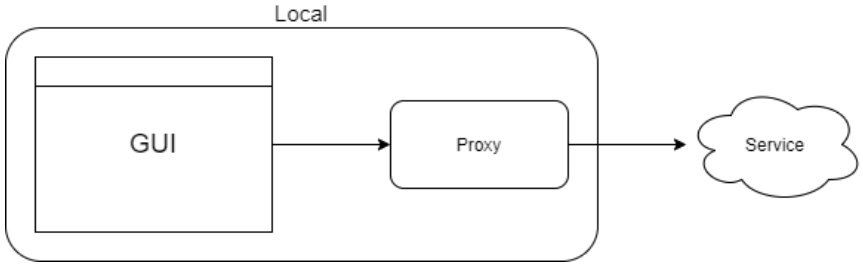
There are two different approaches to pass parameters between remote methods invocation:

- *byValue:* make a copy in memory of the actual parameter's value that is passed in;
- *byReference:* pass the real object. Then the object could be modified by client-side.

We chose the *by value* approach because we wanted to manage all the data structures by server-side, in order to control the operations and don’t allow the client to manage data without server supervision. To achieve this goal, we extended the *Serializable* interface defined by *Java* in all the classes involved.

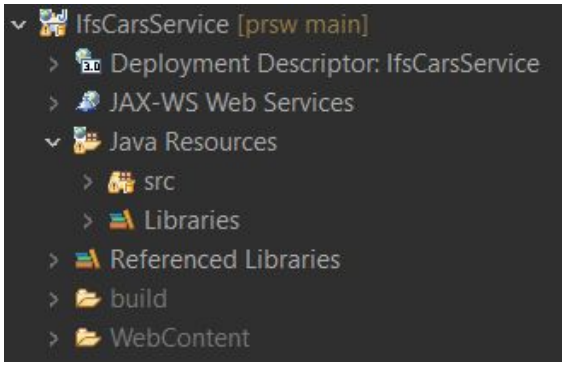
1.2.3 Proxies

One of the "*best practises*" that we adopted is to separate the graphical interface from the business logic to avoid the direct communication among GUIs and servers. In order to make it possible we decided to create a proxy, which contains the service entrypoint, between the gui class and the service class.



1.3 Encountered problems

The first problems we encountered are related to creating and configuring projects with Github. We have in total five projects and one tomcat server instance. Every project comprends more folders in addition to the source folder, so we have decided to synchronize just the "*src*" folder and exclude all the others. To manage it, each member of the group created his own projects and then synchronized the folder with git.



The second problem was related to the JAX-RPC specifications: we used java.util.List as data structure to exchange the lists of vehicles between the IfsCarsWebService and its client. Unfortunately, List is not supported by the JAX-RPC 1.1. In fact, in chapter 5, "Java to XML/WSDL Mapping" of the JAX-RPC specification explains which Java types are supported. These include:

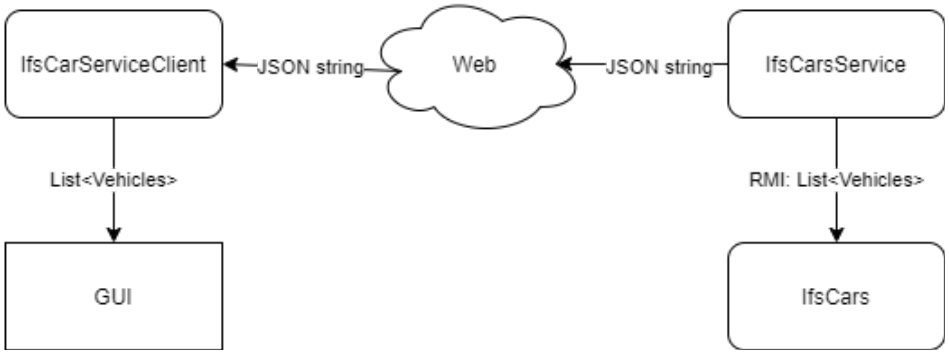
The following are the JAX-RPC supported Java types:

- *One of the Java primitive types as specified in the section 5.1.1*
- *A subset of the standard Java classes (as specified in the J2SE APIs) as specified in the section 5.1.3*
- *An array of a supported Java type as specified in the section 5.1.2*
- *An exception class as specified in the section 5.2.1*
- *A JAX-RPC value type as specified in the section 5.4*

A JAX-RPC runtime system implementation must support transmission of the values of a JAX-RPC supported Java type between a service client and service endpoint at runtime. Values of a JAX-RPC supported Java type must be serializable to and from the corresponding XML representation.

***source**: https://cysun.org/archive/S07/cs520/extras/jaxrpc-1_1-fr-spec.pdf*

To solve this problem, we decided to implement methods that are able to convert a list of vehicles into a JSON file, and parse one JSON file to reconstruct the list



1.4 Remarks

To implement this project we used old technologies, this was the reason for several incompatibilities with the new java standards, also with the Eclipse environment and the different S.O of our machines. If there are any problems during the building, we guess that this problem concerns to this.

2. IfsCars

This project is a Java application that manages the rental service for the employees based on Java RMI. IfsCars vehicles can be rented by all Eiffel Corp employees. In particular, these employees can rent these vehicles with a special discount using a discount code, and they can add notes on the vehicles and their condition when they return them.

This application manages the vehicle database and implements all requisites specified. For instance, when a person requests a rental vehicle, and the vehicle is already on loan to another person, the person is placed on a waiting list and as soon as the requested vehicle becomes available, the first person in the queue on that vehicle, is notified and he/she rents the vehicle.

To simulate the vehicles database we used a JSON file which contains the information about the vehicles offered by the rental service. if it's permitted, we can say that we used an archaic version of NoSQL database in this project.

Note: the discount code used in our simulations is EMP001 and offers a 10% discount on the price.

2.1 CarRental

The class CarRental is the core of this project. It simulates the behaviour of a CarRental and it implements all the methods required by the ICarRentalObservable interface. As can be easily understood from the name of the interface, this Class implements the Observable design pattern, overriding the notifyObservers, attach and detach methods. These methods are used to notify the employees in the waiting list.

In the CarRental class can also be found all the methods that allow the users to visualize, rent or return a car from the graphical user interface.

2.2 CarRentalServer

This is the class that starts the server. In particular:

- it creates and exports a Registry instance on the local host that accepts requests on the specified port
- it sets the policy allowing all permissions
- it registers the service calling the Naming.rebind method

2.3 Encountered problems

We had some problems with the server policy and reading the json file, but both were solved with a minimum effort. To read json files we used the org.json.simple external library.

3. EiffelCorp

This project is the RMI client, it also contains the graphic user interface (GUI) which allows the employee an easy way to rent or return vehicles. In particular, this project manages the access to the rental system with a login form, it also manages all the information about the employees including the way they are stored.

3.1 ClientProxy

The class ClientProxy is the bridge between the client and the server. It allows us to use all the methods in CarRental using the lookup method to take the reference of the CarRental's object.

It also allows us to load the list of the employees from the json file where they are stored, in the same ways of vehicles seen in the previous section.

3.2 LoginGUI

This GUI was created for the employees with the aim of making the rent and return operations easier and more effective. For these reasons we implemented a login GUI, and one authenticated a rent GUI where it is possible to rent or return vehicles.

The login GUI is a simple sign-in form where all the employees, stored in a json file with all their information, can access the rental service. The information about the employees are the first name, last name, age, email and password.

We chose the json file to store the employees because it was an efficient and easy way to test all the features of the project immediately and simulate the company's database, without the necessity to use other more complex kinds of storage system.

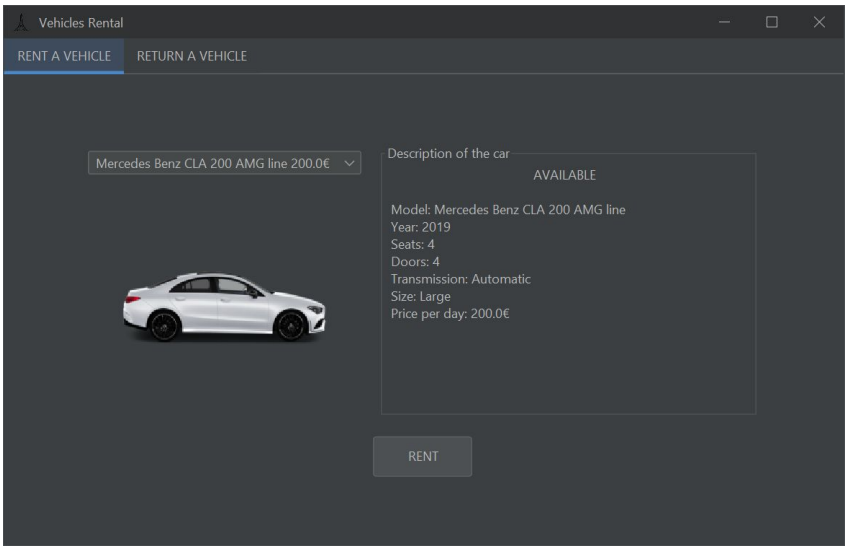
Furthermore, we thought to build a registration form, but we decided to don't follow this idea because for us the employees are already stored and if there is a new employee the company (EiffelCorps) will register him directly in their databases.

3.3 VehicleRentalGUI

Once authenticated, the employee can rent or return cars via user graphical interface. This interface has two tabs, one for the rent and one for the return. Each tab corresponds to one panel which implements all the graphical components.

3.3.1 RentPanel

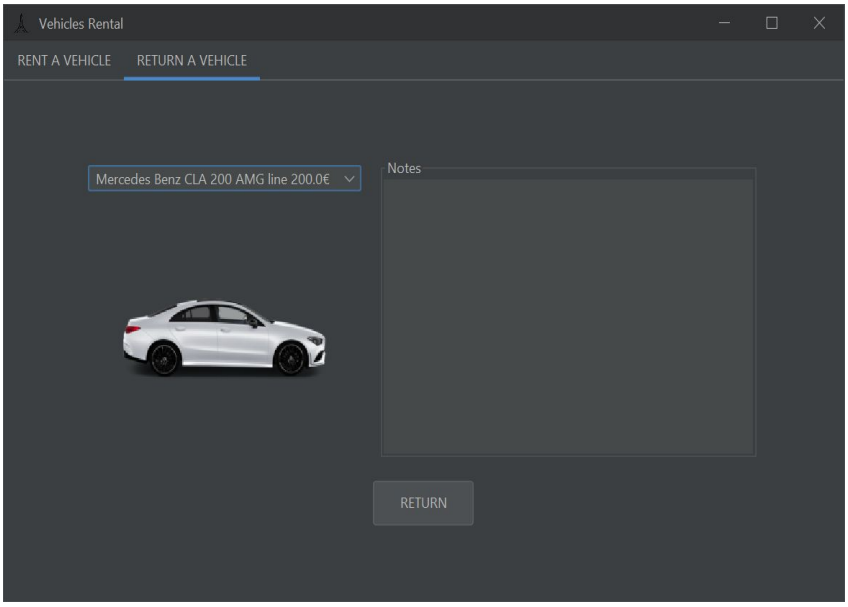
When an employee logs-in successfully, it is displayed the rent panel which contains all the vehicles which could be rented by an employee, the image and the description of the selected vehicle. In the description there are specified the vehicle's model, year of manufacture, number of seats and doors, the kind of transmission, the size and the price per day.



In addiction, it has also shown the availability of the vehicle: if the vehicle it is free or not; particularly in the description it is written "AVAILABLE" if none is currently renting the car or "AVAILABLE FROM ..." if the vehicle is currently rented, with also the information about the end date of the currently rent. If an employee wants to rent a car not available he/she will be added in the vehicle's wait list. In particular, the wait list is a map where for each vehicle there is a list of rents; when a vehicle is returned, then if there are other rents in the waitlist we update the vehicle information with the new rent, otherwise the vehicle is again available.

3.3.2 ReturnPanel

An employee can also switch to the return panel where the set of rented vehicles is displayed. The employee could also write notes before returning the vehicle. Then, the vehicle will be removed from the rented vehicles list and it will be again available for rent. At the end of the rent, a message box is shown with the total amount paid.



3.4 Encountered problems

The main difficulty designing the rental gui was to refresh the two panels when a rent or return operation was performed. In general the difficulty was to refresh every component of the gui depending on the events. The solution that we adopted is to recreate the panel, or the specific components, every time the state of the application changes in order to display the new information as soon as possible.

4. IfsCarsService

This Dynamic Web Project offers one web service that permits to sell vehicles. In particular, the service retrieves from the IfsCars project the list of the available cars (rented at least once) via RMI invocation and offers this list to the world. This service is also a web service client because it uses the Bank service to check the availability of funds and make the payment.

4.1 CarSeller

CarSeller is the main class used to sell vehicles, it has:

- A reference to a CarRental object, used to get all the information about available vehicles, it also contains.
- A reference to a Bank object, used to make the payment and convert the amount of money in different currencies.

The web service based on this class offers these 3 operations:

- ***getAvailableForSaleVehicles*** returns all vehicles available for sale, requesting available vehicles from the IfsCars RMI server and then verifying which ones are ready to be sold (vehicles at least rented once).
- ***sellVehicle*** actually allows the purchase of a car, taking available vehicles from ifsCars RMI server and removing the purchased ones.
- ***convert*** realizes, using the BankWebService, currency conversions.

It also contains the methods ***createJSONString*** and ***reconstructFromJSONString*** that convert the list of vehicles into a json file and vice versa. Then, to have a real correspondence between the rebuilt objects and the objects contained in IfsCars, we override the methods equals and hashCode (a vehicle is used as a key for the waiting list map) in Vehicle's class.

4.2 Encountered problems

We struggled with a lot of problems within this project:

- We had problems defining the policy file because some of us need to specify the policy, but some don't, and debugging to identify the problem wasn't easy.
- Find a solution due to exchange objects between client and server, due to the JAX-RPC limitations.
- Create the json file to offer vehicles and then parse the response and rebuild vehicles.

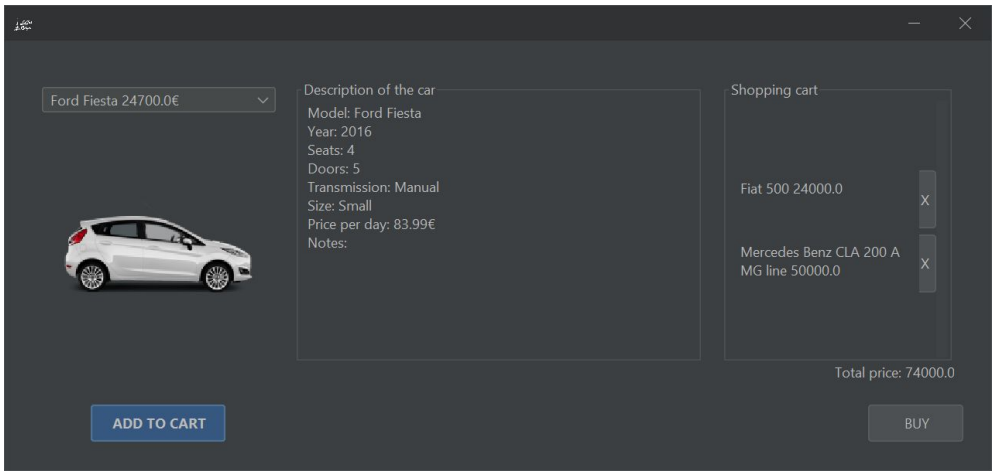
5. IfsCarsServiceClient

This project is a Web Service Client which allows the client to buy vehicles. This project also contains the graphic user interface (GUI) which allows the users to buy vehicles.

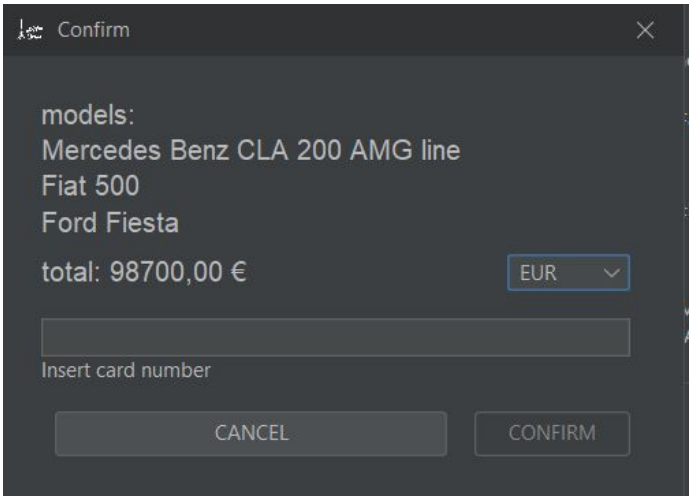
The class IfsCarsServiceClient allows us to use the CarSeller service methods (IfsCarsService) used to get and sell vehicles or convert an amount of money in a given currency. In addition, it manages the shopping basket.

5.1 ClientGUI

It was necessary to create a gui for users who want to buy vehicles. So the first thing the user can do is the choice of the vehicle he/she wants to buy, with its description, the price and the notes released by the EiffelCorp's employees who rented previously the selected vehicle. When the client clicks on the "ADD TO CART" button the vehicle chosen is placed in the shopping cart panel, where it is displayed the model and the price of the vehicles selected.



In addition, when the client clicks on the "BUY" button it is displayed on a panel where the user can see all the chosen vehicle's model and the total price. He could also change the currency between euro, american dollars, english pounds and yen.



Finally, there is a text field where the client can write his credit card number which must have 10 numbers and it has to be in the bank account list of the BankWebService.

5.2 Encountered problems

The main difficulty designing this gui was designing the basket. The first part of the interface is similar to the rent gui, so we re-used the same code. The shopping cart was something new and we used nested panels to make it, the difficulty was to manage them and make the cart versatile.

6. BankWebService

This project is a Web Dynamic Project that allows users to make transactions to buy vehicles using their accounts. This web service uses another web service, specified **here**, to make conversions between world currencies.

6.1 Bank

The bank web service offers 2 operations:

- ***makePayment***, which realizes the payment for a bank account, after checking that there is disponibility in the bank account.
- ***getExchangeFromEUR***, which returns the amount of money in another currency. To allow this feature, it uses the web service specified above.
  The exchange rates is found in real time

The bank contains a Collection of BankAccount. A BankAccount is composed of an id and an amount of money.

To store and recuperate data about bank accounts, we used a simple file json, *bank_accounts_list.json* and the *org.json.simple* library. Then, to load it, we created a private static method ***loadBankAccountsFromFile***, which is called in the constructor of the class.

6.2 Encountered problems

We didn't encounter many problems developing this web service, except reading the json file from a web dynamic application because all the code is executed on the tomcat server and we had to put the json file ther, instead of the main source directory.

7. Conclusion

This project was challenging for us, especially because we had to use a lot of knowledge learned in the class such as RMI (Remote Method Invocation) to share a service or objects without sharing its code and Web Services to support interoperable interaction across a network mixed with previous knowledge about java development, managing data structures. The best part was the search for the solution of the JAX-RPC limitations: we were looking for a tricky solution to avoid the limitations of the context, it wasn't enough to just use the services, but we had to understand them and, try, to improve them.

7.1 Future developments

The final projects have a lot of limitations, especially in the way we manage data due to the fact that we don't have persistent data storage such as databases. Of course it could be better, by considering implementing more features. Some of the possible future developments are:

- Add a login/registration form for clients who want to buy vehicles.
- Make data persistent with databases, and another web service that manages it.
- Transform this JAVA project into a web application.
- Use new technologies instead of JAX-RPC 1.1 such as JAX-WS.