Assignment 2.2

Car Tax and Appraisal System

## Student: IEREMIAȘ Viorel

## Group: 30442

# **Requirements analysis**

## **Assignment specification**

The application is used in in the computation of taxes and selling prices of cars.

## **Functional requirements**

Users introduce the information of their cars using a simple form (Web or Desktop):

o int year – fabrication year

o int engineSize – engine size

o double price- purchasing price

The application uses RPC to send the car information to the distributed object from the server that computes the following information depending on the client request:

Tax for a car

𝑡𝑎𝑥=(𝑒𝑛𝑔𝑖𝑛𝑒𝑆𝑖𝑧𝑒200)∗𝑠𝑢𝑚, where sum depends on the engine size

Selling price of a car

𝑝𝑟𝑖𝑐𝑒𝑠𝑒𝑙𝑙𝑖𝑛𝑔={ 𝑝𝑟𝑖𝑐𝑒𝑝𝑢𝑟𝑐ℎ𝑎𝑠𝑖𝑛𝑔 − 𝑝𝑟𝑖𝑐𝑒𝑝𝑢𝑟𝑐ℎ𝑎𝑠𝑖𝑛𝑔 / 7 ∗ (2018 – 𝑦𝑒𝑎𝑟) 𝑖𝑓 2018−𝑦𝑒𝑎𝑟< 7, otherwise 0

The result of the invoked operation, tax, respectively selling price, is displayed on the client GUI.

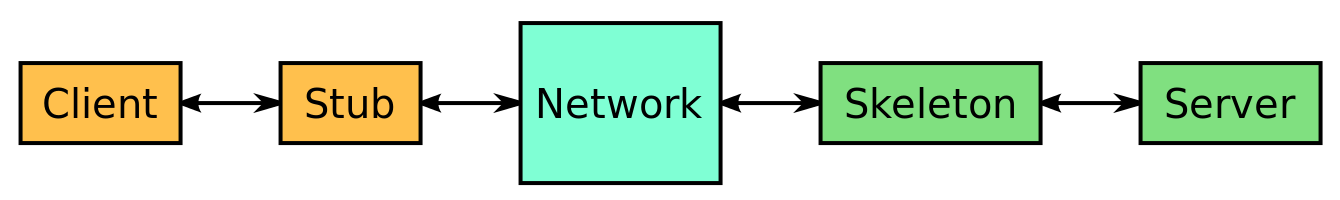
# **Design and architecture**

## **Conceptual architecture**

*Conceptual architecture diagram*

The solution is a three-tiered distributed system, consisting of the following components: Server, Client and Web application. All the tiers are run are separate processes and communicate with the other processes in different ways.

The client exposes a REST API to the web application, with which it communicates via HTTP request. But the client application doesn’t process the request itself. The client makes remote procedure calls (RPC) to the server, which contains the actual processing entities.

RPC uses the client-server model. The requesting program is a client and the service providing program is the server. Like a regular or local procedure call, an RPC is a synchronous operation requiring the requesting program to be suspended until the results of the remote procedure are returned. However, the use of lightweight processes or threads that share the same address space allows multiple RPCs to be performed concurrently. When program statements that use RPC framework are compiled into an executable program, a stub is included in the compiled code that acts as the representative of the remote procedure code. When the program is run and the procedure call is issued, the stub receives the request and forwards it to a client runtime program in the local computer. The client runtime program has the knowledge of how to address the remote computer and server application and sends the message across the network that requests the remote procedure. Similarly, the server includes a runtime program and stub that interface with the remote procedure itself. Response-request protocols are returned the same way.

*RPC architecture diagram*

In particular, the project uses Java Remote Method Invocation (Java RMI), a Java API that performs remote method invocation, the object-oriented equivalent of remote procedure calls (RPC), with support for direct transfer of serialized Java classes and distributed garbage-collection.

The original implementation depends on Java Virtual Machine (JVM) class-representation mechanisms and it thus only supports making calls from one JVM to another. The protocol underlying this Java-only implementation is known as Java Remote Method Protocol (JRMP). In order to support code running in a non-JVM context, programmers later developed a CORBA version.

Usage of the term RMI may denote solely the programming interface or may signify both the API and JRMP, IIOP, or another implementation, whereas the term RMI-IIOP (read: RMI over IIOP) specifically denotes the RMI interface delegating most of the functionality to the supporting CORBA implementation.

In terms of high-level architecture, the application is structured in layers, based on the principles of the layered architecture. The principle of the layered architecture is to separate the components of the system that perform similar functions into isolated groups which share information inside the layer they form, but only expose through an interface the communication with other layers. Such a system behaves like a linear pipeline of modules where each layer uses the functions of the layer immediately beneath itself and data passes through the layers being processed at each step. The difference is that data flows in both directions, either from the data source towards the user, or from the user, who has access to input mechanisms, towards the data source.

The advantage is that the layers are decoupled, while inside the layers, cohesion is high, making the system more stable and easier to extend, maintain and test. The disadvantage is that there may be layers in which some data is not processed too much or is not processed at all, which affects performance for no gain. Also, such a system is more complex and more difficult to design.

Furthermore, this architecture is well suited for web-based applications, since it allows a clean separation of responsibilities between the server and the client. In this case, processing and rendering is accomplished server-side, while the browser clients only display that view and pass the user inputs.

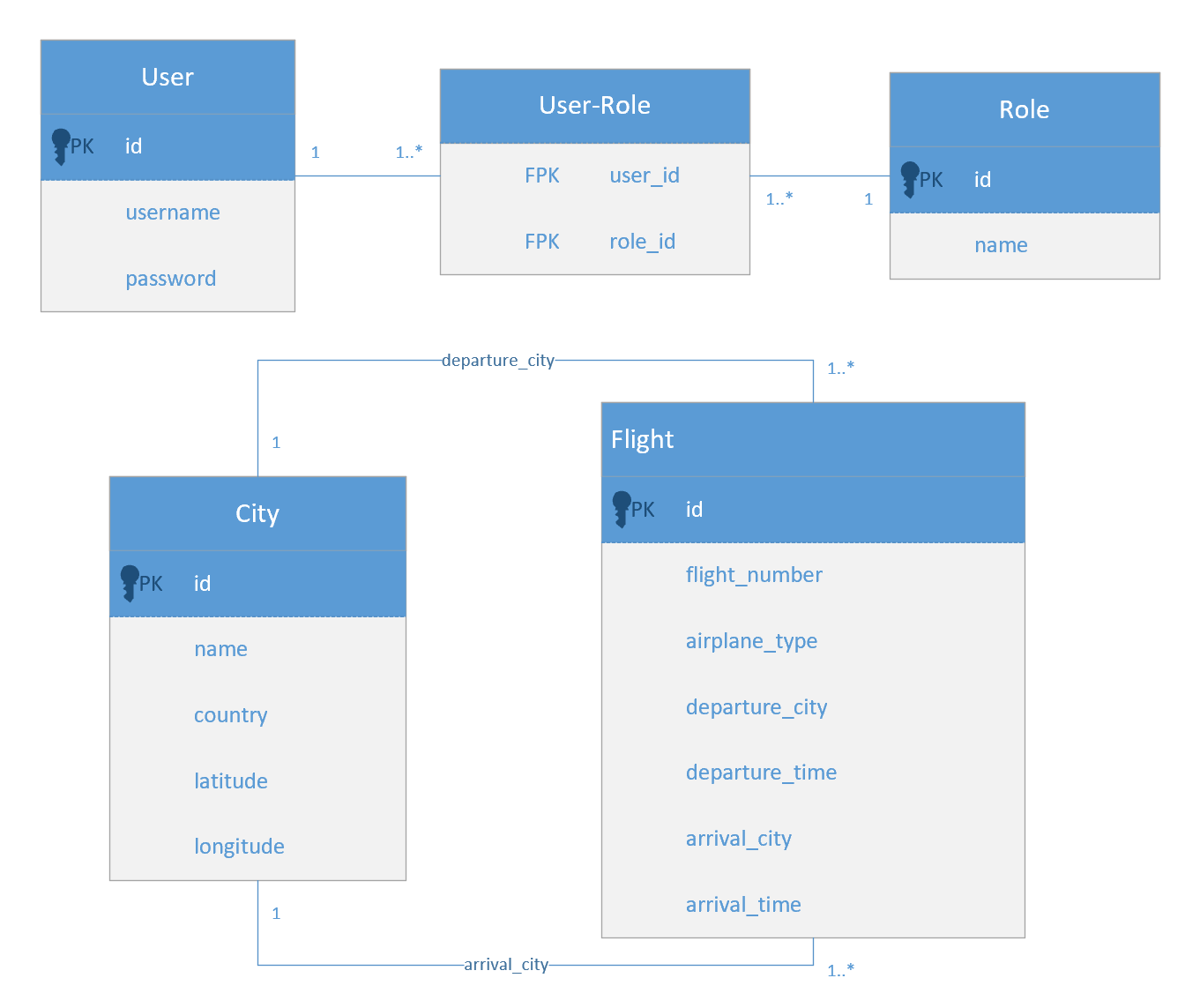
Deployment diagram

## **The server**

The design of the database is highly dependent on the requirements of the application, because they influence both the entities and the relationships between them.

The following entities are included in the database:

* User: stores basic authentication information for the users of the application
* Role: used to define the access rights for a user
* User-Role: maps users to roles in a Many-To-Many relationship, which offers the flexibility to assign multiple roles to a user, in case of implemented a more fine-grained role-based access control policy
* City: stores geographical information (country, name, latitude, longitude) about a city which hosts an airport
* Flight: main business entity, storing the following information – flight number, airplane type, departure location, departure date and time, arrival location and arrival date and time

E-R diagram

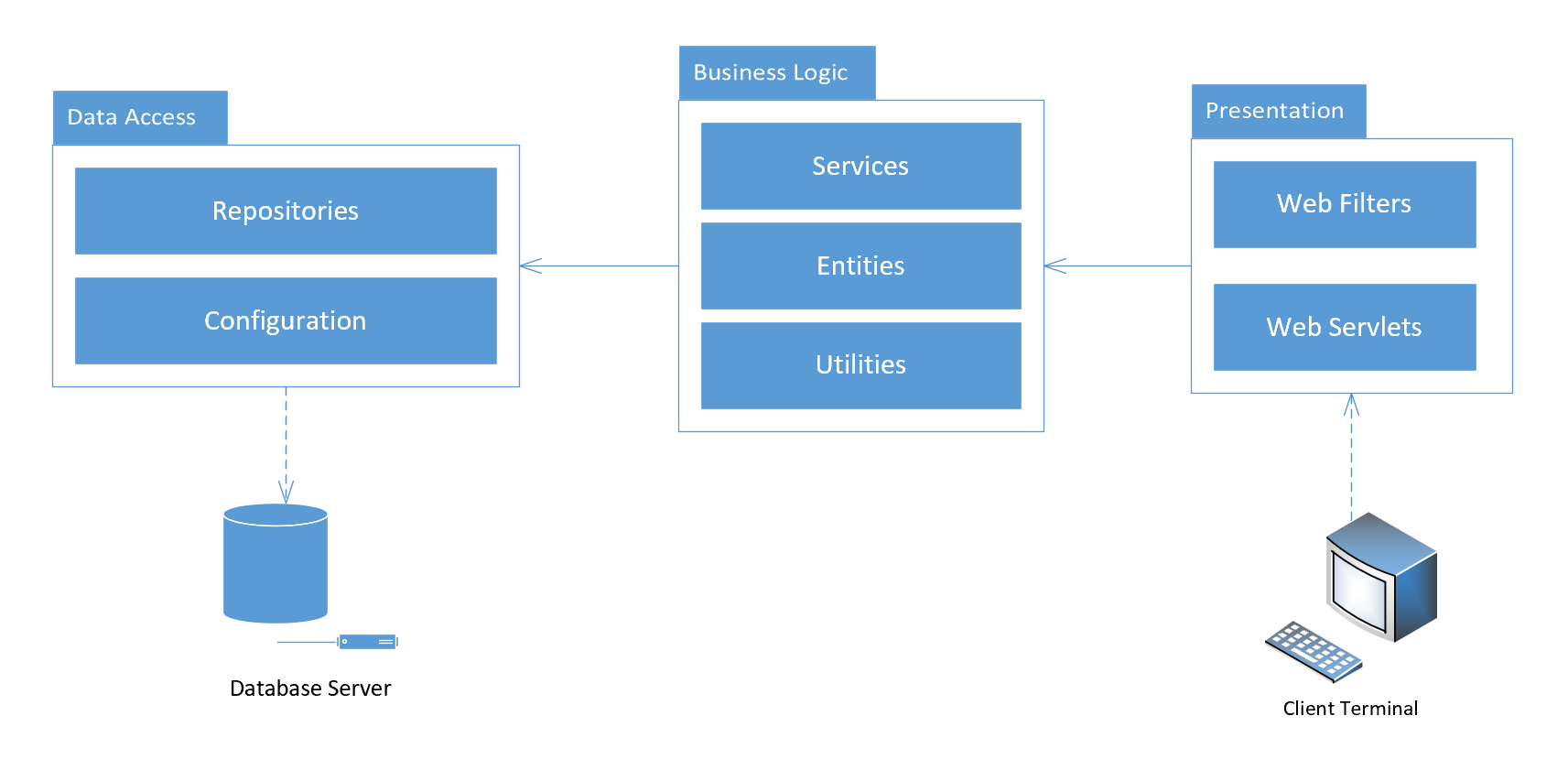
## **Package architecture**

As previously mentioned, the application is layered. Each layer abstracts out one type of process and communicates with the layer directly below it in terms of some contracts established by a set of interfaces. The main layers are Data Access, Business Logic and Presentation.

The Presentation layer manages the communication with the client, from whom it receives requests and to which it sends back responses. The web servlets map the endpoints of the application and parse the requests, before delegating the work to the services, which process the request. The web filters have different functions, ranging from logging to application access rights management for the two types of users.

The Business layer is the most complex part of the system. It processes the requests that are parsed and forwarded to it by the Servlets. The Entities package contains the classes for the business objects, User, Role, Flight and City. The Services package contains the classes that process the data, generate the result and send the data that needs to be persisted to the data access layer. The Utilities package contains additional classes, that are used by the Services, like data transfer objects or filter objects.

The Data Access layer connects the business to the

Package diagram