Projet: Mashup

19th September 2025

1 Introduction

This project will consists in developing an application that combines different web services technologies (mainly `REST` and `RPC`). More precisely, you are going to develop a "`Mashup`": a web application by following an approach based on the *design by layers*. The term "mashup" refers to an application that relies on a set of remote services (not necessarily web services) to provide its functionality.

2 First Part: VirtualCRMService and its companions

In this section we present the first part of the project where several webservices will be developed and others will be just queried in order to develop a *mashup* application.

2.1 Context of the Project

To define the context of the project, we will assume that we are working in a company that, for historical reasons, it stores its customer information in their own Customer Relationship Management (CRM) systems[1]. This `CRM` is a web application installed on one of the company's machines, and uses a database (perhaps installed on another machine) to store customer information.[2]

The model provides with the (internal) data type `ModelTO` shown in Figure 1. The service layer is implemented via Apache Thrift [1] and its interface is shown in Figure 1.

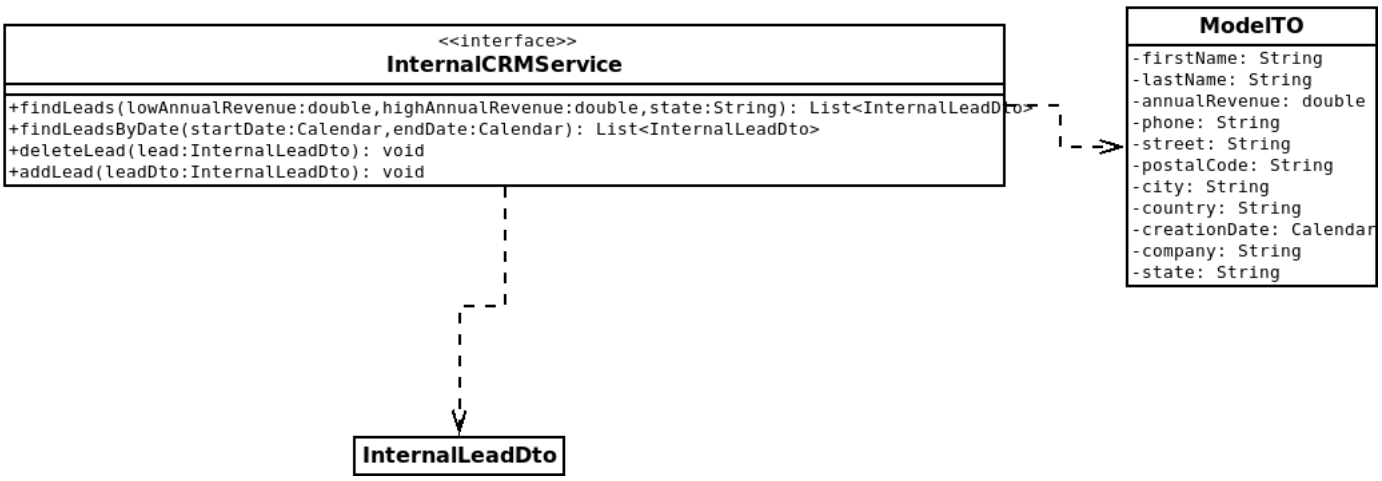

Figure 1: InternalCRM interface

[1] See `https://en.wikipedia.org/wiki/Customer_relationship_management`.

[2] **In this project we will not consider the modelisation of the database. Instead, we will simulate the database by means of a list.**

One month ago, our company acquired another company in a similar field of activity. This company had subscribed to a CRM service from Salesforce, so nowadays, our company has to deal with two different CRM services.

Companies like `Salesforce`[3] offer CRM services via internet. The CRM is managed by an external company (Salesforce in this case), and a REST API is published in order to provide operations such as adding/removing leads or retrieve leads.

For the moment, the company notes that it has customer data in both `CRM`. For certain functionalities (probably numerous), it would be desirable to have an application offering a unified view of customer data, thus avoiding the need for employees to explicitly use both `CRM`, which would be very tedious. In particular, our company decided to build an intermediate webservices to retrieve customer information (*Lead*). The interface and the main dependencies of this webservice is shown in Figure 2. This webservice, named Virtual CRM service provides two operations to find Leads (by date and/or by revenue) and it delegates such operations on both the internalCRM and Salesforce. To each Lead that is retrieved, the virtualCRMService will add the spatial coordinates (longitude and latitude) related to the address of the client (see Figure 2). To retrieve such information, the virtualCRMService will query `OpenStreetMap`[4] in order to obtain such data from the address of the Lead.

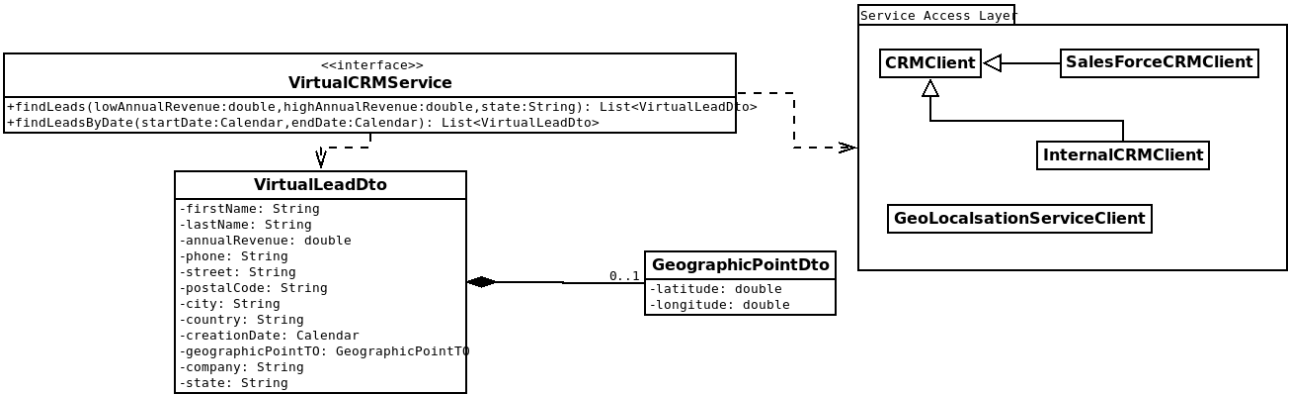

Figure 2: General Architecture of the VirtualCRMService

Figure 3 shows the architecture of the mashup application to be implemented. The code is structured in a module: *virtualcrm*.

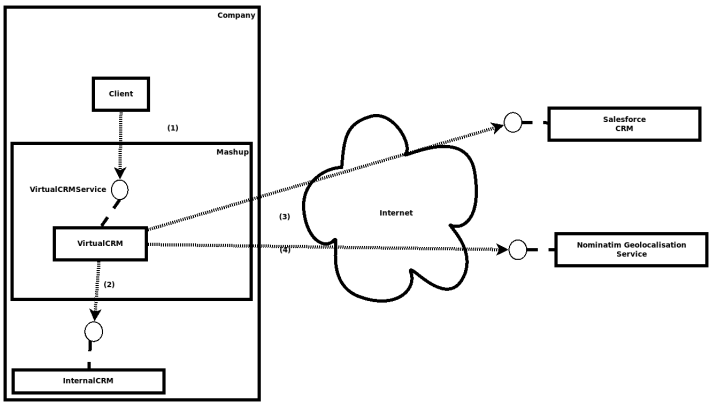

Figure 3: Architecture the `mashup` application

.

[3]See `https://www.salesforce.com/fr/`

[4]`https://www.openstreetmap.org/`

When the user runs the client, we invoke the `findLeads`(1) operation of the `VirtualCRMService` interface (Figure 2), provided by the "virtualcrm" module. The `findLeads` operation is given three parameters. The first two represent the annual revenue range that customers must have (`lowAnnualRevenue` < `lowHighRevenue`), and the third the name of the province in which they are located. The operation returns information on customers matching these search criteria.

For each customer (`VirtualLeadDto`), we return their surname, first name, the company they work for, their expected annual income, their telephone number, street, postcode, town, department and country, the date they were registered in the CRM and their geographical position (`GeographicPointDto`). If the geolocation service is unable to return a customer's position, the attribute `geographicPointTO` must be `null`.

The client Should maintain a single instance (Singleton) of the class implementing the `VirtualCRMService` interface. This behaviour can be achieved by means of the use of `Factories`. In this case, the implementation of this interface must initialise the state in the constructor and must never modify it during the execution of the interface operations.

The implementation of the `VirtualCRMService` interface must invoke the search services of the internal `CRM` (2) and `SalesForce`(3) to obtain the data of potential customers, and the geolocation service (4) to obtain their geographical position.

Note that, the access to the Salesforce webservice requires an authentication procedure. Such authentication does not need to be performed at any request on Salesforce. Instead the service access layer should do the authentication when the serviceClient is created and when the authentication token is expired it should be regenerated automatically.

The interface must return the information of all customers (a list of objects of the type `VirtualLeadDto`, see Figure 2), sorted by potential profit (from highest to lowest). Independently of the origin of the Lead (salesforce or internalCRM), the lead will be converted to a type `VirtualLeadDto`.

The client accessing to `virtualCRMService` will consist on a simple commandline application that will print the information o each retrieved `Lead` on the command-line. A similar approach should be implemented for the operation of `findLeadsByDate`.

The next section explains the different parts of this exercise that should be implemented.

2.2 Service InternalCRM

A `RPC` service will be implemented with Apache Thrift. This service provides the operations presented in Figure 1. Internally it will delegate the operations to the model and it will export the type `InternalLeadDto` in order to `hide` the use of the types of the model from external applications. Moreover it will provide operations for creating and deleting Leads from the service. Those operations will be used by a third application presented in Section 3.

One special feature worth mentioning is that the `internalCRM` service will return (type `InternalLeadDto`) the first and last name of each contact in a single field using the format Last Name, First Name (for example Laporte, Aymeric). Contrary, that the type `VirtualLeadDto` of "`virtualCRMservice`" has two different fields for the first and last name respectively.

2.3 Salesforce Webservice

Salesforce (`https://www.salesforce.com/`) offers its customers a CRM service and, since not a long time, our company has access to it. Our `VirtualCRMService` can easily access and retrieve its information contained in its `CRM` via a REST API [5].

In order to facilitate the development and testing of applications which access their data, `Salesforce` allows developers to create **Developer Account** identical to real accounts but containing fictitious data. In practice, we are going to access the data of a fictitious company created in this way. To do this, each group will create its own developer account with `Salesforce`.

The `Salesforce` data model consists of a number of data entities, each of which contains various types of information. In our case, we will use the "`Lead`" entity which contains various data about potential customers. This data includes the various elements such as address, street, state, postcode or country, as well as the annual revenue of the company to which the contact belongs (`AnnualRevenue`).

To query the data of an entity, it is necessary to use the `query` method of the `API` of the Web service. This method receives among its parameters a query expression written in a language called `SOQL` [6], which is a very simplified version of `SQL`. Before invoking the query, it is necessary to authenticate to the service using the logical method of the `API`.

In `https://developer.salesforce.com/` you can find various examples and resources of interest.

2.4 Geolocalisation Service

Several geolocation web services are available on the Internet [2, 3]. Some of them, such as `Google Maps`, require a `APIKey` and also follow a *pay-as-you-go* pricing model.

The Nominatim [3] API of `OpenStreetMaps` does not require any type of registration and provides a `REST` service which takes an address as a parameter and returns `latitude` and `longitude` in `JSON` or `XML` formats among others. For example, if we want to obtain the location of the `UFR` (Figure refufr) of sciences at the University of Angers using the `API` *Nominating search* [4], this can be done by calling the following `URL` (see [4] for more details)

`https://nominatim.openstreetmap.org/search?<params>`

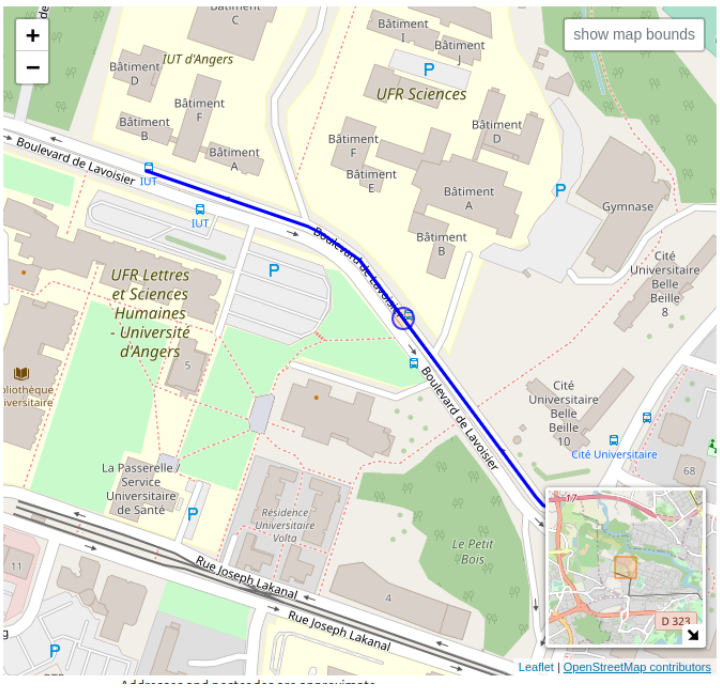

Figure 4: Visualisation OpenStreetMap de l'UFR des sciences.

`https://nominatim.openstreetmap.org/search?city=angers&country=france&postalcode=49100&street=2+boulevard+de+lavoisier&format=json&limit=1`

and analysing the response. This can be received in a `JSON` format. In the case of the example, we obtain the following (simplified) `JSON` (see Figure 5):

La réponse doit être analysée afin d'en extraire la **latitude** et la **longitude**.

```
0:
    place_id:       251271841
  licence:          "Data © OpenStreetMap contributors, ODbL 1.0. http://osm.org/copyright"
    osm_type:       "way"
    osm_id:         119321018
    lat:            "47.4795093"
    lon:            "-0.6003698"
    class:          "highway"
    type:           "residential"
    place_rank:     26
    importance:     0.10000999999999993
    addresstype:    "road"
    name:           "Boulevard de Lavoisier"
  display_name:     "Boulevard de Lavoisier, Belle-Beille, Angers, Maine-et-Loire, Pays de la Loire, Metropolitan France, 49045, France"
  boundingbox:
      0:            "47.4782032"
      1:            "47.4805084"
      2:            "-0.6029448"
      3:            "-0.5986745"
```


Figure 5: JSON code obtained.

2.5 VirtualCRMService

This service retrieves the information from the internalCRMService, Salesforce and the geolocalisation service in order to provide a common interface for retrieving customers. This module is also a web-service in its turn since provices an API implemented with `Spring` containing two operators shown in Figure 2. The architecture explained in that figure is very very general, a better design must be devised by means of the use of design patters (factories, injection of dependencies, facades, templates , etc).

2.6 The Client

The client consists of a simple command-line tool that by using the `VirtualCRM REST` api it retrieves the leads from both `CRM`s and prints their information in text on the command-line.

3 Second Part: Tool for Merging the InternalCRM and Salesforce

In the long term, wants to use an unique CRM: the internalCRM. For that reason we will develop a command-line tool that will retrieve all Leads from salesforce and will add them to the `internalCRM` via the `thrift` API. A very general architecture of this tool is shown in Figure 6.

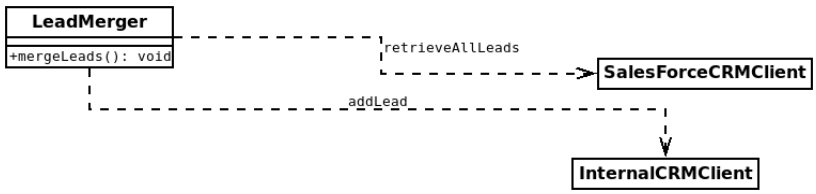

Figure 6: Architecture of the tool for adding all Leads of Salesforce to the internalCRM.

4 Evaluation

4.1 Groups

The project will be carried out in groups of 3 people (preferably).

4.2 Coding Norms

In order to write high-quality, easy-to-read code, a common coding system is used, which defines rules for naming classes, attributes and methods, identification rules, etc. This allows a development team to maintain the appearance of the code, regardless of who wrote it, which makes maintenance easier.

This enables a development team to maintain the appearance of the code, regardless of who wrote it, which makes maintenance easier. The [7] coding standard will be used for the project. The examples seen in the course follow this naming system.

It will be mandatory to use a `git` version control system. github or gitlab. The link to the project will be sent to the teacher by email.

Finally, it will not be necessary to document the classes (for example, JavaDoc).

4.3 Submission of the Final Version

A copy of the sources of the project (containing all the applications requested) in `.tar.gz` or `.zip` and a report in `.pdf` format will be uploaded to `moodle`.

Concerning the project sources, the compressed file must contain a project which can be compiled via the command-line `gradle`. This project must contain the sources and configuration files, but it is not necessary to add other file types such as `.class` or `.war`.

4.4 Format of the Report

As a general rule, diagrams should use `UML` notation. Diagrams should be of high quality (not reverse engineered - design precedes implementation!) and should be briefly but clearly explained. The quality of the report will be essential for the evaluation of the project.

The report will contain the following parts:

1. **Design**

   (a) **Global Architecture**: briefly explain the modules that have been designed and implemented. Two "high-level" UML diagrams will be used to support the explanation.

   - A class diagram illustrating the abstractions (interfaces, *factories*, interface implementation classes, etc.) of the various modules involved in implementing the `VirtualCRMService` interface. This diagram should not show the names of the operations and attributes in the interfaces and classes. Its purpose is to illustrate the dependencies between the main abstractions.
   - A sequence diagram illustrating the processing of an invocation of the `findLeads` operation of the `VirtualCRMService` interface, in terms of the abstractions presented in the class diagram above.
   - A sequence diagram illustrating the invocation of the method `mergeLeads` from the class `LeadMerger`.

   (b) **i-th service/module**: For each service/module, it is necessary to write a section explaining its design. Each explanation should include :

- The overall structure of the module package. It is not necessary to use UML notation. Nor is it necessary to show all the packages, just the top-level packages that reflect the architecture of the module and its main sub-packages. For each package, a brief explanation of what it contains should be given.
- One or more diagrams illustrating the architecture of the module. In particular, the diagrams specify in detail the operations of the interfaces and classes for `transfer objects` (*transfer object*). For the rest of the abstractions (concrete classes), it will only be necessary to specify the operations and attributes that are considered relevant to understanding the explanation of the module (for example, an attribute that is important for understanding the explanation of an interface). In other words, UML diagrams should not bring together all the abstractions implemented, but only those that are relevant and, for each of them, show only what is necessary to understand the design of the module.

2. **Compiling and installing the application**: It should be clearly explained how to compile and install the practice, assuming a correctly configured environment (e.g. application server installed and configured, software packages installed in the same directories as in the laboratory, etc.). The installation instructions should be as simple as possible. In particular, they should specify :
   - How to unpack the source distribution
   - Any particular aspects of the configuration that need to be highlighted
   - The project must be able to be run using `gradle`.
   - **When correcting the final version of each application, the installation instructions will be followed**
3. **Known problems**: list of known bugs/errors in the code.

4.5 Iterations and Deliveries

For each application, an iterative approach will be followed. Each iteration incorporates more functionality than the previous one, until the last iteration ends with software that implements all the functionality. In particular, each implementation will be carried out in two iterations.

- **The correction of the first iteration will not count in the final mark and it will not be necessary to submit a report**.
- It will be sufficient to show the diagrams linked to this iteration using a UML design tool.
- The aim of this first iteration is to try to detect major errors and, if necessary, to guide the learner towards their resolution.
- The second iteration will correspond to the (final) correction.

4.6 Evaluation

The evaluation of the project will take into account :

- How well it works
- The quality of the design
- The quality of the memory

A copied project means failure for the group that authorised the copy and for the group that copied, no distinction will be made.

References

[1] Apache thrift, https://nominatim.org/release-docs/develop/api/Search/

[2] Google geocoding api, https://developers.google.com/maps/documentation/geocoding/overview

[3] Nominatim: Open street map geocoding, https://nominatim.org/release-docs/develop/api/Overview/

[4] Nominatim: Open street map geocoding, https://nominatim.org/release-docs/develop/api/Search/

[5] Salesforce rest api developer guide, https://developer.salesforce.com/docs/atlas.en-us.api_rest.meta/api_rest/intro_rest.htm

[6] Soql, https://developer.salesforce.com/docs/atlas.en-us.soql_sosl.meta/soql_sosl/sforce_api_calls_soql.htm

[7] Java code conventions (1997), https://www.oracle.com/technetwork/java/codeconventions-150003.pdf