CA 2 - Object Relational Mapping, REST, Test, Ajax and JQuery

This CA builds a proof of concept solution, for what could have been a (agreed very simplified) real life system. For this reason we assume there will be a notable difference in how far the different teams will come.

This CA will strengthen your skills and knowledge related to:

- Use an Object Relational Mapper (JPA) to map between Java object and Relational Tables
- Use an Object Relational Mapper (JPA) to map Java Inheritance to Relational Tables
- Expose business functionality via a REST-API and JSON.
- Handle exceptions via a REST-API and JSON
- Test a REST-API, using both unit-tests on the data model, and tools like REST Assured to test the API.
- Build dynamic web-pages that builds content via JQuery (using AJAX calls to a server REST-API).

Hand-in and documentation

- The code must be made available via GIThub.
- All documentation, the REST-API and the projects web-pages must be made available via a small web-site published on your WEB-server.
- Documentation must include (each with a separate heading or a separate page):
  - A complete description of the REST API, error responses and JSON format.
  - A section explaining your test strategy, including test results
    - For plain Unit Test
    - Testing the Database façade(s)
    - Testing the REST API
  - A section stating who did what. This should also include your own suggestion for the amount of studypoints each member should be awarded for this part (0-5).
  - A description of the strategy chosen to implement inheritance and why this strategy was chosen.
- A clear and precise description of how to test the system. We will not do a class demo for this CA, so the documentation must include a description of how:
  - we can test the API via your implemented test cases using REST Assured, and via Postman
  - and how to use the web pages that uses the API.

- You must all include the following elements in your CA-2 front-page, to identify your group.
  - Feel free to style it anyway you like, but the ids must be exactly as below:

```
<div id="authors"> Peter Hansen, Ole Jensen, Ida Hansen </div>
<div id="class">A or B or COS</div>
<div id= "group"> Group number </div>
```

  This is meant for an exercise in Module-3, so please ensure that you're CA-2 web-site stays live, at least until the exam.

How to spend your four days

You are supposed to develop the system over 2 consecutive two-day mini sprints as sketched below:

- Day-1+2: Before the end of day two you must demonstrate your REST-API (backend)
- Day 3+4: Complete the Front-end, missing back-end parts, the documentation and deploy the system

How to complete this CA in only four days.

This CA is time intensive and if you are pressed for time, it is more important that you complete a little bit of all tasks, than completing all the steps of only a few tasks.

This CA is meant to be impossible (for an average student) to complete in only four days. So you must take advantage of the fact that you are two or three persons in a team and distribute (sprint) tasks between team members.

Studypoints for this (4-week) period + CA are given below:

For your participation in the class (one for each day)	16 points
For the three Study Points Exercises	15 points
Each member in a team can earn additional 19 points for the CA as sketched below:	
For your contribution to the code and documentation (verified via Git, attendance in the class + your own suggestion in the documentation)	Up to 5 points
For the demonstration + test of the REST-API (the backend) Day-2	Up to 3 points
The quality of your design and documentation	Up to 3 points
Quality/coverage and description of your tests (backend test and API-test)	Up to 4points
For the testing/demo (monday) of the final system	Up to 4 points

Bonus Points:

While you develop this system you will acquire knowledge about many small details, which initially might take you a lot of time to figure out.
You should try to help each other, so if you have a problem, ask if any of your classmates can help (and respect if the reply is – wait we are doing something else right now).
If you feel you have received helpful hints from somebody during this sprint, you can suggest (via your documentation) that they are rewarded an **EXTRA study point.**
A team/person can receive up to five extra study points in this way, if suggested by more than one team (max one point per team)

Hand-in: Saturday (09.10.2016) before 18.00.

Before you start:

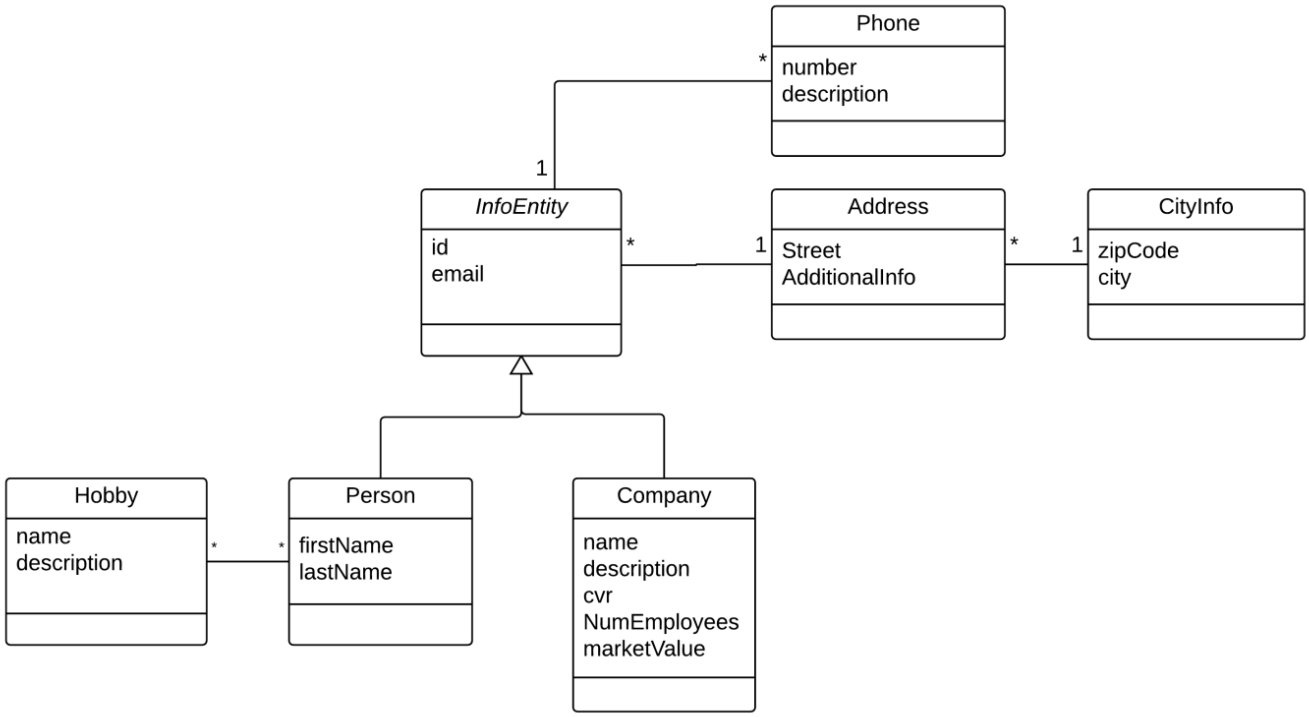
Before you start it is recommended that you read the section " Bacground, REST-API's, DTO's and JSON" at the end of this document.

The Business **Domain**

A new company plans to join the market for *information about people and companies* (like Krak, De Gule Sider etc.). They plan to offer services like:

- Get information about a person (address, hobbies etc) given a phone number
- Get information about a company given a phone number or cvr.
- Get all persons with a given hobby
- Get all persons living in a given city (i.e. 2800 Lyngby)
- Get the count of people with a given hobby
- Get a list of all zip codes in denmark
- Get a list of companies with more than xx employes
- Etc.
- In order to set up data, the API must also provide methods to add, delete and edit the Entities

Initiallly the following simplified version of the busines domain has been approved as the background for a quick prototype application.



What to do in this CA, with a suggested time schedule.

Part-1 (day 1) Design your Entity Classes, tables and Façade(s)

a)

Design Your Entity classes and tables (you decide whether to design tables, and create Entity classes from the tables, or the other way around, but we suggest the last strategy)

b)

Implement one (or more) façade classes to simplify the operations on the entity classes

The façade must have methods matching the initial requirements given on the previous page, that is:

```
Person getPerson(int id)[1];
List<Person> getPersons();
List<Person> getPersons(int zipCode);
Company getCompany(cvr);
…
```

Note: You don't need to complete your façade in day-1. Complete as much as you can on day 1 (including testing) and continue with the API on day 2. You can add the missing parts later if you have time.

c)

Implement **JUnit** tests to verify the behaviour of your façade. Make sure to have a specific test database for this part (see the exercises for hints about, how to have two persistence-units, and an in-memory test database)

d) Test Data

We want the final demonstration of this system to be as realistic as possible, so we need (lots of) Test Data

Hints:

Use the script **populateZip.sql** to set up values for all Danish zip-codes.

Use the study point exercise *exam-preparation_REST_JSON.pdf* as an inspiration of; how to create large amounts of test data (don't create JSON, but an SQL-script)

For a Group with only two members you may use a modified version of a script provided by another team

[1] This should provide a Person with all details. When you build your JSON for a specific REST CALL you decide for how much to include according to you API description

Part-2 (day 2) Design the REST API

Design the REST API

Design the actual REST API for the application (using the guidelines from the slides) which will make it possible to operate on the Database via the façade.

The API should include all four CRUD operations + most of the business services described in the section "The Business Domain".

GET	
api/person/complete	Return all Persons, with all details, as a JSON object (As in *JSON example-1)*
api/person/complete/id	Return the Person (with the given id) with all details as a JSON object (as in *JSON example-1)*
api/person/contactinfo	Return all Persons, with only contact info, as a JSON object (As in *JSON example-2)*
api/person/contactinfo/id	Return the Person (with the given id) with only contact info, as a JSON object (As in *JSON example-2)*
...	
POST	
api/person	Create a new Person given a JSON object (as in *JSON example-1)*
Complete the API description for all services[2]	

Error Handling

- Complete the API description with a supplementary description of the responses for erroneous scenarios.
- The response must include a proper HTTP status code and a JSON response with a description of the error and (only, when in debug mode. Hint: use web.xml to declare this attribute) the Exception Stack Trace (if any).
- You must include this description (API description + Error Responses) in your documentation.

Implement the REST API

Implement the REST API using JAX-RX and GSON

Test the API

Test the API using REST Assured

[2] As a check for whether your API description is accurate enough, consider yourself in a situation where you were the API user having to write the client code, given only the API description. Or consider that you were going to outsource this part, and do the front-end part in parallel with this.

Part-3 (day 3) Design WEB pages that uses the API

Implement a number of web pages that uses the API, using HTML, CSS, -Bootstrap, JavaScript, JQuery and AJAX.

Feel free to do this in any way you like, but a single page that allows for reading, Creating Editing and Deleting a Person would be nice.

Also consider page/pages like:

- Get all persons with a given hobby
- Get all persons living in a given city (i.e. 2800 Lyngby)
- Get the count of people with a given hobby
- Get a list of all zip codes in denmark
- Get a list of companies with more than xx employes

Part-4 (day 4) Complete the missing parts, the documentation and upload to Digital Ocean.

Complete the missing parts

Complete the missing parts. Write the documentation and deploy your project (including documentation) to Digital Ocean

Use the guidelines given in *Tomcat On Your Server V2.pdf* to prepare you project for deployment.

Bacground, REST-API's, DTO's and JSON

This CA involves the following skills:

- Design/implement a RESTful API
- Use JSON as a mean to transfer data between disparate systems
- Objekt Relation Mapping (JPA) to simplify DataBase access
- JQuery, JavaScript and Ajax to build dynamic pages on the client (browser)
- Deploy code to a cloud provider

The following pages will provide some background and hints which should help you in fullfilling this CA

Data Transfer Objects (DTO's)

An important thing to understand is that the diagram, given on the previous side, represents the internal business logic on the server. What we chose to expose to the "outside world" via our REST-API can and will be quite different.

One reason for this can be network performance. For example we could decide to combine a number of calls into one (returning a result representing the individual calls) since we know that the TCP/IP-stack adds an overhead to each call. For small amounts of data this overhead could easily exceed the size of the actual data. As an example, let's see how we send/receive information about a Person. Internally a person is represented by 6 different entities (InfoEntity, Person, Address etc.). When transferring data between our services and clients however, this could easily be combined into a single object as sketched below (newlines added for readability):

```
 {
  "firstName": "xxx",
  "lastName" : "xxx",
  "email" : "xxx",
  "phones": [{"number":"xx","description":"xx"},{"number":"xx","description":"xx"},..]
  "street": "xxxx",
  "additionalInfo" : "xxxx",
  "zipcode": "nnnn",
  "city" : "xxxx"
}
```

JSON example-1(hobbies left out)

We could also go the other way around, and decide to return less data in a call because the client might not need all information available. As an example, let's take a look at the diagram and imagine we needed contact information only for a client. Here we could return a JSON object as sketched below:

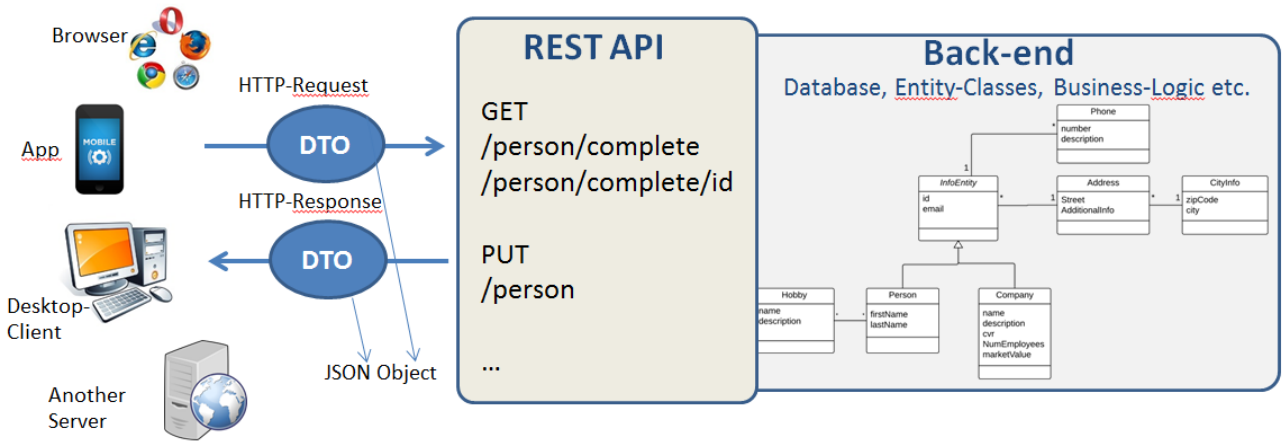
```
{
 "id":1,
 "name": "Lars Mortensen",
 "email":"lam@lam.dk",
 "phones": [{"number":"xx","description":"xx"},{"number":"xx","description":"xx"},..]
}
```

JSON example-2

Objects, like the ones above, are sometimes referred to as a Data Transfer Objects (DTO)[3]

Another reason for defining DTO's[4] could be that we don't won't to expose the internal design so, as an example, we could rename all variables so for example firstName could be fn and lastName ln etc.

The figure below is an attempt to visualize DTO's and the difference between the actual business logic and what is exposed in JSON-objects via REST.



From Java to JSON and from JSON to Java

Since we will implement the REST API using Java, it is vital that we can convert our Java objects into JSON and the other way around. When there is a one-to-one mapping between Java and JSON we have seen that this is simple, using for example the Gson library.

If we have a JSON String like:

```
{"id":1,"name": "Lars Mortensen","email":lam@lam.dk, "phone" : "12345678"}
```

We can transform it into a matching Java object (assuming we have a Java Person class with the three attributes, and the JSON above as a string) as easy as:

```
Person p = New Gson().fromJson(jsonString, Person.class);
```

And we can transform a Person object into a string like:

```
String jsonString = new Gson().toJson(p);
```

If you take a look at the document https://sites.google.com/site/gson/gson-user-guide you will see that we can easily handle arrays and many more complex situations.

[3] http://en.wikipedia.org/wiki/Data_transfer_object
[4] In this CA we will refer to all objects exposed via the REST-API as DTO's

From Java to JSON and from JSON to Java with POJO's

The problem with the strategy above is that is requires us to have a matching Java Class (POJO) to do the conversion. Take for example *JSON example-1* on the previous page. This object doesn't match any of our existing Entity Classes, so we must create a matching Java class, `PersonDetail`, with the only purpose of serving as a template for our JSON-conversions. These classes however, are very simple to create, since they are data-only classes (no behaviour), so it's a common strategy to use. Add mapper classes like this, in a specific package (`jsonmapper`, `dto` or similar) to clearly distinguish them from the Entity classes.

From Java to JSON and from JSON to Java without POJO's

If you don't like the strategy with a matching Java Class (POJO) for each JSON object you can use Gson's low level parser as sketched below (using hardcoded data values) to build a JSON string. The advantage is that you dynamically can add code to decide which properties to include:

```
JsonObject jo = new JsonObject();
jo.addProperty("firstName", "Lars");
jo.addProperty("lastName", "Mortensen");
jo.addProperty("email", "lm@aaa.dk");
jo.addProperty("street", "Lyngbyvej 23");
jo.addProperty("additionalInfo", "");
jo.addProperty("zipCode", "2800");
jo.addProperty("city", "Lyngby");
JsonArray phones = new JsonArray();
JsonObject phone1 = new JsonObject();
phone1.addProperty("number", "12345678");
phone1.addProperty("description", "Mobile");
JsonObject phone2 = new JsonObject();
phone2.addProperty("number", "23232323");
phone2.addProperty("description", "Fixed");
phones.add(phone1);
phones.add(phone2);
jo.add("phones", phones);
String jsonStr = gson.toJson(po); //The JSON string is ready
System.out.println(gson.toJson(jsonStr));
```

This low level API obviously also works the other way around, so you can take an incoming JSON string and grab each of the individual values as sketched below:

```
JsonObject o = new JsonParser().parse(jsonAsString).getAsJsonObject();
System.out.println(o.get("firstName"));
System.out.println(o.get("lastName"));
System.out.println(o.get("email"));
System.out.println(o.get("street"));
System.out.println(o.get("additionalInfo"));
System.out.println(o.get("zipCode"));
System.out.println(o.get("city"));
JsonArray phonesArr = o.getAsJsonArray("phones");
for(JsonElement pjo : phonesArr){
  System.out.println(pjo.getAsJsonObject().get("number"));
  System.out.println(pjo.getAsJsonObject().get("description"));
}
```