Java Practical Work: Advanced Java Programming - f2023

The Goal of this practical work is to assess the student's ability to write and complete Java code, to show his operational skills in real situation.

This Practical Work is composed of 5 domains

- Usage of maven,
- Unit testcases implementation,
- Dependency Injection using JSR-330 (Inject) and Spring,
- Persisting data thanks to JPA and repository approach.
- Exposing REST services using Spring Boot or other technology.

Each domain needs to be completed to go on, as those domains are ordered from the most general to the most specific.

Remark: you'll have a global bonus If you follow good coding practices, like putting comments on critical code, using loggers, Javadoc, code formatting, naming conventions, code organization…

Expected delivery structure:

```
project-root/
    output/
    core-module/
        src/main/java
            <your_code>
        src/test/java
            o TestMVD2
            o TestMVD3
            o TestJUN1
            o TestJUN2
            o TestSPR1
            o TestSPR2
            o TestSPR3
            o TestJPA1
            o TestJPA2
            o TestJPA3
            o TestJPA4
    rest-module/
        src/main/java
            <your_code>
        src/test/java
            o TestRST1
            o TestRST2
            o TestRST3
            o TestRST4
```

Domain 1: Maven and data

Exercise MVD1 – project setup

15 minutes ***1pt***

Create a maven project, it should have these dependencies (format given with groupid:artifactid:version) :

- org.springframework.boot:spring-boot-starter-web:2.7.1
- org.springframework.boot: spring-boot-starter-test:2.7.1
- com.h2database:h2: 2.1.214
- org.hibernate: hibernate-core: 5.6.9.Final
- org.springframework: spring-orm:5.3.20
- org.springframework: spring-test:5.3.20
- org.springframework: spring-context:5.3.20
- org.junit.jupiter: junit-jupiter-engine:5.8.2
- javax.inject:javax.inject:1

do a mvn clean install and copy the console output to the directory “out” in the file “mvd1.out”

Exercise MVD2 – data modeling

10 minutes ***1pt***

Have a look to the sql file “base.sql”, it contains definitions and data about 2 tables “MEMBERS” and “FACILITIES”.

- Prepare 2 java classes (datamodel) to represent the data structures.

Initialize 1 instance of each class with hardcoded values, then display them in the console. This code should be put in the class `TestMVD2`. The code is executed through a `main()` method

Exercise MVD3 – data importation

20 minutes – ***2pts***

Import the data contained in base.sql in an h2 in memory database instance, then read them back from the database as list of instances of the model class defined in MVN2. Your code has to read from sql file(s) and execute the sql code contained in it.

Write this code in `TestMVD3.` The code is executed through a `main()` method

Hint: you can reorganize the base.sql file into multiple files to load them easily. You can for instance split it into :

- create-members.sql
- insert-members.sql
- create-facilities.sql
- insert-facilities.sql

beware that jdbc will not like lines ending with “;” so remove them manually if you want your script to be imported correctly.

Domain 2: JUnit

Exercise JUN1 – BDD (Given-When-Then)

*10 minutes – **1pt***

Copy the code you have made for **MVD3** exercise in a class called ***`TestJUN1`***. It should be a junit test class

How can you assert that the size of the list fetched from the database is correct? Put your solution in ***`TestJUN1`***.

Exercise JUN2 – Using the JUnit annotations.

*10 minutes – 1**pt***

Copy TestJUN1 into TestJUN2. In ***`TestJUN2`***, use the proper Junit annotations to properly setup the database import before and cleanup the database after **each** test

Domain 3: DI using Spring

Exercise SPR1

*10 minutes – **1pt***

Setup Dependency injection for this project by creating an appropriate Java class configuration. Use this context to inject a bean of type String, containing “test from spring!” and display it in the test class ***`TestSPR1`***

Exercise SPR2

*15 minutes – 2**pt***

Create a javax.sql.Datasource bean representing the h2 connection;

- Initialize the database in the bean declaration as in MVD3 (by reading from the sql files).
- Inject this bean in a ***`TestSPR2`*** class
- Assert the members count in the database is as expected

Domain 4: JPA with Hibernate

Exercise JPA1 setting up the context

*15 minutes – 2**pt***

Setup JPA in this project using the configuration defined for SPR1 and SPR2 exercises.

- Annotate you datamodel classes properly
- Configure then Inject an EntityManagerFactory (SessionFactory) in a test class named ***TesJPA1***.
- Create a hibernate session,
- Try to persist an instance of ***Member*** datamodel.
- Assert that it has been correctly integrated in the database.

Exercise JPA2 – Dealing with DAOs

*25 minutes – 2**pt***

- Create 2 DAOs, one for ***Members*** and the other for ***Facilities***.
- Create a class named ***TestJPA2*** in which you will use these daos to fetch all data from the 2 tables

Hint: in JPQL, the sql equivalent of “select * from MEMBERS” will be “from Member”.

Exercise JPA3 – Modifying the model to add the booking entity

*20 minutes – 2**pt***

- Create a third datamodel class matching the data structure defined by the sql script “bookings.sql”.
- Annotate this class so that it has 2 ManyToOne Relationships (1 for member, 1 for facilities). You can decide what will be the column name containing the relationship by using the ***@JoinColumn*** annotation, as follows

```
@ManyToOne

@JoinColumn(name="memid", referencedColumnName="memid")

Member member
```

- Fetch in the ***`TestJPA3`*** all the instances of bookings from JPA, assert that the size of the list is 4044 elements.

Exercise JPA4 – designing a repository to manage bookings

*40 minutes –4**pt***

Remark: this exercise is not mandatroy for next domain

Define a BookingRepository which will be responsible for managing bookings. It should be possible to use this repository to create a booking, providing an instance of member and instance of facilities (previously existing and fetched thanks to respective DAOs – important notice), the date and time of the reservation and the number of slots taken by the member for the facility.

The usage should be like :

```
repo.createBooking(member,facility, bookingDateTime, slotsTaken);
```

Test this method in the test class ***`TestJPA4`***

Domain 5: REST

Exercise RST1

*10 minutes –1****pt***

Initialize a Spring boot application and make it run in a class named ***`TestRST1`***.

Exercise RST2

*10 minutes –1****pt***

Define a REST controller ***`MemberRestController`*** exposing the POST and GET http methods (output message "hello from get!" and "hello from post" in the console for each) on base url "members"¸ restart ***`TestRST1`*** and check your services are available.

Exercise RST3

*20 minutes – 3****pts***

Define a REST controller ***`FacilitiesRestController`*** exposing POST and GET methods with baseUrl "facilities".

- Design the messages classes to define your external datamodel (DTOs).
- (optional) adapt the repository defined in the JPA section to accept the written dtos.
- Inject in this controller the JPA DAO of Facilities (you have to reimport the hibernate configuration in this spring boot application, as shown in class)
- Make the GET http method return a list of all the facilities (input: nothing, result: json array)
- Make the POST http method create a new facilities (input: json structure, result: nothing)

Start your spring boot application through a ***`TestRST3`*** class. The code is executed through a ***`main()`*** method

Exercise RST4 – BONUS

*20 minutes – 3****pts***

Define integration tests with junit in TestRST4 to make sure your rest service is operational on the 2 methods (GET and POST).