Information Systems Institute

Distributed Systems Group (DSG)
VU Distributed Systems Technologies SS 2015 (184.260)

Assignment 1

Submission Deadline: 9.4.2015, 18:00

General Remarks

- Group work is not allowed in the lab. You have to work alone. Discussions with colleagues (e.g., in the TUWEL forum) are allowed, but the code has to be written alone. If we find two students, who submitted the same, or very similar assignments, these students will be graded with 0 points (no questions asked). We will use automated plagiarism checks to compare solutions. Note that we will also include the submissions of previous years in our plagiarism checks.
- No deadline extensions are given. Start early and, after finishing your assignment, upload your submission as a zip file to TUWEL. If you think that you will be hard-pressed making the deadline you should upload a first version well before time runs out! We will grade whatever is there at the deadline, there is no possibility to submit later on.
- Make sure that your solution compiles and runs without errors. If you are unsure, test compiling your submission on different computers (e.g., one of the ZID lab computers). Preferred target platform for the entire lab (all 3 assignments) is JDK 7[1]. Before you submit, make sure to test your solution with JDK 7 - points will be deducted if your code does not build/run out of the box.
- The assignment project is set up using Apache Maven[2]. We provide a project template that contains some code interfaces and JUnit tests, which will assist you in developing your code. Please stick exactly to the provided interfaces as we will check your solutions in an automated test environment. The Maven project is split up into multiple modules, which correspond to the different sub-parts of the assignments. **Note:** If all unit tests are passing in your solution, it does *not* necessarily mean that you will receive all the points. We will perform additional code checks and run tests that are not provided in the template. The tests included in the template should merely help you to get started and assist you in developing your solution. If you want to further increase your test coverage, you may also add new unit tests, but this is optional. Do not modify any of the pre-defined tests - in any case, we will check your code with the original tests from the template.
- A good introduction into JPA can be found in Part VI of the Java EE 6 Tutorial[3].
- Before asking questions about Hibernate, check out the Hibernate Documentation[4] page. If you cannot find an answer to your question, consult the Hibernate forum[5].
- For the MongoDB tasks, consult the MongoDB Java tutorial[6].
- We use (as can be seen in the persistence-unit configuration of persistence.xml) a H2[7] database, named `dst` that can be accessed without credentials. You may of course change the settings of the configuration for your work at home, but please reset them to the original values in your submitted solution (and make sure it still works). Again: make sure that all settings are as expected before you submit!
- Please make sure to add reasonable logging output to help us keep track of what your solution does. No debug output is very bad, and too much (e.g., many screen pages) is just as bad. Aim for a good middle ground, which allows us to check your solution quickly.
- **Later assignments will be based on (parts of) the solution of this lab. Keep this in mind while implementing your code**.

[1] `http://www.oracle.com/technetwork/java/javase/downloads/`
[2] `http://maven.apache.org/`
[3] `http://download.oracle.com/javaee/6/tutorial/doc/bnbpy.html`
[4] `http://hibernate.org/docs`
[5] `https://forum.hibernate.org/`
[6] `http://www.mongodb.org/display/DOCS/Java+Tutorial`
[7] `http://www.h2database.com/html/main.html`

A. Code Part

Command for building/testing parts 1.-4.:	`mvn install -Pass1-jpa`
Command for building/testing part 5.:	`mvn install -Pass1-nosql`
Command for building all parts without testing:	`mvn clean install -Pall -DskipTests`
Command to force a check for updated releases:	`mvn clean install -U`

1.1. Mapping Persistent Classes and Associations (16 Points)

Throughout this lab, it is your task to build the enterprise application for managing a Massive Online Course (MOC) platform. A lecturer can create and publish lectures on the platform. The platform then allows the streaming of the lecture in one or more online classrooms, i.e., makes it available to a group of students listening to that course. Our simple assumption here is that each classroom has a fixed number of student capacity it can satisfy. Lectures may be assigned freely to classrooms to satisfy the total number of attending students. For instance, we assume that a lecture that is attended by 100 students can either be processed by 4 classrooms with a student capacity of 25 or by 5 classrooms with capacity 20. Classrooms are organized in virtual schools. Every classroom belongs to exactly one virtual school. It is also possible that a virtual school is recursively composed of other virtual schools to facilitate a hierarchical organisation. Each virtual school is maintained and moderated by exactly one moderator.

Please note that your task in this and the following assignments is to build the Massive Online Course Management System, not the Massive Online Course Platform itself. We will only simulate the actual streaming of lectures.

The domain model that we use as starting point is defined in the following UML class diagram. We will come back and slightly extend this data model in future tasks and assignments.

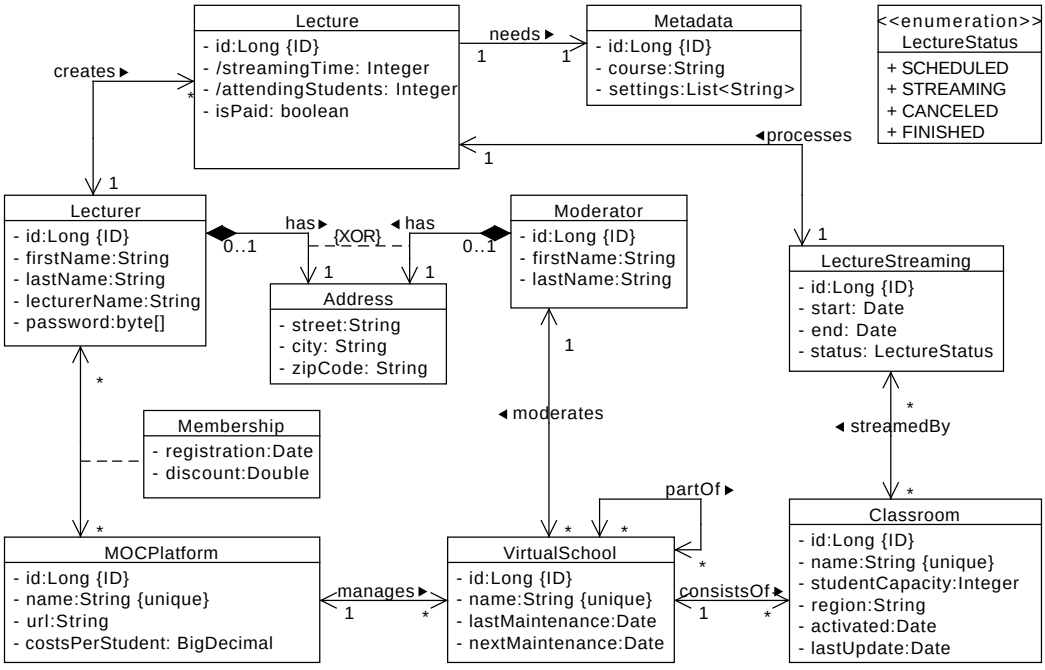

Figure 1: Massive Online Course Platform - Domain Model

Note that each lecturer and each moderator have an address. These addresses shall be mapped as embedded classes. Also notice that each lecture needs some kind of metadata. The metadata consists of a course it belongs to and a list of settings. The ordering of the settings in this list is important, i.e., the order should not change after persisting and loading the entity. Lectures are streamed to a number of classrooms, possibly in parallel (a lecture can be streamed to many classrooms, all at the same time). Lecturers do not need to be members of the MOC platform to assign lectures. Memberships are identified via the platform's and the lecturer's ID. If the lecturer has a membership, the platform may also give discounts (in percent) to lecturers for assigned lectures. Finally, the password of lecturers is stored as a cryptographic hash of a password string (never store plain text passwords to the database). For simplicity and due to its wide availability you should use MD5 as hashing function (although in practice MD5 is known to have some vulnerabilities). MD5 hashes are fixed size, so make sure that Hibernate uses the optimal data type for your password column.

In the template, **interfaces** (starting with capital letter "`I`") are provided for all **model classes** and **data access objects** (DAOs). You should create an implementation class for each of the interfaces and make sure to correctly instantiate objects using the **ModelFactory** and the **DAOFactory**. Put all the implementation classes you add into a separate **impl** sub-package, i.e., **dst.ass1.jpa.model.impl** and **dst.ass1.jpa.dao.impl**.

1.1.a. Basic Mapping

Map these classes and all associations using the Hibernate JPA implementation (i.e., only make use of the package `javax.persistence.*` in your mappings and the respective test code). For all classes, use annotations for the mappings. The only entity you should use XML mappings for is the **Classroom** entity. Make sure that you implement the navigation in the data model as specified in the diagram (for instance, the Lecture-LectureStreaming association is bidirectional, so implement it that way).

Note that you will also need your solution for this part for almost all tasks in later assignments, so we strongly recommend that you solve at least this part of the assignment.

1.1.b. Inheritance

While solving task (1.a.), you may have noticed that lecturers and moderators resemble each other in many ways. Therefore, we will now implement an inheritance relationship via the abstract entity *Person*. Expand and adapt your code from task (1.a.) by choosing one of the three well-known inheritance patterns. Be prepared to argue your choice during the practice lesson! At this point, we also add information about the lecturer's bank account to be able to debit money for the streamed lectures. The required changes are shown in the following diagram. You also need to implement the two new unique constraints, as well as the new not-null constraint.

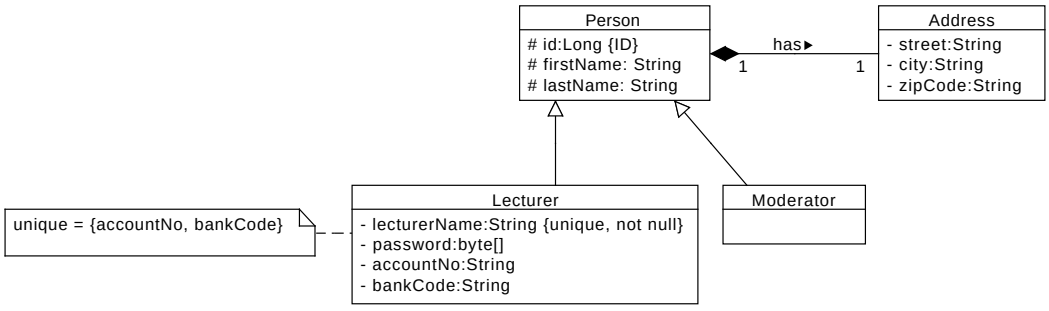

Figure 2: Massive Online Course Platform - Inheritance Model

Now that you have finished mapping the platform's domain, you need to test/demonstrate your mapping. The unit tests in the template store a reasonable amount of entities in the database, retrieve them, update

some of them, and delete some of them again. However, you may extend the pre-defined tests and also include additional test classes to cover any special situations and corner cases, if necessary. Make sure that your solution works in its entirety, that is, no unnecessary tables get created, no unintended information is lost when deleting entities (e.g., because of unexpected cascades), and, of course, that there are no exceptions thrown when doing any of it.

1.1.c. Additional Mapping

For later assignments our system requires a manageable pricing model in order to charge lecturers, who are using the Massive Online Course platform. Therefore, we will introduce the Price entity. Furthermore, to keep track of operations executed in the MOC platform we will use the AuditLog and AuditParameter entities (Figure 3).

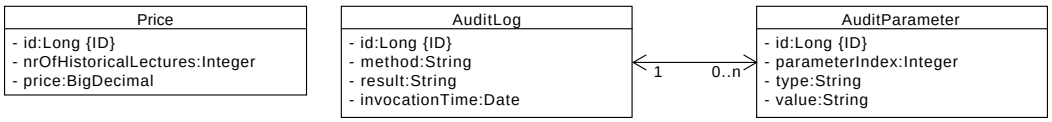

Figure 3: Massive Online Course Platform - Additional Entities

Note that these entities (dst.ass2.ejb.model) and their respective DAOs (dst.ass2.ejb.dao) will be used in the next assignments. Furthermore the following DAOs contain convenience methods, which will be helpful in later assignments:

- **IClassroomDAO.findByPlatform(...)**
- **ILecturerDAO.findByName(...)**
- **IMembershipDAO.findByLecturerAndPlatform(...)**

You can choose whether you implement them now or later. But keep in mind that they will not be tested/graded as part of this assignment.

1.2. JPA-QL and Hibernate Criteria API (8 Points)

1.2.a. Write the following **Named Queries** (no plain SQL, and no inline queries!) for the domain model implemented in Task 1. Your queries should solely accept the specified parameters and return the requested types. Make sure to name your queries exactly as specified in the pre-defined tests of the template.

- Find all lecturers who have an active membership for a specific platform and assigned at least x lectures to this platform (given the name of the platform and the number of lectures).
- Find the most active lecturer(s), i.e., the lecturer(s) who assigned the largest number of lectures.
- Find all completed lectures (i.e., all lectures with lecture streaming status FINISHED).

1.2.b. Implement a named query to find all moderators with first name starting with 'Alex' and, for each of them, calculate the next upcoming maintenance (get the earliest *nextMaintenance* attribute among all the virtual schools of the respective moderator). You do not need to find a single query to solve this task (you can use a combination of Java code and named queries), but you have to keep ORM-performance in mind, i.e., make sure that your solution is also reasonably fast if you have many virtual schools on many moderators. During the presentation you should be able to answer which problems you can run across here. For the unit tests to succeed, implement the corresponding DAO interfaces (methods are already defined in the interfaces).

1.2.c. Write the following queries using the **Hibernate Criteria API** (i.e., you are not able to conform to JPA this time). All parameters should be optional (i.e., indicated by `null` values). Again, put the code into your implementation of the corresponding DAO methods.

- Find all lectures that were created by a specified lecturer, which are part of a given course (lecturer is given by his lecturerName).
- Use **Query-by-Example** to find all lectures with LectureStatus `FINISHED` and a specified start and end date.

1.2.d. Now we want to optimize the following query by adding an **index** with Hibernate:

- `"Select l from Lecturer l where l.password = MD5(?)"`

Assume that the lecturer base will be very large, and this query will be executed often (i.e., whenever a lecturer logs in). Assign the correct index value to speed up this query. Prepare to argue during the practice lessons, in which cases this index is helpful, and when it might even be harmful to the performance of the application. We encourage you to also run some large-scale tests (adding huge amounts of lecturers, running the query, taking the time, deleting the lecturers again) to check whether (and under which circumstances) the index has an influence on the performance. Implementing large-scale tests is optional, but could nicely contribute to our discussion during the lab session.

1.3. Bean Validation (3 Points)

The Bean Validation API is a feature in Java EE version 6 and above. You can use it to introduce restrictions on class fields or entire class instances, e.g., to verify that a certain String contains a well-formed email-address or that an Integer field has a value in the range between two specified constants. Bean Validation becomes especially useful when validating user input before storing it to the database, but it is explicitly not tied to any programming model or application layer.

1.3.a. In a first step we will start by applying some built-in constraints of the **Bean Validation API** to our classroom (this time, you are allowed to use annotations for the classroom entity):

- A classroom has dates for activated and lastUpdate, both have to be in the past.
- Every classroom has a name not shorter than 5 and no longer than 25 characters.
- The region of the classroom starts with three upper-case alphabetic characters (for the Country) followed by a '-', another 3 upper-case alphabetic characters (for the City) followed by a '@' and at least 4 digits (for the zip-code). Example: 'AUT-VIE@1040'.

1.3.b. The platform only uses classrooms, which have a specific amount of capacity. Therefore, provide a constraint annotation and a constraint validator to ensure that the capacity is between a defined upper and lower bound. If validation fails, return meaningful error messages. Put the code into **dst.ass1.jpa.validator**. The annotation should look like this:

```
@StudentCapacity(min = 40, max = 80)
```

The pre-defined tests in the template instantiate valid and invalid classroom entities, and use an instance of `javax.validation.Validator` to validate each entity. Invalid entities should violate every single constraint. We advise you to also create your own test class with additional test cases to verify each constraint individually.

1.4. Entity Manager, Entity Listeners and Interceptor (5 Points)

1.4.a. Entity Manager - Lifecycle

Up to this point, you already gained some experience storing and retrieving entities using the EntityManager. Now it is time to study the lifecycle of Persistent Entities. Create a transaction

sequence to illustrate the persistence lifecycle of a lecture object (there has to be an association with a lecturer). Additionally, make some source code comments that describe the current state of the activity object. Put the resulting code into **dst.ass1.jpa.lifecycle.EMLifecycleDemo**. Make sure that all possible states are covered by your example.

1.4.b. Entity Listener

Entity Listeners allow us to register callback methods for specified lifecycle events. Provide an entity listener which sets the activated- and lastUpdate-field every time a new classroom gets stored and overwrites the lastUpdate-field every time an already existing classroom entity is updated. Implement your callback methods in the class **dst.ass1.jpa.listener.ClassroomListener**.

1.4.c. Default Listener

Now your task is to implement a simple default listener. This listener should record the number of entities loaded, updated and removed. This listener also records how often a persist operation was successfully called and how much time all store-operations took in total. Implement your listener by completing the class **dst.ass1.jpa.listener.DefaultListener**. Make sure that your listener implementation is thread-safe and concurrent!

1.4.d. Interceptor

Task 1.2.b. aimed at minimizing the number of selects issued to the database. Now we will use the Hibernate Interceptor interface to count the number of database select statements that Hibernate actually uses in the background. The Hibernate Interceptor interface provides callbacks from the session to the application, allowing the application to inspect and/or manipulate properties for a specific event. Implement the provided **dst.ass1.jpa.interceptor.SQLInterceptor** and count the number of selects for moderators and virtual schools. It is ok to check if the SQL-String starts with the respective select-statement (also think about distinct selects). In addition, you should provide methods to reset the select statement counter. Use the interceptor to validate that your solution for Task 1.2.b. works efficiently. Make sure that your implementation is thread-safe and concurrent!

1.5. NoSQL Database (8 Points)

So far, all tasks made use of a traditional SQL database and JPA/Hibernate. For the last practical tasks of this assignment, we will look at a different database paradigm, so-called NoSQL (Not only SQL) databases, exemplified on the basis of *MongoDB*. The project template already contains a dependency to MongoDB, and upon initialization the class **dst.ass1.NoSQLTestData** runs an embedded MongoDB server. That is, no local installation is required on your machine. The template project uses the default MongoDB port 27017 (make sure this port is free), and skips any authentication features (evidently, you would not want to do this in production, but for our tests this is the easiest way to get started).

The package **dst.ass1.nosql** in the project template contains an interface for loading test data into MongoDB (**IMongoDbDataLoader**), and one interface for running queries against the MongoDB database (**IMongoDbQuery**). Use these interfaces for the remaining parts of this section, and put your implementation classes into a sub-package **dst.ass1.nosql.impl**. Also make sure to properly instantiate and return your implementation classes in **MongoDbFactory**.

1.5.a Storing lecture data in MongoDB

So far, all data that we wanted to store in our MOC Platform management system was fairly well-structured and uniform. However, assume now that we also need to attach some custom data uploaded by the lecturer for each lecture-streaming (denoted as 'streams' for this part of the assignment), e.g. custom input data for homework assignments. These data is evidently quite unstructured, as different types of lectures might require different types of assignment data. For instance, a mathematics lecture might require a matrix for performing the assignment, while a bioinformatics lecture might require a genome sequence sample. Furthermore, there is no way at design time to know which data we will need to store. Hence, we will use MongoDB, a schema-free NoSQL database for this part of the system.

Every lecture data can be represented as a JSON document with 2 or more properties. The first property is **lecture_id**, which refers back to the relational database ID of the lecture that produced this output. The second property is **lecture_finished**, containing a UNIX timestamp when the lecture's stream ended. Finally, stream outputs can have 0 or more additional properties, containing additional information. An example is provided below.

```
{
  "lecture_id" : 2 ,
  "lecture_finished" :  1329469221971,
  "lecture_data" : {
    "matrix": [[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]],
  }
}
```

Your first task is to add lecture data for every finished lecture to your MongoDB installation. Add the 3 provided example documents in JSON notation (see class **MongoTestData**).

Put the code into your implementation of the method **IMongoDbDataLoader.loadData**. Inside the **loadData** method, you should first load all finished lectures from the H2 database (using the named query created earlier in this assignment - 1.2.a.), loop over all lectures and store a JSON object with the stream data to MongoDB. Use the official MongoDB Java driver (already included in the Maven dependencies of the template). Since it does not really matter which output you assign to which lecture, you can select any random result data using the **getData** and **getDataDescription** methods from the JSON documents in **MongoTestData**. Make sure to correctly set the `lecture_id` and `lecture_finished` attributes in all JSONs you save to MongoDB. Furthermore, as we will often retrieve documents by `lecture_id`, you should add an index to speed up such queries.

1.5.b Querying lecture data

MongoDB primarily uses the query-by-example pattern for retrieving data, which we have already used in Task 2.c. In your implementation of the interface **IMongoDbQuery**, you have to write two simple queries, which (1) fetch the lecture data for a given `lecture_id` (e.g., for the lecture id 1) and return the finished timestamp. (2) fetch all results with a finished field later than the given timestamp. For the second query, use a filter to retrieve only the actual `lecture_id` of the document (no other properties should be contained in the delivered result). Provide reasonable logging output with information about the results of your queries.

1.5.c Map/Reduce queries

In addition to query-by-example, we can also use the more powerful, but also more complex, map/reduce pattern for data retrieval and processing. MongoDB supports map/reduce via JavaScript, that is, you will need to write both the map and the reduce functions as JavaScript functions that operate on your stored documents. You can find a good example online[8]. Another blog post shows how you can call map/reduce via the Java driver[9]. Your task is now to implement the method **IMongoDbQuery.mapReduceStreaming** to use map/reduce for reporting some rudimentary statistics about the stored stream output documents. The output of your map/reduce query should be a list of all existing (top-level) document properties, along with the number of documents that they occur in. Do not consider the properties containing the `lecture_id` or the `lecture_finished` field. Similarly, skip the `_id` field which MongoDB automatically generates for each document. Essentially, this report allows us to reason about how many documents there are for each type of stream. See below for an example output. As before, print the results of your query to the system output stream.

```
{ "_id" : "alignment_block" , "value" : 3}
{ "_id" : "logs" , "value" : 1}
{ "_id" : "matrix" , "value" : 4}
```

[8] `http://architects.dzone.com/articles/walkthrough-mongodb-mapreduce`
[9] `http://blog.evilmonkeylabs.com/2011/02/28/MongoDB-1_8-MR-Java/`

B. Theory Part

The following questions will be discussed during the practice lesson. Before the actual lesson, you can specify the questions you have prepared and are willing to present in the respective ticking activity (**Practice Session 1 - Ticking**) in TUWEL. We will then select students at random who checked a question to discuss the question (you know the procedure from your math courses). If you are asked to discuss a question but fail to provide a strong answer, you will lose **all** points for the theory part of this assignment.

1.6. Annotations vs. XML (2 Point)

In the previous tasks you already gained some experiences using annotations and XML. What are the pros and cons of each approach, when would you use which one? Answering these questions, also keep in mind maintainability and the different roles usually involved in software projects.

1.7. Versioning (2 Point)

JPA provides a feature called versioning. Why and under which circumstances can this feature be useful? Think about a situation where optimistic locking may result in an (desired) exception.

1.8. Read-Locks (2 Point)

The EntityManager allows the programmer to set Read-locks on specified objects. What are the consequences on concurrent threads when one thread sets such a lock? Think about use-cases this behaviour may be adequate for. What problems can arise?

1.9. Database Indices (2 Point)

What is the purpose and functioning of a database index? Using which data structures do database management systems typically store an index internally, and what are important characteristics of these data structures? What is the basic tradeoff of using an index, what are its limitations? Think of two concrete examples - one in which an index leads to an improvement and one in which the index is useless (i.e., does not lead to an improvement).

1.10. NoSQL Databases (2 Point)

What general types of NoSQL databases exist? Name prominent examples for each type of database, and argue when you should be using them (and also, when you should specifically *not* use them).