KIV/PIA - JPA Association and Inheritance Mapping, Queries

This is project with resolved tasks from the JPA Basics Lab. See the jpa-basics module for blank lab project.

This lab covers:

- mapping of entity classes to database tables - associations and inheritance
- JPQL
- Criteria API

Entity Mapping

In this section we continue with entity mapping we started during the last lab.

Association Mapping

JPA supports mapping depending on the type of association between two entities, each represented by own association. Each of the annotations is used on *getter* of the respective property:

- `@OneToMany` used on collection-type attributes, representing 1..N association.
- `@ManyToMany` used on collection-type attributes, representing M..N association, creates extra table in the process to maintain the database model in 3NF.
- `@ManyToOne` used on an entity attribute type, representing N..1 association.
- `@OneToOne` used on an entity attribute type, representing 1..1 association.
- `@Embedded` used when the associated entity's value should be stored in the same table as the owner entity. This an alternative to `@OneToOne` mapping, preference depends on the particular usecase. Entities which are associated via `@Embedded` must be annotated with `@Embeddable` annotation instead of `@Entity`. Such entities dont have own primary key.

Just like entity and basic attributes have their `@Table` and `@Column` annotations, the association mappings can be complemented by `@JoinTable` for the many-to-many relationship and `@JoinColumn` for the rest.

When mapping associations, it is important to understand the concept of **LAZY** loading. By default all collection mapping are loaded only when they are actually used. I.e. until you access the attribute, it is filled with a proxy object capable of loading the data when needed. The idea behind this is that in many cases you don't require all the associations loaded e.g. (when listing all users in the system, you don't need to read all the Notes they have).

Decision which associations to load eagerly (together with the main object) and which lazily (on access) is crucial to proper optimization of your application. To load an association, by default, another DB query must be run. If done badly, use of JPA may result in hundreds of database

queries run to load a single page (that is bad ;)).

It is a good assumption that all collections should be always lazily loaded and extra DAO method used to retrieve the list when needed. At the same time I would claim that from my experience it is **in many cases** wise to lazily load all associations and use **JOIN FETCH** (see the JPQL section later) instead. We will cover proper query optimization in a single lab.

Inheritance Mapping

There are two main approaches to inheritance mapping:

1. If the parent class is not a stand-alone entity, yet only set of common fields, it should be declared as `@MappedSuperclass`. Such class is not managed as an entity, doesn't have own table. Serves only as a shared definition of field mappings.
2. If the parent class is an entity on its own, it is marked with `@Entity` as usual, plus specific mapping strategy can be chosen using `@Inheritance` annotation on the parent entity class:
   - **JOINED** - parent entity fields are mapped in the parent table, the subclass fields are mapped in the subclass table. Retrieval of the full subclass instance requires JOIN between the tables.
   - **TABLE_PER_CLASS** - each entity has own table, where all the attributes are stored. No JOINs required. List of all instances of the parent class requires multiple queries, though.
   - **SINGLE_TABLE** - there is only one table per class hierarchy. Certain columns are null depending on the particular subclass. Concrete class decided using *discriminator value*.

Concrete strategy depends on the particular use-case - **single table** strategy makes it very easy to retrieve all instance of the hierarchy, but may produce very large tables. **table per class** removes the inheritance from the persistance level, but makes it more difficult to query over multiple classes in the same hierarchy (multiple queries required). Etc.

Java Persistence Query Language

JPQL is SQL-like language independent on the underlying JPA implementation or datastore. It is very similar to SQL, see full specification for details.

The language uses entities and their attributes to form queries in the same way SQL uses tables and columns. The following query would return list of users with the given username.

```
SELECT u FROM User u WHERE u.username=:username
```

where `:username` is a named parameter with name `username` (without the `:`).

JPQL supports SELECT queries and also bulk UPDATE and DELETE queries. Single UPDATE and DELETE and also INSERT operations should be performed using entity manager.

Joins

Joining in JPQL queries follows entity relationships instead of data tables. Check the nice overview.

JPA Criteria API

Criteria API is mechanism for building dynamic, type-based queries in a programmatic way (instead of using the SQL-like JPQL. Their expressive power is equal, hence it is up to the user which one he prefers.

Typed criteria queries are suitable for dynamic cases and allow earlier detection of errors. On the other hand, the string-based JPQL might be easier to read and understand due to its similarity to SQL.

Nice overview of Criteria API basics can be found in the ObjectDB documentation.

JPA Metamodel Generator

JPA Metamodel is a representation of your application's entity graph. The metamodel contains representation of all managed entity classes and their attributes, making them available for use in Criteria API queries instead of string paths. This allows, unlike string paths, for static type check and earlier error detection.

The metamodel can be generated automatically. See `pom.xml` of this project and JBoss Documentation for details.

To generate the metamodel run the following:

```
mvn processor:process
```

Check that the `target` folder now contains class representations with the following name structure: `<EntityName>_`.

License

Base of the JPA setup has been created by Karel Zibar during one of the courses at the University.