KIV/PIA - JPA Basics

This lab covers:

- basic set-up of JPA persistence context backed by Hibernate provider;
- elementary mapping of entity classes to database tables
- JPQL

Java Persistence API

JPA is specification of Java interface for **Object-Relational-Mapping (ORM)**. For details consult the Oracle Documentation.

The specification provides three main areas:

- Entity Mapping Interface - set of annotations for describing how the mapping should be done
- API for entity management
- Query interface - **Java Persistence Query Language (JPQL)** and **Criteria API**

Dependencies

There are two mandatory dependencies in the *pom.xml* file:

- hibernate-jpa-2.1-api - artifact with JPA interfaces (**javax.persistence** package)
- hibernate-entitymanager - entity manager implementation from Hibernate ORM Project.

One optional dependency is C3P0 Connection Pooling Library. Remember the struggle with connection management from the previous lab?

Configuration

In order to use JPA in an application, a so-called **Persistence Unit** must be defined. **Persistence Unit** defines:

- set of entities managed by an entity manager
- data store in which the entities are persisted
- configuration of the entity manager and additional mechanisms

Persistence Units are defined in *META-INF/persistence.xml* file. In a Maven project the location is *src/main/resources/META-INF/persistence.xml*.

Entity Mapping

This section covers field and association mapping of entities.

Class that represents an entity is annotated with `@Entity` annotation. You can use `@Table` to add own specification of the table the entity is mapped to.

```
@Entity
@Table(name="my_entity_table"
public class MyEntity {}
```

Elementary Fields

Primitive data types (such as int, double) and enums are mapped automatically as well as certain common classes such as Date, String. Yet in certain cases it might be necessary to specify additional information about the mapping. For this the following annotations are used:

- `@Basic` - allows to specify whether the property is fetched lazily or whether it is optional. Use of the annotation is optional, if not provided, default settings are used (fetch = EAGER, optional = true)
- `@Temporal` - used to specify type of information persisted in Date attribute - DATE, TIME, TIMESTAMP.
- `@Enumerated` - used to decide whether enum type values are stored as ordinal number or string representation of the value.
- `@Column` - can be used in conjuction with any of the previously mentioned annotations to provide additional information about the mapped column - is it nullable, updatable, etc.

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.

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.

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.

License

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

This work is licensed under the Creative Commons license BY-NC-SA.