KIV/PIA - JPA

This lab covers:

- inheritance mapping
- JPQL and Criteria API

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**

Lab Tasks - Inheritance Mapping

In the first part of this lab we are going to transform are last example so that there is a base class with PK mapping,
which all our entities are going to extend. Additionally we try out several inheritance mapping options.

Base Class for All Entities

1. Create new base class for our entities in the **domain** package

```
package org.danekja.edu.pia.domain;

import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.MappedSuperclass;
import javax.persistence.Transient;

/**
 * Base interface for all entities to make implementation of generic
```

```
dao easier.
  *
  * PK type represents type of the entity's primary key.
  *
  * Date: 26.9.15
  *
  * @author Jakub Danek
  */
 @MappedSuperclass
 public abstract class BaseEntity implements  IEntity<Long> {

     protected Long id;

     @Id
     @GeneratedValue(strategy = GenerationType.AUTO)
     public Long getId() {
         return id;
     }

     public void setId(Long id) {
         this.id = id;
     }

     @Override
     @Transient
     public Long getPK() {
         return getId();
     }

     public abstract String toString();

     public abstract boolean equals(Object o);

     public abstract int hashCode();
 }
```

Note the **@MappedSuperclass** annotation - it means the class is not a stand-alone entity. It is a shared mapping
configuration.

2. Make **User** entity extend the base class - remove the **getPK()** method and **@Id** annotation from the **getUsername()** method. Don’t forget to modify DAO headers

```
public class User extends BaseEntity {


public interface UserDao extends GenericDao<User, Long> {
```

```
public class UserDaoJpa extends GenericDaoJpa<User, Long> implements UserDao {
```

3. Make **Role** entity extend the base class - remove the **id** attribute, its getter and setter and the **getPK()** method.

```
public class Role extends BaseEntity {
```

4. Try *Example 1* in the **App** class to check that everything still works.

Class Hierarchy Mapping Strategies

1. Annotate **Employee** with **@Entity** annotation.

```
@Entity
@Table(name = "danekja_employee")
@DiscriminatorValue("EMPLOYEE")
public class Employee extends User {
```

2. Set User inheritance strategy as SINGLE TABLE.

```
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorValue("USER")
public class User extends BaseEntity {
```

3. Try *Example 2* in the **App** class. Check via *phpMyAdmin* what happened in the database.
4. Change **User** inheritance strategy to JOINED

```
@Inheritance(strategy = InheritanceType.JOINED)
public class User extends BaseEntity {
```

   Note you can remove the @DiscrimantorValue annotation from both entity classes.

5. Try *Example 2* in the **App** class. Check via *phpMyAdmin* what happened in the database.
6. Change **User** inheritance strategy to TABLE PER CLASS

```
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class User extends BaseEntity {
```

7. It is necessary to change ID generation strategy for this to work, so in the **BaseEntity**:

```
@Id
@GeneratedValue(strategy = GenerationType.TABLE)
```

```
    public Long getId() {
        return id;
    }
```

8. Try *Example 2* in the **App** class. Check via *phpMyAdmin* what happened in the database.

Lab Tasks - JPQL and Criteria API

In this part of the lab we are going to implement simple queries using JPQL and Criteria API.

1. First we are going to implement method to search User by his username (since username is no longer a primary key)
   Add the following into **UserDaoJPQL**:

```
    @Override
    public User findByUsername(String username) {
        TypedQuery<User> q = entityManager.createQuery("SELECT u FROM User u WHERE u.username = :username", User.class);
        q.setParameter("username", username);

        try {
            return q.getSingleResult();
        } catch (NoResultException e) {
            return null;
        }
    }
```

2. Try the *Example 3* in the **App** class. Should write user string representation and end with LazyInitializationException
   (do you remember from the last lab?).

3. Let’s try the same thing with the Criteria API. Add the following into the **UserDaoCriteria**:

```
    @Override
    public User findByUsername(String username) {
        CriteriaBuilder cb = entityManager.getCriteriaBuilder();

        CriteriaQuery<User> criteria = cb.createQuery(User.class);
        Root<User> root = criteria.from(User.class);

        Predicate byUsername = cb.equal(root.get("username"), username);
        criteria.where(byUsername);

        TypedQuery<User> q = entityManager.createQuery(criteria);
```

```
     try {
         return q.getSingleResult();
     } catch (NoResultException e) {
         return null;
     }
 }
```

4. Try the *Example 4* in the **App** class. Should write user string representation and end with LazyInitializationException
   (do you remember from the last lab?).

5. Now, let's solve the lazy initialization issue. For that we need to implement **RoleDao.findByUser**.
   First, using JPQL.
   Put into the **RoleDaoJPQL**:

```
 @Override
 public Set<Role> findByUser(String username) {
     Query q = entityManager.createQuery("SELECT u.roles FROM User u W
HERE u.username = :username");
     q.setParameter("username", username);

     return new HashSet<>(q.getResultList());
 }
```

   Note that the JPQL FROM clause is actually **User** entity.

6. Try the *Example 5* in the **App** class.

7. The same thing using Criteria API, in **RoleDaoCriteria**:

```
 @Override
 public Set<Role> findByUser(String username) {
     CriteriaBuilder cb = entityManager.getCriteriaBuilder();

     CriteriaQuery<Role> criteria = cb.createQuery(Role.class);
     Root<User> root = criteria.from(User.class);

     //select roles
     criteria.select(root.get("roles"));

     //where user has username
     Predicate byUsername = cb.equal(root.get("username"), username);
     criteria.where(byUsername);

     Query q = entityManager.createQuery(criteria);
```

```
        return new HashSet<>(q.getResultList());
    }
```

8. Try the *Example 6* in the **App** class.

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.