KIV/PIA - JPA

This lab covers:

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

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*.

Lab Tasks - Basic Mapping

Tasks to complete in this lab.

1. First get acquainted with the *org.danekja.edu.pia.domain* package and its classes. Most importantly the **IEntity** interface.

Finish DAO

1. Open **GenericDaoJpa** in *org.danekja.edu.pia.dao.jpa* - methods **save**, **findOne** and **delete** are missing implementations

2. Implement **save** method
   `if(instance.getPK() == null) { entityManager.persist(instance); return instance; } else { return entityManager.merge(instance); }`

3. Implement **findOne** method
   `return entityManager.find(persistedClass, id);`

4. Implement **delete** method

```
E en = entityManager.find(persistedClass, id);
if(en != null) {
    entityManager.remove(en);
}
```

5. Open **UserDaoJpa** and implement *create* method
   `entityManager.persist(user); return user;`

Annotate Entities

Now we need annotate entities in *domain* package so that the framework recognizes them.
All annotations are from the *javax.persistence* package!

1. Annotate **User** class
   `@Entity @Table(name="danekja_user") public class User implements IEntity<String> {`
2. Specify primary key for the **User** entity
   `@Id @Column(name="username") public String getUsername() {`
3. Mark *getPK()* method of the **User** entity as *transient*
   `@Transient @Override public String getPK() {`
   otherwise hibernate would have attempted to persist it as separate column.

Run Examples

1. Open PhpMyAdmin in browser, login with credentials **pia**:**pia**.
2. Return to project, uncomment *Example 1* in the **App** class and run it.
3. Check the result in phpMyAdmin interface.
4. Comment the *Example 1*.
5. Continue with the examples 2 - 5. Have only a single example uncommented at a single time!

Lab Tasks - Association Mapping

Tasks related to association mapping practise.

Annotate Entities

First we need to annotate additional entities.

1. Annotate **Adress** class

```
@Entity
@Table(name="danekja_address")
public class Address {
```

2. Annotate **Adress**' primary key

```
@Id
@GeneratedValue
public Long getId() {
```

3. Annotate **Role** class

```
@Entity
@Table(name="danekja_role")
public class Role implements IEntity<Long> {
```

4. Annotate **Role**' primary key

```
@Id
@GeneratedValue
public Long getId() {
```

5. Mark *getPK()* method of the **Role** entity as *transient*
   `@Transient @Override public String getPK() {`

6. Annotate **Email** class as *@Embeddable*
   `@Embeddable public class Email {`

Annotate User associations

Now we need to mark User associations properly.

1. Annotate *address* field of the **User** entity (remove the `@Transactional` annotation).

```
@OneToOne(fetch = FetchType.LAZY, cascade = CascadeType.ALL, orphanRemoval = true)
public Address getAddress() {
```

2. Annotate *roles* field of the **User** entity (remove the `@Transactional` annotation).

```
@ManyToMany
@JoinTable(name = "danekja_user_roles", joinColumns = @JoinColumn(name = "user", referencedColumnName = "username"),
inverseJoinColumns = @JoinColumn(name= "role", referencedColumnName = "id"))
public List<Role> getRoles() {
```

3. Annotate *email* field of the **User** entity.
`@Embedded public Email getEmail() {`

Run the examples

1. Open [PhpMyAdmin]() in browser, login with credentials **pia**:**pia**.
2. Return to project, uncomment *Example 6* in the **App** class and run it.
3. Check the result in phpMyAdmin interface.
4. Comment the *Example 6*.
5. Continue with the examples 7 - 8. Have only a single example uncommented at a single time!

Cheatsheet

JPA cheatsheet - annotations, syntax.

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.
- `@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.

License

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