Programação para Web II
Aula 9

Profa. Silvia Bertagnolli

BASEDAO

Repetição de Código

Você deve ter notado que as classes DAO acabam repetindo muito código, então podemos criar a classe BaseDAO que usam genéricos e que vai reduzir esse problema

BaseDAO

```
public abstract class BaseDAO<T, ID>
                                    implements Serializable {

 private static final long serialVersionUID=-595322584650593811 8L;
 private EntityManager em;
 private Class entidade;

 public BaseDAO() {}
```

BaseDAO

```
public abstract class BaseDAO<T , ID>
                                implements Serializable {

  ...
  public Class getEntidade() {
    if (entidade == null) {
        entidade = (Class) ((ParameterizedType) getClass()
            .getGenericSuperclass()).getActualTypeArguments()[0];
    }
    return entidade;
  }
```

BaseDAO

```
public abstract class BaseDAO<T , ID>
                                implements Serializable {

    public void setEntidade(Class entidade) {
        this.entidade = entidade;
    }

    public T pesquisarPorId(long id) {
        em = JPAUtil.getEntityManager();
        return (T) em.find(getEntidade(), id);
    }
```

BaseDAO

```
public abstract class BaseDAO<T , ID>
                                implements Serializable {
    public void remove(long id) {
        em = JPAUtil.getEntityManager();
        em.getTransaction().begin();
        try {
            Object ref = em.getReference(getEntidade(), id);
            em.remove(ref);
            em.getTransaction().commit();
        } catch (EntityNotFoundException e) {
            System.out.println("Não existe o id: " + id);
        } finally {
            em.close();
        }
```

BaseDAO

```
public abstract class BaseDAO<T , ID>
                                implements Serializable {


    public void update(T t) {
        em = JPAUtil.getEntityManager();
        em.getTransaction().begin();
        em.merge(t);
        em.getTransaction().commit();
        em.close();
    }

}
```

BaseDAO

```
public abstract class BaseDAO<T , ID>
                                implements Serializable {

    public void save(T t) {
        em = JPAUtil.getEntityManager();
        em.getTransaction().begin();
        em.persist(t);
        em.getTransaction().commit();
        em.close();
    }
```

BaseDAO

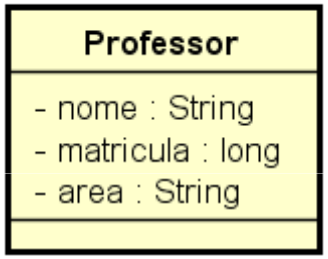
```
public abstract class BaseDAO<T , ID>
                                implements Serializable {


    public List<T> getAll() {
        em = JPAUtil.getEntityManager();
        return em.createQuery("Select entity FROM " +
getEntidade().getSimpleName() + " entity").getResultList();
    }
}
```

Classe PessoaDAO

```
public class PessoaDao extends BaseDAO <Pessoa, Long>{

    private static final long serialVersionUID = 1L;

    //...

    //incluir outros métodos que não estão na classe BaseDAO

}
```

EXERCÍCIOS

1) Refaça a classe Professor e ProfessorDAO, usando o esquema abaixo e a classe BaseDAO:



2) Na classe ProfessorDAO crie os métodos: pesquisarPorNome(), pesquisarPorMatricula(), pesquisarPorArea()

EXERCÍCIOS

3) Monte uma classe de testes para verificar se as ações com os objetos Professor e ProfessorDAO estão funcionando corretamente

JPQL
Criteria

JPQL: TypedQuery

Usada para realizar consultas em uma tabela

Ela substituiu a interface Query que poderia gerar erros de cast

Tutorial:
http://www.objectdb.com/java/jpa/query/jpql/structure

TypedQuery foi substituída pela API Criteria

API Criteria

Permite realizar consultas e estabelecer critérios para essas consultas

Vantagens:

Possibilita identificar erros em tempo de compilação;

Aumenta a segurança, porque reduz as chances de SQL injections;

Realizar verificação de tipos (tipagem forte)

Desvantagem: Muito mais complexa

Exemplo 1

```
TypedQuery<Mensagem> typedQuery = entityManager.createQuery("SELECT
m FROM Mensagem m", Mensagem.class);
List<Mensagem> msgs= typedQuery.getResultList();
```

Com a API Criteria:

```
EntityManager em = JPAUtil.getEntityManager();
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaQuery<Mensagem> query = builder.createQuery(Mensagem.class);
Root<Mensagem> from = query.from(Mensagem.class);
CriteriaQuery<Mensagem> select = query.select(from);

TypedQuery<Mensagem> typedQuery = em.createQuery(select);
List<Mensagem> msgs = typedQuery.getResultList();
```

Exemplo 2

Como listar as mensagens que possuem o id > 2?

```
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaQuery<Mensagem> query = builder.createQuery(Mensagem.class);
        Root<Mensagem> from = query.from(Mensagem.class);
typedQuery = em.createQuery(query.select(from).where(
             builder.gt(from.get("id"), 2)));
msgs = typedQuery.getResultList();
System.out.println("Usando API Criteria");
for (Mensagem m : msgs) {
     System.out.println("Mensagem:"+m.toString());
}
```

Exemplo 3

Como listar as mensagens que possuem o id > 2 de forma ordenada?

```
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaQuery<Mensagem> query =
builder.createQuery(Mensagem.class);
Root<Mensagem> from = query.from(Mensagem.class);
typedQuery = em.createQuery(query.select(from).
             where(builder.gt(from.get("id"), 2)).
             orderBy(builder.asc(from.get("mensagem"))));
msgs = typedQuery.getResultList();
```

Observações

A API Criteria é usada quando queremos utilizar vários filtros usando diversos critérios, por exemplo, selecionar um produto usando o maior preço, o nome, a categoria, etc...

Nesse caso o predicado da consulta é construído por partes e passado como parâmetro na cláusula from do Criteria

Mapeamento ORM com JPA

ORM x JPA

O relacionamento de objetos devem ser traduzidos para entidades do Banco de Dados

JPA é uma solução ORM para fazer isso

JPA permite mapear os relacionamentos entre as classes para tabelas

JPA: Associações

@OneToOne

@OneToMany

@ManyToOne

@ManyToMany

JPA: Herança

@Inheritance

Estratégias:

- Única tabela
- Tabela por subclasse
- Tabela por classe concreta

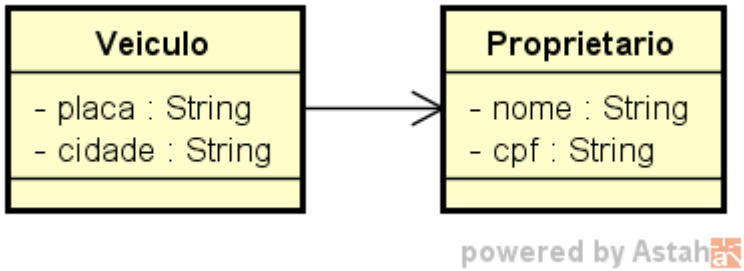
Usar herança de Mapeamento com: MappedSuperClass

@OneToOne

Relacionamentos: @OneToOne

@OneToOne: Usada para mapear um relacionamento "um para um". Ou seja, cada instância de uma entidade A se relaciona com no máximo uma instância de uma entidade B, e vice-versa

Exemplo: Veiculo tem um Proprietario



Classe Proprietário

Classe **Proprietario** é criada como uma entidade sem referência à classe Veículo

```
@Entity
@Table(name="proprietario")
public class Proprietario implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @Column(name="id_proprietario")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nome;
    private String cpf;
...
}
```

Classe Veiculo

Classe **Veiculo** é criada como uma entidade que possui a coluna de junção chamada id_proprietario – isso faz com que o JPA procure em Proprietario a coluna id_proprietario

```
@Entity
@Table(name="veiculo")
public class Veiculo implements Seriali        {
    private static final long serialVer      UID = 1L;
    @Id
    @Column(name="id_veiculo ")
    @GeneratedValue(strategy = Gener     onType.IDENTITY)
    private Long id;
    private String placa;
    private String cidade
    @OneToOne (cascade = CascadeType.PERSIST)
    @JoinColumn (name="id_proprietario")
    private Proprietario proprietario;
...
```

@OneToOne

@JoinColumn – usada para definir o nome da chave estrangeira

Atributo name é usado para indicar qual deve ser o nome da chave localizada na tabela Proprietario

```
@JoinColumn(name="id_proprietario")
```

CASCADE=CASCADETYPE

- **NONE** = Não faz nada com o objeto, portanto é default e não é declarável na enum
- **MERGE** = Faz update nos filhos quando faz update no pai – você só pode dar update no objeto se ele estiver salvo
- **PERSIST** = Salva o filho quando salva o pai, sendo assim, você pode dar um save ou persist no objeto

CASCADE=CASCADETYPE

- **REFRESH =** Salva o pai e mantém o filho sem alterar
- **REMOVE =** Remove o filho quando remove o pai ou vice-versa
- **ALL =** Esse é o cascade do fim do mundo, você vai salvar, fazer update, remover, o que quiser com seu objeto, é altamente não recomendável utilizá-lo.

Combinações de Cascade

@Cascade(cascade={CascadeType.PERSIST,CascadeType.MERGE})

salva em cascata, altera pai e filho em cascata

@Cascade(cascade={CascadeType.PERSIST,CascadeType.MERGE,

CascadeType.REMOVE})

salva em cascata, altera pai e filho em cascata, exclui

em cascata

Combinações de Cascade

```
@Cascade(cascade={CascadeType.PERSIST,CascadeType.REFRESH})
```

salva em cascata, altera apenas o pai e mantém o filho

EXERCÍCIOS

1. Crie a classe Proprietário descrita anteriormente
2. Crie a classe Veiculo descrita anteriormente
3. Crie a classe de testes. Lembre-se que:

- Crie o objeto da classe Proprietario
- Depois, crie o objeto da classe Veiculo e vincule o proprietario usando setProprietario
- Faça a persistência do veiculo e verifique se os objetos foram salvos no BD