Hibernate e JPA

Gustavo Ansaldi Oliva, Márcio Guedes Hasegawa, Pedro Lopes de Souza, Rafael de Oliveira Lopes Gonçalves, Straus Michalsky Martins, Victor Williams Stafusa da Silva

Conteúdo

1. Introdução

ORM (Object-relational mapping) é a solução criada para resolver o problema criado pela diferença de como os dados são armazenados em aplicações orientadas a objetos e bancos de dados relacionais. Estes dois modelos possuem visões diferentes do conceito de entidade, em OO, uma entidade é um objeto que pode possuir relações com outros objetos; em um banco de dados relacional, uma entidade é uma linha em uma tabela, podendo também possuir relações com outras linhas.

2. Java Persistence API (JPA)

JPA é uma especificação de framework relacionado ao ORM. Divulgada na especificação Java EE 5 como parte do EJB 3, tem como funcionalidade a facilitação do gerenciamento da comunicação entre aplicações Java e banco de dados relacionais.

A motivação para criar tal especificação surgiu devido a existência de frameworks muito simples utilizar como o Hibernate e Toplink do que a solução que era fornecida pelo EJB 2, então foi necessária a criação de uma especificação mais simples que padronizasse este cenário.

Hoje, os frameworks que deram origem ao JPA adaptaram-se a esta especificação tornando-se implementações certificadas, um deles, inclusive, se tornou a referência de implementação.

3. HIBERNATE

O Hibernate foi um framework criado para solucionar o mesmo problema que a JPA se propõe. Como citado anteriormente, foi um dos precursores desta especificação e, após a finalização da mesma, incorporou as alterações necessárias para implementá-la.

A ideia inicial era, além de facilitar a comunicação, criar portabilidade entre BDs com pequeno overhead, sendo que o framework seria responsável por gerar o SQL que é executado

4. Annotations

No JPA utiliza-se o @Entity para anotar os POJOs (“Plain Old Java Objects”). É importante ressaltar algumas carateristas dessa entidades:

- Elas não podem ser classes do tipo "final".
- Não podem ter atributos persistentes "final".

- Deve ter um construtor sem argumentos público ou protegido, no entanto ela pode ter outros construtores além deste.
- Pode herdar de outras classes, sejam entidades ou não.
- Outras classes, sejam entidades ou não, podem herdar de entidades.
- Os atributos persistentes não podem ser públicos e só devem ser acessados pela própria classe.
- As anotações são colocadas ou apenas nos atributos, ou apenas nos getters, não se pode misturar.
- Se forem colocadas nos atributos, a classe não é obrigada a ter getters e setters para os atributos.
- Se forem colocadas nos getters, a classe é obrigada a ter getters e setters para os atributos.
- Se puderem ser usadas desacopladas (detached), devem implementar Serializable.

Segue abaixo alguns exemplos de como definir uma entidade:

```
@Entity
public class Funcionario implements Serializable {

        @Id
        @Generatevalue(strategy=GenerationType.AUTO)
        @SequenceGenerator(name="Funcionario_Seq",
sequenceName="Func_Gen")
        private int id;
        private String nome;
        @Temporal(TemporalType.DATE)
        private Date nascimento;

        public Funcionario() {
        }
        public int getId() {
        return id;
        }
        public void setId(int id) {
        this.id = id;
        }
        public void setNome(String nome) {
                this.nome = nome;
        }
        public String getNome() {
                return nome;
        }
        public void setNascimento(Date nascimento) {
        this.nascimento = nascimento;
        }
        public Date getNascimento() {
        return nascimento;
        }
}
```

- @Entity – Define a entidade.
- @Id - Define qual campo que é a chave primária.
- @GeneratedValue - Define como que o JPA gera os valores para a chave primária. Pode

ser de quatro tipos: AUTO (o default), IDENTITY, SEQUENCE e TABLE. No exemplo acima O JPA tenta descobrir qual é a melhor estratégia.

```
@Id
@GeneratedValue(strategy=GenerationType.IDENTITY)
private int id;
```

- strategy=GenerationType.IDENTITY - O banco de dados usa uma coluna de auto incremento.

```
@Id
@SequenceGenerator(name="Funcionario_Seq", sequenceName="Func_Gen")
@GeneratedValue(strategy=GenerationType.SEQUENCE,
                                        generator="Funcionario_Seq")
private int id;
```

- strategy=GenerationType.SEQUENCE - Uma sequence é usada para gerar a chave primária. Essa sequence é descrita em uma anotação @SequenceGenerator.

```
@TableGenerator(
name="Funcionario_Seq",
table="COntadores",
pkColumnName="Nome",
valueColumnName="Proximo",
pkColumnValue="Funcionario"
)
@Id
@GeneratedValue(strategy=GenerationType.TABLE,
                                              generator="Funcionario_Seq")
private int id;
```

- strategy=GenerationType.TABLE - Uma outra tabela é usada para gerar a chave primária. Essa tabela é especificada em uma anotação @TableGenerator.

Para tipos de data, hora e data/hora é necessário as seguentes bibliotecas:

- java.sql.Date mapeia para "data".
- java.sql.Timestamp mapeia para "data/hora".
- java.sqlTime mapeia para "hora".
- java.util.Date e java.util.Calendar mapeiam de acordo com a anotação @Temporal.

```
@Temporal(TemporalType.DATE)
private Date dataNascimento;       //TemporalType pode ser DATE, TIME ou
                                   //TIMESTAMP.
```

Campos ou getters que devem ser ignorados pelo JPA devem ser marcados como “transient” ou ter a anotação @Transient. Campos ou getters de tipos “enum” devem ter a anotação @Enumerated. Essa anotação possui dois valores : ORDINAL, que é o padráo e diz que o método ordinal() do enum será usado para persisti-lo; ou STRING que diz que o método name() será usado para persisti-lo.

```
@Lob
@Basic(optional=true, fetch=FetchType.LAZY)
```

```
private char[] fotografia;
```

- @Lob é usado para mapear campos dos tipos CLOB e BLOB. Normalmente é usado junto com @Basic(fetch=FetchType.LAZY).
- FetchType.LAZY - Indica que o campo só é carregado para a memória quando esse é referenciado.
- FetchType.EAGER - Indica que o campo sempre é carregado para a memória junto com a entidade.
- @Basic é usado para se definir o tipo do "fetch" de uma coluna e se o JPA aceita o valor null em um atributo ou getter (de acordo com propriedade "optional"). O padrão de @Basic é optional=true e fetch=FetchType.EAGER.

Em certas ocasiões, o nome da tabela não corresponde aos da entidade, o nome dos campos
não corresponde aos nomes dos atributos ou setters/getters da entidade ou o nome da
entidade desejado não é o nome da classe. Para resolver isso:

- @Entity tem uma propriedade name, que é o nome que será usado em consultas JPQL.Se omitido o JPA usa o nome da classe.
- @Table contém as propriedades name, catalog e schema. Se omitidos, o JPA tenta deduzir por conta própia. Essa anotação também tem um atributo uniqueConstraints, utilizado apenas por ferramentas que geram o banco de dados automaticamente.

```
@Entity(name="truck")
@Table(name="tb_caminhoes")
public class Caminhao implements Serializable {
...
}
```

A anotação @Column é usado para definir características da coluna no banco de dados.
Os atributos importantes são:

- name - É o nome da coluna. Se ausente o JPA assume que o nome da coluna é o nome do atributo ou getter.
- nullable - Indica se a coluna aceita valores nulos ou não. O padrão é true.
- length, scale, precision, unique, insertable, updatable - Autoexplicativos.
- table - Indica em qual tabela está a coluna (útil em casos onde @SecondaryTable(s) é utilizado).
- columnDefinition - Usado por ferramentas que geram o banco de dados automaticamente.

```
@Entity
public class Carro implements Serializable {
    @Id
    @Column(name="carro_placa")
    private String placa;
    @Column(unique=true, nullable=true)
```

```
        private String renavam;
        ...
}
```

Por vezes, principalmente em bancos de dados ligados, precisamos definir uma entidade que representa duas tabelas ligadas por um relacionamento 1-1. Para isso temos @SecondaryTable. No caso de entidades que representam três ou mais tabelas, usamos @SecondaryTables.

```
@Entity
@SecondaryTable(name="carro_detran")
public class Carro implements Serializable {
        @Id
        private String placa;
        @Column(table="carro_detran")
        private String renavam;
        ...
}
```

Às vezes, um subgrupo de atributos de alguma(s) tabela(s) correspondem a algum tipo de objeto importante. Como por exemplo o endereço.

```
@Entity
public class Empresa implements Serializable {
        ...
        private String rua;
        private String cep;
        private String bairro;
        private String cidade;
...
}

@Entity
public class Funcionario implements Serializable {
        ...
        private String rua;
        private String cep;
        private String bairro;
        private String cidade;
        ...
}
```

Usar campos repetitivos é ruim. Além disso, e se quiséssemos trabalhar com o endereço como se este fosse um objeto? Para isso utiliza-se a anotação @Embeddable ou @Embedded.

```
@Embeddable
public class Endereco {
        private String rua;
        private String cep;
        private String bairro;
        private String cidade;
        ...
}

@Entity
public class Empresa implements Serializable {
```

```
        @Embedded
        private Endereco endereco;
        ...
}

@Entity
public class Funcionario implements Serializable {
        @Embedded
        private Endereco endereco;
        ...
}
```

Mas, e se o nome das colunas não coincidir ? Basta apontar o nome correspondente na tabela.

```
@Entity
public class Empresa implements Serializable {
        ...
        @Column(name="endr_rua")
        private String rua;
        private String cep;
        @Column(name="empresa_bairro")
        private String bairro;
        private String cidade;
        ...

}

@Entity
public class Funcionario implements Serializable {
        ...
        @Column(name="func_endr")
        private String rua;
        @Column(name="func_cep")
        private String cep;
        private String bairro;
        @Column(name="fcidade")
        private String cidade;
        ...
}
```

Caso você não queira ter os mapeamentos @Column com a chave da classe primária, o mesmo se quiser sobrescrever-los, pode-se usar a anotação @Attributeoverrides. Essa anotação é uma lista de arrays da anotação @Atributeoverride.

```
@Entity
public class Empresa implements Serializable {
        @AttributeOverrides({
                @AttributeOverride(name="rua",
                                column=@Column(name="endr_rua")),
                @AttributeOverride(name="bairro",

column=@Column(name="empresa_bairro"))
        })

        @Embedded
```

```
        private Endereco endereco;
        ...
}

@Entity
public class Funcionario implements Serializable {
        @AttributeOverrides({
                @AttributeOverride(name="rua",
                                        column=@Column(name="func_endr")),
                @AttributeOverride(name="cep",
                                        column=@Column(name="func_cep")),
                @AttributeOverride(name="cidade",
                                        column=@Column(name="fcidade"))
        })

        @Embedded
        private Endereco endereco;
        ...
}
```

Para uma chave primária composta podemos usar @IdClass.

```
public class DataComLocal implements Serializable {
        private Date dia;
        private String lugar;

        public DataComLocal() {
        }
        //Getters, setters e outros métodos.
        @Override
        public boolean equals(Object oj) {
        ...
        }

        @Override
        public int hashCode() {
        ...
        }

}

@Entity
@IdClass(DataComLocal.class)
public class Congresso implements Serializable {
        @Id
        @Temporal(TemporalType=DATE)
        private Date dia;
        @Id
        private String lugar;
        ...
}
```

- Importante: A classe da chave primária deve implementar Serializable, ter um construtor público sem argumentos e implementar equals e hashCode de forma consistente. É possível utilizar @AttributeOverride(s) caso seja necessário renomear-se campos ou de alguma outra forma mudar as suas propriedades.

Outra possibilidade para chaves primárias composta é usar @EmbeddedId.

```
public class DataComLocal implements Serializable {
        @Temporal(TemporalType=DATE)
        private Date dia;
        private String lugar;

        public DataComLocal() {
        }
        //Getters, setters e outros métodos.

        @Override
        public boolean equals(Object oj) {
        ...
        }

        @Override
        public int hashCode() {
        ...
        }
}

@Entity
public class Congresso implements Serializable {
        @EmbeddedId
        private DataComLocal dia;
        ...
}
```

Os relacionamentos entre classes são realizados com as seguintes anotações:

- @OneToOne - Cada elemento desta classe se relaciona com um elemento da outra classe.
- @OneToMany - Cada elemento desta classe se relaciona com vários elementos da outra classe.
- @ManyToOne - Vários elementos desta classe se relacionam com um mesmo elemento da outra classe.
- Observação: @OneToMany e @ManyToMany devem ser usados em campos do tipo java.util.Collection, java.util.Set, java.util.List e java.util.Map.
- @ManyToMany - Vários elementos desta classe se relacionam com vários elementos da outra classe.

```
@Entity
public class Empresa implements Serializable {

        @OneToMany
        private Collection<Funcionario> funcionarios;

        @ManyToMany
        private Collection<Cliente> clientes;
        ...
}
```

Todos essas anotações de relacionamento entre classes têm os atributos "cascade",

"fetch" e "targetEntity":

- "cascade", que indica quais operações realizadas na entidade vão cascatear através do relacionamento. Por padrão nada é cascateado.
- "fetch" que indica como as entidades relacionadas são carregadas. @OneToOne e @ManyToOne usam FetchType.EAGER por padrão. @OneToMany e @ManyToMany usam FetchType.LAZY por padrão.
- "targetEntity" define qual é o tipo da entidade relacionada. Necessário em casos onde haja herança entre entidades ou onde o tipo do atributo ou getter for uma coleção não-genérica.

As anotações @OneToOne e @ManyToOne têm também o atributo "optional", para indicar se a associação com a outra entidade é obrigatória ou não. Por padrão é true.

Existe ainda nas anotações @OneToOne, @OneToMany e @ManyToMany o atributo "mappedBy", para resolver casos de relacionamentos bidirecionais. Um dos lados é tido como dono da relação. E o lado oposto usa mappedBy para indicar em qual atributo ou getter do dono da relação ocorre a junção.

```
@Entity
public class Veiculo implements Serializable {

    @ManyToOne
    private Pessoa proprietario;
    ...
}

@Entity
public class Pessoa implements Serializable {

    @OneToMany(mappedBy="proprietario")
    private Collection<Veiculo> veiculos;

    @ManyToOne(fetch=FetchType.LAZY)
    private Pessoa mae;

    @OneToMany(mappedBy="mae")
    private Collection<Pessoa> filhos;
    ...
}
```

- IMPORTANTE: Ao modificar um relacionamento bidirecional no seu programa, lembre-se de fazê-los dos dois lados da relação.

- Problema: não é possível usar @Column em atributos ou getters que se refiram a outras outras entidades com @OneToOne, @OneToMany, @ManyToOne e @ManyToMany. Isso ocorre porque esses objetos não correspondem a uma coluna, e sim a um grupos de colunas existente em uma outra tabela.

```
@Entity
public class Veiculo implements Serializable {

    @ManyToOne
    @Column(name="prop") // NÃO FUNCIONA ! ! !
    private Pessoa proprietario;
    ...
}
```

Para resolver isso basta utilizar a anotação @JoinColumn. Essa anotação é bem parecida com a @Column, porém tem alguns campos a menos. Ela especifica as propriedades da chave estrangeira. Quando a tabela relacionada tiver uma chave composta, use @JoinColumns. @JoinColumn pode ser aplicado em @ManyToOne, especificando qual coluna desta entidade referencia a outra entidade, ou em @OneToMany, especificand qual coluna da outra entidade referencia esta entidade.

```
@Entity
public class Veiculo implements Serializable {

        @ManyToOne
        @JoinColumn(name="cod_pessoa")
        private Pessoa proprietario;
        ...
}
```

Às vezes, em um relacionamento 1-1, não há chave estrangeira, e sim tuplas relacionadas nas duas tabelas que possuem a mesma chave primária. Este seria um caso típico de se usar @SecondaryTable. Mas, se for importante manter as entidades separadas, use @PrimaryKeyJoinColumn, ou @PrimaryKeyJoinColumns para chaves compostas. Se for preciso especificar o nome das colunas da chave primária (ou seja, o nome padrão deduzido pelo JPA não é o correto), use os atributos "name" e "referencedColumnName" de @PrimaryKeyJoinColumn.

```
@Entity
public class Pessoa implements Serializable {
        @Id
        private long cpf;

        @OneToOne
        @PrimaryKeyJoinColumn
        private RegistroCivil registro;
        ...
}

@Entity
public class RegistroCivil implements Serializable {
        @Id
        private long cpf;

        @OneToOne(mappedBy="registro")
        private Pessoa pessoa;
        ...
}
```

Relacionamentos N-N usam uma tabela intermediária. Quando @ManyToMany é usado, o JPA tenta deduzir o nome da tabela intermediária e dos respectivos campos. Se for preciso especificar o nome da tabela intermediária e/ou dos seus campos, use @JoinTable. @JoinTable é quase igual a @Table, no entanto tem os campos joinColumn e inverseJoinColumns a mais.

```
@Entity
public class Pessoa implements Serializable {
        @Id
        private long cpf;

        @ManyToMany
```

```
        @JoinTable(name="visitas",
                joinColumns={@JoinColumn(name="cpf_visitante")},
inverseJoinColumns={@JoinColumn(name="cod_museu")}
        )

        private Collection<Museu> museusVisitados;
        ...
}
```

Ao utilizar um java.util.List para especificar algum relacionamento baseado em coleção, temos um problema: Como manter a lista ordenada?

- Solução: Usar @OrderBy, com um pequeno trecho de JPQL.

```
@Entity
public class Empresa implements Serializable {

        @OneToMany
        @OrderBy("nascimento, nome")
        private List<Funcionario> funcionarios;
        ...
}
```

Ao utilizar um java.util.Map para especificar algum relacionamento baseado em coleção, também temos um problema: Qual é a chave?

- Solução: Especificar o campo da tabela relacionada a ser usada como chave com a anotação @MapKey.

```
@Entity
public class Empresa implements Serializable {

        @OneToMany
        @MapKey(name="cpf")
        private Map<Long, Funcionario> funcionarios;
        ...
}
```

Como modelar herança de entidades? O JPA tem três formas de fazer isso, por meio da anotação @Inheritance:

- Tabela única: @Inheritance(strategy=InheritanceType.SINGLE_TABLE)
- Tabela por classe concreta: @Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
- Tabela por subclasse: @Inheritance(strategy=InheritanceType.JOINED)

Quando a superclasse não é uma entidade, usa-se a anotação @MappedSuperclass.

É possível usar as anotações @AttributeOverride e @AttributeOverrides para redefinir-se campos herdados (tais anotações são colocadas na classe). Para relacionamentos do tipo @OneToOne, @OneToMany, @ManyToOne e @ManyToMany, é possível redefinir colunas de junção por meio das anotações @AssociationOverride e @AssociationOverrides.

As tabelas únicas se baseiam na existência de uma coluna na tabela que identifique o

tipo
de entidade. Essa coluna é especificada com a anotação @DiscriminatorColumn.
O valor da coluna que identifica a entidade é especificado com @DiscriminatorValue.

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="tipo",
                     discriminatorType=DiscriminatorType.STRING)
@DiscriminatorValue("Emp")
public class Empresa implements Serializable {
...
}

@Entity
@DiscriminatorValue("Farma")
public class Farmacia extends Empresa {
...
}
```

A estratégia de tabela única é a mais rápida das três em termos de desempenho, pois não há junções e nem uniões e somente uma única tabela precisa ser administrada. Mas não há normalização e todos os campos das subclasses devem ser marcados como "nullable".

As tabelas por classe concreta funcionam da seguinte forma: Cria uma tabela para cada classe concreta e copia-e-cola os campos da superclasse lá.

```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public class Empresa implements Serializable {
...
}

@Entity
public class Farmacia extends Empresa {
...
}
```

A estratégia de tabela por classe concreta tende a ser mais fácil de modelar e de usar a partir de um banco de dados legado. Além disso ela dispensa a existência de uma coluna discriminadora. Mas, há várias colunas redundantes em tabelas distintas e o modelo não é normalizado. Buscas na superclasse provavelmente exigirão vários UNIONs e o desempenho deixará a desejar.

Para tabela por subclasse cria-se uma tabela para a superclasse e uma para cada subclasse, e realiza junções para uní-las. A união ocorre por meio da anotação @PrimaryJoinColumn(s), nas subclasses.

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class Empresa implements Serializable {
...
}

@Entity
@PrimaryKeyJoinColumn(name="id_empresa")
public class Farmacia extends Empresa {
```

```
...
}
```

A estratégia de tabela por subclasse deixa o modelo normalizado e sem restrições, tendo um desempenho apenas um pouco inferior a estratégia de tabela única.

Quando a superclasse de uma entidade não é uma entidade, usa-se a anotação @MappedSuperclass. Os campos da superclasse serão copiados-e-colados nas subclasses.

```
@MappedSuperclass
public class Empresa implements Serializable {
...
}

@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class Farmacia extends Empresa {
...
}
```

5. EntityManager

O EntityManager é a primeira interface utilizada pelos desenvolvedores da aplicação para interagirem com o JPA. Uma instancia do EntityManager está associada com um contexto de persistência. Um contexto de persistência é um conjunto de instancias de entidades onde cada identidade de uma entidade persistente possui uma única entidade instanciada. E é pelo contexto de persistência que as entidades e seu ciclo de vida são gerenciados. A API do EntityManager é usada para criar e remover as instancias das entidades de persistências, para encontrar entidades por sua chave primaria e para fazer busca entre as entidades.

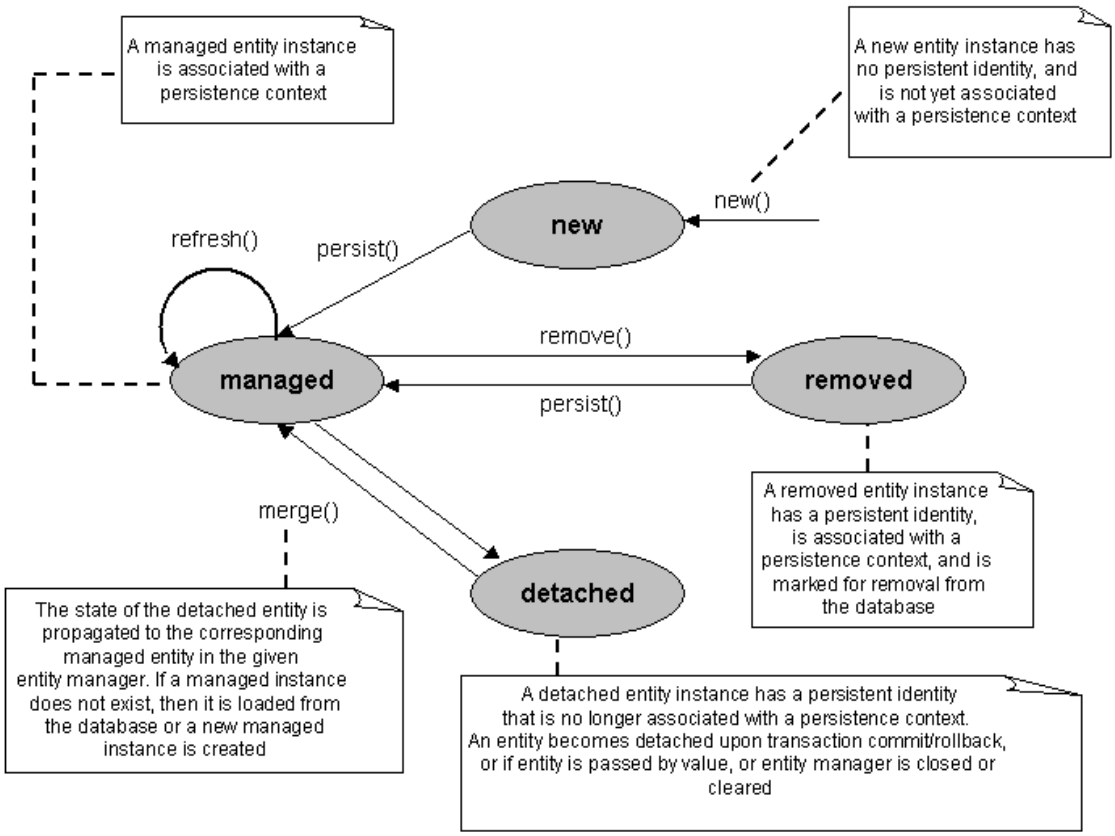
Os métodos do EntityManager serão apresentados a seguir divididos em suas categorias funcionais:

- Associassão de transações (Transaction Association)

```
public EntityTransaction getTransaction();
```

- Gerenciador do tempo de vida de entidades (Entity Lifecycle Management)

```
public void persist(Object entity);
public void remove(Object entity);
public void refresh(Object entity);
public Object merge(Object entity);
public void lock (Object entity, LockModeType mode);
- READ
- WRITE
```



- Gerenciador da identidade de entidades (Entity Identity Management)
public <T> T find(Class<T> cls, Object oid);
public <T> T getReference(Class<T> cls, Object oid);
public boolean contains(Object entity);

- Gerenciador de cache (Cache Management)
public void flush();
public FlushModeType getFlushMode();
public void setFlushMode(FlushModeType flushMode);
- COMMIT
-AUTO
public void clear();

-Gerenciador de identenficadores de entidades (Entity Identity Management)
public <T> T find(Class<T> cls, Object oid);
public <T> T getReference(Class<T> cls, Object oid);
public boolean contains(Object entity);

- Fabrica de querys (Query Factory)
public Query createQuery(String query);
public Query createNamedQuery(String name);
public Query createNativeQuery(String sql);
public Query createNativeQuery(String sql, Class resultCls);
public Query createNativeQuery(String sql, String resultMapping);

-Fechamento (Closing)
public boolean isOpen();

public void close();

6. Enterprise JavaBeans Query Language

EJBQL é a linguagem de manipulação de dados definida no padrão EJB. Ela foi definida para que de maneira portável fosse possível manipular entidades definidas como EJBs.

Similar à SQL (subconjunto de instruções de SQL92) teve seu desenvolvimento inspirado em HQL, a linguagem de query do framework Hibernate.

7. Queries

Os comandos das queries em EJBQL são case insensitive, ou seja select é o mesmo que SELECT porém os nomes relativos aos nomes das classes java não são. Portanto br.usp.ime.Empregado é diferente de br.usp.ime.EmpreGado. Os três comandos básicos são SELECT, UPDATE e DELETE responsáveis respectivamente por seleção de itens, atualização de valores e remoção.

8. SELECT

A forma básica de um select é:

```
SELECT campos FROM entidade [AS nomeParaEntidade]
[WHERE condição]
[GROUPBY clausula]
[HAVING cacondição]
[ORDERBY clausula]
```

As entidades definidas na cláusula FROM podem ser referenciadas em toda a query inclusive em subqueries.

Por exemplo:

```
select emp from br.usp.ime.Empregado as emp
```

o qual devolve uma lista de todas instancias persistidas da entidade br.usp.ime.Empregado.

9. USO DO WHERE

Um exemplo simples de uso da cláusula WHERE em um select pode ser:

```
SELECT emp FROM br.usp.ime.Empregado
WHERE emp.nome = 'Joao'
```

a qual devolve todos os empregados cujos nomes são João.

É possível expressar joins de maneira semelhante a SQL como por exemplo:

```
SELECT departamento
FROM br.usp.ime.Empregado emp
INNER JOIN emp.departamento as departamento
```

ou ainda usando a cláusula de WHERE:

```
SELECT departamento
FROM br.usp.ime.Empregado as emp, br.usp.ime.Departamento as departamento
WHERE departamento = emp.departamento
```

No caso acima,como em todas as comparações do operador = é usado o método equals() da entidade em questão.

A query acima pode ser escrita ainda de maneira implícita e mais compacta:

```
select emp.departamento FROM br.usp.ime.Empregado
```

O exemplo anterior ainda mostra que estas queries podem devolver valores de atributos de entidades.

É possível usando EJBQL simplificar queries quem em SQL tradicionais acabariam usando muitos joins e ficariam muito longas:

```
SELECT emp FROM br.usp.ime.Empregado
WHERE emp.departamento.chefe.endereco.rua = 'Marginal Pinheiros'
```

10. SUBQUERIES

É possível fazer queries dentro de outras queries. Estas devem estar envoltas em parenteses. Por exemplo:

```
SELECT emp.nome
FROM br.usp.ime.Empregado emp
WHERE emp.salario > (
   SELECT avg(departamento.chefe.salario)
   FROM br.ime.usp.Departamento departamento
   )
```

que devolve todos empregados que ganham mais que a média dos chefes de departamento.

É possível também colocar subqueries como parte do FROM:

```
SELECT departamento
FROM br.usp.ime.Departamento as departamento,
IN (departamento.empregados) as empregado
```

```
WHERE empregado.salario > 100
```

que devolve todos departamentos com empregados com salario maior que 100. No ultimo exemplo também é mostrado que queries podem iterar sobre coleções que são membros de outras entidades como é o caso de departamento.empregados

11. UPDATE e DELETE

As operações de remoção e de atualização de valores disponibilizadas através de queries servem para remoção e atualização de grandes quantidades de entidades visto que estas mesmas operações podem ser realizadas usando o *EntityManager* para operações mais pontuais. A sintaxe básica destas operações é:

```
UPDATE entidade SET campo atualizado = valor [WHERE condição]
DELETE entidade [WHERE condição]
```

A cláusula WHERE em ambos os casos funciona como em um comando SELECT. ''ATENÇÃO'' : Estas operações por serem otimizadas não contam com a checagem de trava disponibilizada pelo *EntityManager* assim como a operação de remoção não remove em cascata entidades relacionadas.

12. Definição e Uso das Queries

As queries podem ser definidas de duas formas básicas. Dinamicamente no meio do código:

```
String ejbql = "SELECT e FROM Empregado e WHERE e.nome = 'Joao'";
```

e sendo executadas por meio do *EntityManager*:

```
Query query = em.createQuery(ejbql);
List result = query.getResultList();
```

Ou então definidas como namedQueries no descritor da entidade (*annotations* ou xml):

```
@NamedQuery (
  name="findEmpregadoPorNome",
  query="SELECT e FROM Empregado "
  + " e WHERE e.nome = :nome"
)
})
```

E depois ser realizada através do *EntityManager*:

```
List empregados =
  em.createNamedQuery("findEmpregadoPorNome")
 .setParameter("nome", "Joao")
 .getResultList();
```

Repare que a consulta tem um parametro chamado nome o qual é valorado como 'Joao'. Estes parametros são filtrados de maneira a evitar as chamadas *SQL Injection*s.

13. Hibernate Criteria Query API

14. Introdução

A *Criteria Query API* permite a construção de *queries* (possivelmente aninhadas) utilizando uma API Java, permitindo que a sintaxe seja verificada em tempo de compilação, diferentemente de SQL ou HQL. Trata-se do mecanismo mais fácil de usar do Hibernate para obtenção de dados.

A *Criteria Query API* também inclui uma funcionalidade chamada de *Query by Example* (QBE), permitindo que objetos exemplo que contém determinadas propriedades sejam utilizados como critério de busca. Além disso, a *Criteria Query API* do Hibernate 3 também provê métodos de projeção (*projection)* e agregação (*aggregate functions)* [1], incluindo contagem de linhas.

A seguir, será feita uma breve motivação [2] e em seguida um será mostrado como utilizar os principais recursos da API [3], [4].

15. Motivação

A principal motivação para o uso da *Criteria Query API* é a realização de *queries* com múltiplos critérios. Numa abordagem tradicional, teríamos o seguinte código:

```
Map parameters = new HashMap();
StringBuffer queryBuf = new StringBuffer("from Sale s ");
boolean firstClause = true;

if (startDate != null) {
        queryBuf.append(firstClause ? " where " : " and ");
        queryBuf.append("s.date >= :startDate");
        parameters.put("startDate",startDate);
        firstClause = false;
}
if (endDate != null) {
        queryBuf.append(firstClause ? " where " : " and ");
        queryBuf.append("s.date <= :endDate");
        parameters.put("endDate",endDate);
        firstClause = false;
}
// And so on for all the query criteria...

String hqlQuery = queryBuf.toString();
```

```
Query query = session.createQuery(hqlQuery);

//
// Set query parameter values
//
Iterator iter = parameters.keySet().iterator();
while (iter.hasNext()) {
        String name = (String) iter.next();
        Object value = map.get(name);
        query.setParameter(name,value);
}
//
// Execute the query
//
List results = query.list();
```

Utilizando-se a *Criteria Query API*, poderíamos reescrever o trecho de código acima para:

```
Criteria criteria = session.createCriteria(Sale.class);
if (startDate != null) {
        criteria.add(Expression.ge("date",startDate);
}
if (endDate != null) {
        criteria.add(Expression.le("date",endDate);
}
List results = criteria.list();
```

O código reescrito possui inúmeras vantagens, dentre elas:

- Não há concatenação de Strings

- Sintaxe checada em tempo de compilação

- Manutenção mais simples

16. Primeiros Passos

Para utilizar a API, basta criar um objeto do tipo *org.hibernate.Criteria* através do *Factory Method* createCriteria(...) da *Session*, passando a classe do objeto persistente ou o nome da entidade como parâmetro. O método list() retorna o resultado da busca.

Um exemplo simples:

```
//Get all instances of Person class and its subclasses

Criteria crit = sess.createCriteria(Person.class);
```

```
List results = crit.list();
```

17. Paginação

A *Criteria Query API* consegue lidar com paginação (retornar um número fixo de objetos). Utiliza-se, essencialmente, dois métodos da classe Criteria:

setFirstResult() - seta a primeira linha do resultado

setMaxResult() - seta a quantidade de linhas a retornar

Um exemplo simples:

```
Criteria crit = sess.createCriteria(Person.class);
crit.setFirstResult(2);
crit.setMaxResults(50);
List results = crit.list();
```

18. Restrições (Restrictions)

A API de *Restrictions* é utilizada para se obter objetos que satisfaçam determinadas condições, como por exemplo, objetos do tipo Pessoa cuja idade seja maior do que 20 anos.

Adiciona-se restrições ao objeto Criteria através do método add(). Este método recebe um objeto do tipo *org.hibernate.criterion.Criterion*, que representa uma única restrição. Vale ressaltar que o método add() devolve o próprio objeto Criteria, de modo que é possível realizar *chaining* de restrições.

Segue uma lista com as principais operações estáticas (fábricas de *Criterion*) da classe Restrictions:

```
Restrictions.eq("name", "Shin")
Restrictions.ne("name", "NoName")
Restrictions.like("name", "Sa%")
Restrictions.ilike("name", "sa%")
Restrictions.isNull("name");
Restrictions.gt("price",new Double(30.0))
Restrictions.between("age", new Integer(2), new Integer(10))
```

```
Restrictions.or(criterion1, criterion2)

Restrictions.conjuntion()

Restrictions.disjunction()
```

Um exemplo do uso de disjuntion(), como alternativa a uma árvore de *ORs*:

```
Criteria crit = session.createCriteria(Product.class);

Criterion price = Restrictions.gt("price",new Double(25.0));

Criterion name = Restrictions.like("name","Mou%");

Criterion desc = Restrictions.ilike("description","blocks%");

Disjunction disjunction = Restrictions.disjunction();

disjunction.add(price);

disjunction.add(name);

disjunction.add(desc);

crit.add(disjunction);

List results = crit.list();
```

Um exemplo de *chaining*:

```
// Resgata objetos do tipo Pessoa cujo nome segue um padrão e cuja

// idade é igual a 10 ou null


List people = sess.createCriteria(Person.class)

        .add(Restrictions.like("name", "Shin%"))

        .add(Restrictions.or(

                Restrictions.eq("age", new Integer(10)),

                Restrictions.isNull("age")))

        .list();
```

19. Ordenação de resultados

A API fornece mecanismos de ordenação de resultados. Basta utilizar o método addOrder(..) da classe Criteria e passar um *org.hibernate.criterion.Order* como parâmetro. Objetos Order são criados através dos *factory methods* Order.asc() e Order.desc().

Um exemplo simples de uso de ordenação:

```
List cats = sess.createCriteria(Cat.class)
.add( Restrictions.like("name", "F%")
.addOrder( Order.asc("name") )
.addOrder( Order.desc("age") )
.setMaxResults(50)
.list();
```

20. Associações

É possível especificar restrições em entidades relacionadas utilizando-se *chaining* de Criterias com a operação createCriteria(..)

Um exemplo simples:

```
//Obtém os fornecedores que vendem produtos com preço acima de 25.0

Criteria crit = session.createCriteria(Supplier.class);
Criteria prdCrit = crit.createCriteria("products");
prdCrit.add(Restrictions.gt("price",new Double(25.0)));
List results = crit.list();
```

21. Projeções e Funções de Agregação

A API fornece mecanismos para utilização de projeções e funções de agregação.

As principais funções de agregação são:

```
rowCount()
avg(String propertyName)
count(String propertyName)
```

```
countDistinct(String propertyName)

max(String propertyName)

min(String propertyName)

sum(String propertyName)
```

O exemplo mais simples de projeções é a funcionalidade de contagem de linhas:

```
Criteria crit = session.createCriteria(Product.class);

crit.setProjection(Projections.rowCount());

List results = crit.list();
```

Neste caso, a lista de resultados terá apenas um único objeto, um *Integer* que contém o resultado do *statement* COUNT do SQL.

Alternativamente, pode-se aplicar várias funções de agregação para um mesmo objeto criteria. Segue um exemplo:

```
Criteria crit = session.createCriteria(Product.class);

ProjectionList projList = Projections.projectionList();

projList.add(Projections.max("price"));

projList.add(Projections.min("price"));

projList.add(Projections.avg("price"));

projList.add(Projections.countDistinct("description"));

crit.setProjection(projList);

List results = crit.list();
```

Neste caso, também será retornada uma lista com apenas um único objeto do tipo Object[ ]. O array de objetos contem todos os resultados (valores) em ordem.

22. Query by Example

A funcionalidade de *Query by Example* provê um estilo alternativo de busca de objetos num repositório. Basicamente, cria-se uma instância de um objeto e popula-se seus atributos conforme critérios de busca. Tal objeto é chamado de Exemplo.

A classe *org.hibernate.criterion.Example* implementa a interface *Criterion*, de modo que você pode usá-la como qualquer outra restrição. Utiliza-se a operação estática Example.createExample(...) para criar um exemplo.

Um exemplo simples:

```
// Recupera objetos do tipo Pessoa cujo primeiro nome seja “Maria”


Criteria crit = sess.createCriteria(Person.class);

Person person = new Person();

person.setFirstName("Maria");

Example exampleRestriction = Example.create(person);

crit.add(exampleRestriction);

List results = crit.list();
```

23. Conclusão

Embora a *Criteria Query API* seja bastante simples de ser usada e possua todas as vantagens elucidadas acima*,* o uso de HQL trás algumas vantagens:

- Queries externalizadas podem ser auditadas e otimizadas pelo DBA

- Named queries são fáceis de guardar em cache

- Named queries em arquivos de mapeamento concentram as queries em apenas um lugar

Vale ressaltar, por outro lado, que é possível utilizar ambos mecanismos em um mesmo projeto. Mais detalhes sobre a *Hibernate Criteria API* podem ser vistos em [4], que é a documentação oficial do Hibernate.

24. Bibliografia

[1] – Aggregate functions in SQL

http://databases.about.com/od/sql/l/aaaggregate1.htm - Acessado em 10/04/2009

[2] – Hibernate Querying 102: The Criteria API

http://www.javalobby.org/articles/hibernatequery102/?source=archives – Acessado em 10/04/2009

[3] – Hibernate Criteria Query API

www.javapassion.com/j2ee/**hibernatecriteria**.pdf - Acessado em 10/04/2009

[4] – Hibernate - Relational Persistence for Idiomatic Java

www.**hibernate**.org/hib_docs/v3/**reference**/en/**pdf**/**hibernate_reference**.**pdf** - Acessado em 10/04/2009

[5] – SCBCD 5.0 Study Guide

http://java.boot.by/scbcd5-guide/