

Desenvolvimento de Software para Persistência

JPA - Java Persistence API

Prof. Regis Pires Magalhães
regismagalhaes@ufc.br

Problema

POO – Programação Orientada a Objetos

x

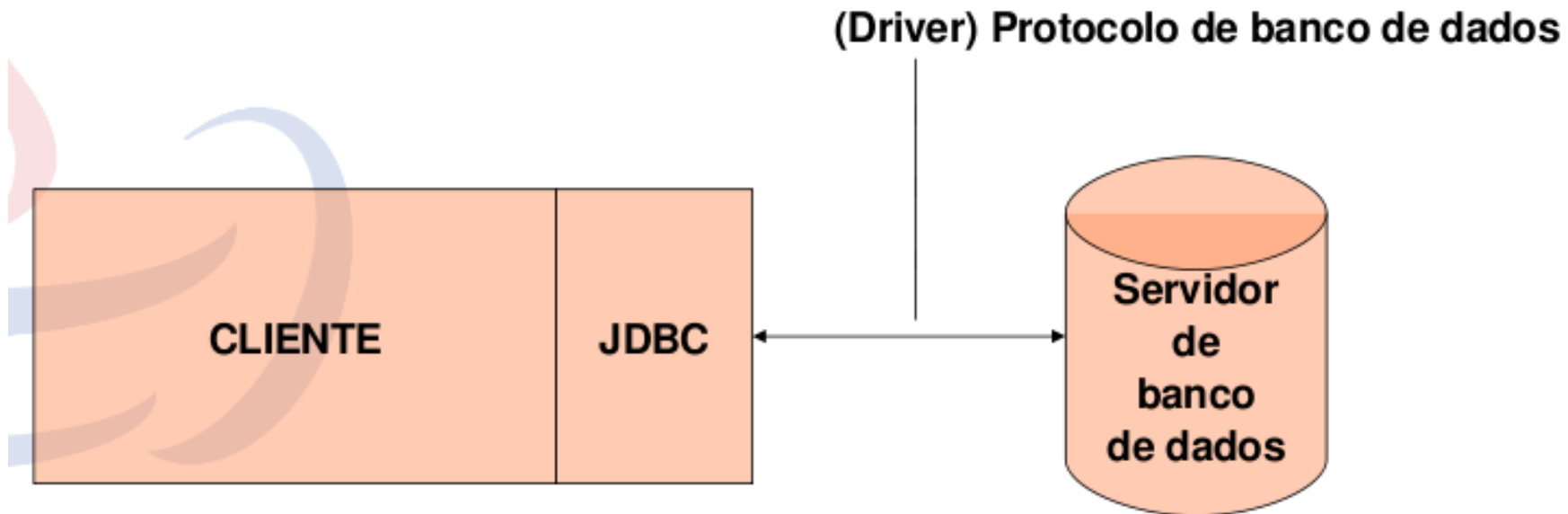
BDR – Banco de Dados Relacionais

=

Paradigmas Diferentes

JDBC

- Trabalha no mesmo nível do banco de dados.



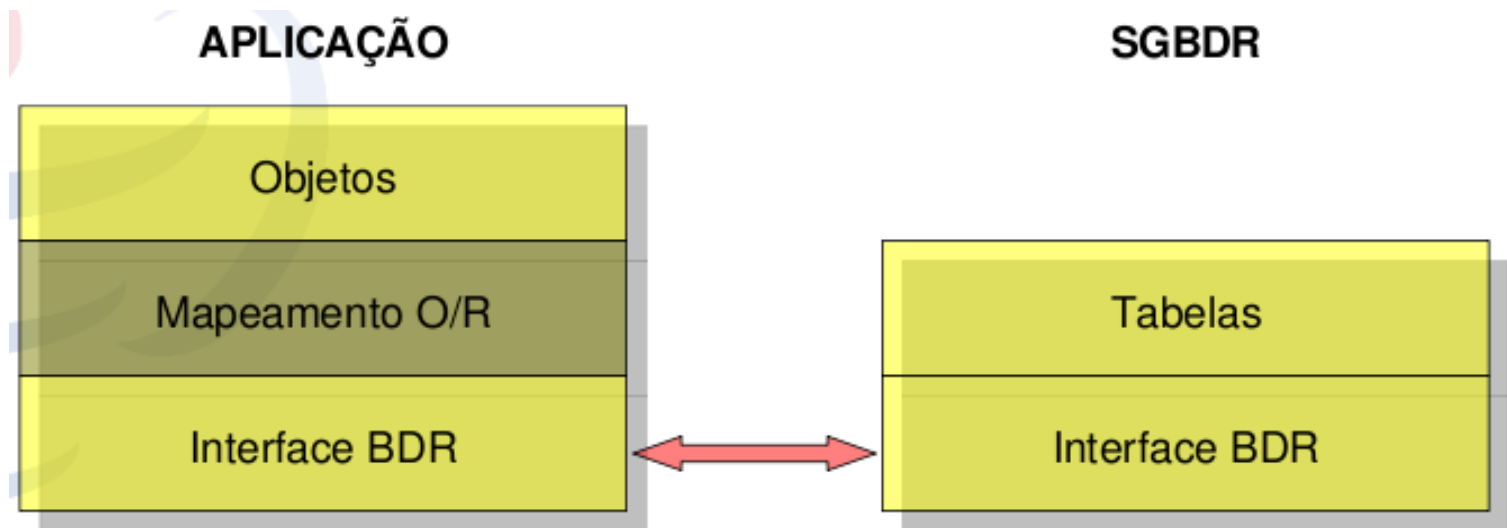
Horstmann, Cay S., Cornell, Gary. Corejava 2 – Volume II – Recursos avançados, pág.181. São Paulo: Makron Books, 2001.

MOR - Mapeamento Objeto Relacional

- Também chamado de ORM
 - Object Relational Mapping
- Resolve o famoso problema de “descompasso de impedância” (impedance mismatch) entre os mundos objeto e relacional.
- Pode ser usado para gerar as tabelas no banco de dados ou atualizar as restrições de integridade.

MOR - Mapeamento Objeto Relacional

- O melhor dos dois mundos:
 - Modelo Orientado a Objetos;
 - Performance e confiabilidade dos bancos de dados relacionais.



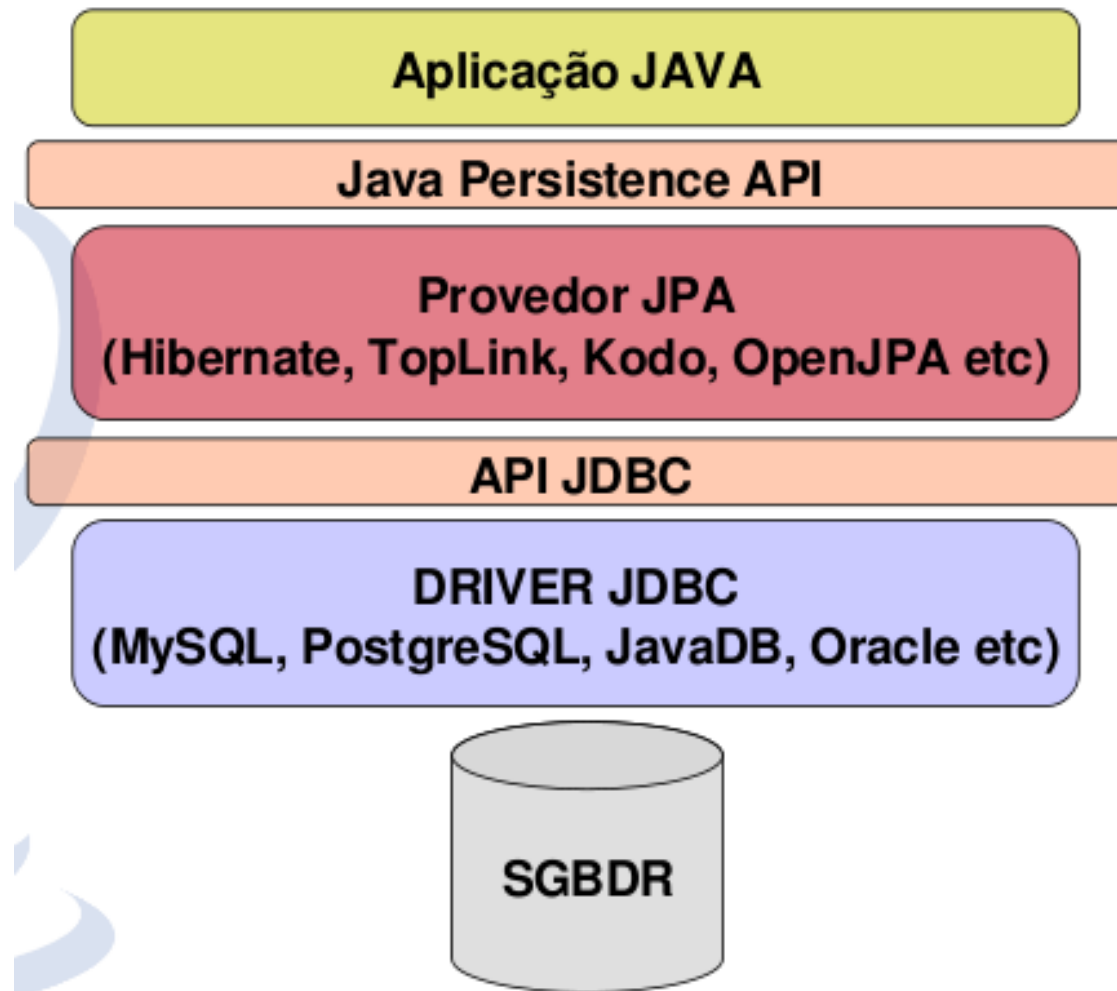
JPA - Características

- Portabilidade
- Persistência transparente
 - Não há necessidade de implementação de interfaces especiais ou classes base. Usa POJOs – Plain Old Java Objects.
 - Atributos são persistidos.
 - Não há necessidade de geração de código.
 - Suporta tipos definidos pelo usuário
 - Controle transparente de restrições de integridade
- Consulta orientada a objetos ou não
 - Query by Criteria
 - JPQL
 - Consultas dinâmicas e nomeadas
 - SQL Nativo

JPA

- Definida na JSR-220 (Java Specification Requests) que especificou o EJB 3.0 (Enterprise JavaBeans 3.0)
 - Padroniza o mapeamento Objeto-Relacional em Java.
- Solução completa para MOR e persistência de dados:
 - Modo declarativo de descrever mapeamento O/R;
 - Linguagem de consulta (JPQL);
 - Ferramentas para manipular entidades.

JPA



Implementações de JPA

- Algumas implementações de JPA:
 - Hibernate
 - Oracle TopLink
 - É a implementação de referência usada pela Sun
 - Mais enxuto e menor que o Hibernate: apenas 2 jars são necessários
 - EclipseLink
 - Apache OpenJPA
 - Kodo
 - EJB + JDO



BEA Kodo™ 4.1

Hibernate

- Poderoso framework de mapeamento objeto-relacional (MOR);
- Framework de Persistência;
- Excelente performance;
- É a mais popular ferramenta de MOR da atualidade;
- Software Livre e Gratuito;
- Permite o uso de uma grande variedade de SGBDs;
- Não é um padrão de indústria.

Hibernate

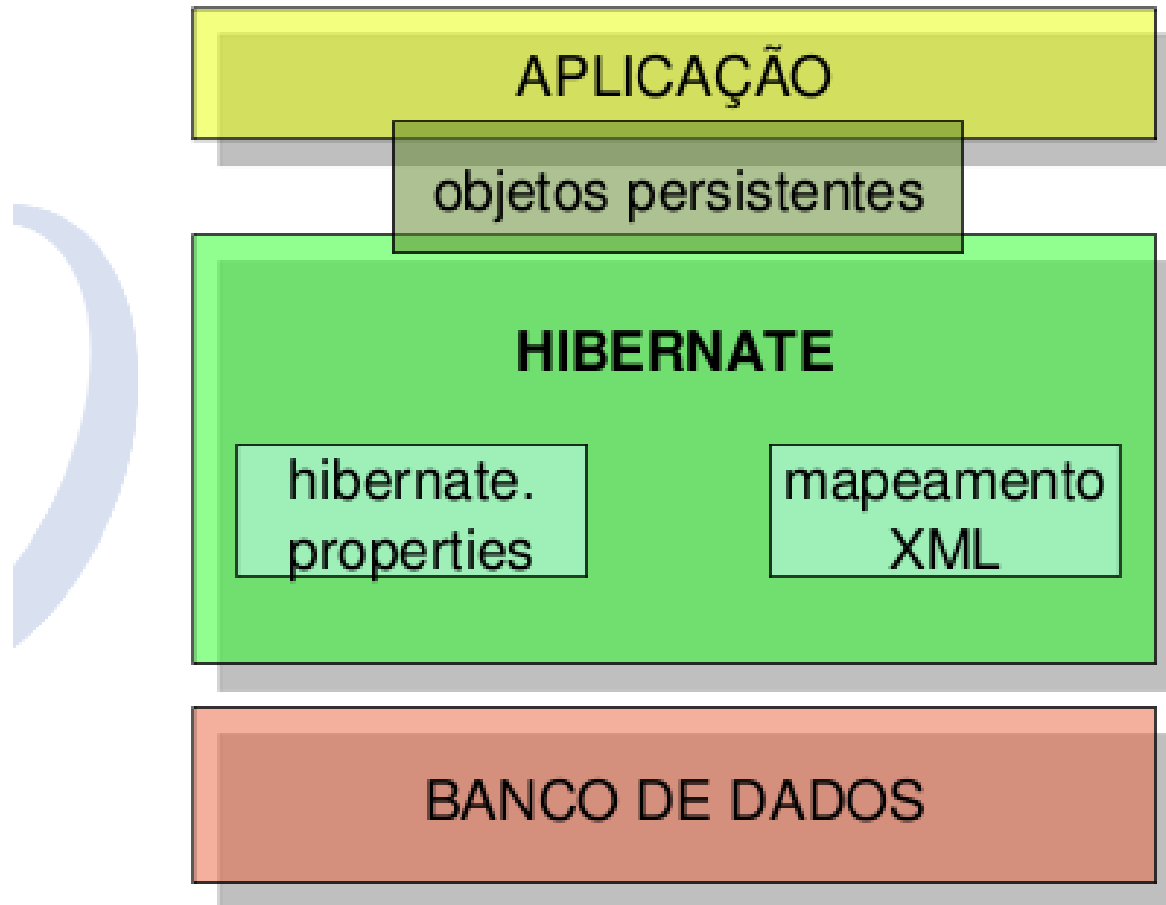
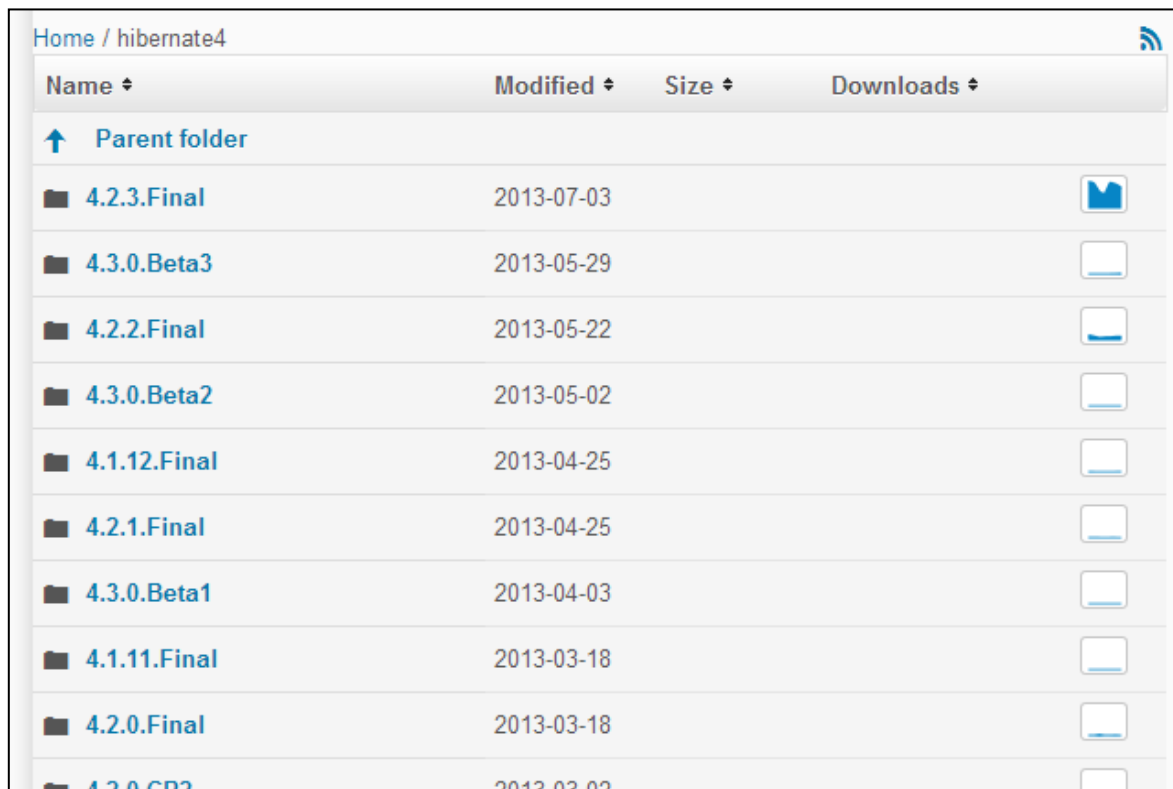


Figura: Boaglio, Fernando. Revista SQL Magazine, ed. 17, p. 40.

Hibernate

- Onde baixar?

- <http://sourceforge.net/projects/hibernate/files/hibernate4/>



The screenshot shows a web interface for downloading Hibernate 4 files. It features a table with columns for Name, Modified date, Size, and Downloads. The files listed are various versions of the Hibernate 4 source code, including 4.2.3.Final, 4.3.0.Beta3, 4.2.2.Final, 4.3.0.Beta2, 4.1.12.Final, 4.2.1.Final, 4.3.0.Beta1, 4.1.11.Final, 4.2.0.Final, and 4.2.0.CR2. Each row includes a folder icon, the version name, the modification date, and a download icon.

Name	Modified	Size	Downloads
Parent folder			
4.2.3.Final	2013-07-03		
4.3.0.Beta3	2013-05-29		
4.2.2.Final	2013-05-22		
4.3.0.Beta2	2013-05-02		
4.1.12.Final	2013-04-25		
4.2.1.Final	2013-04-25		
4.3.0.Beta1	2013-04-03		
4.1.11.Final	2013-03-18		
4.2.0.Final	2013-03-18		
4.2.0.CR2	2013-03-02		

Hibernate - O que usar

- lib/required
 - antlr-2.7.7.jar
 - dom4j-1.6.1.jar
 - hibernate-commons-annotations-4.0.2.Final.jar
 - hibernate-core-4.2.3.Final.jar
 - hibernate-jpa-2.0-api-1.0.1.Final.jar
 - javassist-3.15.0-GA.jar
 - jboss-logging-3.1.0.GA.jar
 - jboss-transaction-api_1.1_spec-1.0.1.Final.jar
- lib/jpa
 - hibernate-entitymanager-4.2.3.Final.jar

Hibernate - O que usar

- Driver JDBC do SGBD
 - Exemplos:
 - **PostgreSQL**: postgresql-9.2-1002.jdbc4.jar
 - **MySQL**: mysql-connector-java-5.1.20.jar
 - **H2**: h2-1.3.166.jar

Hibernate/JPA - Complementos

- ◆ lib/optional

- ◆ Connection Pool

- c3po

- proxool

- ◆ Cache de Segundo Nível

- ehcache

- infinispan

Hibernate x JPA

- JPA provê um subconjunto das funcionalidades do Hibernate.
- Então, porque usar JPA?
 - ◆ Padrão de indústria
 - ◆ Independência de implementação
 - ◆ Não há necessidade de se listar todas as classes anotadas em um arquivo de configuração.
 - ◆ ...

JPA - Principais Conceitos

- EntityManager
 - responsável por praticamente todas operações de persistência de objetos
- PersistenceContext
 - área de memória que mantém os objetos que estão sendo manipulados pelo EntityManager
- Provedores:
 - A especificação é implementada por diversos fabricantes.
- Persistence Unit
 - Configuração do provedor JPA para localizar o banco de dados e estabelecer conexões JDBC.

Persistence Unit

- src/META-INF/persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
  xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">

  <persistence-unit name="dev" transaction-type="RESOURCE_LOCAL">
    <properties>
      <property name="javax.persistence.jdbc.url"
        value="jdbc:postgresql://localhost/aula-jpa" />
      <property name="javax.persistence.jdbc.driver"
        value="org.postgresql.Driver" />
      <property name="javax.persistence.jdbc.user"
        value="postgres" />
      <property name="javax.persistence.jdbc.password"
        value="postgres" />

      <property name="hibernate.hbm2ddl.auto" value="update" />
      <property name="hibernate.show_sql" value="true" />
      <property name="hibernate.format_sql" value="true" />
    </properties>
  </persistence-unit>
</persistence>
```

JPA - Mapeamento

- Classes e interfaces estão no pacote:
 - ◆ `javax.persistence`
- Uso anotações – @
- Configuração por exceção

JPA - Mapeamento via anotações

@Entity

```
public class Contato {  
  
    public Contato() {  
        super();  
    }  
  
    public Contato(String nome, String fone) {  
        super();  
        this.nome = nome;  
        this.fone = fone;  
    }  
  
}
```

Funciona bem no:

- MySQL
- PostgreSQL
- H2
- ...



@Id

@GeneratedValue(strategy=GenerationType.IDENTITY)

```
private int id;  
  
private String nome;  
  
private String fone;  
  
// Getters e Setters  
}
```

Inserindo dados via JPA

```
public class InsereDados {  
  
    public static void main(String[] args) {  
        EntityManagerFactory emf =  
            Persistence.createEntityManagerFactory("dev");  
        EntityManager em = emf.createEntityManager();  
        EntityTransaction tx = em.getTransaction();  
        try {  
            tx.begin();  
            em.persist(new Contato("João", "(88) 3452-4567"));  
            em.persist(new Contato("Maria", "(88) 3452-4568"));  
            em.persist(new Contato("José", "(88) 3452-4569"));  
            tx.commit();  
        } catch (Exception e) {  
            tx.rollback();  
            e.printStackTrace();  
        } finally {  
            em.close();  
        }  
    }  
}
```

Formas de Consultas

- Há basicamente 3 formas de realizar consultas:
 - ◆ JPQL (Java Persistence QL)
 - Várias melhorias em relação à EJBQL.
 - ◆ Criteria Query
 - ◆ Nativas (SQL)
- As consultas JPQL ou Nativas podem ser:
 - ◆ Dinâmicas
 - ◆ Nomeadas (NamedQuery)

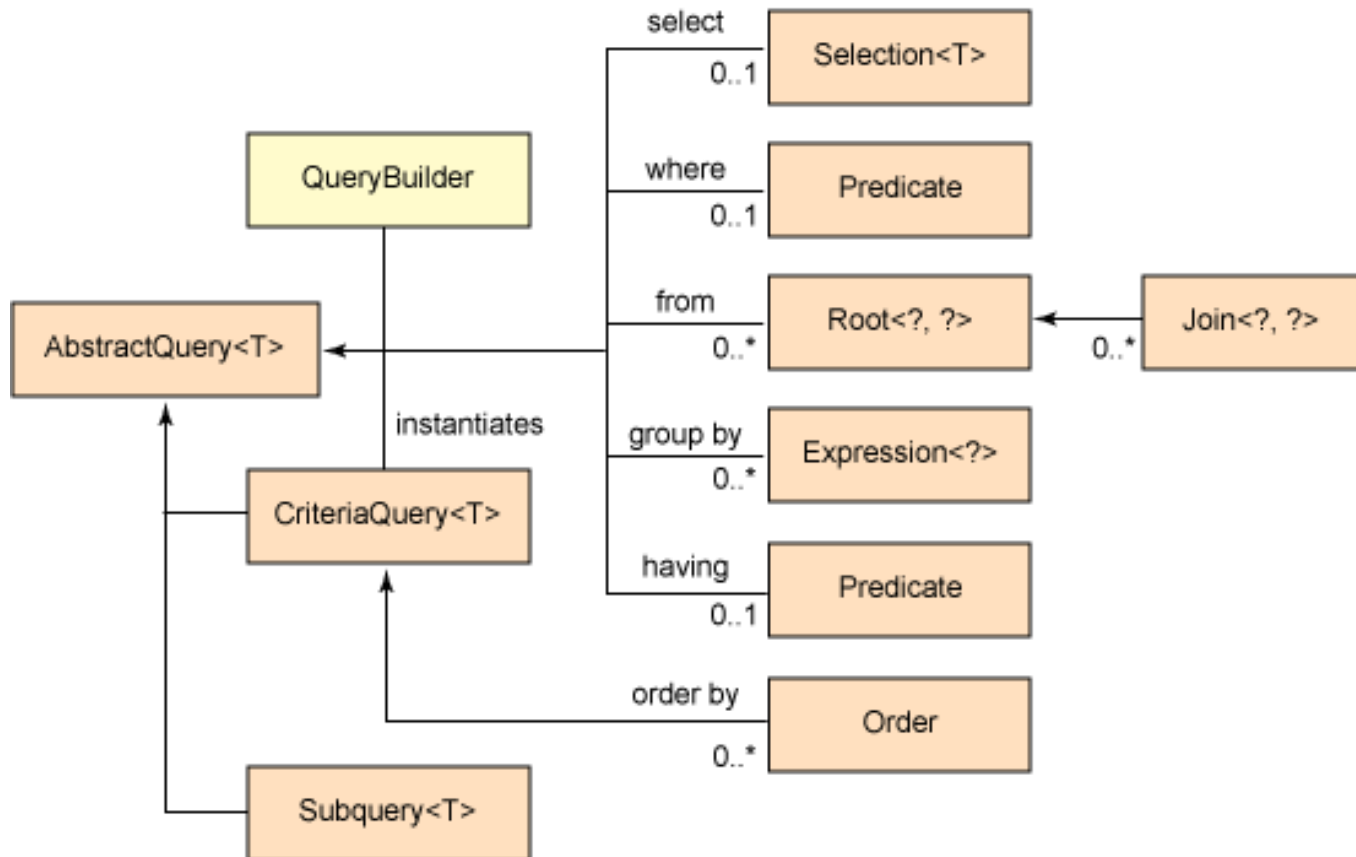
Consultando dados via JPQL

```
public class Principal {  
  
    public static void main(String[] args) {  
        EntityManagerFactory emf =  
            Persistence.createEntityManagerFactory("dev");  
        EntityManager em = emf.createEntityManager();  
  
        List<Contato> l = em.createQuery("from Contato",  
                                         Contato.class).getResultList();  
  
        for (Contato c : l) {  
            System.out.println("Nome: " + c.getNome());  
        }  
  
        em.close();  
    }  
}
```

Consultando dados via Criteria Query

```
public class Principal2 {  
  
    public static void main(String[] args) {  
        EntityManagerFactory emf =  
            Persistence.createEntityManagerFactory("dev");  
        EntityManager em = emf.createEntityManager();  
  
        CriteriaBuilder cb = em.getCriteriaBuilder();  
        CriteriaQuery<Contato> cq = cb.createQuery(Contato.class);  
        cq.from(Contato.class);  
        List<Contato> l = em.createQuery(cq).getResultList();  
  
        for (Contato c : l) {  
            System.out.println("Nome: " + c.getNome());  
        }  
  
        em.close();  
    }  
}
```


Criteria encapsula as cláusulas de uma consulta tradicional



Generic type arguments

<T> : CriteriaQuery <T> selects instances of type <X>

Definições de Named Queries

```
@Entity
@NamedQueries ({
    @NamedQuery (name="Contato.findContato",
                  query="from Contato")
})
public class Contato {

    public Contato() {
        super();
    }

    public Contato(String nome, String fone) {
        super();
        this.nome = nome;
        this.fone = fone;
    }

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

Consultando dados via Named Query

```
public class Principal3 {  
  
    public static void main(String[] args) {  
        EntityManagerFactory emf =  
            Persistence.createEntityManagerFactory("dev");  
        EntityManager em = emf.createEntityManager();  
  
        List<Contato> l = em.createNamedQuery("Contato.findContato",  
                                             Contato.class).getResultList();  
  
        for (Contato c : l) {  
            System.out.println("Nome: " + c.getNome());  
        }  
  
        em.close();  
    }  
}
```

Native Query

```
public class NativeQuery1 {  
  
    public static void main(String[] args) {  
        EntityManager em = Persistence.createEntityManagerFactory("dev-h2")  
            .createEntityManager();  
        Query query = em.createNativeQuery("SELECT NOME, SOBRENOME, IDADE  
FROM PESSOA");  
        List<Object[]> list = query.getResultList();  
        for (Object[] obj : list) {  
            System.out.println("Nome: " + obj[0]);  
            System.out.println("Sobrenome: " + obj[1]);  
            System.out.println("Idade: " + obj[2]);  
        }  
        em.close();  
    }  
}
```

Native Query

...

```
Query q = em.createNativeQuery("SELECT a.firstname, a.lastname FROM Author a");  
List<Object[]> authors = q.getResultList();
```

```
for (Object[] a : authors) {  
    System.out.println("Author "  
        + a[0]  
        + " "  
        + a[1]);  
}
```

...

Native Query

...

```
Query q = em.createNativeQuery("SELECT a.id, a.version, a.firstname, a.lastname FROM  
Author a", Author.class);
```

```
List<Author> authors = q.getResultList();
```

```
for (Author a : authors) {  
    System.out.println("Author "  
        + a.getFirstName()  
        + " "  
        + a.getLastName());  
}
```

...

JPQL ou Criteria?

- Qualquer consulta definida com JPQL pode ser definida com Criteria e vice-versa.
- Algumas consultas são mais facilmente definidas em JPQL, enquanto outras são mais facilmente definidas em Criteria.
- Consultas sem filtros ou com **filtros fixos** são mais facilmente definidas com **JPQL**.
- Consultas com **filtros variáveis** são mais facilmente definidas com **Criteria**.
- A tipagem usada em **Criteria** é verificada em tempo de compilação (**tipagem segura**).
- **Criteria** geralmente possui **melhor desempenho** comparado com consultas dinâmicas usando JPQL.
- A API de **Criteria** é relativamente **complexa** e o código não fica muito legível.
- **Named queries** são **checadas** quando o contexto JPA é iniciado.
 - Consultas **checadas** durante a **inicialização** da aplicação (+ segurança, inicialização + lenta).

Centralizando a criação e fechamento do EntityManager

- Até agora, cada classe que usava JPA especificava um PersistenceUnit e criava um EntityManager.
- Toda vez que era necessário alterar o persistence unit, tínhamos que alterar cada classe.
- Uma melhoria que pode ser realizada é a centralização da definição do PersistenceUnit a ser usado.

ThreadLocal

- Em aplicações WEB toda requisição gera uma nova Thread no servidor.
- Muitas vezes é necessário usar o EntityManager em várias partes da aplicação.
- Para facilitar essa tarefa, pode-se associar o EntityManager à Thread da requisição.
- ThreadLocal é uma funcionalidade do Java que permite armazenar um objeto em um Map em que a chave é a Thread em uso e o valor, o objeto.
 - O método set associa o objeto à Thread atual.
 - O método get obtém o objeto que está associado à Thread atual.

ThreadLocal

```
import javax.persistence.*;

public class JPAUtil {
    private static final EntityManagerFactory emf = Persistence.createEntityManagerFactory("dev-h2");

    private static ThreadLocal<EntityManager> ems = new ThreadLocal<EntityManager>();

    /**
     * Obtém o EntityManager vinculado à Thread atual. Se não existir, é criado e vinculado à Thread atual.
     */
    public static EntityManager getEntityManager() {
        EntityManager em = ems.get();
        if (em == null) {
            em = emf.createEntityManager();
            ems.set(em);
        }
        return em;
    }

    /**
     * Fecha o EntityManager atrelado à Thread atual e retira-o da ThreadLocal.
     */
    public static void closeEntityManager() {
        EntityManager em = ems.get();
        if (em != null) {
            EntityManagerTransaction tx = em.getTransaction();
            if (tx.isActive()) {
                tx.commit();
            }
            em.close();
            ems.set(null);
        }
    }
    ...
}
```

ThreadLocal

...

```
public static void beginTransaction() {  
    getEntityManager().getTransaction().begin();  
}
```

```
public static void commit() {  
    EntityTransaction tx = getEntityManager().getTransaction();  
    if (tx.isActive()) {  
        tx.commit();  
    }  
}
```

```
public static void rollback() {  
    EntityTransaction tx = getEntityManager().getTransaction();  
    if (tx.isActive()) {  
        tx.rollback();  
    }  
}  
}
```

InsererDados com JPAUtil

```
public class InsererDados {  
  
    public static void main(String[] args) {  
        EntityManager em = JPAUtil.getEntityManager();  
        EntityTransaction tx = em.getTransaction();  
        try {  
            tx.begin();  
            em.persist(new Contato("João", "(88) 3452-4567"));  
            em.persist(new Contato("Maria", "(88) 3452-4568"));  
            em.persist(new Contato("José", "(88) 3452-4569"));  
            tx.commit();  
        } catch (Exception e) {  
            tx.rollback();  
            e.printStackTrace();  
        } finally {  
            JPAUtil.closeEntityManager();  
        }  
    }  
}
```

JPQL com JPAUtil

```
public class PrincipalJPQL {  
  
    public static void main(String[] args) {  
        EntityManager em = JPAUtil.getEntityManager();  
  
        // Consulta usando JPQL  
        List<Contato> l = em.createQuery("from Contato",  
                                         Contato.class).getResultList();  
        for (Contato c : l) {  
            System.out.println("Nome: " + c.getNome());  
        }  
  
        JPAUtil.closeEntityManager();  
    }  
}
```

Criteria com JPAUtil

```
public class PrincipalCriteria {

    public static void main(String[] args) {
        EntityManager em = JPAUtil.getEntityManager();

        // Consulta usando Criteria Query
        CriteriaBuilder cb = em.getCriteriaBuilder();
        CriteriaQuery<Contato> cq = cb.createQuery(Contato.class);
        cq.from(Contato.class);
        List<Contato> l = em.createQuery(cq).getResultList();

        for (Contato c : l) {
            System.out.println("Nome: " + c.getNome());
        }
        JPAUtil.closeEntityManager();
    }
}
```

NamedQuery com JPAUtil

```
public class PrincipalNamedQuery {  
  
    public static void main(String[] args) {  
        EntityManager em = JPAUtil.getEntityManager();  
  
        // Consulta usando Named Query  
        List<Contato> l = em.createNamedQuery("Contato.findContato",  
                                             Contato.class).getResultList();  
        for (Contato c : l) {  
            System.out.println("Nome: " + c.getNome());  
        }  
  
        JPAUtil.closeEntityManager();  
    }  
}
```

Padrão de projeto: DAO

- DAO – Data Access Object.
- Fornece uma interface padrão de acesso a dados, independente da tecnologia utilizada.
- Permite que uma entidade que solicita a manipulação de dados os utilize mesmo sem ter noção sobre a tecnologia que permite o armazenamento persistente.
- Consiste em:
 - Uma interface que abstraia a tecnologia de armazenamento.
 - Usualmente o nome da interface possui o sufixo DAO.
 - Classe(s) concreta(s) que implementa(m) a interface DAO.

Padrão de projeto: DAO

```
package br.ufc.dsweb.dao;

import java.util.List;

public interface GenericDAO<T> {
    public void save(T entity);
    public void delete(T entity);
    public T find(Object id);
    public List<T> find();
    public void beginTransaction();
    public void commit();
    public void rollback();
    public void close();
}
```

Padrão de projeto: DAO

```
public abstract class GenericJPADAO<T> implements GenericDAO<T> {

    protected Class<T> persistentClass;

    public void save(T entity) {
        getEm().merge(entity);
    }

    public void delete(T entity) {
        getEm().remove(getEm().merge(entity));
    }

    public T find(Object id) {
        return getEm().find(persistentClass, id);
    }

    public List<T> find() {
        CriteriaQuery<T> cq = getEm().getCriteriaBuilder().createQuery(persistentClass);
        cq.from(persistentClass);
        return getEm().createQuery(cq).getResultList();
    }

    ...
}
```

Padrão de projeto: DAO

```
public abstract class GenericJPADAO<T> implements GenericDAO<T> {  
    ...  
    public EntityManager getEm() {  
        return JPAUtil.getEntityManager();  
    }  
  
    public void beginTransaction() {  
        JPAUtil.beginTransaction();  
    }  
  
    public void commit() {  
        JPAUtil.commit();  
    }  
  
    public void rollback() {  
        JPAUtil.rollback();  
    }  
  
    public void close() {  
        JPAUtil.closeEntityManager();  
    }  
}
```

Padrão de projeto: DAO

```
public interface ContatoDAO extends GenericDAO<Contato> {  
  
}
```

```
public class ContatoJPADAO extends GenericJPADAO<Contato>  
    implements ContatoDAO {  
  
    public ContatoJPADAO() {  
        this.persistentClass = Contato.class;  
    }  
  
}
```

Padrão de projeto: DAO

Acrescentando novos métodos ao DAO específico.

Inserir Dados com DAO

```
public class InserirDados {  
  
    public static void main(String[] args) {  
        ContatoDAO contatoDAO = new ContatoJPDAO();  
        try {  
            contatoDAO.beginTransaction();  
            contatoDAO.save(new Contato("João", "(88) 3452-4567"));  
            contatoDAO.save(new Contato("Maria", "(88) 3452-4568"));  
            contatoDAO.save(new Contato("José", "(88) 3452-4569"));  
            contatoDAO.commit();  
        } catch (Exception e) {  
            contatoDAO.rollback();  
            e.printStackTrace();  
        } finally {  
            contatoDAO.close();  
        }  
    }  
}
```

Consulta dados com DAO

```
public class Principal {  
  
    public static void main(String[] args) {  
        ContatoDAO contatoDAO = new ContatoJPADA0();  
  
        List<Contato> l = contatoDAO.find();  
  
        for (Contato c : l) {  
            System.out.println("Nome: " + c.getNome());  
        }  
        contatoDAO.close();  
    }  
}
```

Mapeamento via Anotações

```
import javax.persistence.*;

@Entity
@Table(name = "MESSAGES")
public class Message {

    @Id
    @GeneratedValue
    @Column(name = "MESSAGE_ID")
    private Long id;

    @Column(name = "MESSAGE_TEXT")
    private String text;

    public Message(String text) {
        this.text = text;
    }

    // Getters and Setters...
}
```


Anotações comuns

→ @Entity

→ @Table

- ◆ Por padrão, a JPA assume que todos os campos persistentes de uma entidade serão armazenados em uma tabela com o mesmo nome da entidade.

- ◆ @Table(name="FUN")

→ @Column

- ◆ Por padrão, a JPA assume que o nome de cada atributo corresponde ao mesmo nome na tabela

- ◆ @Column(name="FUN_ID")

→ @Id

Anotações Comuns

→ @GeneratedValue

- ◆ geração automática de identificadores

→ @Temporal

- ◆ para informações relacionadas ao tempo (DATE, TIME e TIMESTAMP)

→ @OneToOne

`@OneToOne(optional=false)`

`@JoinColumn(name="ADDRESS_ID",
unique=true, nullable=false)`

`Query Address getAddress() { ... }`

Geração de Chave Primária

→ AUTO

- ◆ @Id @GeneratedValue(strategy=GenerationType.AUTO)

→ IDENTITY

- ◆ @Id @GeneratedValue(strategy=GenerationType.IDENTITY)

→ SEQUENCE

- ◆ @Id @GeneratedValue(strategy=GenerationType.SEQUENCE)

→ TABLE

- ◆ @Id @GeneratedValue(strategy=GenerationType.TABLE)

Geração de Chave Primária

→ As opções para geração de chave primária são estas:

◆ AUTO

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

◆ IDENTITY

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

◆ SEQUENCE

```
@Entity  
@SequenceGenerator(name="SEQUENCE", sequenceName="person_id_seq")  
public class Person {  
    @Id @GeneratedValue(strategy=GenerationType.SEQUENCE,  
                        generator="SEQUENCE")  
    private Integer id;  
    ...  
}
```

Geração de Chave Primária

ID_GEN

◆ TABLE

@Entity

```
public class Inventory implements Serializable {
```

@Id

```
@GeneratedValue(generator="InvTab")
```

```
@TableGenerator(name="InvTab", table="ID_GEN",  
    pkColumnName="ID_NAME", valueColumnName="ID_VAL",  
    pkColumnValue="INV_GEN")
```

```
private long id;
```

ID_NAME	ID_VAL
INV_GEN	<last generated value >

◆ **NONE** – é o padrão. De nada for dito, não haverá geração automática de chave primária. A geração passa a ser responsabilidade da aplicação.

→ A estratégia mais portátil entre os diferentes bancos de dados é **TABLE**.

Hibernate

→ Comparação de termos com o JPA:

- ◆ Hibernate → JPA
- ◆ SessionFactory → EntityManagerFactory
- ◆ Session → EntityManager
- ◆ Query → Query
- ◆ Transaction → EntityTransaction

Hibernate

→ Propriedades:

◆ `hibernate.hbm2ddl.auto`

- `create` – apaga as tabelas mapeadas e depois as recria.
- `create-drop` – cria as tabelas mapeadas e ao fechar o `EntityManager`, remove-as.
- `update` – usa `SchemaUpdate` para evolução da base de dados de acordo com o que está mapeado nas classes.
- `validate` – usa `SchemaValidate` para comparar o que está no banco com o que está mapeado.
- `none` (default)

Parâmetros de Consultas

- Os parâmetros de consultas podem ser nomeados ou numerados.

```
@NamedQuery(name="findEmployeesAboveSal",  
            query="SELECT e " +  
                  "FROM Employee e " +  
                  "WHERE e.department = :dept AND " +  
                  "      e.salary > :sal")
```


Parâmetros Nomeados

```
@NamedQuery (name="findEmployeesAboveSal",  
              query="SELECT e " +  
                    "FROM Employee e " +  
                    "WHERE e.department = :dept AND " +  
                    "      e.salary > :sal")
```

```
public class EmployeeServiceBean implements EmployeeService {  
    @PersistenceContext (unitName="EmployeeService")  
    EntityManager em;  
    public List findEmployeesAboveSal (Department dept, long minSal) {  
        return em.createNamedQuery ("findEmployeesAboveSal")  
            .setParameter ("dept", dept)  
            .setParameter ("sal", minSal)  
            .getResultList ();  
    }  
    // ...  
}
```

Parâmetros Numerados

```
public class EmployeeServiceBean implements EmployeeService {
    @PersistenceContext(unitName="EmployeeService")
    EntityManager em;
    public List findEmployeesHiredDuringPeriod(Date start, Date end) {
        return em.createQuery("SELECT e " +
                               "FROM Employee e " +
                               "WHERE e.startDate BETWEEN ?1 AND ?2")
                .setParameter(1, start, TemporalType.DATE)
                .setParameter(2, end, TemporalType.DATE)
                .getResultList();
    }
    // ...
}
```

Consultas

```
public class ProdutoDAO ...  
    . . .  
  
    public Produto findByName(final String nome) {  
        Produto temp = null;  
        try {  
            temp = (Produto) em.createQuery(  
                "SELECT p FROM Produto p WHERE p.nome LIKE :nome")  
                .setParameter("nome", nome + "%").getSingleResult();  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
        return temp;  
    }  
    . . .  
}
```

Consultas Nomeadas

```
@Entity
@Table(name = "produtos")
@NamedQueries( {
    @NamedQuery(
        name = "Produto.findByDescricao",
        query = "SELECT p FROM Produto p WHERE p.descricao = "
            + ":descricao"),
    @NamedQuery(
        name = "Produto.findByPreco",
        query = "SELECT p FROM Produto p WHERE p.preco = ?1")
} )
public class Produto implements Serializable {
    . . . // Implementação do Entity Bean.
}
```

Consultas Nomeadas

```
public class ProdutoDAO ...  
.  
.  
.  
  
    public List<Produto> findByPreco(final BigDecimal preco) {  
        return em.createNamedQuery("Produto.findByPreco")  
            .setParameter(1, preco).getResultList();  
    }  
  
.  
.  
.  
}
```

Paginação em Consultas

- Métodos `setFirstResult()` e `setMaxResults()`

```
List l = em.createQuery("select c from Cliente c")  
    .setFirstResult(100) .setMaxResults(10)  
    .getResultList();
```

JPA 2

- JPA 2 tem suporte a validadores nos beans.
 - Parecido com o Hibernate Validador.

```
public class Aluno {  
    @NotNull  
    private String nome;  
  
    @Length(max = 10)  
    @NotNull  
    public String endereco
```

Obtenção de EntityManager

→ Container-managed

- ◆ O container abre e fecha o entity manager. Além disso, ainda fica responsável pelos limites das transações.
- ◆ Obtenção via injeção de dependência ou JNDI.
 - Via injeção de dependência:

```
@PersistenceContext
```

```
private EntityManager em;
```

- Via JNDI:

```
EntityManager em = (EntityManager)  
    ctx.lookup("java:comp/env/accounting");
```

→ Application-managed

- ◆ EntityManager gerenciado via código.

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory(  
    "jpa-hibernate-derby");
```

```
EntityManager em = emf.createEntityManager();
```


Mapeamento de membros não persistidos

- Membros que não são persistidos devem ser anotados como transientes.

```
@Transient
```

```
private int idade;
```

Associações

→ As anotações usadas para definir os tipos de associações são essas:

- ◆ @OneToOne
- ◆ @ManyToOne
- ◆ @OneToMany
- ◆ @ManyToMany

Associações one-to-one

```
@Entity
public class Customer implements Serializable {
    ...
    @OneToOne
    @JoinColumn(name="CUSTREC_ID", unique=true,
                nullable=false, updatable=false)
    public CustomerRecord getCustomerRecord() {
        return customerRecord;
    }
    ...
}
```

```
@Entity
public class CustomerRecord implements Serializable {
    ...
    @OneToOne(mappedBy="customerRecord")
    public Customer getCustomer() {
        return customer;
    }
    ...
}
```

Associações one-to-one

- Em associações one-to-one bidirecionais, usamos o atributo **mappedBy** para explicitar o lado não proprietário (inverso) da associação.
- O lado proprietário da associação em relacionamentos **one-to-one** e **one-to-many** é o lado que possui a chave estrangeira (foreign key).

Associações unidirecionais e bidirecionais

- Associação unidirecional
 - Possui somente o lado proprietário da associação.
- Associação bidirecional
 - Possui um lado proprietário e um lado não proprietário (inverso) da associação.
 - O lado não proprietário (inverso) da associação deve fazer referência ao lado proprietário através do uso do elemento `mappedBy` da anotação `OneToOne`, `OneToMany` ou `ManyToMany`.
 - `mappedBy` define o nome do atributo do lado proprietário da associação.
- Lado proprietário da associação
 - Determina as atualizações ao relacionamento no banco de dados.

Associações one-to-many e many-to-one

```
@Entity
public class Customer implements Serializable {
    ...
    @OneToMany(cascade=ALL, mappedBy="customer")
    public Set<Order> getOrders() {
        return orders;
    }
    ...
}
```

```
@Entity
public class Order implements Serializable {
    ...
    @ManyToOne
    @JoinColumn(name="CUST_ID", nullable=false)
    public Customer getCustomer() {
        return customer;
    }
    ...
}
```

Associações one-to-many e many-to-one

```
@Entity
public class Company {
    @OneToMany(fetch = FetchType.LAZY, mappedBy = "company")
    private List<Branch> branches;
}
```

```
@Entity
public class Branch {

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "companyId")
    private Company company;
}
```

Associações one-to-many e many-to-one

- Em caso de associações bidirecionais, em uma classe temos um atributo **OneToMany** e na outra classe (a que detém a chave estrangeira) temos um atributo **ManyToOne**.
- No lado **OneToMany** de uma associação bidirecional, usamos o atributo **mappedBy** para explicitar que o relacionamento (chave estrangeira) fica no lado oposto (inverso) da associação.
- O lado **ManyToOne** é o lado proprietário e, portanto, não pode ter o elemento **mappedBy**.

Associações many-to-many

```
@Entity
public class Customer implements Serializable {
    ...
    @ManyToMany
    @JoinTable(
        name="CUST_PHONE",
        joinColumns=
            @JoinColumn(name="CUST_ID", referencedColumnName="ID"),
        inverseJoinColumns=
            @JoinColumn(name="PHONE_ID", referencedColumnName="ID")
    )
    public Set<PhoneNumber> getPhones() {
        return phones;
    }
    ...
}
```

Associações many-to-many

- Em associações many-to-many, qualquer lado pode ser escolhido para ser o lado proprietário da associação.

```
@Entity
public class PhoneNumber implements Serializable {
    ...
    @ManyToMany(mappedBy="phones")
    public Set<Customer> getCustomers() {
        return customers;
    }
    ...
}
```

Associações many-to-many

- Se for necessário criar atributos na tabela de junção, deve-se criar uma nova classe com os novos atributos.

Associações many-to-many

```
class User {
    @OneToMany(mappedBy = "user")
    private Set<UserService> userServices = new HashSet<UserService>();
}

class UserService {
    @ManyToOne @JoinColumn(name = "user_id")
    private User user;

    @ManyToOne @JoinColumn(name = "service_code")
    private Service service;

    @Column(name = "blocked")
    private boolean blocked;
}

class Service {
    @OneToMany(mappedBy = "service")
    private Set<UserService> userServices = new HashSet<UserService>();
}
```

Cascadeamento

→ Cascadeamento pode ser usado para propagar operações para classes associadas.

→ Tipos:

- ◆ **ALL** – Todas as operações possíveis são cascadeadas para as classes associadas.
- ◆ **MERGE** – Se a entidade fonte estiver “merged”, o merge é cascadeado para as classes associadas.
- ◆ **PERSIST** – Se a entidade fonte for persistida, a persistência é cascadeada para as classes associadas.
- ◆ **REFRESH** – Se a entidade fonte for atualizada, a atualização é cascadeada para as classes associadas.
- ◆ **REMOVE** – Se a entidade fonte for removida, as classes associadas também serão removidas.

→ Exemplo:

```
@OneToMany (cascade={ CascadeType . REMOVE } )  
private List<Pessoa> dependentes ;
```

Carregamento de Associações

- As associações podem ser carregadas de duas formas:
 - ◆ **EAGER** – A associação é carregada juntamente com a classe base.
 - `@OneToMany(fetch=FetchType.EAGER)`
 - ◆ **LAZY** – A associação somente é carregada quando for usada.
 - `@OneToMany(fetch=FetchType.LAZY)`
- Em associações **ManyToOne** e **OneToOne**, **EAGER** é padrão.
- Em associações **OneToMany** e **ManyToMany**, **LAZY** é padrão.
 - ◆ As coleções só são inicializadas quando houver necessidade.
 - ◆ Apenas quando algum método que acesse o conteúdo da coleção for executado é que ela será preenchida.

Carregamento de Associações

→ Definição de um atributo comum como LAZY:

```
@Entity
public class Book implements Serializable {
    ...
    @Basic(fetch=LAZY)
    protected String toc;
    ...
}
```

Carregamento de Associações

- Relacionamentos LAZY podem poupar recursos, principalmente quando há muitos modelos na aplicação.
- No entanto, requerem cuidados como manter a sessão aberta.

Modo de Carregamento

- **@Fetch** é uma anotação do **Hibernate** usada no relacionamento que informa como criar o comando SQL.
- A anotação recebe a enumeração **FetchMode** (**SELECT**, **JOIN** ou **SUBSELECT**).
- **FetchMode.SELECT** – realiza selects separados para buscar os dados.
- **FetchMode.JOIN** – cria um único comando SQL que busca todas as informações de uma única vez através de cláusulas join.
- **FetchMode.SUBSELECT** – executa subconsultas, se o banco tem suporte para essa funcionalidade.

Herança

- JPA disponibiliza 3 estratégias para o uso de herança:
 - ◆ **SINGLE_TABLE** – É a opção default. Há uma única tabela para toda a hierarquia de classes.
 - Há uma coluna na tabela para determinar a classe (*DiscriminatorColumn*).
 - ◆ **JOINED** – Define que há uma tabela para cada classe da hierarquia de classes.
 - Atributos de uma superclasse são persistidos na tabela da superclasse.
 - Assim, para obter um objeto de uma subclasse, é necessário fazer um join envolvendo todas as tabelas das superclasses. **Isso pode ser bastante oneroso para o banco de dados e comprometer a performance da aplicação, especialmente quando a hierarquia é complexa e composta de muitas classes.**
 - Há uma coluna na tabela da classe base para determinar a classe (*DiscriminatorColumn*)

Herança

- `TABLE_PER_CLASS` – Semelhante à estratégia `JOINED`, mas cada tabela possui os campos dos atributos herdados e os campos dos seus próprios atributos (não herdados).

Herança - SINGLE_TABLE

PROJECT

ID	TYPE	DESCRIPTION	BUDGET
1	P	small project	
2	L	large project	1000000

```
@Entity
@Table(name="PROJECT")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="TYPE",
    discriminatorType=DiscriminatorType.STRING,length=20)
@DiscriminatorValue("P")
public class Project {
    @Id protected BigInteger id;
    protected String description;
    ...
}
```

```
@Entity
@DiscriminatorValue("L")
public class LargeProject extends Project {
    protected BigInteger budget;
    ...
}
```

Herança - JOINED

PROJECT

ID	TYPE	DESCRIPTION
1	P	small project
2	L	large project

L_PROJECT

ID	BUDGET
2	1000000

```
@Entity
@Table(name="PROJECT")
@Inheritance(strategy=InheritanceType.JOINED)
@DiscriminatorColumn(name="TYPE",
discriminatorType=DiscriminatorType.STRING,length=20)
@DiscriminatorValue("P")
public class Project {
    @Id protected BigInteger id;
    protected String description;
    ...
}
```

```
@Entity
@Table(name="L_PROJECT")
@DiscriminatorValue("L")
public class LargeProject extends Project {
    protected BigInteger budget;
    ...
}
```

Herança - MappedSuperclass

- É possível determinar uma superclasse da qual uma entidade herda atributos persistentes através da anotação **@MappedSuperclass**.
- Recurso interessante quando várias classes compartilham atributos persistentes.
- Pode-se usar as anotações **@AttributeOverride** ou **@AssociationOverride** na subclasse para sobrescrever a configuração da superclasse.

Herança - MappedSuperclass

```
@MappedSuperclass
public class Employee {
    @Id
    protected Integer empId;

    @Version
    protected Integer version;

    @ManyToOne
    @JoinColumn(name="ADDR")
    protected Address address;
    ...
}
```

```
@Entity
@AttributeOverride(name="address", column=@Column(name="ADDR_ID"))
public class PartTimeEmployee extends Employee {
    @Column(name="WAGE")
    protected Float hourlyWage;
    ...
}
```

Composição - Classes Embutidas

- JPA permite o mapeamento de objetos embutidos.
- Cada atributo do objeto embutido é mapeado para a tabela da entidade base.
- Isso permite trabalhar com mais de uma classe, usando composição e armazenando tudo em uma única tabela.

Composição - Classes Embutidas

```
@Embeddable
// Não persistente
public class Intervalo {
    private int inicio;
    private int fim;
    ...
}
```

```
@Embedded
@AttributeOverrides({
    @AttributeOverride(name="inicio",
        column=@Column(name="VALOR_INICIAL")),
    @AttributeOverride(name="fim",
        column=@Column(name="VALOR_FINAL"))
})
Intervalo getIntervalo() { ... }
```

Connection Pool

- Conexão é um recurso caro demais para ficar criando e fechando o tempo todo.
- Um pool é uma reserva de objetos prontos para serem usados, em vez de demandar a criação de novos objetos sempre que precisarmos.
- O Hibernate tem integração nativa com o pool c3po.

Connection Pool

```
<persistence...>
  <persistence-unit name="default" transaction-type="RESOURCE_LOCAL">
    ...
    <properties>
      <property name="javax.persistence.jdbc.url" value="..." />
      <property name="javax.persistence.jdbc.driver"
        value="..." />
      <property name="javax.persistence.jdbc.user" value="..." />
      <property name="javax.persistence.jdbc.password"
        value="..." />
      <property name="hibernate.c3p0.min_size" value="5" />
      <property name="hibernate.c3p0.max_size" value="20" />
      <property name="hibernate.c3p0.timeout" value="300" />
      <property name="hibernate.c3p0.max_statements" value="50" />
      ...
    </properties>
  </persistence-unit>
</persistence>
```

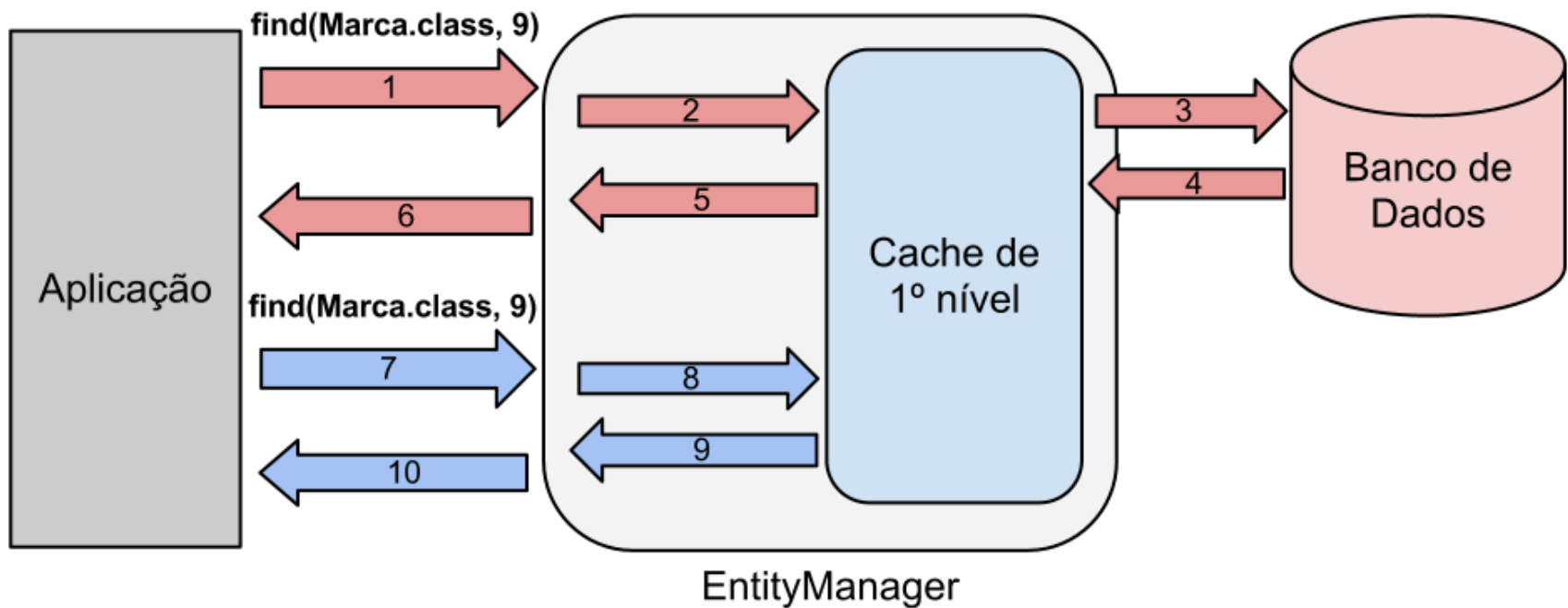
Cache

- JPA possui dois níveis de cache que são usados para diminuir os acessos ao banco de dados:
 - Cache de primeiro nível.
 - Cache de segunda nível.
- O cache de primeiro nível é usado automaticamente no contexto JPA.
- Cada objeto que é carregado pelo EntityManager acaba ficando no cache desse EntityManager.
- A partir de então, toda vez que o mesmo objeto for buscado pela chave, ele será devolvido imediatamente, sem necessidade de um novo acesso ao banco para isso.

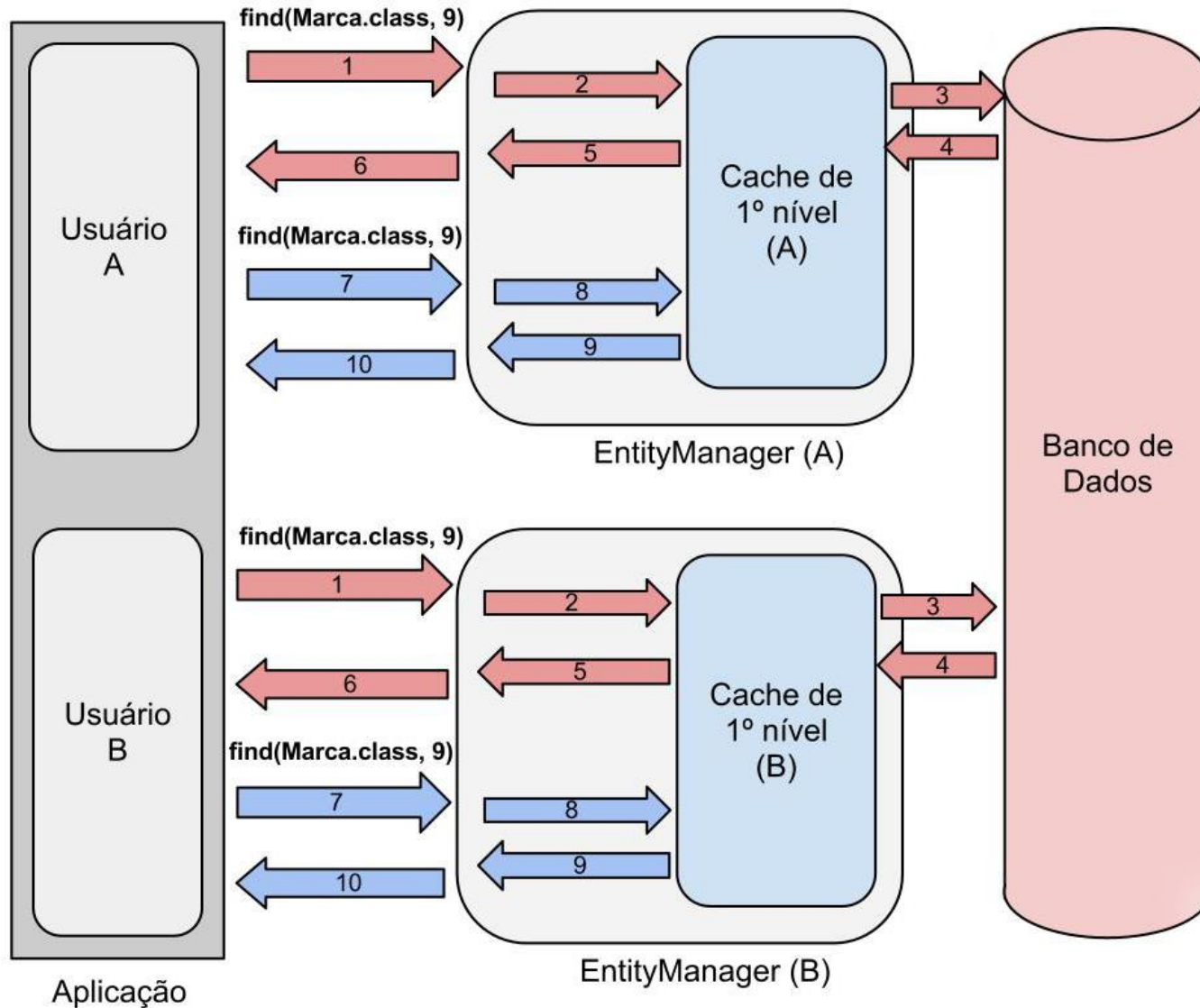
Cache de primeiro nível

- Ao invés de realizar atualizações a cada comando dentro de uma transação, as atualizações são feitas somente ao final da transação.
 - Exemplo: Se um objeto é modificado várias vezes dentro de uma transação, somente um único comando SQL UPDATE é gerado ao final da transação, contendo todas as modificações.

Cache de primeiro nível



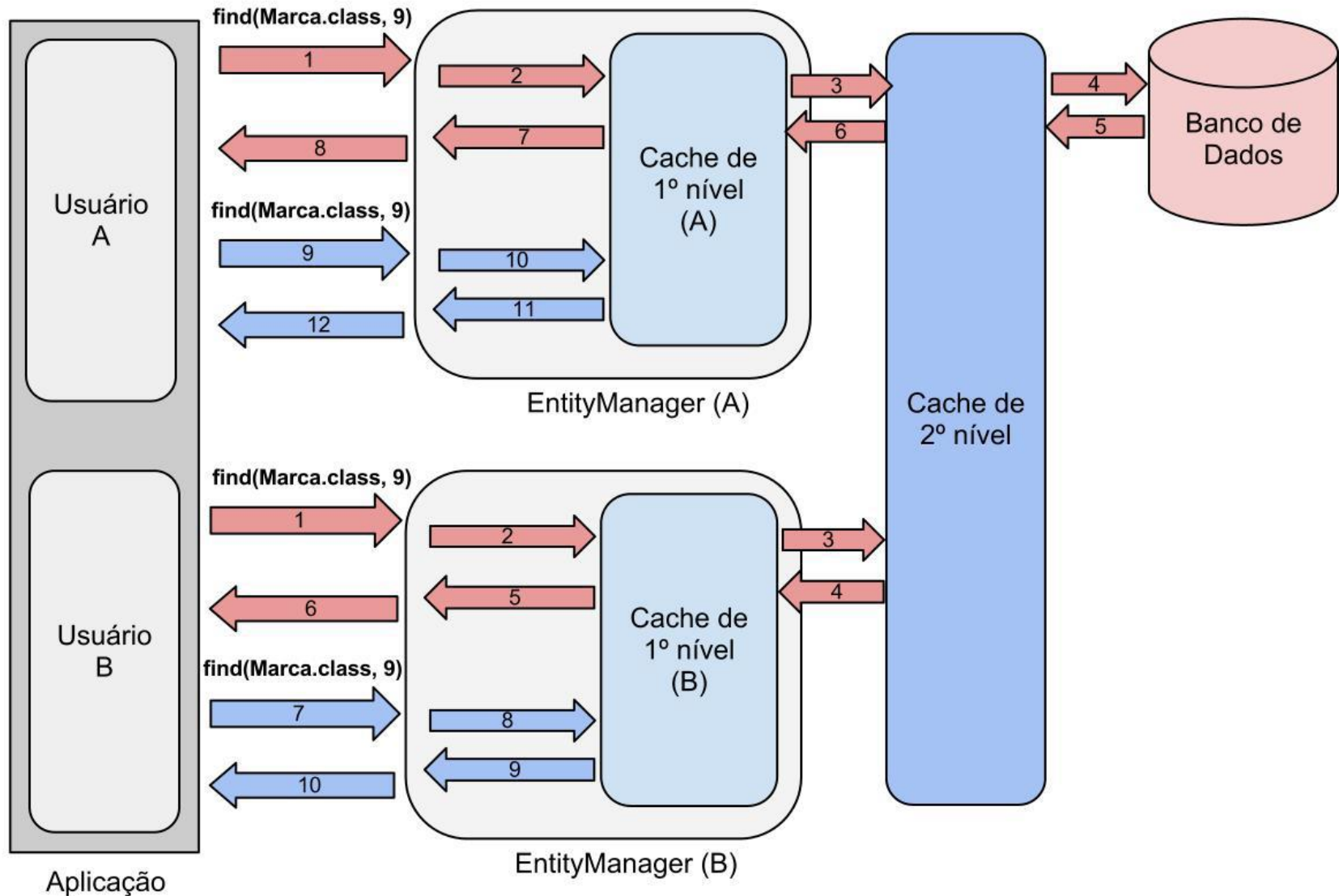
Cache de primeiro nível



Cache de segundo nível

- Mantém os objetos em memória e os utiliza mesmo entre diferentes EntityManagers de uma mesma EntityManagerFactory.
 - Não é usado por default, devendo ser explicitamente configurado para que possa ser usado.
 - Deve-se configurar que entidades vão usá-lo, ou seja, quais vão ficar no cache.

Cache de segundo nível



Cache de segundo nível

- JPA 2.0 tem suporte nativo ao cache de segundo nível e ele é configurado através do elemento `shared-cache-mode` do `persistence.xml`.
- Os modos diferentes de cache são:
 - **ALL**: automaticamente habilita todas as entidades;
 - **NONE**: desabilita para todas as entidades;

Cache de segundo nível

- **ENABLE SELECTIVE**: habilita para todas as entidades que estiverem anotadas com `@Cacheable`.
- **DISABLE_SELECTIVE**: habilita para todas as entidades, desabilitando somente as que estiverem anotadas com `@Cacheable(false)`;
- **UNSPECIFIED**: valor padrão assumido pelo `shared-cache-mode`. Cada implementação de JPA tem liberdade para definir qual modo será habilitado.

Cache de segundo nível

- As implementações mais de cache mais usadas são: EHCache e Infinispan.
- Nelas podemos configurar para cada entidade o numero máximo de objetos que ficarão em memória, e depois o que será armazenado em disco, etc.
- Usaremos o módulo hibernate-ehcache.
 - EHCache (Easy Hibernate Cache)
 - Rápido, leve, fácil de usar.
 - Suporta cache read_only e read_write.
 - Suporta cache em memória e disco.

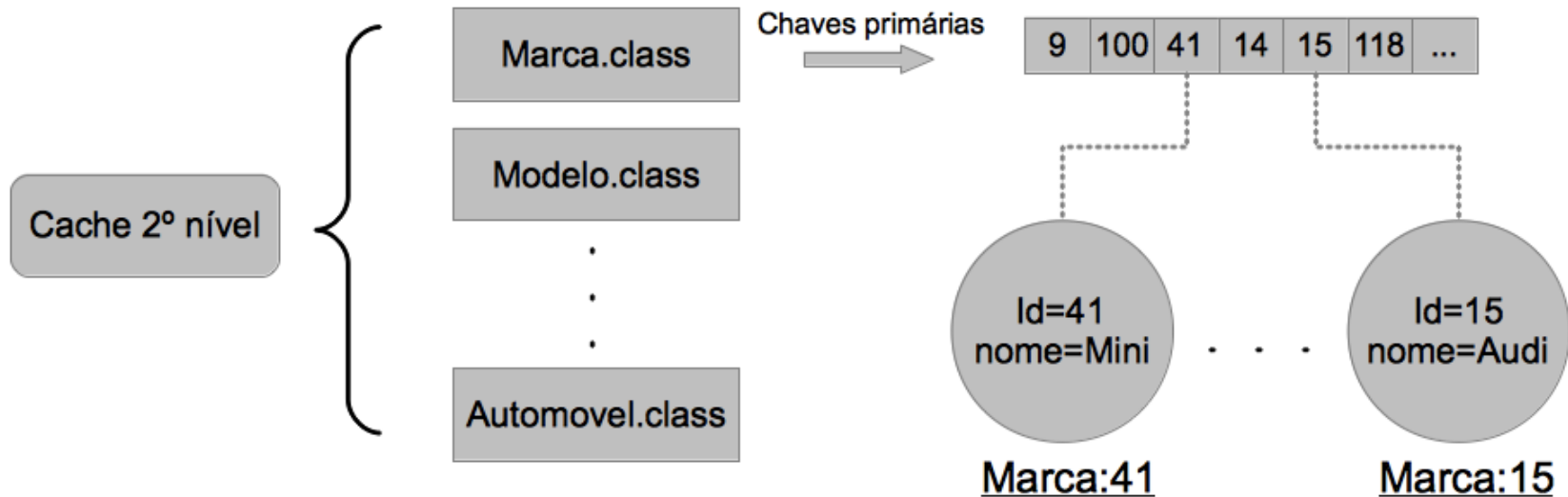
Cache de segundo nível

```
<persistence ...>
  <persistence-unit name="default"
    transaction-type="RESOURCE_LOCAL">
    <shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>
    <properties>
      <property name="hibernate.cache.region.factory_class"
        value=
          "org.hibernate.cache.ehcache.EhCacheRegionFactory"/>
      ...
    </properties>
  </persistence-unit>
</persistence>
```

```
@Entity @Cacheable
public class Marca {
  ...
}
```

Cache de segundo nível

- Recupera objetos de qualquer entidade, desde que baseados na chave primária.



Cache de consultas

- Quando não buscamos pela chave precisamos de um cache diferente: o cache de consultas.
- Não faz parte da JPA.
- É um recurso específico do Hibernate.
- Só funciona se o cache de segundo nível estiver ativado.
- Somente será possível guardar no cache consultas que retornem objetos que estão com o cache habilitado com a anotação `@Cacheable`.
- Além disso, precisamos habilitar o cache em cada consulta que realizarmos.

Cache de consultas

```
...  
<persistence-unit name="default" transaction-type="RESOURCE_LOCAL">  
  <shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>  
  <properties>  
    ...  
    <property name="hibernate.cache.use_query_cache"  
              value="true" />  
  </properties>  
</persistence-unit>  
...
```


Cache de consultas

```
@NamedQuery(name=Automovel.LISTAR_DESTAQUES,  
            query="select a from Automovel a",  
            hints={  
                @QueryHint(name="org.hibernate.cacheable", value="true"),  
                @QueryHint(name="org.hibernate.cacheRegion",  
                            value=Automovel.LISTAR_DESTAQUES)})  
  
@Entity  
@Cacheable  
public class Automovel {  
    public static final String LISTAR_DESTAQUES =  
        "Automovel.buscarDestques";  
    ...  
}
```

Cache de consultas

```
Query query = JpaUtil.getEntityManager()  
    .createQuery("select a from Automovel a",  
        Automovel.class);  
query.setHint("org.hibernate.cacheable", true);  
query.setHint("org.hibernate.cacheRegion",  
    Automovel.LISTAR_DESTAQUES);  
List<Automovel> automoveis = query.getResultList();
```

Cache de consultas

```
...  
public class JpaUtil {  
...  
    public static Query enableCache(Query query, String region) {  
        query.setHint("org.hibernate.cacheable", true);  
        query.setHint("org.hibernate.cacheRegion", region);  
        return query;  
    }  
}
```

```
Query query = JpaUtil.getEntityManager()  
    .createQuery("select a from Automovel a", Automovel.class);  
JpaUtil.enableCache(query, Automovel.LISTAR_DESTAQUES);  
List<Automovel> automoveis = query.getResultList();
```

Invalidando o cache de consultas

```
public class JpaUtil {  
    ...  
    public static void evictCache(EntityManager em, String region){  
        try {  
            Session session = (Session) em.getDelegate();  
            Cache cache = session.getSessionFactory().getCache();  
            cache.evictQueryRegion(region);  
        }  
        catch(Exception e) {  
            // provavelmente a implementação não é o Hibernate  
        }  
    }  
}
```

Invalidando o cache de consultas

```
@ManagedBean @ViewScoped
public class AutomovelBean {

    public String salvar(Automovel auto) {
        EntityManager em = JpaUtil.getEntityManager();
        em.persist(auto);
        JpaUtil.evictCache(em, Automovel.LISTAR_DESTAQUES);
        return "listar";
    }
}
```

Callbacks

- JPA permite métodos de callback para acessar o EntityManager.
- Eventos disponíveis:
 - PostLoad
 - PrePersist
 - PostPersist
 - PreRemove
 - PostRemove
 - PreUpdate
 - PostUpdate

Método de Callback

```
@PrePersist
void validateCreate() {
    if (idade > 130) {
        throw new IllegalStateException();
    }
}
```

Método Callback em classe Listener

```
@Entity
@EntityListeners (br.ufc.Monitor.class)
public class Pessoa {
    ...
}
```

```
public class Monitor {
    @PostPersist
    public void alertaNovaPessoa (Pessoa p)
    {
        emailRH (p) ;
    }
}
```


Open Entity Manager in View

- Uma questão comum em aplicações Web na camada de visão é o acesso a associações Lazy de objetos desconectados do EntityManager.
- Uma solução simples é deixar o EntityManager aberto mesmo na camada de apresentação e só fechá-lo após o processamento completo da requisição.
- O padrão “Open Entity Manager in View” faz isso.
- Usamos um interceptador que filtra a requisição.
- Nele podemos usar qualquer código antes e após a requisição.
- O fluxo de processamento da requisição fica assim:
 - Início da Requisição → Filtro → Servlet/Action → Filtro → Fim da Requisição.

Open Entity Manager in View

- Para que o filtro funcione no container web, é necessário defini-lo no arquivo web.xml ou através de anotação:

```
<filter>
  <filter-name>JPAFilter</filter-name>
  <filter-class>br.ufc.web.EntityManagerFilter</filter-class>
</filter>

<filter-mapping>
  <filter-name>JPAFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Controle de Transações no Filtro

```
import java.io.IOException;
import javax.servlet.*;

@WebFilter("/*")
public class EntityManagerFilter implements Filter {
    public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain) throws IOException, ServletException {
        try {
            JPAUtil.beginTransaction();
            chain.doFilter(request, response);
            JPAUtil.commit();
        } catch (Throwable e) {
            System.out.println(e.getMessage() + "\n" + e.getCause());
            e.printStackTrace();
            JPAUtil.rollback();
            throw new ServletException(e);
        } finally {
            JPAUtil.closeEntityManager();
        }
    }

    public void destroy() {}
    public void init(FilterConfig arg0) throws ServletException {}
}
```

Referências

- CORDEIRO, Gilliard. Aplicações Java para web com JSF e JPA. Casa do Código.
- Apostila da K19: K22 - Desenvolvimento Web com JSF 2 e JPA 2
 - <http://www.k19.com.br/downloads/apostilas/java/k19-k12-desenvolvimento-web-com-jsf2-e-jpa2>
- Apostila da K19: K22 - Persistência com JPA2 e Hibernate
 - <http://www.k19.com.br/downloads/apostilas/java/k19-k21-persistencia-com-jpa2-e-hibernate>
- Hibernate EntityManager
 - <http://docs.jboss.org/hibernate/orm/4.0/hem/en-US/html/>
- Dynamic, typesafe queries in JPA 2.0
 - **How the Criteria API builds dynamic queries and reduces run-time failures**
 - <http://www.ibm.com/developerworks/java/library/j-typesafejpa/>
- Apresentação – JPA: Persistência padronizada em Java – Flávio Henrique Curte
 - http://www.javanoroeste.com.br/tech_days_votuporanga/palestras/Java_Persistence_API.pdf
- Screencast da Caelum – Primeiros Passos para JPA – Fábio Kung
 - <http://blog.caelum.com.br/2007/05/15/screencast-primeiros-passos-para-a-jpa/>
- Resumo de JPA – Quick Ref Card
 - <http://www.solarmetric.com/resources/ejb-api-quickref.pdf>
- Exemplo de JPA – Java Persistence Example
 - <https://glassfish.dev.java.net/javaee5/persistence/persistence-example.html>

Referências

- Os 7 hábitos dos desenvolvedores Hibernate e JPA altamente eficazes (por Paulo Silveira da Caelum)
 - <http://blog.caelum.com.br/os-7-habitos-dos-desenvolvedores-hibernate-e-jpa-altamente-eficazes/>
- JPA Annotation Reference
 - <http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html>
- JPA Inheritance Strategies
 - http://www.jpox.org/docs/1_2/jpa_orm/inheritance.html

Referências

- Open Session in View (Hibernate)
 - <http://www.hibernate.org/43.html>
- EJB 3.0 Persistence API – Quick Reference Guide
 - <http://www.solarmetric.com/resources/ejb-api-quickref.pdf>
- Hibernate Avançado
 - Paulo Silveira e Guilherme Moreira
 - Revista Mundo Java – N° 19
- Gerenciando a Sessão do Hibernate em Aplicações WEB
 - Nico Steppat e Guilherme Moreira
 - Revista Mundo Java – N° 25

Referências - Recursos

- Hibernate
 - <http://www.hibernate.org/>
- Introduction to Hibernate Caching
 - <http://www.javabeat.net/articles/37-introduction-to-hibernate-caching-1.html>
- Hibernate Caching
 - <http://www.javapassion.com/j2ee/hibernatecaching.pdf>

Referências - Livros

- CORDEIRO, Gilliard. Aplicações Java para web com JSF e JPA. Casa do Código.
- Pro EJB 3: Java Persistence API (Pro) – Mike Keith e Merrick Schincariol. 2006.
- EJB 3 in Action – Debu Panda, Reza Rahman e Derek Lane. 2007.
- Beginning EJB 3 Application Development: From Novice to Professional – Raghu R. Kodali, Jonathan R. Wetherbee e Peter Zadrozny. 2006.
- Java Persistence with Hibernate – Christian Bauer e Gavin King
 - Capítulo 2: <http://www.manning.com/bauer2/>

Referências – Revistas

- Revista Java Magazine
 - Ed. 25 – Persistência Turbinada I: Como usar o JDBC de modo eficiente para criar classes DAO – Osvaldo Pinali Doederlein
 - Ed. 26 – Persistência Turbinada II: Recursos avançados do JDBC – Osvaldo Pinali Doederlein
 - Ed. 39 – Java EE 5: Um Enterprise Edition muito mais fácil – Osvaldo Pinali Doederlein
 - Ed. 39 – Persistência no Java EE 5: Iniciando com a Java Persistence API – André Dantas Rocha e Sérgio Oliveira Kubota
 - Ed. 41 – Java EE 5 na Prática: Criando uma aplicação passo a passo com EJB 3.0, JPA e NetBeans 5.5 – Osvaldo Pinali Doederlein
 - Ed. 42 – Dados e Mapeamento: Explorando técnicas e tecnologias para persistência de dados – Osvaldo Pinali Doederlein