Fundação Educacional do Município de Assis
Instituto Municípal de Ensino Superior de Assis - IMESA

Rodolfo Lourenço Tretel

Abordagem Comparativa das Linguagens de Consultas JPQL e CRITERIA em Ambiente JAVA com uso de Objeto-Relacional (ORM)

ASSIS
2012

Rodolfo Lourenço Tretel

Abordagem Comparativa das Linguagens de Consultas JPQL e CRITERIA em Ambiente JAVA com uso de Objeto-Relacional (ORM)

Trabalho de Conclusão de Curso apresentado ao Instituto Municipal de Ensino de Assis – IMESA e à Fundação Educacional do Município de Assis – FEMA, como requisito do Curso de Bacharelado em Ciências da Computação para obtenção do Certificado de Conclusão.

Orientador: Prof. Fernando Cesar de Lima

Área de Concentração: Linguagem de consulta; ORM; JPQL; CRITERIA.

Assis

2012

Dedicatória

Dedico este trabalho aos meus pais, que sempre apoiaram as minhas decisões e aos professores, que contribuíram para uma formação acadêmica de alto nível.

Resumo

O ORM proporciona uma solução para o problema de incompatibilidade entre a linguagem Java, por ser orientada a objeto, e aos Bancos de Dados de Relacionais, que trabalham com tabelas e colunas, colaborando com que o desenvolvedor foque mais tempo desenvolvendo do que efetuando Querys em SQL, sejam elas simples ou complexas, mantendo o código mais limpo. O Hibernate, como pioneiro na área de framework de ORM, mantém dois tipos de consultas de dados: a Criteria, totalmente voltada a orientação a objeto e a HQL, que é correspondente à JPQL da JPA, a qual é mais voltada para Strings de *Querys*. As linguagens de consultas focam em tirar do desenvolvedor da aplicação a tarefa de ter vasto conhecimento das tabelas as quais está trabalhando, apenas necessitando o conhecimento da configuração dos seus mapeamentos na parte do Java.

Palavras-chave: JPA; CRITERIA; Linguagem de Consulta; Java.

Abstract

The ORM provides a solution to the problem of incompatibility between the Java language, which is object-oriented, and Relational Databases, working with tables and columns, helping the developer focus more time developing than in SQL Queries, be they simple or complex, and keeping the code cleaner. Hibernate, as a pioneer in the field of ORM framework, maintains two types of data queries: Criteria, which is focused totally on object orientation and HQL, which is corresponding to the JPA JPQL, which is more focused on Strings queries. Query languages focus on the application developer to not worry about having extensive knowledge of the tables that are working, and just need the knowledge about the configuration of its mapping in Java.

Keywords: JPA, CRITERIA, Query Language, Java.

Lista de Figuras

Lista de Siglas e Abreviaturas

JPA	*Java Persistence API*
ORM	*Object-relational mapping*
SGDB	Sistema de gerenciamento de banco de dados
POO	Programação Orientada Objeto
SQL	*Structured Query Language*
OO	Orientação a Objeto

Sumário

1 INTRODUÇÃO

Quando se fala em sistemas de informações nos dias atuais é imprescindível falar de como os dados serão armazenados e controlados por suas aplicações, já que os dados estão no maior escopo de um sistema de informação, já que os mesmos iriam ser um sistema reiniciável em que todas as vezes que fosse fechado começaria sempre vazio. Esse trabalho tem a intenção de mostrar modelos de consultas dos dados com conceito de ORM no ambiente de desenvolvimento ECLIPSE com uso da tecnologia JAVA.

Por sua maturidade e tecnologia mais avançadas, grande parte dos sistemas de informações nos dias atuais trabalham com SGBD Relacional, como MYSQL, ORACLE, SQLSERVER, dentre outros.

Como o Java possui um paradigma de programação POO (Programação Orientada Objeto) não podendo ter uma 'Conversação' direta com os SGBD Relacionais, assim surgindo o chamado paradigma da incompatibilidade. Então como tirar os melhores proveitos dessas duas grandes tecnologias com paradigmas tão diferentes? Eis que surge o ORM (OBJECT RELATIONAL MAPPING) que é uma técnica de desenvolvimento em que tabelas de um banco de dados viram classes e suas linhas virão instâncias de sua classe, podendo-se fazer abstração de associações, tipos de dados e herança.

A SUN, detentora da Tecnologia Java, aproveitando a ascensão do HIBERNATE, que era o maior framework ORM da época, criou a especificação JPA (Java Pesistence API) de como os dados na sua plataforma deveriam ser persistidos, padronizando o modo de se mapear os objetos para os Bancos de Dados Relacionais e efetuando consultas abrindo porta para várias outras tecnologias, como TOPLINK, MYECLIPSE e até o HIBERNATE criando uma implementação JPA, onde eles deveriam trabalhar com ORM (Object Relational Mapping).

O HIBERNATE, além de sua implementação da JPA(Java Pesistence API) possui o seu próprio framework que é o mais usado até hoje tendo diferenças distintas de uma

implementação de JPA. Uma das partes que mais se destaca é a CRITERIA, que é uma das suas linguagens de consulta, ainda que o mesmo possua, além dessa, a HQL (Hibernate Query Language), em vez de trabalhar apenas com Querys, a Criteria trabalha inteiramente com as classes e atributos e ainda abre espaço para o uso de Querys caso seja preciso.

Já a JPA faz uso da linguagem JPQL, onde suas consultas ainda são baseadas em Querys/SQL com uma interação com o paradigma de orientação a objeto, em vez de colocar na consulta o nome da tabela de onde iria se abstrair os dados coloca-se o nome da classe.

1.1 OBJETIVO

Construir duas aplicações, uma usando HIBERNATE e outra JPA, com MYSQL como banco de dados, implementar casos de consultas complexas que possam simular como agiria essa linguagem em um ambiente de desenvolvimento real, demonstrar a construção de cada consulta, além de servir em apoio para estudantes e desenvolvedores na hora de escolher a maneira que irão cuidar das consultas em seus projetos/trabalhos .

1.2.1 JUSTIFICATIVA

A justificativa pela escolha desse tema foi pelas várias possibilidades que se tem para trabalhar com os dados na linguagem Java, bem como conseguir definir qual é a mais apropriada e que oferece mais vantagens como agilidade, para o desenvolvedor e empresas que possam optar por essa tecnologia.

1.3 MOTIVAÇÃO

O que levou a escolher esse tema foi a necessidade de adquirir novos conhecimentos na área da persistência em Java para que em futuros projetos e trabalhos possa ter uma base de qual seriam as ferramentas mais ideais para se usar, além de utilizar frameworks com futuros bem promissores como Hibernate.

1.4 ESTRUTURA DO TRABALHO

Este trabalho esta dividido em nove capítulos, sendo o primeiro capítulo está a Introdução. No segundo capítulo será apresentada uma descrição sobre ORM e como funciona essa tecnologia, com demonstração de mapeamentos. No terceiro capítulo é apresentado um breve estudo sobre SGBD e do modelo relacional. No quarto a uma apresentação rápida a documentação JPA estrutura e sobre a linguagem de consulta JPQL. No quinto será feito uma apresentação sobre o framework Hibernate e sobre a linguagem de consulta Criteria. No sexto é apresentada a estrutura do experimento que será construído para rodar os casos de teste, os quais serão demonstrados no sétimo capítulo, levando à conclusão no oitavo capítulo.

1.5 METODOLOGIA

Serão desenvolvidas duas aplicações com uso da plataforma Eclipse e Linguagem Java, usando em uma HIBERNATE/CRITERIA e em outra JPA/JPQL, onde servirão como base

de testes para casos de consultas para serem obtidas as conclusões sobre os resultados deste trabalho.

2 ORM (Mapeamento objeto-relacional)

Segundo Bauer (2005) Mapeamento Objeto Relacional é “persistir de maneira automática e transparente, os objetos de um aplicativo para tabelas em um banco de dados relacional. Em essência, transforma dados de uma representação para a outra”.

A linguagem orientada a objetos tem um paradigma diferente do banco de dados relacional, mas que por suas qualidades e maturidade fazem deles um bom conjunto, gerando um paradigma de incompatibilidade. Usando um framework de ORM, o desenvolvedor não necessita ter conhecimento de SQL apenas precisa saber fazer os arquivos de mapeamentos da aplicação com o banco de dados.

É um padrão de desenvolvimento que permite que se possa trabalhar com objetos e Classes na persistência de dados com o uso de bancos de dados relacionais.

O Mapeamento funciona de modo que cada Classe represente uma tabela do Banco de dados, fazendo com que as linhas de uma tabela se transformem em objetos e as colunas virem atributos.

“Ele faz o mapeamento da sua classe para o anco de dados e cada ORM tem suas particularidades, para gerar o SQL referente a inserção do objeto que corresponde a uma tabela no banco de dados e realizar a operação. Utilizando um ORM, também se ganha produtividade, pois deixa-se de escrever os comando SQL para deixar que o próprio ORM, faça isto por você”
(CADU, 2011)

2.1 ANNOTATION

Sempre tem uso do símbolo “@” como identificador. Utilizam-se os atributos da *annotation* direto na Classe que será mapeada.

```
package Mapeamento;
import java.util.Date;

@Entity
@Table(name="cliente")
public class ClienteBE {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer idcliente;

    @Column(name="nome")
    private String nome;

    @Column(name="sobrenome")
    private String sobrenome;

    @Column(name="data_nascimento")
    private Date data_nascimento;

    @Column(name="telefone")
    private Integer telefone;

    @ManyToOne
    @JoinColumn(name="idrua",referencedColumnName="idrua")
    private RuaBE rua;
```

Figura 1 Exemplo de Mapeamento usando annotation na aplicação JPA

“@Entity” diz que a classe é uma entidade. “@Table” define o nome da tabela do banco que está sendo mapeada. “@Id” marca o atributo que corresponde à *Primary key* e o “@Generated”, a estratégia de população dela. “@Column” fala com qual coluna da tabela o atributo está relacionado e caso o atributo contenha o mesmo nome da coluna, não é necessário o uso dessa notação. “@manyToOne” define o tipo do relacionamento que a classe vai ter com o atributo, podendo também ser ainda “@OneToMany”, “@OneToOne” e “@ManyToMany”. Sendo o OneToMany(Um para muitos) onde um registo numa tabela está relacionado com vários registos numa segunda tabela, mas os registos na segunda tabela estão apenas relacionados com um registo na primeira tabela. O OneToOne(Um para

um) existe quando a chave primária do registro de uma determinada tabela pode ser utilizada uma única vez em um dos registros da outra tabela. E o último ManyToMany(Muitos para muitos) é um tipo de relacionamento que acontece de forma indireta entre duas tabelas, pois para que ele possa ser concebido é necessário a geração de uma terceira tabela.

2.2 XML

Usa um arquivo separado (hbm.xml) para cada classe que será utilizada na aplicação, deixando a Classe do objeto livre de comandos , apenas com seus atributos e gets/sets.

Um exemplo do Mapeamento da Classe ClienteBE usado na aplicação de HIBERNATE de teste é demonstrado abaixo.

```xml
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd"

<hibernate-mapping package="Mapeamento">
<class name="ClienteBE" table="cliente">
    <id name="idcliente" column="idcliente">
        <generator class="native"></generator>
    </id>
    <property name="nome"></property>
    <property name="sobrenome"></property>
    <property name="data_nascimento"></property>
    <property name="telefone"></property>
    <property name="email"></property>
    <property name="rg"></property>


     <many-to-one name="rua" class="Mapeamento.RuaBE"
     fetch="select" column="idrua" not-null="true"></many-to-one>

</class>
</hibernate-mapping >
```

```java
public class ClienteBE {

    private Integer idcliente;
    private String nome;
    private String sobrenome;
    private Date data_nascimento;
    private Integer telefone;
    private RuaBE rua;

    public RuaBE getRua() {
        return rua;
    }

    public void setRua(RuaBE idrua) {
        this.rua = idrua;
    }

    public Integer getIdcliente() {
        return idcliente;
    }

    public void setIdcliente(Integer idcliente) {
        this.idcliente = idcliente;
    }
```

Figura 2 Exemplo Mapeamento XML

```
<class name="ClienteBE" table="cliente">
```

Name mostra o nome da Classe que está sendo Mapeada e tabela do banco que está relacionada.

```
<id name="idcliente" column="idcliente">
<generator class="native"></generator>
```

Name mostra o nome do atributo que será *Primary Key*, *column* sua coluna no banco e *class* a estratégia da *Key*.

```
<property name="nome"></property>
```

Nome das propriedades da classe correspondente com as do banco de dados, se não fosse o mesmo nome teria que ter uma *column* indicando o nome que está no banco.

```
<many-to-one name="rua" class="Mapeamento.RuaBE"
fetch="select" column="idrua" not-null="true"></many-to-one>
```

Mostra o modelo de relacionamento com o atributo “rua”, podendo ser “<many-to-many>”, “<one-to-many>”, “<one-to-one>” ou “<many-to-one>.

3 SGBD Sistema de Gerenciamento de Banco de Dados

“Um SGBD é uma coleção de arquivos e programas inter-relacionados que permitem aos usuários o acesso para consulta e alterações desses dados. O maior benefício de um banco de dados é proporcionar ao usuário uma visão abstrata dos dados. Isto é, o sistema acaba por ocultar determinados detalhes sobre a forma de armazenamento e manutenção desses dados.”

(SILBERSCHATZ, 1999)

O SGBD consiste em uma coleção de dados e de programas, que permite o usuário manipular, definir e controlar os bancos de dados, (KORTH, 1995).

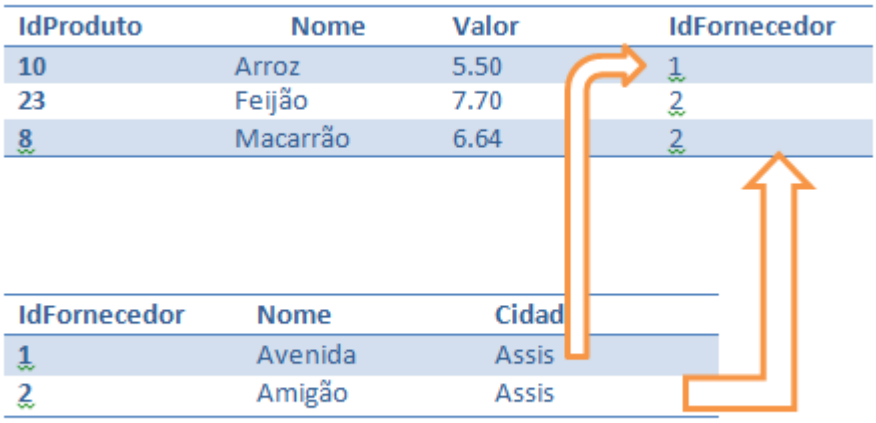
Esses sistemas servem como interface para diferentes tipos de usuários para o mesmo banco de dados e possuem algumas características, como independência dos dados, integridade dos dados, acesso simultâneo dos dados e diferente vista do banco de dados o seu principal objetivo é armazenar grande volume de dados.

Seus principais modelos de dados são hierárquico, rede, relacional dedutivo e objeto. O mais usado e mais avançado até os dias de hoje é o Relacional.

3.1 SGBD Relacional

São softwares que irão controlar o armazenamento, recuperação, exclusão, segurança e a integridade do Banco de Dados. Nesse modelo se trabalha com tabelas, que são organizadas em colunas. Os dados como simples “instancia” de uma tabela são as linhas.

Tipicamente são usada chaves que servem para relacionar ou identificar uma linha na tabela. (NETO, 2011).



IdProduto	Nome	Valor	IdFornecedor
10	Arroz	5.50	1
23	Feijão	7.70	2
8	Macarrão	6.64	2

IdFornecedor	Nome	Cidad
1	Avenida	Assis
2	Amigão	Assis

Figura 3 Modelo Relacionamento

A figura de tabela representa a relação entre as tabelas, de modo que na primeira tabela o produto Arroz tem a como chave estrangeira a chave primaria da avenida e no produto arroz tem referencia a chave primaria do amigão.

3.2 SQL

Structured Query Language é uma linguagem simples de consulta, que fornece as instruções para se modificar a base de dados. Entre seus principais comandos estão:

Exemplos de consulta usando SQL:

Select – Retornar,Consultar;

SELECT **[COLUNAS A SEREM RETORNADAS]** ***FROM*** **[TABELA];**

Insert – Inserir;

INSERT **INTO [TABELA] VALUES [VALORES] ;**

Update – Modificar;

UPDATE **[TABELA] SET [COLUNA=VALOR];**

Delete – excluir;

DELETE **FROM [TABELA];**

Exemplos de condições básicas de consultas, existem muitas situações de consultar com SQL, mas não abordaremos isso aqui.

4 JPA

A JPA(*Java Persistence API)* define a gestão da persistência e mapeamento ORM no ambiente Java. Ela combina os melhores recursos dos mecanismos de persistência já existente, como as APIs Java Database Connectivity (JDBC), Mapeamento Objeto Relacional (ORM) e Java Data Objects (JDO). A especificação JPA define o mapeamento objeto-relacional, internamente, em vez de confiar em implementações específicas do fornecedor de mapeamento, e usa tanto anotações ou XML para mapear objetos em tabelas de banco de dados.

Para o uso da JPA deve ser fornecido um provedor, entre um dos muitos populares nos dias atuais são o HIBERNATE, TOPLINK, esse provedor deve usar dos seguintes elementos para manutenção dos dados.

Entity objects: Uma entidade é uma classe Java simples que representa uma linha em uma tabela de banco de dados. Entidades podem ser classes concretas ou classes abstratas. Eles mantêm estados usando as propriedades ou campos.

EntityManager: Um objeto EntityManager mantém a coleção de objetos de entidade ativa que estão sendo usados pela aplicação. O objeto EntityManager manipula a interação de banco de dados e metadados para o objeto-relacional mapeamentos. Uma instância de um objeto EntityManager representa um contexto de persistência. Uma aplicação em um recipiente pode obter o EntityManager através de injecção na aplicação ou por pesquisá-lo no namespace componente Java. Se a aplicação gere a sua persistência, o EntityManager é obtido a partir do EntityManagerFactory. Os recipientes do servidor de aplicativos normalmente supre essa função, mas o EntityManagerFactory é necessário para usar JPA na aplicação de gestão de persistência.

EntityManagerFactory: A fábrica é usada para criar um EntityManager para interações de banco de dados.

Unidade de persistência: Uma unidade de persistência consiste na metadados declarativos que descreve a relação de objetos de classe de entidade para um banco de dados relacional. O EntityManagerFactory usa esses dados para criar um contexto de persistência que pode ser acessado através do EntityManager.

Persistence: O contexto de persistência é o conjunto de instâncias ativas que o aplicativo está segurando. O contexto de persistência pode ser criado manualmente ou através de injeção.

A figura ilustra as relações entre os componentes primários da arquitetura JPA.

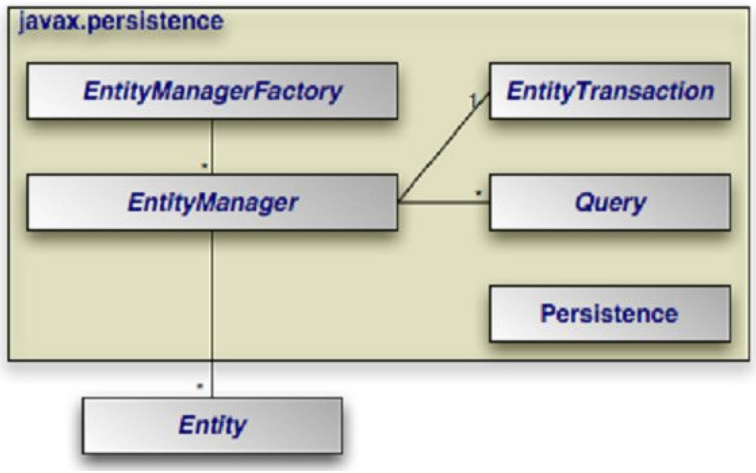


Figura 4 Componentes JPA (In: Java Persistence API Architecture ,Apache OpenJPA User's Guide)

Exemplo de uma *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="test" transaction-type="RESOURCE_LOCAL">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>

        <properties>
            <property name="hibernate.show_sql" value="true" />
            <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver" />
            <property name="javax.persistence.jdbc.url" value="jdbc:mysql://127.0.0.1:3306/test"/>
            <property name="javax.persistence.jdbc.user" value="root" />
            <property name="javax.persistence.jdbc.password" value="rodolfo" />
            <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect"/>
            <property name="hibernate.hbm2ddl.auto" value="update"/>
        </properties>
    </persistence-unit>
</persistence>
```

Figura 5 Configuração JPA

4.1 JPQL: *Java Persistence Query Language*

Com o JPQL deixou-se de trabalhar com tabelas e colunas do modelo do relacional e passou-se a utilizar classes e atributos em suas consultas. É usada para fazer diferentes consultas em um objeto de persistência, trabalha com os conceitos de SQL como *SELECT*, *INSERT*, *DELETE* e baseia-se um pouco na orientação a objeto, deixando mais fácil para pessoas que já trabalham com banco de dados relacionais trabalharem com essa linguagem de Consulta do que com Criteria, mas saindo do critério de orientação a objeto.

```
String SQL = "SELECT produto FROM ProdutoBE produto " +
                    "WHERE produto.name = mouse";
```

```
Query query = em.createQuery(SQL);

List<ProdutoBE> reList =  (ArrayList< ProdutoBE >) query.getResultList();
```

O Código acima mostra a consulta dos dados da tabela produto onde em vez de se consultar a tabela, se consulta a classe da mesma mapeada na aplicação.

5 HIBERNATE

HIBERNATE é o *FRAMEWORK* ORM mais popular na atualidade é um software livre de código aberto distribuído com a licença LGPL. Ele dá suporte ao Mapeamento com associações entre objetos, herança, polimorfismo, composições e coleções, que formam uma estrutura lógica de banco de dados relacionais. Além do mapeamento, permite as consultas por meio de HQL e a CRITERIA de uma forma orientada a objeto.

Acaba com a necessidade de escrever muito código SQL e trabalha no intermédio entre a aplicação e o banco de dados, deixando o desenvolvedor com mais tempo livre para outras atividades.

As principais interfaces do HIBERNATE são a *Session*, *Session Factory*, *Configuration*, *Query* e *Criteria*.

SessionFactory: E o responsável por manter o ORM na memoria. Trabalha como uma fábrica de Session como o nome já diz.

Session: Faz a comunicação entre a aplicação e a persistência, através do JDBC, e possui um cache local.

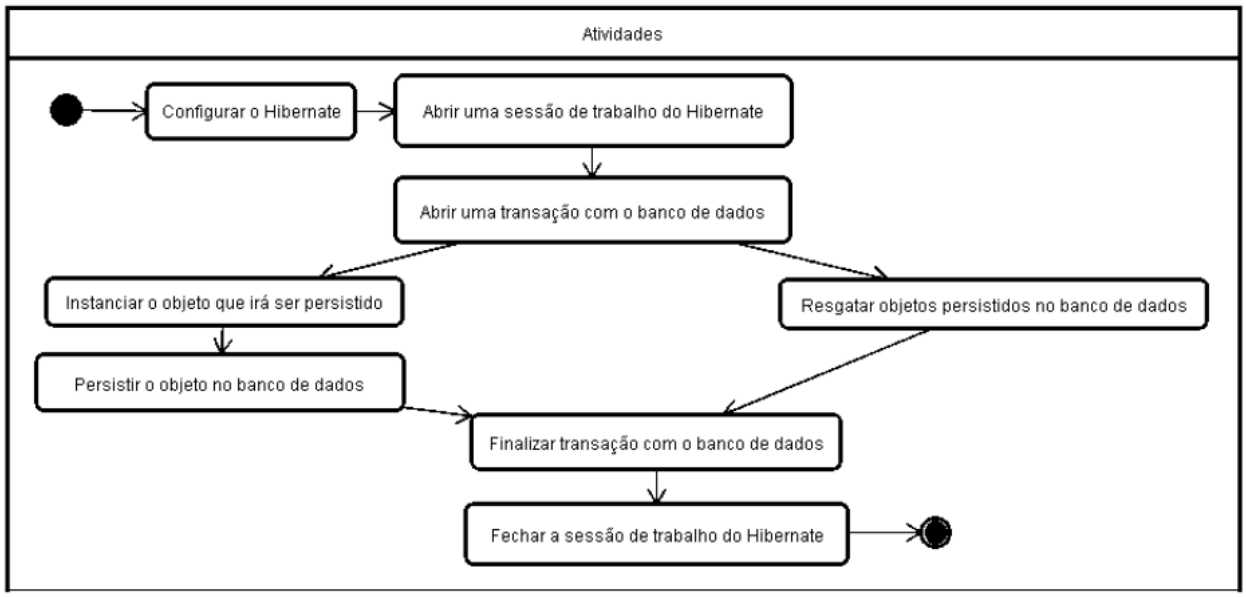


Figura 6 Diagrama de Atividades de um mapeamento objeto relacional (BAUER, 2007).

Configurar o Hibernate: Inicialização do arquivo XML que contém as configurações do Hibernate.

Abrir sessão de trabalho do Hibernate: Sessão de trabalho é a parte que vai trabalhar com o ORM.

Instanciar classes: Fazer uma instancia do objeto usado na persistência.

Persistir o objeto: o objeto é enviado para a persistência, onde o Hibernate vai verificar seus atributos e gerar o comando e executá-lo no banco de dados.

Finalizar transação com o banco de dados: parte de tratamento de erros, caso exista, retorna erro e desfaz a transação.

Fechar sessão de trabalho do Hibernate: Fechar a sessão que já não esta mais em uso para evitar a perca de desempenho da maquina.

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

    <session-factory>
        <!-- Database connection settings -->
        <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name="connection.url"> jdbc:mysql://127.0.0.1:3306/test</property>
        <property name="connection.username">root</property>
        <property name="connection.password">rodolfo</property>

        <mapping resource="Mapeamento/CidadeBE.hbm.xml"/>
        <mapping resource="Mapeamento/EstadoBE.hbm.xml"/>
        <mapping resource="Mapeamento/PaisBE.hbm.xml"/>
        <mapping resource="Mapeamento/BairroBE.hbm.xml"/>
        <mapping resource="Mapeamento/RuaBE.hbm.xml"/>
        <mapping resource="Mapeamento/ClienteBE.hbm.xml"/>

    </session-factory>

</hibernate-configuration
>
```

Figura 7 Configuração Hibernate

5.1 CRITERIA

O uso da *Criteria* acabou com a situação de ter várias linhas de *Querys* na camada de persistência e começou a ter apenas código Java no código.

Ela Facilita a detecção de erros pelo fato de sinalizar erros no momento de compilação uma das coisas que não é possível observar quando se está trabalhando com linhas de *Querys*.

```
Criteria crit = session.createCriteria(Product.class);
crit.add(Restrictions.eq("nome","Mouse"));
List results = crit.list()
```

Exemplo de consulta com *Criteria* acima, trazendo todos os resultados da tabela Produto, e adicionando uma restrição, a qual consiste em que o nome do produto tenha que ser “*Mouse*” e gerando a lista desses resultados.

6 DESENVOLVIMENTO DA APLICAÇÃO DE TESTE

A aplicação está sendo desenvolvida utilizando a IDE ECLIPSE e trabalhando com as classes das tabelas da estrutura da 6.1. Consistirá de telas de Cadastro e Consulta para as mesmas, e telas contendo as consultas dos estudos de caso do Capítulo 8.

6.1 Estrutura do Banco de Dados

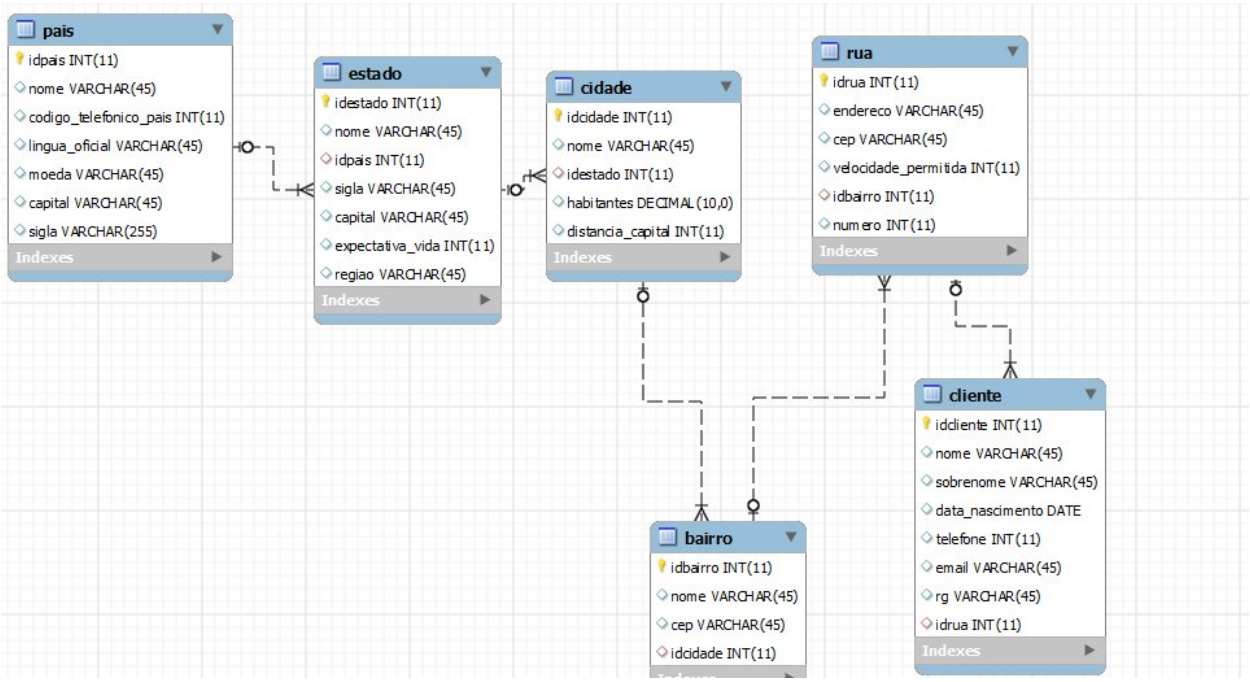


Figura 8 Estrutura do Banco de Dados da Aplicação

Serão criadas seis tabelas, cada uma contendo atributos distintos e com relacionamentos entre elas. Sempre com uma chave primaria para uso de identificação e de uso para fazer os relacionamentos.

6.2 Aplicação Hibernate

6.2.1 Tela Principal

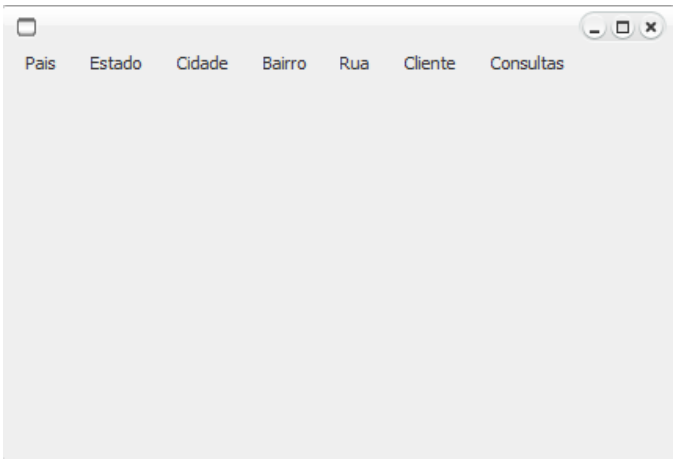


Figura 9 Tela Principal Hibernate

6.2.2 Tela de Cadastro

Figura 10 Tela de Cadastro Hibernate

6.2.3 Tela de Consulta

Figura 11 Tela de Consulta Hibernate

6.2.4 Tela de Caso de Consulta

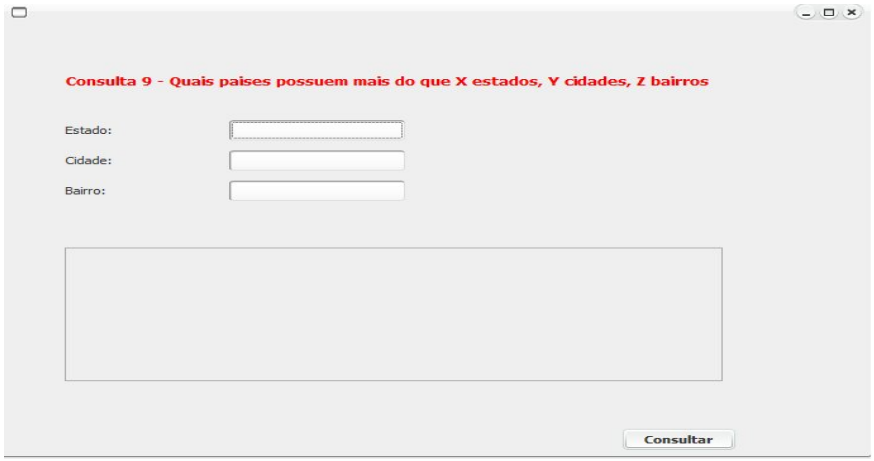


Figura 12 Tela de Caso de Consulta Hibernate

6.3 APLICAÇÃO JPA

6.3.1 Tela Principal

Figura 13 Tela Principal JPA

6.3.2 Tela de Cadastro

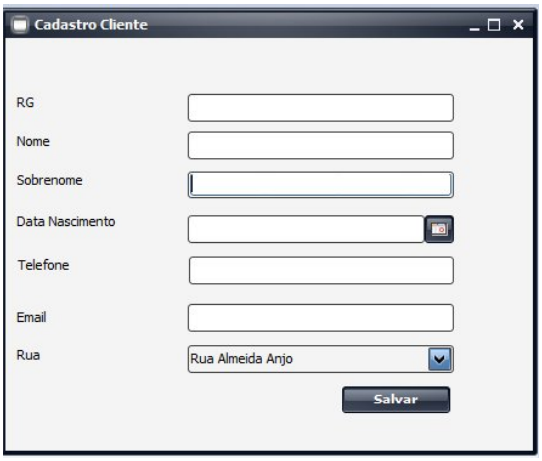


Figura 14: Tela de Cadastro JPA

6.3.3 Tela de Consulta

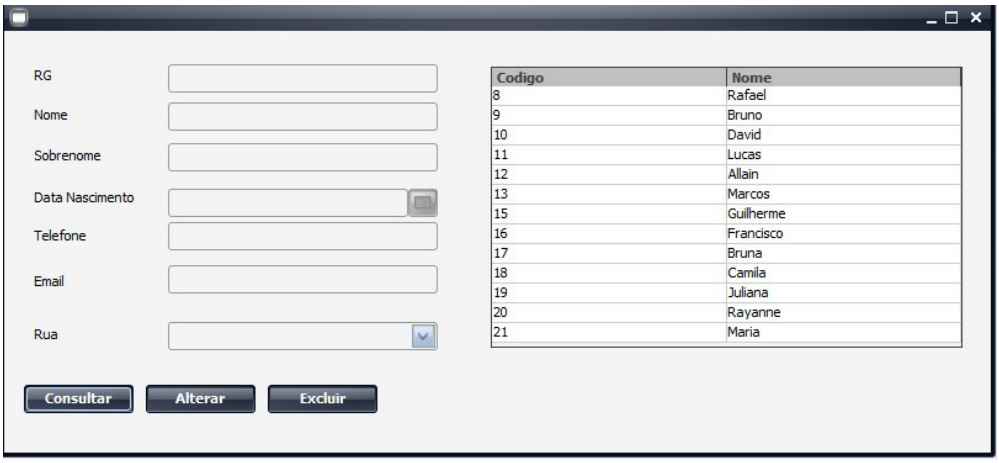


Figura 15 Tela de Consulta JPA

6.2.4 Tela de Caso de Consulta

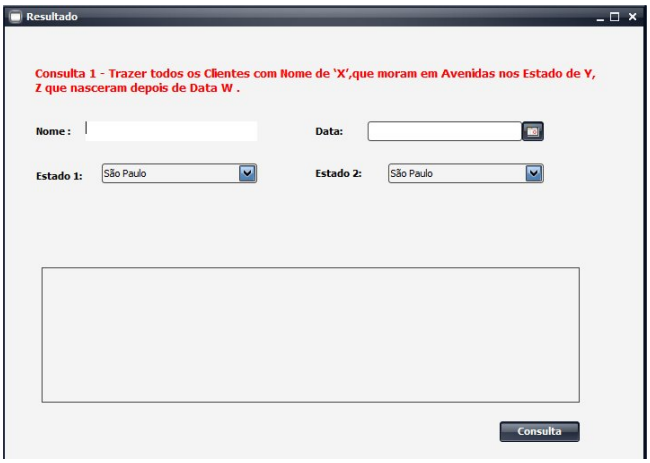


Figura 16 Tela de Caso de Consulta Hibernate

7 CASOS DE CONSULTAS

Serão testados diferentes tipos de consultas nessa seção a fim de prover informações para a conclusão do trabalho.

7.1 Caso Um

Buscar todos os Clientes com nome com ‘X’, que moram em Avenidas nos Estado de Y, Z e que nasceram depois de Data W.

Figura 17 Primeiro caso

Resolução Criteria

```
Criteria criteriaCliente =
HibernatUtils.getSession().createCriteria(ClienteBE.
class);
criteriaCliente.add(Restrictions.like("nome", nome,
MatchMode.ANYWHERE));
criteriaCliente.add(Restrictions.gt("data_nascimento
",data));
Criteria criteriaRua =
criteriaCliente.createCriteria("rua");
criteriaRua.add(Restrictions.ilike("endereco", "Av",
MatchMode.ANYWHERE));
Criteria criteriaBairro =
criteriaRua.createCriteria("bairro");
Criteria criteriaCidade =
criteriaBairro.createCriteria("cidade");
Criteria criteriaEstado =
criteriaCidade.createCriteria("estado");
Criterion estadoSao =
Restrictions.eq("nome",estadoBE.getNome());
Criterion estadoRio = Restrictions.eq("nome",
estadoBE2.getNome());
LogicalExpression ouExp = Restrictions.or(estadoRio,
estadoSao);
criteriaEstado.add(ouExp);

List<ClienteBE> reList = criteriaCliente.list();
```

Figura 18 Primeiro Caso - Criteria

Resolução JPA

```
String SQL = "SELECT cliente FROM ClienteBE cliente JOIN cliente.rua rua JOIN
rua.bairro bairro"
+ "JOIN bairro.cidade cidade JOIN cidade.estado estado JOIN estado.pais pais
"
+ "where cliente.nome like '%"+nome+"%' and cliente.data_nascimento > '" +
data.getDate() + "' "
+ " and rua.endereco like '%Av%' "
+"and   (estado.nome   =   '"+estadoBE.getNome()+"'   or   estado.nome   =
'"+estadoBE.getNome()+"') ";

Query query = em.createQuery(SQL);


List<ClienteBE> reList =  (ArrayList<ClienteBE>) query.getResultList();
```

Figura 19 Primeiro Caso - JPQL

Foi necessário o uso de mais comandos na *Criteria* do que na JPQL e de um conhecimento maior da ferramenta. Já no caso com JPQL um leve conhecimento de SQL levaria a resolução do caso.

O uso de comandos para as expressões lógicas na parte da *Criteria* acaba deixando difícil de entender a regra da lógica.

Tempo de execução Hibernate: 271 Milissegundos.

Tempo de execução JPA: 322 Milissegundos.

7.2 Caso Dois

Buscar o mínimo, máximo, soma e média dos habitantes que falam X e Z línguas, que residem em estados que a expectativa de vida seja entre W – K e que estejam a pelo menos 200 km de sua capital.

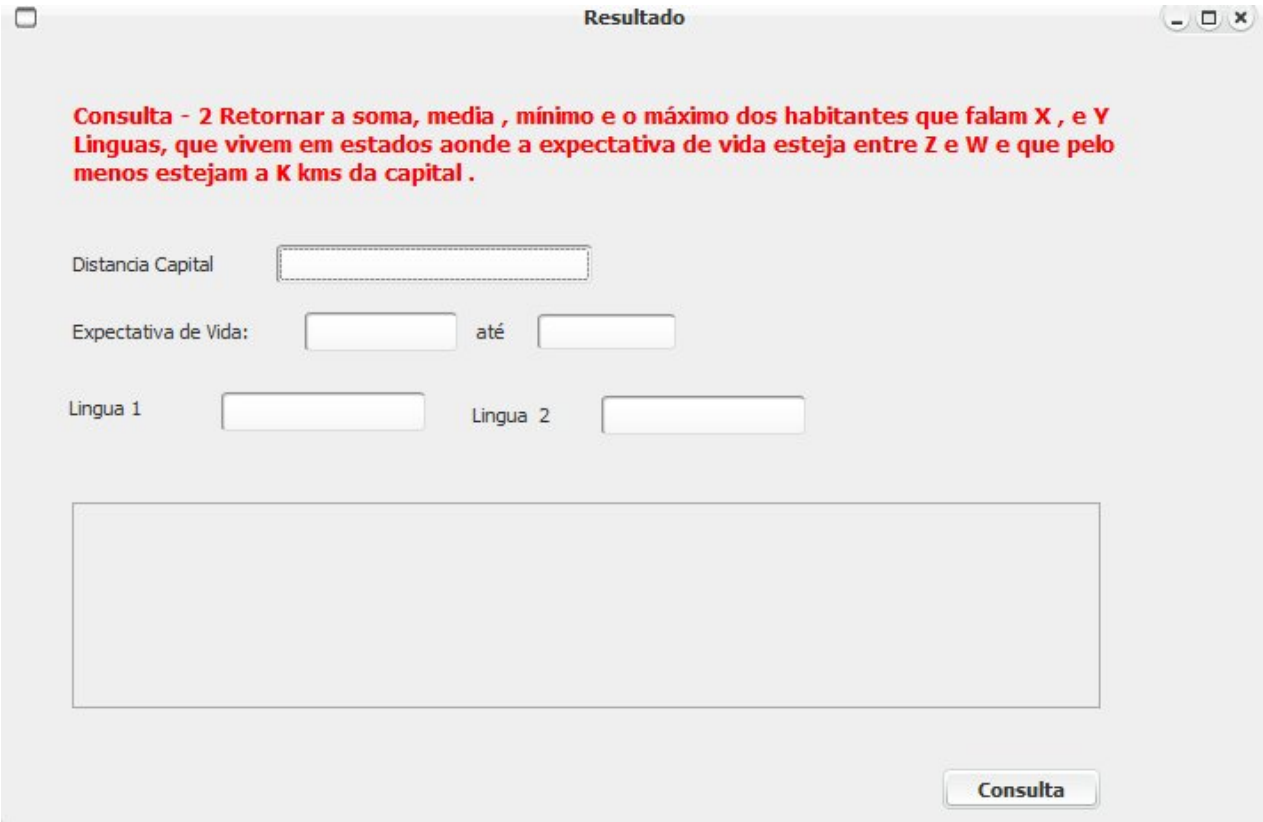


Figura 20 Segundo caso

Resolução Criteria

```
HibernatUtils.getSession().beginTransaction();
DefaultTableModel modelo = new DefaultTableModel();
Criteria criteriaCidade =
HibernatUtils.getSession().createCriteria(CidadeBE.class);
criteriaCidade.add(Restrictions.lt("distancia_Capital",
Integer.parseInt(txtDist.getText())));
Criteria criteriaEstado = criteriaCidade.createCriteria("estado");
criteriaEstado.add(Restrictions.between("expectativa_vida",
Integer.parseInt(txtEx.getText()),
Integer.parseInt(texthab2.getText())));
Criteria criteriaPais = criteriaEstado.createCriteria("pais");
Criterion LinguaPor = Restrictions.eq("lingua_oficial",
textLingua.getText());
Criterion LinguaIngl = Restrictions.eq("lingua_oficial",
textLingua_1.getText());
LogicalExpression ouExp = Restrictions.or(LinguaPor, LinguaIngl);
criteriaPais.add(ouExp);
criteriaCidade.setProjection(Projections.min("habitantes"));
Integer min = (Integer) criteriaCidade.uniqueResult();
criteriaCidade.setProjection(Projections.max("habitantes"));
Integer max = (Integer) criteriaCidade.uniqueResult();
criteriaCidade.setProjection(Projections.sum("habitantes"));
Long soma = (Long) criteriaCidade.uniqueResult();
criteriaCidade.setProjection(Projections.avg("habitantes"));
Double media = (Double) criteriaCidade.uniqueResult();
```

Figura 21 Segundo Caso - Criteria

Resolução JPQL

```
String SQL = "SELECT sum(cidade.habitantes ),max(cidade.habitantes
),min(cidade.habitantes)," + " avg(cidade.habitantes ) FROM  CidadeBE
cidade JOIN cidade.estado estado " + " " + "JOIN estado.pais pais " + "
where cidade.distancia_Capital < " + txtDist.getText().toString() + " and
" + "  (pais.lingua_oficial = '" + textLingua_1.getText() + "' or
pais.lingua_oficial = '" + textLingua.getText() + "') " + " and
(estado.expectativa_vida between " + txtEx.getText() + " and " +
texthab2.getText() + ")";

Query query = em.createQuery(SQL);

Object[] valores = (Object[]) query.getResultList().get(0);
```

Figura 22 Segundo Caso - JPQL

A Consulta com *Criteria* se tornou mais complexa em fato de se ter que usar projeções para poder ver o resultado de somas, média etc, já a JPQL traz o mesmo resultado com a sintaxe mais simples e não precisar restringir a um único resultado já que no SQL é automático.

No ambiente Java, não se pega o valor diretamente em *Int*, o que acaba precisando no final ficar fazendo muitas conversões, e como na JPQL esta se trabalhando com texto acabar não precisando ficar utilizando essa técnica.

Tempo de execução Hibernate: 146 Milissegundos.
Tempo de execução JPA: 63 Milissegundos.

7.3 Caso Três

Buscar as ruas que não sejam Avenidas, que fiquem em cidades com nome começadas por X, que não estejam em estados com expectativa de vida igual a Y, e que a moeda corrente não seja W. Os resultados devem ser ordenados em ordem decrescente pelo nome da rua.

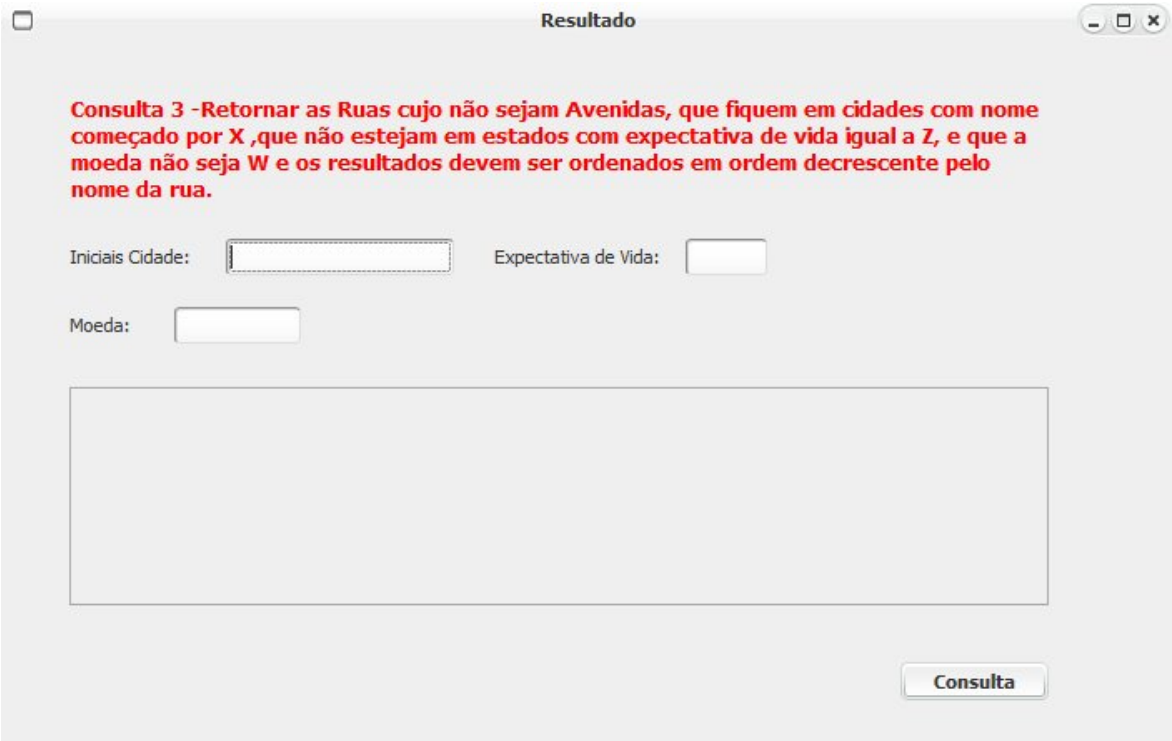


Figura 23 Terceiro caso

Resolução Criteria

```
Criteria criteriaRua =
HibernatUtils.getSession().createCriteria(RuaBE.class);
criteriaRua.add(Restrictions.not(Restrictions.ilike("endereco
", "av", MatchMode.ANYWHERE)));
Criteria criteriaBairro =
criteriaRua.createCriteria("bairro");
Criteria criteriaCidade =
criteriaBairro.createCriteria("cidade");
criteriaCidade.add(Restrictions.ilike("nome",nome,MatchMode.S
TART));
Criteria criteriaEstado =
criteriaCidade.createCriteria("estado");
criteriaEstado.add(Restrictions.ne("expectativa_vida",Expec))
;
Criteria criteriaPais =
criteriaEstado.createCriteria("pais");
criteriaPais.add(Restrictions.ne("moeda", moeda));
criteriaRua.addOrder(Order.desc("endereco"));
List<RuaBE> reList = criteriaRua.list();
```

Figura 24 Terceiro Caso - Criteria

Resolução JPA

```
String SQL = "SELECT rua FROM RuaBE rua" +
 "  JOIN  rua.bairro bairro JOIN bairro.cidade cidade
JOIN cidade.estado estado " +
 " JOIN estado.pais pais " +
" WHERE rua.endereco not like '%av%' and cidade.nome like
'"+nome+"%'  " +
" and estado.expectativa_vida != "+Expec+" and pais.moeda
!= '"+moeda+"'  " +
" ORDER by  rua.endereco desc";

Query query = em.createQuery(SQL);
```

Figura 25 Terceiro Caso - JPQL

As duas consultas precisaram de poucas linhas e comandos para funcionar. O comando *addOrder* consegue superar a facilidade do Clausula *Order By*, pois já traz todos os tipos de ordenações com apenas um clique.

A restrição do *like* na *Criteria* acabou precisando de vários parâmetros como o MatchMode.Start, o que deixava mais complicado de entender, no JPQL a restrição *like* é feita com maior facilidade .

Tempo de execução Hibernate: 29 Milissegundos.

Tempo de execução JPA: 337 Milissegundos.

7.4 Caso Quatro

Buscar as cidades que não possuem Rua com nome X.

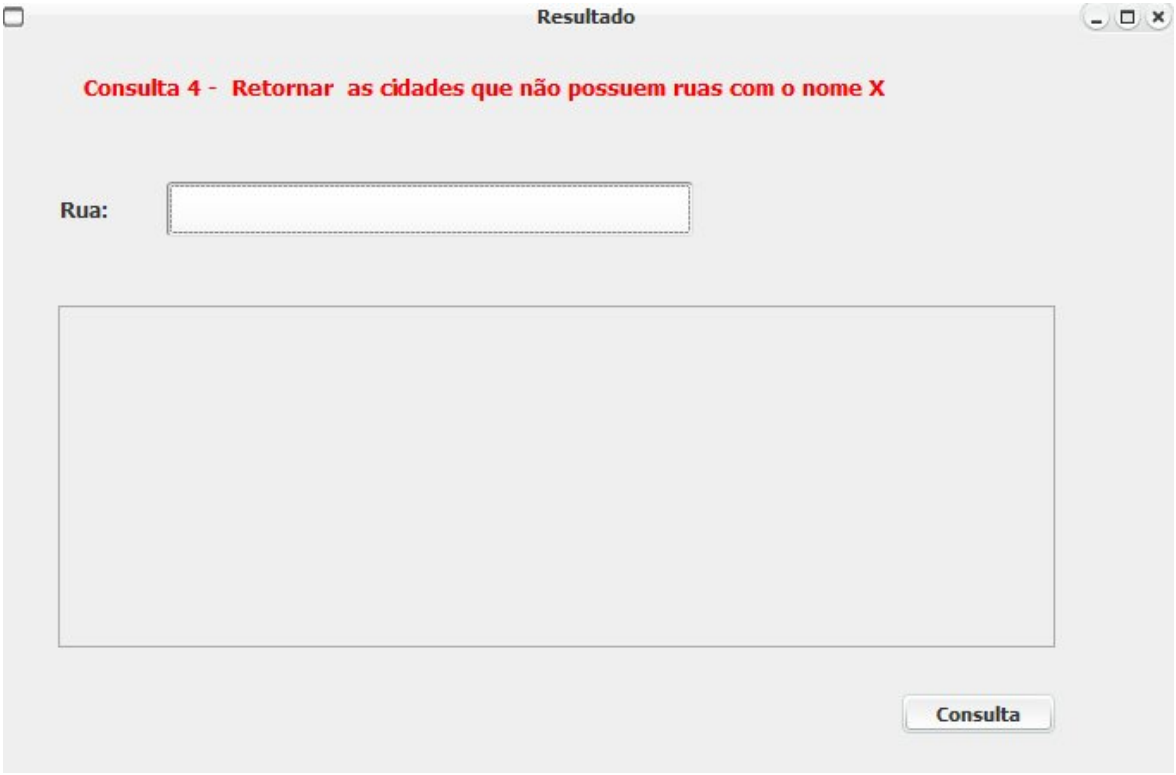


Figura 26 Quarto caso

Resolução Criteria

```
HibernatUtils.getSession().beginTransaction();

Criteria criteriaCidade =
HibernatUtils.getSession().createCriteria(CidadeBE.class);
Criteria criteriaBairro =
criteriaCidade.createCriteria("bairros");
Criteria criteriaRua =
criteriaBairro.createCriteria("ruas");
criteriaRua.add(Restrictions.not(Restrictions.like("enderec
o", txtNome.getText(), MatchMode.ANYWHERE)));
criteriaCidade.setResultTransformer(DistinctRootEntityResul
tTransformer.INSTANCE);

List<CidadeBE> reList = criteriaCidade.list();
```

Figura 27 Quarto Caso - Criteria

Resolução JPA

```
String SQL = "SELECT cidade FROM CidadeBE cidade JOIN
cidade.bairros bairro " +
" JOIN bairro.ruas ruas WHERE ruas.endereco not like '%" +
txtNome.getText() + "%' " + "GROUP by cidade.idcidade ";

Query query = em.createQuery(SQL);
```

Figura 28 Quarto Caso - JPQL

Na *Criteria* a necessidade de usar o *SetResultTransfomer* é confuso inicialmente e acaba complicado o que na JPQL é só usar o conceito do *Group by* para chegar no resultado desejado.

A *Criteria* não tem uma restrição direta de *ilike* o que faz com que o usuário necessite fazer uma restrição do tipo *.not* e em seguida a restrição do modo *like* o que acabou fazendo o

desenvolvimento das mesmas muito mais pratica na JPQL que o simples comando *not like* faz todas as restrições necessárias.

Tempo de execução Hibernate: 132 Milissegundos.

Tempo de execução JPA: 40 Milissegundos.

7.5 Caso Cinco

Retornar as cidades do estado X que possuam um número de habitantes maior que Z.

Figura 29 Quinto caso

Resolução Criteria

```
Criteria criteriaCidade =
HibernatUtils.getSession().createCriteria(CidadeBE.class);
criteriaCidade.add(Restrictions.eq("estado", estado));

criteriaCidade.add(Restrictions.gt("habitantes",quantidade
Habitantes));

List<CidadeBE> reList = criteriaCidade.list();
```

Figura 30 Quinto Caso - Criteria

Resolução JPQL

```
String SQL = "SELECT cidade FROM CidadeBE cidade JOIN
cidade.estado estado " + " WHERE cidade.habitantes > " +
txtNome.getText() + " and estado.nome " + " = '" + nome +
"'";

Query query = em.createQuery(SQL);
```

Figura 31 Quinto Caso - JPQL

No caso com CRITERIA houve a restrição do objeto Estado inteiro, em vez de ter que se pegar apenas o valor do nome do mesmo, o que mantém o código mais padronizado nas práticas de OO.

As duas trabalharam de maneiras eficientes em que apenas foi necessário o uso da restrição *grater than* na primeira consulta e na segunda apenas a > fez toda a parte de restrição.

Tempo de execução Hibernate: 88 Milissegundos.

Tempo de execução JPA: 55 Milissegundos.

7.6 Caso Seis

Retornar as cidades que possuam mais de X ruas.

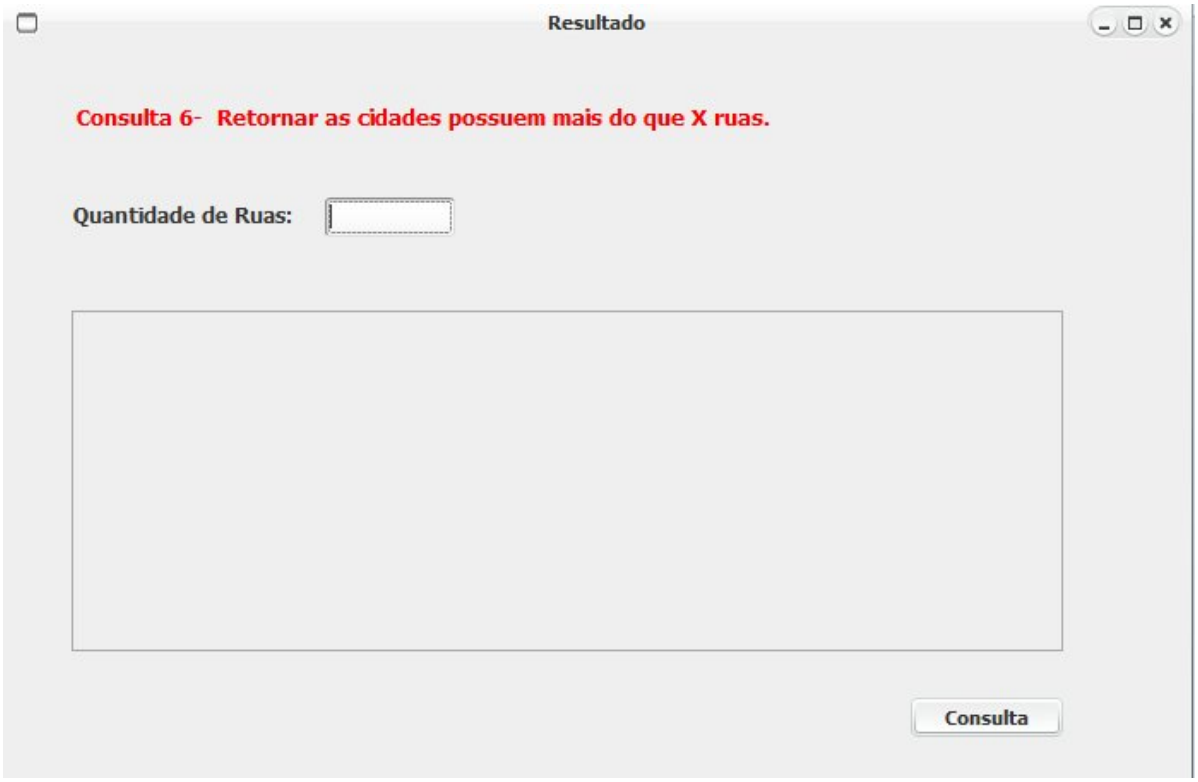


Figura 32 Sexto caso

Resolução Criteria

```
Criteria criteriaCidade =
HibernatUtils.getSession().createCriteria(CidadeBE.class);
Criteria criteriaBairro =
criteriaCidade.createCriteria("bairros");
criteriaBairro.add(Restrictions.sizeGt("ruas",
Integer.parseInt(txtRuas.getText())));

List<CidadeBE> reList = criteriaCidade.list();
```

Figura 33 Sexto Caso - Criteria

Resolução JPQL

```
String SQL = "SELECT cidade FROM CidadeBE cidade JOIN
cidade.bairros bairro " +
" JOIN bairro.ruas ruas WHERE count(SELECT * FROM ruas) >
+ ruas";

Query query = em.createQuery(SQL);
```

Figura 34 Sexto Caso - JPQL

No caso com JPQL foi gasto mais tempo na criação da consulta em torno do relacionamento *One-To-Many*. Para que se pudessem restringir valores na JPQL teve que criar *subquerys* com mais condições, já na *Criteria* a forma de trabalhar mesmo sendo *One-To-Many* continua simples e objetiva, funcionando do mesmo jeito que funcionava com os outros tipos de relacionamentos.

Tempo de execução Hibernate: 53 Milissegundos.

Tempo de execução JPA: 45 Milissegundos.

7.7 Caso Sete

Retornar os bairros que não contém clientes.

Figura 35 Sétimo caso

Resolução Criteria

```
Criteria criteriaBairro =
HibernatUtils.getSession().createCriteria(BairroBE.class);
Criteria criteriaRua =
criteriaBairro.createCriteria("ruas");
criteriaBairro.setResultTransformer(DistinctRootEntityResu
ltTransformer.INSTANCE);
criteriaRua.add(Restrictions.sizeEq("clientes", 0));


List<BairroBE> reList = criteriaBairro.list();
```

Figura 36 Sétimo Caso - Criteria

Resolução JPQL

```
String SQL = "SELECT bairro FROM BairroBE bairro WHERE
(SELECT COUNT(*) FROM RuaBE rua  "+
"WHERE rua.bairro = bairro) = 0 GroupBy bairro.idbairro";

Query query = em.createQuery(SQL);
```

Figura 37 Sétimo Caso - JPQL

CRITERIA usa mais comandos para execução, mas JPQL se apresenta mais complexa pelo uso da *subQuerys*, já que com apenas a restrição *sizeEq* no atributo que esta carregando os valores do relacionamento do bairro, faz o trabalho todo ao contrario da JPQL que necessariamente precisa fazer uma outra linha de *Query* apenas para restringir o relacionamento a 0.

Tempo de execução Hibernate: 53 Milissegundos.

Tempo de execução JPA: 45 Milissegundos.

7.8 Caso Oito

Retornar os estados que possuam cidades com menos de X bairros.

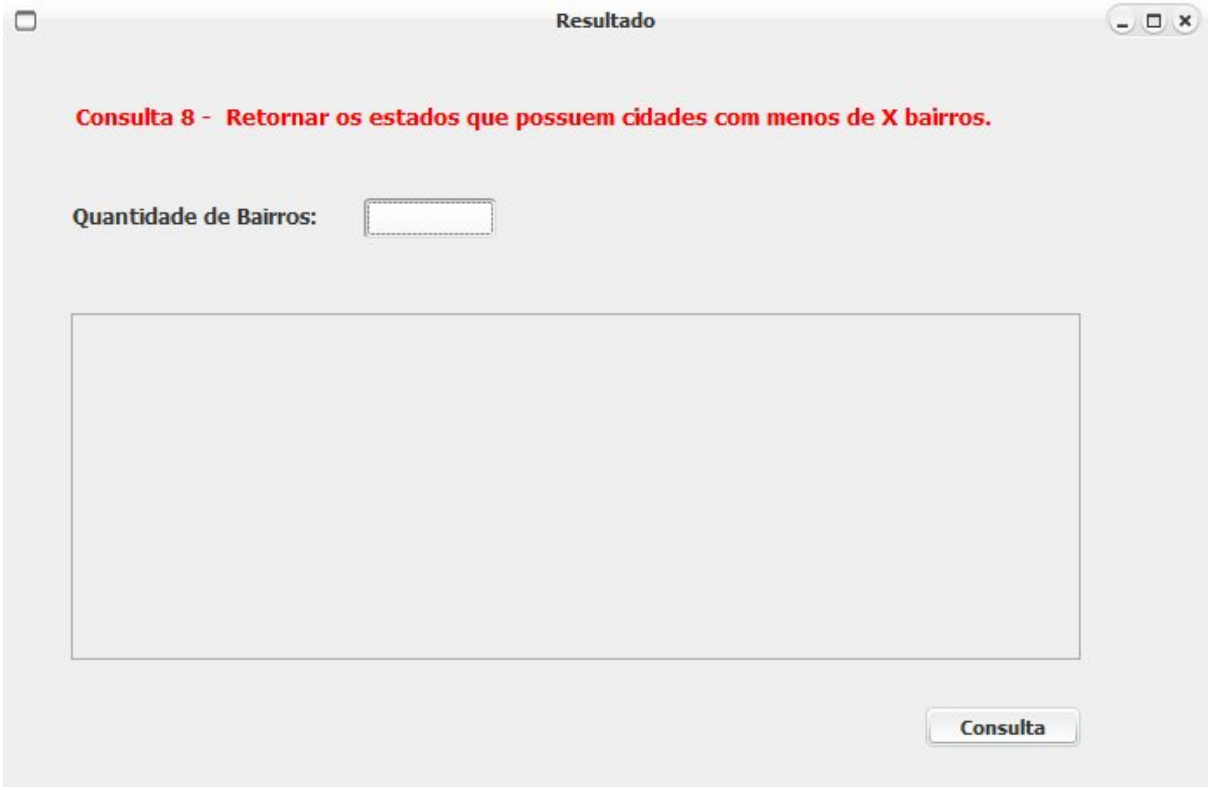


Figura 38 Oitavo caso

Resolução Criteria

```
Criteria criteriEstado =
HibernatUtils.getSession().createCriteria(EstadoBE.class);
Criteria criteriaCidade =
criteriEstado.createCriteria("cidades");
criteriaCidade.add(Restrictions.sizeLt("bairros",
xBairros));
criteriEstado.setResultTransformer(DistinctRootEntityResul
tTransformer.INSTANCE);

List<EstadoBE> reList = criteriEstado.list();
```

Figura 39 Oitavo Caso - Criteria

Resolução JPQL

```
String SQL = "SELECT estado FROM EstadoBE estado WHERE
(SELECT COUNT(*) FROM CidadeBE cidade  "+
" JOIN cidade.bairros bairro JOIN bairro.ruas rua  JOIN
rua.clientes cliente WHERE cidade.estado = estado) = 0 " ;

Query query = em.createQuery(SQL);
```

Figura 40 Oitavo Caso - JPQL

Como em outros casos, sempre que envolve restringir um atributo de tabelas relacionadas na JPQL a necessidade de fazer *SubQuery* complica o desenvolvimento da consulta.

Precisou fazer como restrição no *where* um *select* nas cidades com muitos *joins* para chegar no resultado da JPQL.

Na Criteria a restrição Lt(*Less than*) já realizou todo o trabalho, deixando o programador livre da maneira como lidaria com esses relacionamentos.

Tempo de execução Hibernate: 205 Milissegundos.

Tempo de execução JPA: 39 Milissegundos.

7.9 Caso Nove

Retornar os países possuam mais do que X estados, Y cidades, Z bairro.

Figura 41 Nono caso

]

Resolução Criteria

```
Criteria criteriaPais =
HibernatUtils.getSession().createCriteria(PaisBE.class);
Criteria criteriaEstado =
criteriaPais.createCriteria("estados");
Criteria criteriaCidades =
criteriaEstado.createCriteria("cidades");


criteriaPais.add(Restrictions.sizeGe("estados",
quantEstado));
criteriaEstado.add(Restrictions.sizeGe("cidades",
quantCidade));
criteriaCidades.add(Restrictions.sizeGe("bairros",
quantBairro));

List<EstadoBE> reList = criteriEstado.list();
```

Figura 42 Nono Caso - Criteria

Resolução JPQL

```
String SQL = "SELECT pais FROM PaisBE pais WHERE (SELECT
COUNT(*) FROM EstadoBE estado WHERE pais = estado.pais) >
" +quantEstado.toString() +" AND "+
"(SELECT COUNT(*) FROM EstadoBE estado JOIN estado.cidades
cidade  WHERE pais = estado.pais) > "
+quantCidade.toString() +" AND " +
"(SELECT COUNT(*) FROM EstadoBE estado JOIN estado.cidades
cidade JOIN cidade.bairros  WHERE pais = estado.pais) > "
+quantBairro.toString() +"";

Query query = em.createQuery(SQL);
```

Figura 43 Nono Caso - JPQL

A CRITERIA apenas precisou de uma restrição no tamanho em cada *criteria* com o uso do Ge(*Grater*,*Equal*) dizendo que a lista de valores dos relacionamentos deveriam ser maior ou igual o valor informado. Na JPQL foi necessário fazer uma *select* com 3 *subQuery* para conseguir o resultado desejado onde ainda em cada *SubQuery* tinha que ficar lidando com a questão dos relacionamentos que gera mais complexidade.

A operação logica *AND* que não precisou ser especificado na Criteria por ser nativo.

Tempo de execução Hibernate: 378 Milissegundos.

Tempo de execução JPA: 209 Milissegundos.

7.10 Caso Dez

Retornar os estados não possuam clientes cadastrados.

Figura 44 Decimo caso

Resolução Criteria

```
Criteria criteriEstado =
HibernatUtils.getSession().createCriteria(EstadoBE.class);
Criteria criteriaCidade =
criteriEstado.createCriteria("cidades");
Criteria criteriaBairro =
criteriaCidade.createCriteria("bairros");
Criteria criteriaRuas =
criteriaBairro.createCriteria("ruas");

criteriaRuas.add(Restrictions.sizeEq("clientes", 0));

criteriEstado.setResultTransformer(DistinctRootEntityResul
tTransformer.INSTANCE);

List<EstadoBE> reList = criteriEstado.list();
```

Figura 45 Decimo Caso - Criteria

Resolução JPQL

```
String SQL = "SELECT estado FROM EstadoBE estado WHERE
(SELECT COUNT(*) FROM CidadeBE cidade  "+
" JOIN cidade.bairros bairro JOIN bairro.ruas rua  JOIN
rua.clientes cliente WHERE cidade.estado = estado) = 0 " ;

Query query = em.createQuery(SQL);
```

Figura 46 Decimo Caso - JPQL

Na CRITERIA foi necessário apenas um comando para se conseguir restringir, mas acabou precisando criar varias *Criterias* até conseguir chegar na tabela desejada. Na JPQL precisou fazer o uso de *SubQuery* na restrição *where* com o auxilio do *COUNT* pra restringir os resultados e ainda realizar os *joins* de ligação, no final acabou tendo 2 *where* de restrição enquanto na Criteria apenas uma restrição conseguiu fazer o desejado.

Tempo de execução Hibernate: 15 Milissegundos.

Tempo de execução JPA: 22 Milissegundos.

8 CONCLUSÃO

O uso de *frameworks* de ORM, seja com persistências JPA ou HIBERNATE, consequentemente deixa o desenvolvimento de aplicações muito mais rápido e prática, evitando gasto de tempo na parte de alocação, manipulação de dados e mapeamento, que acaba sendo um trabalho árduo, deixando também mais tempo para se desenvolver as regras de negócios ou as interfaces que o usuário ira interagir.

O ORM já esta presente em uma boa parte das empresas de sistema de informação, e em um futuro não muito longe deverá ser adotada pela maioria, já que a manipulação dos dados é uma parte que demanda tempo e as vezes muitas linhas acabam atrapalhando a visualização geral do programador e com esses Frameworks toda essa parte de manipulação fica rápida e fácil, deixando os programadores com mais tempo para focar no desenvolvimento e menos no banco.

No requisito desempenho, as duas se mostraram rápidas e com poucas diferenças entre desempenho, mas levando em conta que os testes foram feitos em uma base pequena de dados e trabalhando com uma quantidade massiva pode haver maiores variações.

Nos casos 7, 8, 9, 10 que acontece de começar forçar mais os usos dos relacionamentos fica claro que a Criteria tem um comportamento melhor nessa situação, tendo o desenvolvimento das consultas muito mais otimizado do que com a JPQL onde havia muitas linhas de código e *SubQuerys* mais avançadas e complicadas de trabalhar.

Outro forte ponto do HIBERNATE com *Criteria* é que caso queira mudar um atributo pode fazer a alteração apenas na classe do objeto e fim de alterações, já na JPA usando JPQL se mudar o nome de um atributo tem de ir até o código JPQL e alterar manualmente os campos que continham aquele atributo.

A JPA permite uma persistência “plugável”, sendo que se for decidido mudar o provedor de Persistência na aplicação, não terá problemas, contanto que o provedor escolhido esteja nas especificações contidas na JPA. Já o HIBERNATE não dá essa liberdade sendo que se for

preciso mudar de provedor da persistência, muito código terá que ser mudado para que isso aconteça.

Usando essas ferramentas é obtida mais portabilidade à aplicação, pois ela não ficará presa a um banco de dados específico, porque bancos diferentes usam dialetos diferentes, podendo mudar sem problemas e o desenvolvedor não precisa ter um vasto conhecimento no dialeto SQL.

O *Hibernate* com *Criteria* se mostrou mais complexo inicialmente do que a JPA, mas em questão de tempo se tornou uma alternativa melhor e mais fácil de trabalhar, pois se utilizado com um paradigma orientado a objeto, não é preciso ter um vasto conhecimento em SQL para poder se efetuar consultas complexas, mostrando maior facilidade de se encontrar os seus erros do que em uma *Query* muitas linhas ou condições além de quando é preciso trabalhar com os relacionamentos e restrições a eles, mostrou-se mais eficiente, onde as consultas podem ser resolvidas com poucas linhas e pouca lógica, ao contrário da JPQL, que fez uso de SUBQUERYS para restringir atributos em seus relacionamentos.

As duas ferramentas são poderosas e fornecem uma ótima alternativa para se trabalhar com os bancos de dados relacionais que estão fortemente estabelecidos no mercado, mas em um paradigma OO a melhor escolha ainda seria a CRITERIA para se manter uma padronização, deixando o código inteiro em JAVA e não com fragmentos de QUERYS no mesmo, e com maior facilidade de se encontrar erros nas restrições, já que fica extremamente visível onde e o que está acontecendo do que ficar procurando em várias linhas de QUERYS pelo erro.

REFERÊNCIAS

BAUER, Christian, **Java Persistence com Hibernate**, Rio de Janeiro: Ciência Moderna, 2007.

IBM, **Developing JPA applications,** Disponível em < http://publib.boulder.ibm.com/infocenter/radhelp/v7r5/index.jsp?topic=%2Fcom.ibm.jpa.doc%2Ftopics%2Fc_jpa.html>. Acesso em: 26 de dezembro de 2012

SILBERSCHATZ, Abraham, KORTH, Henry F. E SUDARSHAN, **Sistema de Banco de Dados. 3™ Ed**.Makron Books, 1999.

GUERRA, Rafael Laurino; ZAINA, Dra. Luciana Aparecida Martinez.**MAPEAMENTO OBJETO RELACIONAL: UM ESTUDO DE CASOUTILIZANDO O HIBERNATE.** Disponível em: <http://fatecindaiatuba.edu.br/reverte/index.php/revista/article/view/9/10>. Acesso em: 10 nov. 2012.

Cadu***, ORM : Object Relational Mapper,*** Disponível em < http://www.devmedia.com.br/orm-object-relational-mapper/19056>. Acesso em: 28 de maio de 2012.

CLÁUDIO DIAS NETO, Arilo, ***Bancos de Dados Relacionais***, SQL Magazine ed. 86,2011.

LINWOOD, Jeff; MINTER, Dave. ***Beggining Hibernate***. 2 ed. Apress, 2010.

BAUER, Christian; KING, Gavin. ***Java Persistence with Hibernate***. Manning Publications, 2007.

BAUER, Christian; KING, Gavin. ***Hibernate in Action***. Manning Publications, 2005.

KEITH, Mike ; SCHINCARIOL, Merrick. ***Pro EJB 3 Java Persistence API***. Apress, 2006.

KRAEMER, Fabiano; VOGT JARDEL, Jerônimo. **Hibernate, um Robusto Framework de Persistência Objeto-Relacional**. Disponível em: <http://saloon.inf.ufrgs.br/twiki/viewfile/Disciplinas/Old/PODWebSis2004/WEB04Hibernate?rev=1.1;filename=WEB04Hibernate.pdf >. Acesso em: 24 de abril de 2012.

VIVIAN REGINA, Renata;WERNER,Claudete. **Utilização Tecnologia Java e Framework Hibernate para Desenvolvimento de Software**. Disponível em <http://web.unipar.br/~seinpar/artigos/Renata-Regina-vivian.pdf>. Acesso em: 10 de maio de 2012.

BARROS, Bruno Alberth Silva; BARROS, Raul Silva; CORTES, Omar Andres Carmona. Um estudo comparativo entre APIS de persistência utilizando grandes volumes de dados. In**: Congresso de Pesquisa e inovação da Rede Norte e Nordeste e Educação Tecnológica**. 2009. Belém, Brasil.

ROCHA, Gabriel, FILHO, Hildebrando; JURITY, Rutemberg. **Camada de Persistência de Dados para Aplicações Java: O Hibernate**. Diposnível em: <https://disciplinas.dcc.ufba.br/pub/MATA60/WebHome/Hibernate.pdf>. Acesso em: 25 de março 2012.

JÚNIOR, Herval Freire de A. **Persistência Fácil de Objetos em Java**; Disponível em: <http://pt.scribd.com/doc/4484783/Artigo-Camadas-de-Persistencia-de-Objetos>. Acesso em: 5 de Abril 2012.

LIPITSAINEN, Arvo. ***ORM – Object Relational Mapping***, Disponível em: <www.dbtechnet.org/labs/dae_lab/Orm.pdf>. Acesso em: 14 de maio de 2012.

IBM. ***Apache OpenJPA User's Guide***. Diponível em: <http://pic.dhe.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=%2Forg.apache.jpa10.for.ejbfep.multiplatform.doc%2FApacheOpenJPAUsersGuide.htm>. Acesso em 27 de maio de 2012.