Persistindo com Hibernate

Francisco do Nascimento (IFPE)
francisco.junior@jaboatao.ifpe.edu.br

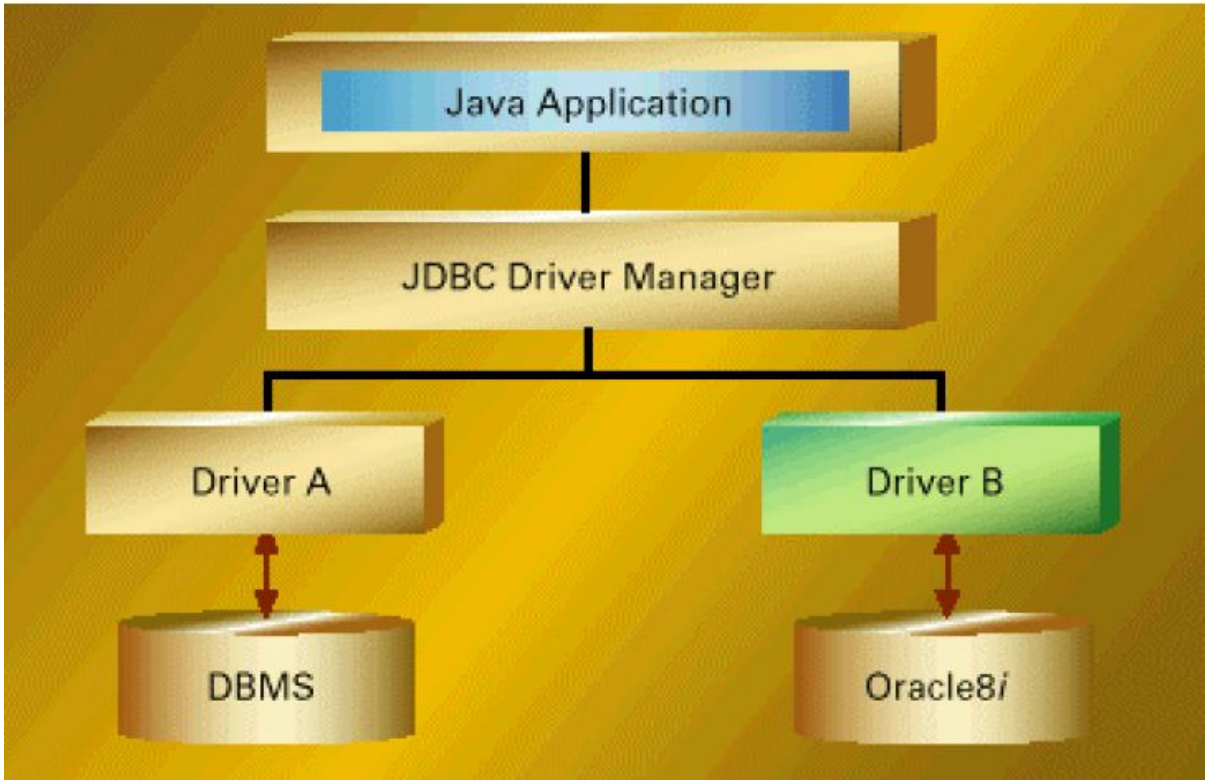
UERN, JULHO/2017

JPA/Hibernate :: Etapas

1. Mapeamento Objeto-Relacional
2. JPA/Hibernate
    a. Mapeamento de Entidades
    b. Operações sobre entidades
    c. Mapeamento de Relacionamentos
    d. Linguagem de consulta JPQL/HQL

[1]
Mapeamento Objeto-Relacional

Desde os primórdios...



Persistir dados usando Java, remete a:

- JDBC
- Driver
- Connection
- Statement
- ResultSet

Salvando um objeto no BD usando JDBC

1. Carregar o driver correspondente ao SGBD
2. Abrir conexão com o banco de dados
3. Criar um comando SQL usando os dados do objeto
4. Enviar o comando ao banco
5. Executar o comando e obter o resultado
6. Processar o resultado e gerar objetos
7. Fechar a conexão

Mudança de paradigma: Comandos para Objetos

- Uma aplicação OO deveria focar exatamente nos objetos, ao invés de linhas e colunas de uma base de dados.

Modelo Relacional	Orientação a Objetos
Tabelas, linhas	Classes, Objetos
Colunas	Atributos
Chave primária	Identidade
Chave estrangeira	Relacionamento entre classes
Stored procedures	Métodos
-	Herança

Como salvar um objeto num banco de dados relacional?

Mapeamento Objeto-Relacional (Object-Relational Mapping - ORM)

Mapeamento Objeto-Relacional (MOR)

- Técnica de desenvolvimento utilizada para reduzir a impedância da programação OO em relação ao armazenamento num banco de dados relacional.
- SQL gerado automaticamente utilizando metadados
- Vantagens:
  - Manutenibilidade: menos código, mais facilidade para manutenção e refatoração
  - Produtividade: um tempo maior disponível para regras de negócio
  - Performance: frameworks maduros com otimizações
  - Portabilidade: suporte a diversos SGBDs
  -

Partes principais de um MOR

1) Mecanismo para especificação de metadados
2) API para realização de operações básicas (CRUD: Create, Read, Update, Delete)
3) Linguagem ou API para realização de consultas ao BD
4) Técnicas de otimização
    a) Dirty checking
    b) Lazy association fetching

Aplicando um MOR

- Um time contém um código, nome, estado, pontos e vários jogadores
- Um jogador tem um código, nome, número da camisa.

Classe Time
int código
String nome
String estado
List jogadores

Classe Jogador
int código
String nome
int numeroCamisa

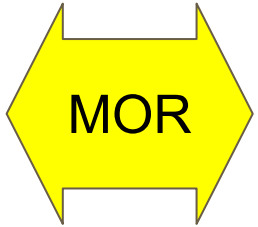


Tabela TIME
INTEGER CODIGO (PK)
VARCHAR NOME
CHAR(2) ESTADO

Tabela JOGADOR
INTEGER CODIGO (PK)
VARCHAR NOME
INTEGER NUMCAMISA
INTEGER COD_TIME (FK)
Tecnico tecnico

[2]
JPA / Hibernate

Mapeamento de Entidades

JPA: Java Persistence API

- É um padrão definido pelo JCP para trabalhar com persistência de dados.
- O líder da especificação foi Gavin King, o criador do Hibernate.
- JPA é uma tecnologia POJO (Plain-Old Java Object) orientada à metadados
  - Classes básicas não precisam implementar uma interface, nem estender uma classe
- Existem várias implementações disponíveis no mercado.
  - **Hibernate**, Glassfish, SAP Netweaver AS, TopLink, EclipseLink, OpenJPA, Kodo, JPOX, Amber, entre outras

Utilização de Metadados

- JPA fornece duas opções: **Annotation** e XML
- Serão usados para indicar respostas para as perguntas:
  - Quais são os objetos de domínio que serem persistidos? **@Entity**
  - Como identificar exclusivamente um objeto de domínio persistente? **@Id**
  - Quais são os relacionamentos entre objetos? **@OneToOne, @OneToMany, @ManyToMany**
  - Como o objeto de domínio é mapeado para tabelas de BDR? **@Table, @Column, @JoinColumn**

Mapeando uma entidade básica

```
import javax.persistence.Entity;

@Entity
public class Time {
    private Integer codigo;
    private String nome;
}
```

Mapeando Entidades: @Entity, @Table

- Cada entidade, tipicamente, representa uma tabela no banco de dados.
- Cada instância de uma entidade corresponde a uma linha na tabela.
- Requisitos:
  - Ser anotada com a anotação @Entity.
  - Deve ter um construtor sem argumentos, público ou protegido.
  - Não pode ser declarada final. Nenhum método ou variável de instância pode ser declarada como final.
  - Variáveis de instância persistentes não devem ser declaradas públicas e só podem ser acessadas pelos métodos da classe.
- @Table
  - informa o nome da tabela em que os objetos serão armazenados
  - Opcional, caso não seja adicionada, o nome da classe deverá ser igual ao nome da tabela

Mapeando atributos: @Column

- @Column: Opcional, uma anotação para cada atributo
  - Usada para dar mais informações sobre a persistência deste atributo
  - Para atributos cujo tipo não seja uma entidade
- Por default, o nome da coluna será o mesmo do atributo
- Propriedades:
  - name: nome da coluna na tabela
  - length: tamanho do VARCHAR, usado no tipo String
  - nullable: true/false para definir se o atributo permite ***null***
- Ao mapear uma entidade com @Entity, todas as propriedades serão **persistentes**
- Propriedades não persistentes devem ser marcadas com **@Transient** ou o modificador **transient**

Exemplo: Classe Fornecedor

```
@Entity
@Table(name="TBFORNECEDOR")
public class Fornecedor {

    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer codigo;
    @Column(length=14)
    private String cnpj;
    private String razaoSocial;
```

Mapeando atributos: @Colum (tipos)

Mapping type	Java type	Standard SQL built-in type
integer	int or java.lang.Integer	INTEGER
long	long or java.lang.Long	BIGINT
short	short or java.lang.Short	SMALLINT
float	float or java.lang.Float	FLOAT
double	double or java.lang.Double	DOUBLE
big_decimal	java.math.BigDecimal	NUMERIC
character	java.lang.String	CHAR(1)
string	java.lang.String	VARCHAR
byte	byte or java.lang.Byte	TINYINT
boolean	boolean or java.lang.Boolean	BIT
yes_no	boolean or java.lang.Boolean	CHAR(1) ('Y' or 'N')
true_false	boolean or java.lang.Boolean	CHAR(1) ('T' or 'F')

Mapeando atributos: @Colum (tipos)

Mapping type	Java type	Standard SQL built-in type
date	java.util.Date or java.sql.Date	DATE
time	java.util.Date or java.sql.Time	TIME
timestamp	java.util.Date or java.sql.Timestamp	TIMESTAMP
calendar	java.util.Calendar	TIMESTAMP
calendar_date	java.util.Calendar	DATE

Mapeando o identificador: @Id, @GeneratedValue

- @Id: Obrigatório, define o atributo que identifica o objeto
  - É mapeado para a chave primária
  - Por definição, se duas instâncias tiverem o mesmo identificador, elas representam a mesma linha da tabela
- @GeneratedValue: Controla a geração do valor do identificador
  - Parâmetro strategy especifica a estratégia de geração do identificador:
    - AUTO – estratégia é escolhida de acordo com o banco utilizado
    - IDENTITY – DB2, MySQL, SQL Server
    - SEQUENCE – DB2, Postgre, Oracle
    - TABLE – Valor da chave armazenado numa tabela

```
@Id
@GeneratedValue(strategy=GenerationType.AUTO)
private Integer codigo;
```

Mapeando datas: @Temporal

- Propriedades podem ser do tipo java.util.Date ou java.util.Calendar
- Precisão pode ser alterada através da Annotation @Temporal:
  DATE, TIME, TIMESTAMP (default)

```
@Temporal(TemporalType.DATE)
private Date date1;

@Temporal(TemporalType.TIME)
private Date date2;

@Temporal(TemporalType.TIMESTAMP)
private Date date3;
```

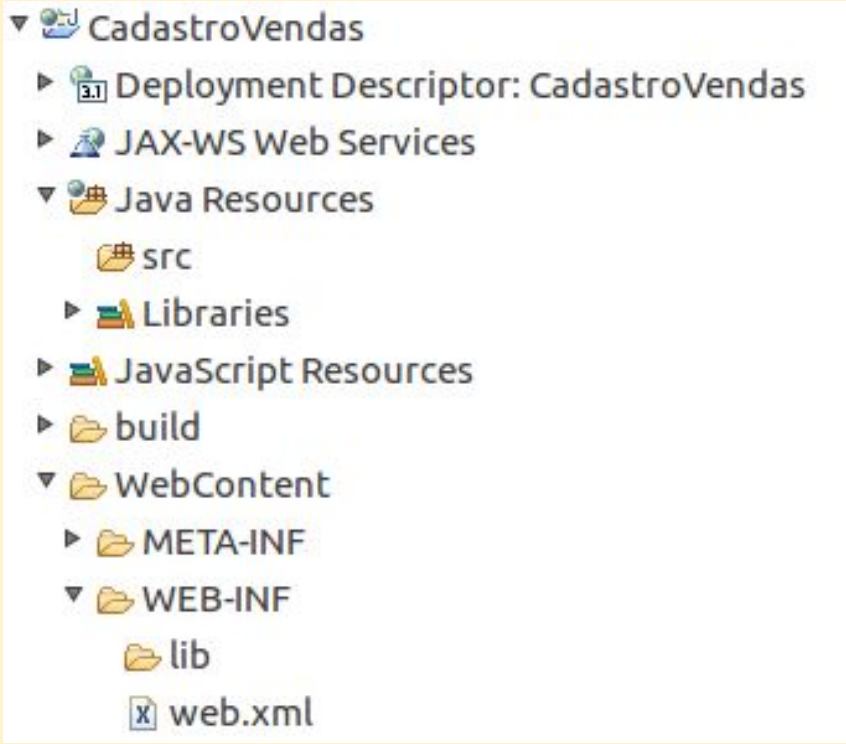
Mapeando tipos enumerados: @Enumerated

- Duas opções de salvar o valor:
  - ORDINAL: salva a ordem do valor
  - STRING: salva uma string com o valor

```
public enum SituacaoProduto {ATIVO, INATIVO};
...
@Enumerated(EnumType.STRING)
private SituacaoProduto situacao;
```

Prática 01: Criar e configurar um projeto no Eclipse

- Criar projeto ‘CadastroVendas’
- Eclipse JEE
  New > Project:
    Web > Dynamic Web Project
- **Pasta src: classes**
- **META-INF: persistence.xml**
- WebContent: jsp, html, css, js
- **WEB-INF/lib: bibliotecas**



Prática 01: Obter bibliotecas

- Como JPA é uma especificação, precisamos escolher uma implementação: **Hibernate (JBoss)**, Toplink (Oracle), OpenJPA (Apache)
- Site do Hibernate: http://hibernate.org/orm/ > Download
  - Versão atual (5.2.10)
- Descompactar o zip e adicionar as bibliotecas da pasta **required** ao projeto do eclipse
- Acrescentar o jar do driver do banco:
  - MySQL: mysql-connector-java-5.1.42-bin.jar (https://dev.mysql.com/downloads/connector/j/)

Prática 01: Arquivo de configuração

- JPA: persistence.xml (ou Hibernate: hibernate.cfg.xml)
- Local: src/META-INF/persistence.xml

```
<persistence-unit name="minicurso.vendas">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <properties>
        <property name="javax.persistence.jdbc.driver"
            value="com.mysql.jdbc.Driver" />
        <property name="javax.persistence.jdbc.url"
            value="jdbc:mysql://localhost:3306/vendas" />
        <property name="javax.persistence.jdbc.user" value="root" />
        <property name="javax.persistence.jdbc.password" value="1234" />
        <property name="dialect" value="org.hibernate.dialect.MySQLDialect" />
    </properties>

</persistence-unit>
```

Prática 01: Criar e mapear classes

1. Criar um pacote model
2. Criar classe Fornecedor com os atributos codigo, cnpj, razaoSocial
3. Adicionar Annotations
4. Criar um banco de dados no MySQL (nome=vendas)
5. Alterar o arquivo persistence.xml:

   javax.persistence.jdbc.url=jdbc:mysql://localhost:3306/vendas

Prática 01: Criar e mapear classes

1. Criar um pacote model
2. Criar classe Fornecedor com os atributos codigo, cnpj, razaoSocial
3. Adicionar Annotations
4. Criar um banco de dados no MySQL (nome=vendas)
5. Alterar o arquivo persistence.xml:

   javax.persistence.jdbc.url=jdbc:mysql://localhost:3306/vendas

Prática 01: Criar classe de teste

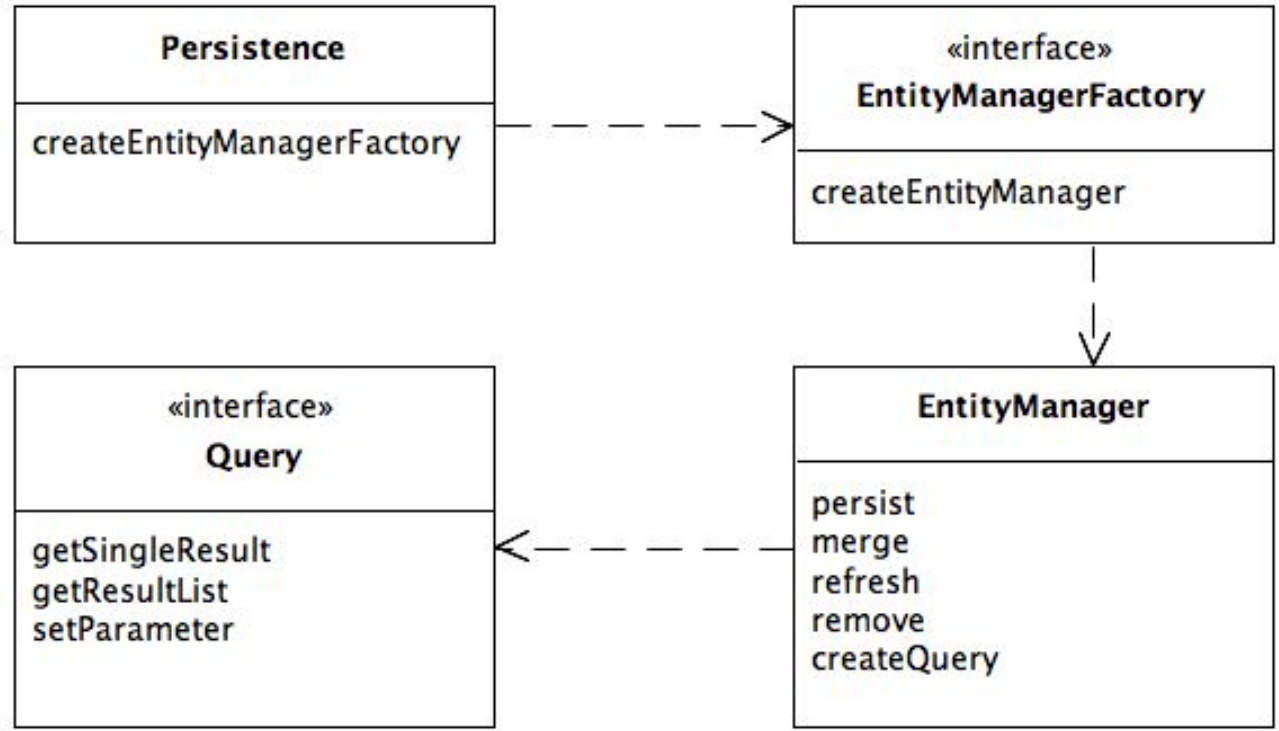
1. Criar uma classe Teste com o método main

```
public class Teste {

    public static void main(String[] args) {

        EntityManagerFactory emf = Persistence
            .createEntityManagerFactory("minicurso.vendas");
    }

}
```

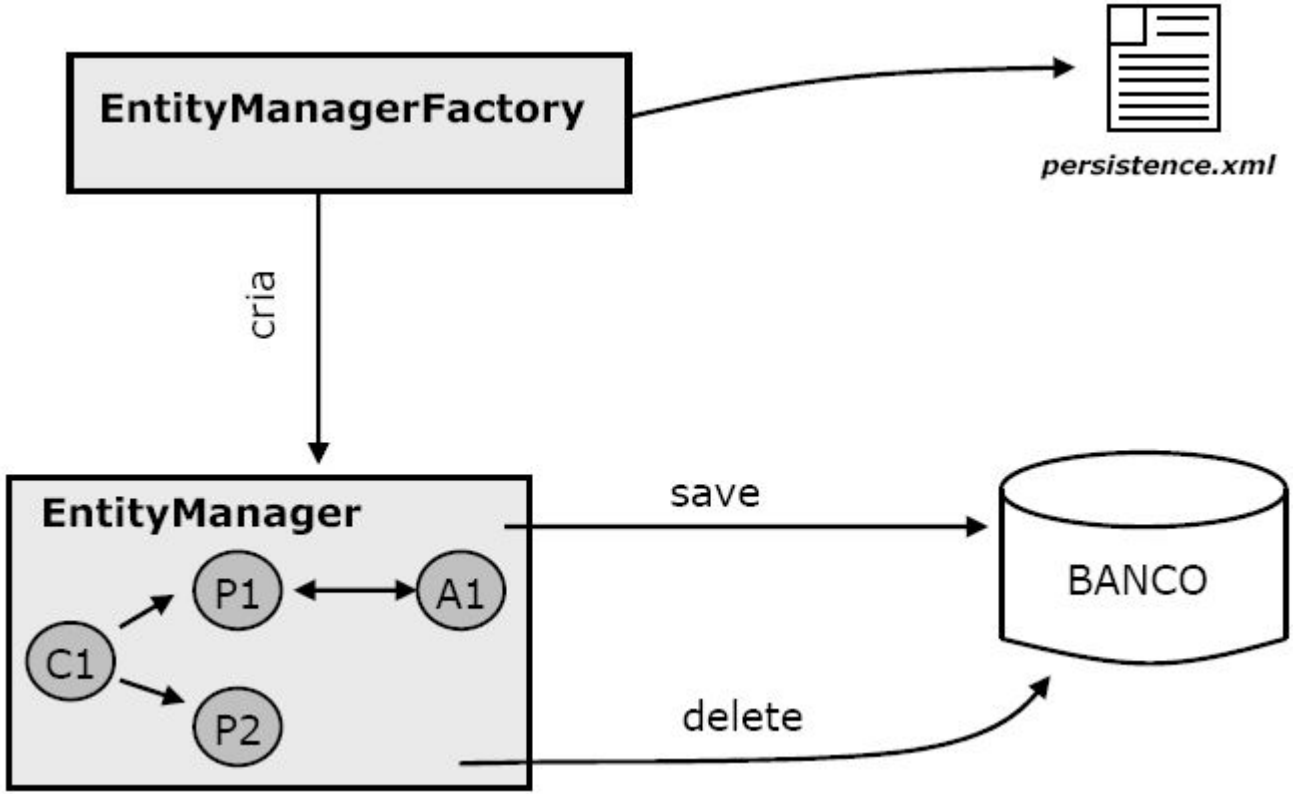
[2]
JPA / Hibernate

Operações sobre entidades

API Entity Manager

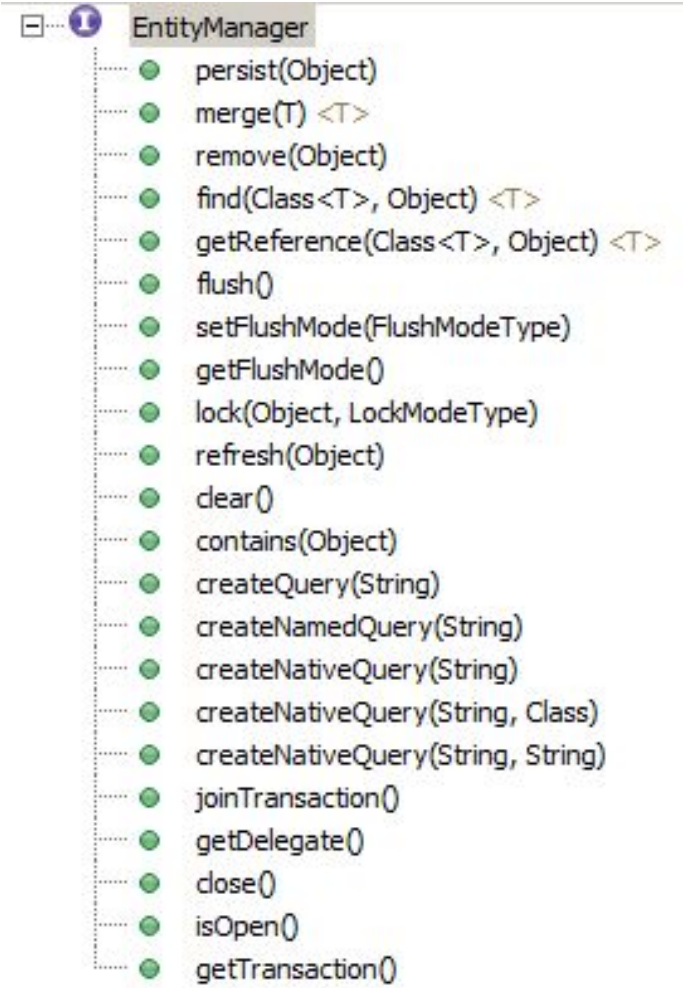


Criando EntityManager



Métodos do EntityManager

- EntityManager gerencia o ciclo de vida dos objetos que estão marcados com @Entity
- Métodos para salvar (***persist***), alterar (***merge***), consultar pela PK (***find***) e excluir (***remove***)
- Método para executar comandos JPQL (Java Persistence Query Language) - ***createQuery***
- Método para executar comandos nativos SQL (***createNativeQuery***)
- Operações que atualizam o banco devem usar uma transação (***getTransaction***)



EntityManager: salvar objeto

- void persist(T entity): salva o objeto no BD

```
// Instanciar objeto
Fornecedor f = new Fornecedor();
f.setCnpj("123123000123");
f.setRazaoSocial("Livraria Solemar");
// Salvar objeto
em = emf.createEntityManager();
em.getTransaction().begin();
em.persist(f);
em.getTransaction().commit();
```

EntityManager: consultar objeto

- T find (Class<T> classe, Object PK): encontra uma instância de entidade pelo identificador

```
// Consultar objeto
em = emf.createEntityManager();
Fornecedor f = em.find(Fornecedor.class, 1);
System.out.println(f.getRazaoSocial());
```

- Caso não encontre, o método find retornará **null**.
- T getReference(Class<T> classe, Object PK): idêntico ao find, porém levanta uma exceção EntiyNotFoundException na ausência do objeto.

EntityManager: alterar objeto

- T merge(T entity): atualiza o BD com os dados do objetos, retornando uma referência atualizada

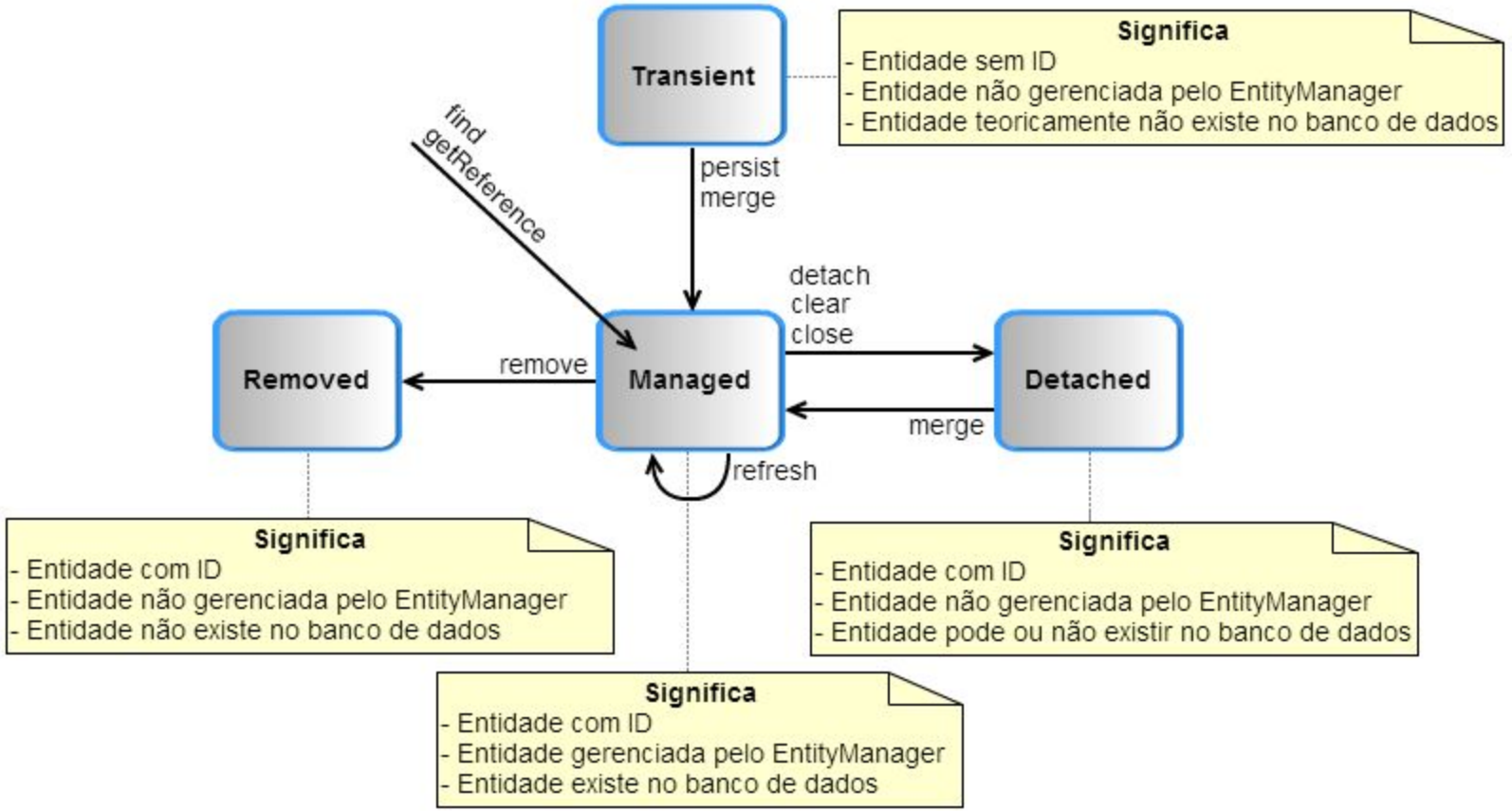
```
// Consultar objeto
em = emf.createEntityManager();
Fornecedor f = em.find(Fornecedor.class, 1);
System.out.println(f.getRazaoSocial());

f.setRazaoSocial("Livraria Sol e Mar");
em.getTransaction().begin();
f = em.merge(f);
em.getTransaction().commit();
```

EntityManager: remover objeto

- void remove(T entity): remove do BD uma entidade baseada na identidade do objeto

```
// Consultar objeto
em = emf.createEntityManager();
Fornecedor f = em.find(Fornecedor.class, 2);
System.out.println(f.getRazaoSocial());
// Remover objeto
em.getTransaction().begin();
em.remove(f);
em.getTransaction().commit();
```

Ciclo de vida das entidades



[2]
JPA / Hibernate

Mapeamento de Relacionamentos

Relacionamento entre entidades

1. Um para muitos: OneToMany
    - Um fornecedor pode entregar vários produtos
2. Muitos para um: ManyToOne
    - Vários produtos podem vir de um fornecedor
3. Um para um: OneToOne
    - Uma nota fiscal é de um pedido
4. Muitos para muitos: ManyToMany
    - Um produto pode estar em vários pedidos
    - Um pedido pode ter vários produtos

Associações 1:N <-> N:1

- **Muitos** produtos para **um** fornecedor
- Produto e Fornecedor devem ser @Entity
- Annotations
  - @ManyToOne: **obrigatória** sobre no atributo do relacionamento
  - @JoinColumn: opcional para definir a coluna da FK

```
public class Produto {

    @ManyToOne
    @JoinColumn(name="forn_id")
    private Fornecedor fornecedor;
```

Associação 1:1

- Um pedido tem uma nota fiscal
- Pedido e NotaFiscal devem ser @Entity
- Annotations:
  - @OneToOne: obrigatória, sobre o atributo do relacionamento
  - @JoinColumn: opcional, para definir o atributo da FK

```
@Entity
public class Pedido {

    @OneToOne
    @JoinColumn(name="ntfs_id")
    private NotaFiscal notaFiscal;
```

Associação N:N

- Um pedido tem vários produtos
- Pedido e Produto devem ser @Entity
- Annotation:
  - @ManyToMany: obrigatória, sobre o atributo do relacionamento (sempre coleção)
  - @JoinTable: opcional, para definir a tabela de junção

```
@Entity
public class Pedido {

    @ManyToMany
    private List<Produto> produtos;
```

Salvando entidades ManyToOne

- Objetos associados só precisam ter o identificador preenchido

```
// Instanciar objeto
Produto p = new Produto();
p.setNome("Livro Maestria, Roberto Greene");
Fornecedor f = new Fornecedor();
f.setCodigo(1);
p.setFornecedor(f);
// Salvar objeto
em = emf.createEntityManager();
em.getTransaction().begin();
em.persist(p);
em.getTransaction().commit();
```

Salvando entidades ManyToMany

```
// Instanciar objeto
Pedido ped = new Pedido();
ped.setCpf("10900239212");
ped.setNomeCliente("Maria Elizabeth");
ped.setSituacao(Situacao.PENDENTE_PAGAMENTO);
ped.setDataCompra(new Date());

em = emf.createEntityManager();

Produto p1 = em.find(Produto.class, 1);
Produto p2 = em.find(Produto.class, 2);
List<Produto> lista = new ArrayList<Produto>();
lista.add(p1);
lista.add(p2);

ped.setProdutos(lista);

// Salvar objeto
em.getTransaction().begin();
em.persist(ped);
em.getTransaction().commit();
```

[2]
JPA / Hibernate

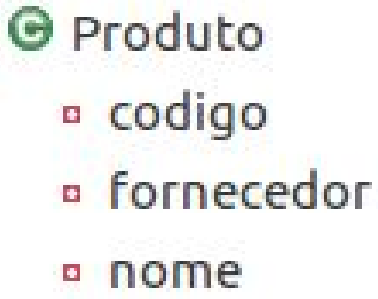
JPQL / HQL

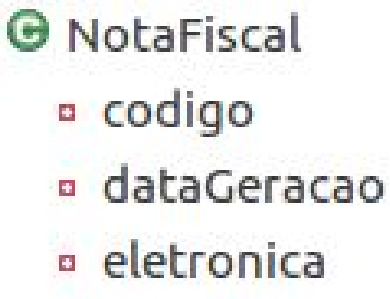
API de Consulta

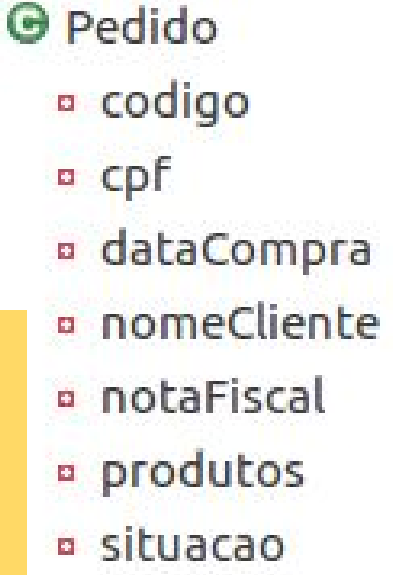
- Java Persistence Query Language (JPQL) / Hibernate Query Language (HQL)
- Permite a criação de consultas usando o modelo de entidades
- Nomes das classes e dos atributos serão usados nas consultas (case-sensitive)
- Agrega todo conhecimento que já temos de SQL
- Possibilita a criação de consultas dinâmicas

Principais classes: Query, TypedQuery

Nosso caso de estudo



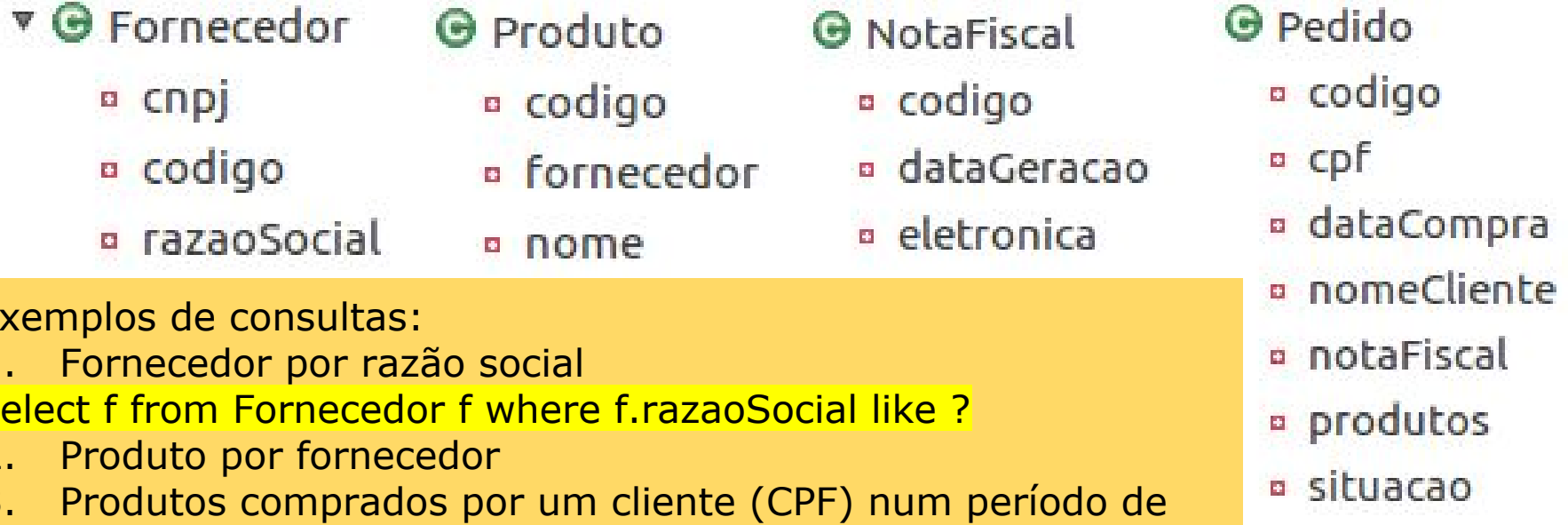




Exemplos de consultas:

1. Fornecedor por razão social
2. Produto por fornecedor
3. Produtos comprados por um cliente (CPF) num período de datas, ordenados pela situacao
4. Notas fiscais eletrônicas de pedidos que tenham produtos de um fornecedor (CNPJ)
5. Pedidos com mais de 5 produtos realizados no mês de junho

Nosso caso de estudo



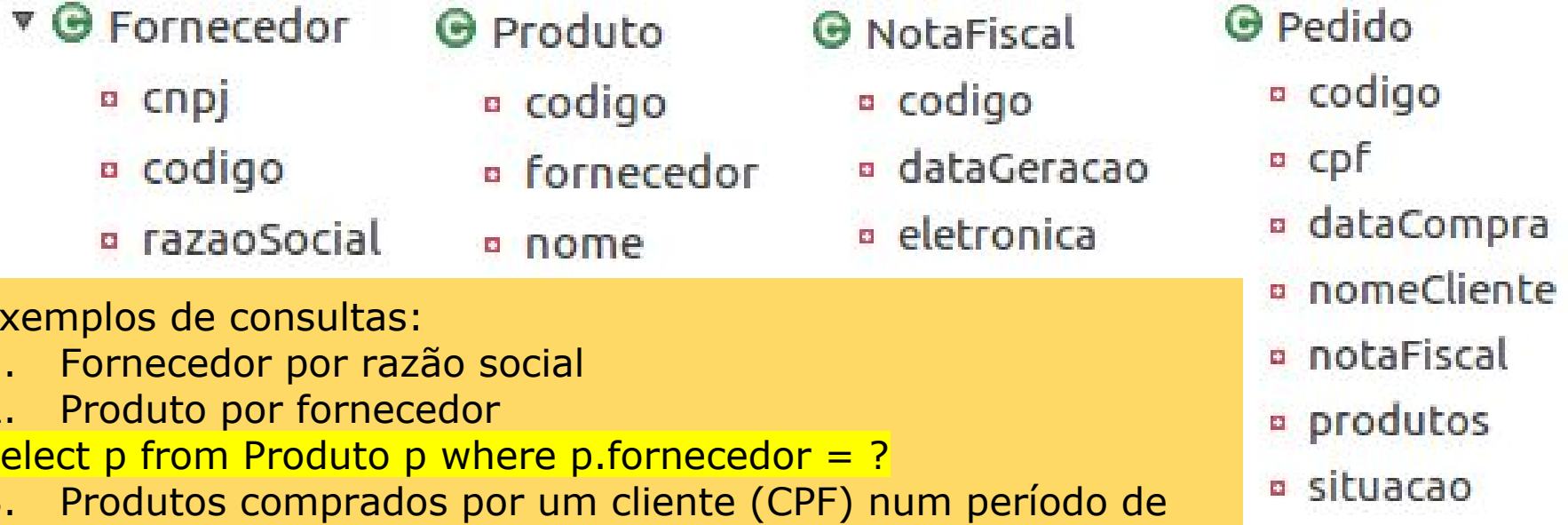
Exemplos de consultas:

1. Fornecedor por razão social

Select f from Fornecedor f where f.razaoSocial like ?

2. Produto por fornecedor
3. Produtos comprados por um cliente (CPF) num período de datas, ordenados pela situacao
4. Notas fiscais eletrônicas de pedidos que tenham produtos de um fornecedor (CNPJ)
5. Pedidos com mais de 5 produtos realizados no mês de junho

Nosso caso de estudo



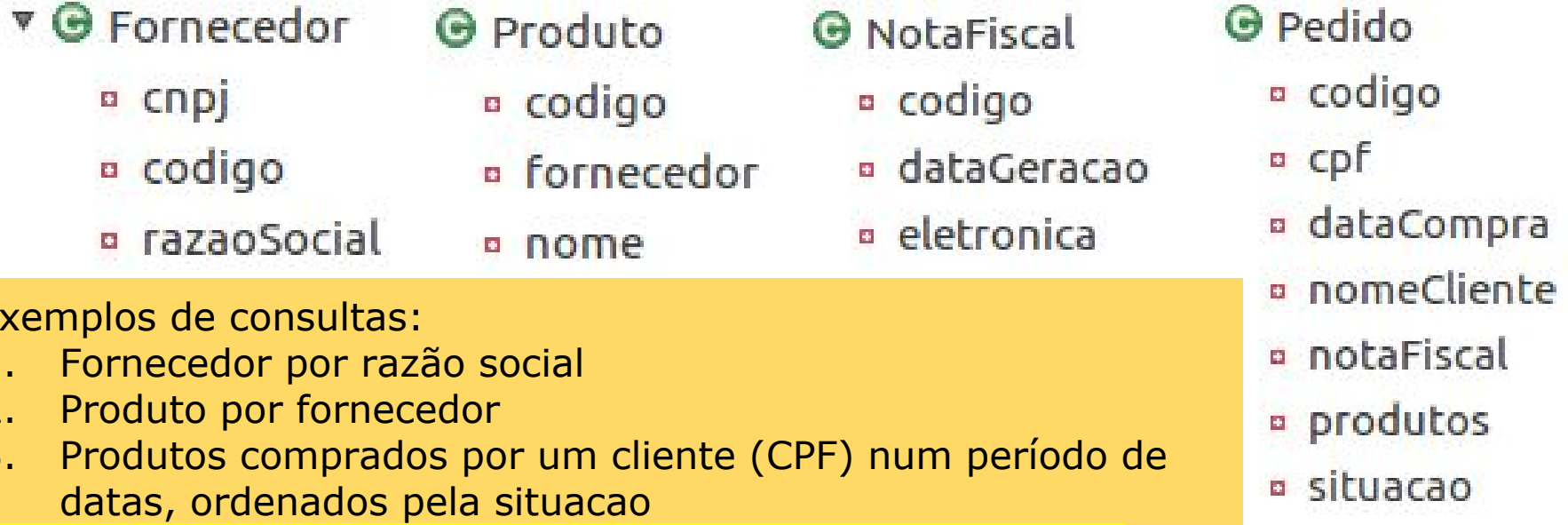
Exemplos de consultas:

1. Fornecedor por razão social
2. Produto por fornecedor

Select p from Produto p where p.fornecedor = ?

3. Produtos comprados por um cliente (CPF) num período de datas, ordenados pela situacao
4. Notas fiscais eletrônicas de pedidos que tenham produtos de um fornecedor (CNPJ)
5. Pedidos com mais de 5 produtos realizados no mês de junho

Nosso caso de estudo



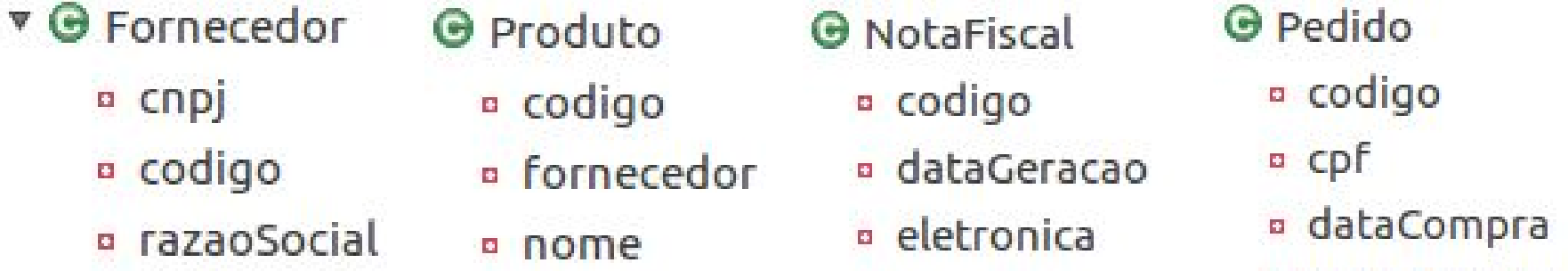
Exemplos de consultas:

1. Fornecedor por razão social
2. Produto por fornecedor
3. Produtos comprados por um cliente (CPF) num período de datas, ordenados pela situacao

Select ped.produtos from Pedido ped where ped.cpf = ? and ped.dataCompra between ? and ?

4. Notas fiscais eletrônicas de pedidos que tenham produtos de um fornecedor (CNPJ)
5. Pedidos com mais de 5 produtos realizados no mês de junho

Nosso caso de estudo



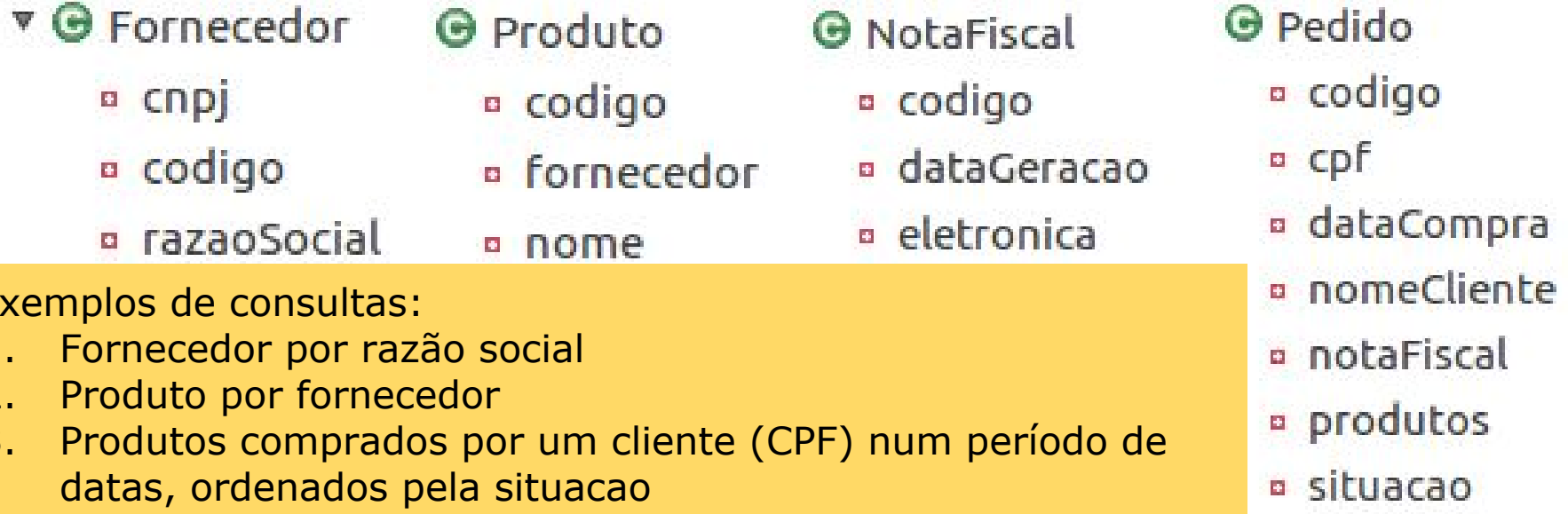
Exemplos de consultas:

1. Fornecedor por razão social
2. Produto por fornecedor
3. Produtos comprados por um cliente (CPF) num período de datas, ordenados pela situacao
4. Notas fiscais eletrônicas de pedidos que tenham produtos de um fornecedor (CNPJ)

Select p.notaFiscal from Pedido p join p.produtos pr where p.notaFiscal.eletronico = ? and pr.fornecedor.cnpj = ?

5. Pedidos com mais de 5 produtos realizados no mês de junho

Nosso caso de estudo



Exemplos de consultas:

1. Fornecedor por razão social
2. Produto por fornecedor
3. Produtos comprados por um cliente (CPF) num período de datas, ordenados pela situacao
4. Notas fiscais eletrônicas de pedidos que tenham produtos de um fornecedor (CNPJ)
5. Pedidos com mais de 5 produtos realizados no mês de junho

```
Select p from Pedido p where size(p.produtos) > 5 and
month(p.dataCompra) = 6
```

Executando as consultas com JPQL

- Query: retorna resultados genéricos (Object)

```
Query q0 = em.createQuery("from Fornecedor f");
List resultados = q0.getResultList();
for (Object object : resultados) {
    System.out.println(object);
}
```

- TypedQuery: retorna resultados de um tipo específico

```
TypedQuery<Fornecedor> q1 =
    em.createQuery("from Fornecedor f", Fornecedor.class);
List<Fornecedor> resultados = q1.getResultList();
for (Fornecedor f : resultados) {
    System.out.println(f.getRazaoSocial());
}
```

JPQL: Parâmetros

Duas formas de adicionar parâmetros numa Query

1) **Nominal:** Dois-pontos seguidos de um identificador (case-sensitive)

```
TypedQuery<Produto> q1 =
    em.createQuery("select p from Produto p where p.nome like :n",
        Produto.class);
q1.setParameter("n", "%" + "Java" + "%");
List<Produto> resultados = q1.getResultList();
for (Produto p : resultados) {
    System.out.println(p.getCodigo() + "-" + p.getNome());
}
```

JPQL: Parâmetros

Duas formas de adicionar parâmetros numa Query

2) Ordinal: Interrogação seguida de um número (iniciando de 1)

```
TypedQuery<Produto> q1 =
    em.createQuery("select p from Produto p where p.nome like ?1"
            + " and p.fornecedor.codigo = ?2",
        Produto.class);
q1.setParameter(1, "%" + "Java" + "%");
q1.setParameter(2, 1);
List<Produto> resultados = q1.getResultList();
for (Produto p : resultados) {
    System.out.println(p.getCodigo() + "-" + p.getNome());
}
```

Métodos de execução

- **List getResultList():** usado para executar um SELECT e retornar uma coleção
- **Object getSingleResult():** usado para executar um SELECT e retornar um único objeto como resultado
  - *Exceções:* ***NoResultExcetion*** *(sem resultados),* ***NonUniqueResultException*** *(mais de um)*
- **int executeUpdate():** usado para executar comandos de atualização - UPDATE, DELETE, INSERT e retornar a quantidade de linhas alteradas

Fora do escopo, porém importante

1) Mapear classes com herança e consultas polimórficas
2) Mapear relacionamentos N:N com atributos
3) Detalhes de definição de colunas simples, colunas de junção e tabelas de junção
4) JPQL: Paginação, agrupamento, NamedQuery

Referências

- **Código no Github**
  https://github.com/francisco-nascimento/hibernate-vendas.git
- **Apostilas:**
  - **Caelum:**
    https://www.caelum.com.br/apostila-java-web/uma-introducao-pratica-ao-jpa-com-hibernate/
  - **K19:**
    http://dkapostilas.blogspot.com.br/2014/10/apostila-de-persistencia-com-jpa-2-e.html
- **Livros:**
  - EJB3 em Ação - Debu Panda, Reza Rahman, Derek Lane
  - Java Persistence with Hibernate - Christian Bauer, Gavin King

Persistindo com Hibernate

Francisco do Nascimento (IFPE)
francisco.junior@jaboatao.ifpe.edu.br

UERN, JULHO/2017