Java Persistence API

- Entity
  - Entity – Campos e Propriedades
  - Entity – Chaves Primárias
  - Entity – Associações

- Entity Manager
  - Entity Manager API
  - Java Persistence Query Language (JPQL)
  - Persistence Units

Java Persistence API

- JPA é uma *framework* JAVA para a persistência de dados em modelos relacionais

- JPA permite mapear objetos nas tabelas de uma base de dados relacional

- A tecnologia tanto pode ser usada no âmbito JEE, como num ambiente *Standard Edition* (JSE) fora do contexto de um servidor aplicacional

- JPA engloba três áreas distintas:
  - A API propriamente dita definida no pacote *javax.persistence*
  - A linguagem JPQL (Java Persistence Query Language)
  - Os meta-dados (anotações ou XML) utilizados no mapeamento O/R

Java Persistence API (2)

- JPA é apenas uma especificação. É necessário um *persistence provider* que forneça uma implementação:
  - Hibernate
  - Oracle Toplink
  - Apache OpenJPA
  - Eclipselink

Entity (1)

Conceito *de Entity*

- Classe Java (POJO) que representa uma tabela numa base de dados relacional

- As instâncias da classe representam linhas numa tabela

- O mapeamento O/R baseia-se nas anotações aplicadas sobre os campos ou sobre as propriedades da *entity*

Entity (2)

Requisitos

- **Construtor** público sem argumentos (pode ter outros construtores)
- Anotação **@javax.persistence.*Entity***
- Identificador/**Chave primária** (*@Id ou @EmbeddedId*)
- Tipos permitidos nos campos e propriedades:
  - Primitivos Java e seus Wrappers, Strings, Date/Time e Math
  - Collections, Sets, Maps, Lists , Arrays e tipos Enumerados
  - Tipos definidos pelo utilizador desde que Serializáveis
  - Outras Entities

Entity (3)

Fields* vs. *Properties

- As anotações podem ser opcionalmente aplicadas:

-Diretamente sobre as variáveis da instância (*persistent fields*)

```
@Basic
private String name;
```

- Aos *getters* que seguem as convenções JavaBeans (*persistent properties*)

```
@Basic
public String getName() {
 return this.name;
}
```

Entity (4)

Exemplos de anotações

- @Id – identifica a chave primária

- @Basic – mapeamento por omissão; aplicável aos tipos primitivos, enumerados, *serializable*, etc.

- @Transient – indica que o valor não deve ser persistido

- @Table(name="TABLE_NAME") – por omissão o nome da tabela é o nome da classe

- @Column(name="COLUMN_NAME") – por omissão o nome da coluna é do campo

Entity – Definição da Chave Primária

Tipos de Chaves Primárias

- Automática

```
@Entity
public class Example {
    @ Id
    @GeneratedValue
    long id;
```

- Inicializada pela aplicação

```
@Entity
public class Example {
    @ Id
    long id;
```

Entity – Definição da Chave Primária

Tipos de Chaves Primárias (2)

- Composta

```
@Entity
public class Example {
    @ EmbeddedId
    ExampleId  id;
```

```
@Embeddable
public class ExampleId {
      int departmentId;
      long projectId;
}
```

Validation Constraints

- Bean validation é implementada através de um conjunto de anotações do javax.validation package

- Bean Validation Constraints podem ser aplicadas aos campos ou propriedades de uma classe persistente

- Exemplos:

```
@NotNull
private String firstName;   // campo  obrigatório

@Pattern(regexp="^\\(?(\\d{3})\\)?[- ]?(\\d{3})[- ]?(\\d{4})$",
message="{invalid.phonenumber}")
private String mobilePhone;                    // formato (xxx) xxx–xxxx
```

Entity – Associação

- **Multiplicidade**

  - Um para Um (@OneToOne)
  - Vários para Um (@ManyToOne)
  - Um para Vários (@OneToMany)
  - Vários para Vários (@ManyToMany)

- **Direção da associação**
- Bidirecional

  A associação tem um “owning side “ e um “inverse side”

- Numa associação bidirecional, cada entidade tem um campo que refere a outra entidade

- Unidirecional

  Tem apenas um “owning side”

Entity – Associação (2)

- O lado “inverse” de uma associação bidirecional deve referir qual o seu “owning side” através do elemento mappedBy das anotações @OneToOne, @OneToMany ou @ManyToNany

- O elemento mappedBy designa o campo na entity que é o owner da associação.

- O lado Many de uma associação bidirecional Many-To-One não deve definir o elemento mappedBy. O lado Many é sempre o owner da associação

- Para associações one-to-one bidirecionais o owner corresponde ao lado que contém a chave estrangeira.

- Para associações bidirecionais many-to-many ambos os lados podem ser o owner

Entity – Associação (3)

- Exemplo:

- Um Cliente tem vários endereços, e a um endereço corresponde um Cliente. Associação um-para-vários,

- Um cliente sabe que endereços tem e dado um endereço sabe-se a que cliente pertence. Associação Bidirecional

```
public class Customer implements Serializable {
  …
@OneToMany(mappedBy = "customerId")
private Collection<Address> addressCollection;
```

Coluna que contém a chave estrangeira

```
public class Address implements Serializable {
  …
 @JoinColumn(name = "CUSTOMER_ID",
 referencedColumnName = "CUSTOMER_ID")
 @ManyToOne
 private Customer customerId;
```

Coluna que contém a chave primária na tabela do lado One

Entity – Associação 1 para 1

- **One-To-One**

```
// On Customer class:

@OneToOne(optional=false)
@JoinColumn(
  name="CUSTREC_ID", unique=true, nullable=false, updatable=false)
public CustomerRecord getCustomerRecord() { return customerRecord; }

// On CustomerRecord class:

@OneToOne(optional=false, mappedBy="customerRecord")
public Customer getCustomer() { return customer; }
```

Entity – Associação 1 para N

- **One-To-Many, Many-To-One**

```
// In Customer class:

@OneToMany(cascade=ALL, mappedBy="customer")
public Set getOrders() { return orders; }

In Order class:

@ManyToOne
@JoinColumn(name="CUST_ID", nullable=false)
public Customer getCustomer() { return customer; }
```

Entity – Associação N para N

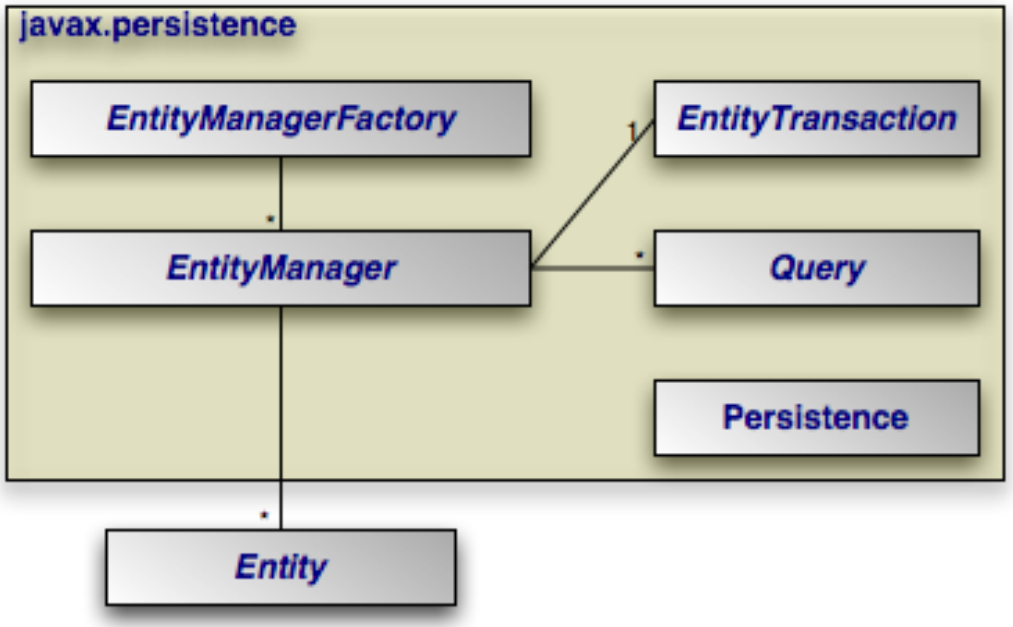
- **Many-To-Many**

```
// In Customer class:

@ManyToMany
@JoinTable(name="CUST_PHONES")
public Set getPhones() { return phones; }

// In PhoneNumber class:

@ManyToMany(mappedBy="phones")
public Set getCustomers() { return customers; }
```

Arquitetura JPA

- As *Entities* representam os dados e a forma como são mapeados.
- É ainda necessária toda uma infraestrutura para fazer a sua gestão.

Arquitectura JPA



Arquitetura JPA (2)

EntityManagerFactory

- Utilizada para criar um *EntityManager*.
- Os application servers geralmente automatizam este passo, mas a factory é utilizada para o programador gerir JPA fora do *container*.

EntityManager

- Mantém o conjunto de entities activas (usadas pela aplicação).
- Gere a interação com a base de dados e o mapeamento O/R.
- Uma aplicação obtém um *EntityManager* por *injection*, Java lookup (application server) ou através da *EntityManagerFactory*.

Arquitetura JPA (3)

EntityTransaction

- Cada *EntityManager* tem associado uma *EntityTransaction.*

- Permite agrupar várias operações sobre os dados em unidades que falham ou são bem sucedidas como um todo (i.e. transações) .

- Visa assegurar a integridade dos dados.

Persistence Units

- O Entity Manager necessita de um contexto para operar: Classes geridas, definições de acesso à base de dados, entre outros tipos de configurações;

- O contexto é fornecido por uma ***Persistence Unit*** identificada por um nome e definida num ficheiro ***persistence.xml*** colocado na diretoria META-INF

- Exemplo de *persistence.xml*

```
<persistence>
    <persistence-unit name="OrderManagement">
        <description>This unit manages orders and customers.
            It does not rely on any vendor-specific features and can
            therefore be deployed to any persistence provider.
        </description>
        <jta-data-source>jdbc/MyOrderDB</jta-data-source>
        <jar-file>MyOrderApp.jar</jar-file>
        <class>com.widgets.Order</class>
        <class>com.widgets.Customer</class>
    </persistence-unit>
</persistence>
```

Persistence Units (2)

- Um Entity Manager é associado ao contexto através da anotação:

  @PersistenceContext (unitName=“OrderManagement”)

- A *jta-data-source* referenciada é definida algures no servidor aplicacional (e.g. ficheiro XML). Contém além de um JNDI name, diversas definições de acesso à base de dados:
  - User
  - Localização da BD
  - Driver JDBC
  - Nº máximo de ligações à BD
  - etc.

Entity Manager

- Fornece uma API que fornece as operações CRUD (create, read, update, delete) sobre o modelo de dados

- O Entity Manager pode ser gerido pela aplicação ou pelo container JEE, neste último pode ser injetado e inicializado através da anotação @PersistenceContext;

```
@PersistenceContext
EntityManager em;
...
```

Entity Manager (2)

- Persistir um objeto;

```
public LineItem createLineItem(Order order, Product product,
        int quantity) {
    LineItem li = new LineItem(order, product, quantity);
    order.getLineItems().add(li);
    em.persist(li);
    return li;
}
```

- Procurar por chave e remover um objeto;

```
public void removeOrder(Integer orderId) {
    try {
        Order order = em.find(Order.class, orderId);
        em.remove(order);
    }...
```

Entity Manager - JPQL

- A pesquisa pela chave primária (EntityManager.*find*) constitui a *query* mais básica;

- O JPA define ainda uma linguagem genérica e orientada aos objectos (vs. tabelas) designada por JPQL (*Java Persistence Query Language*);
  - JPQL é traduzido para linguagem BD nativa (MySQL, PostgreSQL, PL/SQL, etc.);

- Invocando *createQuery* na entity manager é possível definir uma *javax.persistence.Query*

  - **Query** query = em.**createQuery**(String);

Entity Manager – JPQL (2)

- Sintaxe muito inspirada em SQL:
  - SELECT x FROM *Entity x* WHERE *conditionA* AND *conditionB*
  - DELETE FROM *Entity* x WHERE *conditionC*

- Parâmetros de entrada são definidos na *Query string*, pela sintaxe

  : *myParameter*

  Em que *myParameter* identifica o nome do parâmetro

- Os valores são atribuídos através do método da Query *setParameter*

  *E.g. query.**setParameter**(“myParameter”, myValue)*

Entity Manager – JPQL Sintaxe (1)

- O resultado de um SELECT pode ser extraído da Query por:
  - *getSingleResult()* – se esperarmos apenas 1 elemento; 0 ou >1 elementos lança exceção)
  - *getResultList ()* – resultado numa lista de 0 ou mais elementos

- O resultado de um DELETE ou UPDATE é ser executado por:
  - *executeUpdate()*

- Operadores usuais
  - *Comparação:* =, >, >=, <, <=, <>, BETWEEN, LIKE
  - Lógicos: NOT, AND, OR

Entity Manager – JPQL Sintaxe (2)

- Funções agregadoras
  - COUNT, MAX, MIN, AVG, SUM
- Manipulação de strings
  - CONCAT, SUBSTRING, TRIM
  - LOWER, UPPER
  - LENGTH
- Cláusulas principais
  - SELECT, DELETE, UPDATE
  - FROM, WHERE
  - JOIN
  - ORDER BY
  - GROUP BY

Entity Manager – JPQL (Exemplo)

```
public List findWithName(String name) {
return em.createQuery(
    "SELECT c FROM Customer c WHERE c.name LIKE :custName")
    .setParameter("custName", name)
    .setMaxResults(10)
    .getResultList();
}
```

- Notar no exemplo acima:
  - O método *createQuery* define uma query em JPQL
  - Customer é uma classe e não uma tabela (ainda que tenha o mesmo nome)
  - *custName* é um parâmetro que recebe o seu valor em *setParameter*
  - O número máximo de ocorrências é limitado pelo método *setMaxResults*
  - A query é executada com a invocação de *getResultList()*