Java Persistence API

JPA

- O sistema é desenvolvido na grande maioria utilizando somente objetos, sem referências específicas de cada banco de dados.
- Oferece uma linguagem semelhante ao SQL chamada JPA Query, que possibilita escrever consultas e instruções de persistência.
- Principais Características
  - Persistência de POJOs (Plain Old Java Objects)
  - Permite elaboração de domínios ricos (herança, polimorfismo, navegabilidade, multiplicidade, associações e composições, etc)
  - Acessos sob demanda (Lazy)
  - Suporte à annotations - sem arquivos de mapeamento
  - Permite conectar frameworks de persistência (ex: Hibernate)

Entidade

- Entidade não é uma nova abordagem adotada no JPA. Pelo contrário, esse termo já vem sendo utilizado há muito tempo e é utilizado em diversas linguagens de programação.
- Uma entidade continua sendo essencialmente um nome ou um grupo de estados relacionados compondo uma unidade simples, muitas vezes representada - como uma tabela.
- Uma entidade pode participar de relacionamentos com uma ou muitas outras entidades e, esses relacionamentos, podem ser de diversos tipos, conforme os conceitos de multiplicidade e navegabiliade definidos no paradigma de Orientação a Objetos.
- Numa aplicação orientada a objetos (utilizando ou não JPA), qualquer objeto definido pode ser uma entidade.

Entidade - Características

- Persistenciabilidade (persistabiliity):
  - Esta é a mais importante característica para considerar um objeto como uma Entidade: ele deve ser persistido/armazenado ou o ciclo de vida é apenas volátil? Além disso, o objeto deve ser capaz de ser persistido, ou seja, o seu estado (conteúdo de seus atributos) devem ser capazes de serem persistidos em um banco de dados, por exemplo.
- Identidade (identity):
  - Um objeto deve ser identificado no sistema, seja por um código ou por um estado. Quando um objeto é persistido em um banco de dados ele deve possuir uma forma de identificação, chamada de identifier. Este identificador é a chave (key) que identifica unicamente uma instância de um objeto no sistema, ou seja, com ela é possível recuperá-la.

Entidade - Características

- Transacionalidade (transactionality):
  - é a possibilidade de realizar transações (inserção, alteração, consulta e deleção) destes objetos em qualquer contexto.
- Granularidade (granularity):
  - capacidade de detalhar um objeto em uma série de atributos (um tipo primitivo nunca será uma entidade), a fim de que cada objeto representará uma linha de uma tabela e cada atributo, uma coluna, por exemplo. Um objeto muito grande, pode ser detalhado, talvez, através de outros objetos, e o JPA oferece suporte à isto.

Metadados das Entidades

- Uma entidade, entretanto, possui um metadados. Ou seja, cada entidade no sistema deve ter informações sobre os dados que estão armazenando, com o objetivo de guiar o JPA no mapeamento adequado dos Objetos vs. Banco de Dados Relacional.
- Os metadados são, basicamente, realizados de duas formas:
  - Arquivos XML: utilizados nas versões mais antigas
  - Annotations: utilizados a partir do JPA 2
- As annotations vem para simplificar a implementação, reduzindo a quantidade de arquivos de configuração e facilitando a visualização e leitura do código.

Configuração Inicial

- A configuração inicial do JPA consiste basicamente em:
- Configurar o persistence unit;
  - Instalar os jars do JPA + Persistence Provider (o Persistence Provider é o mecanismo de persistência que será acoplado ao JPA, no nosso caso, utilizaremos o Hibernate)
  - Usar um projeto Maven
- Configuração do Banco, incluindo:
  - Instalação do banco;
  - Driver JDBC do banco devidamente instalado em WEB-INF/lib (ou no classpath)
  - Setup do usuário e base de dados
  - Ajuste das propriedades no persistence.xml

Definindo uma PersistenceUnit

- Uma persistence unit é definida a partir de um arquivo XML chamado de persistence.xml que deverá ser salvo dentro da pasta META-INF contida no SourceFolder.
- O conteúdo deste arquivo é o seguinte:

```
<persistence 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"
        version="2.0">
  <persistence-unit name="EmployeeService"
            transaction-type="RESOURCE_LOCAL">
    <properties>
      <property name="javax.persistence.jdbc.driver"
              value="com.mysql.jdbc.Driver" />
       <property name="javax.persistence.jdbc.url“
          value="jdbc:mysql://localhost:3306/dbprojetos?createDatabaseIfNotExist=true"/>
<property name="javax.persistence.jdbc.user" value="APP"/>
       <property name="javax.persistence.jdbc.password" value="APP"/>
    </properties>
  </persistence-unit>
</persistence>
```

Construindo uma entidade

- Qualquer classe Java (não abstrata) pode ser uma Entidade. Porém, somente as classes que geram objetos com as características citadas acima realmente fazem sentidos serem implementadas como Entidades.
- Considerando este requisito fundamental, basta utilizar uma série de annotations para construir uma Entidade a partir de uma classe Java.

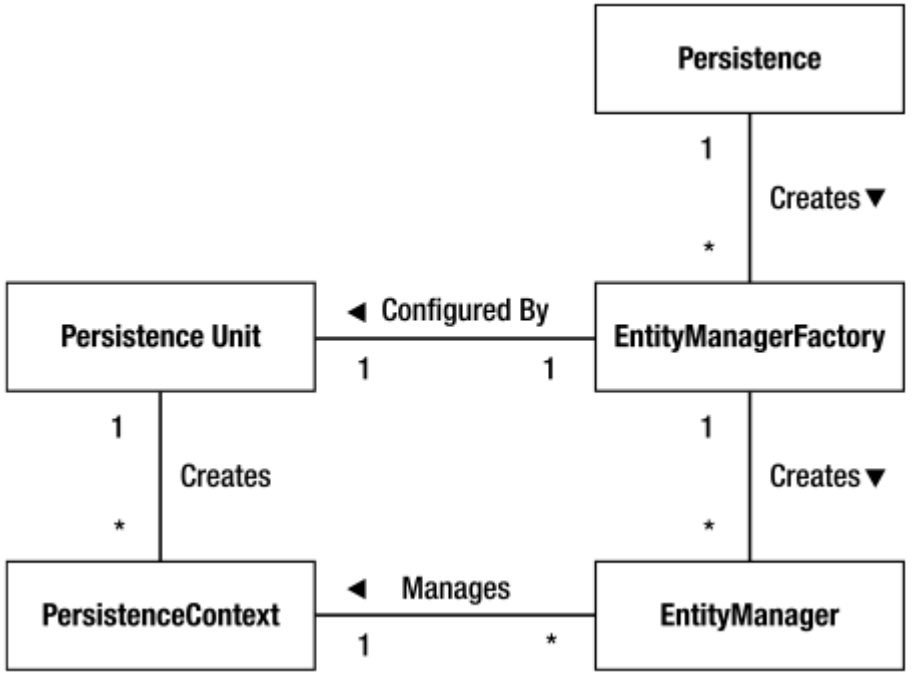
```
@Entity
 public class Employee {
        @Id
        private String name;
        private long salary;

        public Employee() {}
        public Employee(int id) { this.id = id; }
        public gets();
        public sets();
}
```

EntityManager

- O entity manager compreende o mecanismo que, como o próprio nome diz, gerencia as entidades do sistema. Ou seja, é responsável por realizar operações de inserção, busca, alteração e remoção de entidades.
- É importante destacar:
  - Um EntityManager é criado a partir de uma fábrica, chamada de EntityManagerFactory;
  - Uma factory pode criar vários entity manager;
  - Uma entity manager gerencia entidades, indentificando-a pelo Nome e verificando se a mesma está devidamente registrada no JPA.

Obtendo uma EntityManager

- O EntityManager é obtido através da fábrica, configurada por uma PersistenceUnit em um determinado contexto.
- O relacionamento entre os elementos do JPA, considerando, principalmente, a instância única do EntityManagerFactory e as n instâncias de EntityManager que podem ser criadas a partir desta fábrica.



Obtendo uma EntityManager

- Uma persistence Unit é identificada no sistema por um nome e é responsável por estabelecer uma unidade de persistência, ou seja:
  - possui informações de conexão
  - possui tipo do banco de dados
  - schema
  - outras informações
- Cada módulo pode ter uma unidade de persistência, porém, usualmente utilizamos somente 1 PersistenceUnit em sistemas mais simples.
- Para obter uma entity manager, usamos a factory e passamos qual é a persistence unit responsável.

```
EntityManagerFactory emf =
Persistence.createEntityManagerFactory("EmployeeService");

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

- Com a instância da EntityManager, é possível realizar operações com as Entidades

```
Employee emp = new Employee(158);
em.persist(emp);
```

Obtendo uma EntityManager

```
(....)
 public Employee createEmployee(int id, String name,
long salary) {
        Employee emp = new Employee(id);
        emp.setName(name);
        emp.setSalary(salary);
        em.persist(emp);
        return emp;
}
(....)
EntityManager em = ????
public void persistir(Employee emp) {
em.persist(emp);
}
```

EntityManager

- Buscar
- Uma vez que uma entidade encontra-se no banco de dados, é possível recuperá-lo utilizando o método find da EntityManager.

```
Employee emp = em.find(Employee.class, 158);
```

- Remover
- Após carregar o objeto no atual contexto, é possível removê-lo.

```
Employee emp = em.find(Employee.class, 158);
em.remove(emp);
```

EntityManager

- A partir de uma EntityManager, é possível criar transações. Uma transação é utilizada principalmente quando se altera um conjunto de entidades e o valor somente deve ser efetivamente persistido se, no final de todas as operações, nenhum erro for encontrado.
- Informando que uma transação é iniciada: em.getTransaction().begin();
- Informando que a transação foi realizada com sucesso e, portanto, os dados dever efetivamente ir para o banco de dados: em.getTransaction().commit();
- Informando que a transação toda deve ser cancelada: em.getTransaction().rollback();

```
em.getTransaction().begin();
try {
	createEmployee(158, "John Doe", 45000);
	em.getTransaction().commit();
}catch(Exception e) {
	em.getTransaction().rollback();
}
```

EntityManager

- Consultas (Queries)
- É possível escrever consultas com JPA utilizando uma linguagem semelhante ao SQL padrão Ansi, porém, deve ficar claro que:
  - Independentemente do fabricante do Banco de Dados, o SQL utilizado no JPA será o mesmo;
  - A consulta não é em uma tabela e sim em uma Entidade (classe do java);
  - O retorno geralmente é uma lista de Objetos. Este retorno depende do modo que o Select foi construido e, geralmente, a lista é de objetos do tipo da Entidade selecionada. Deve-se ter cuidado em Queries mais complexas.

```
TypedQuery<Employee> query = em.createQuery("SELECT e FROM
Employee e", Employee.class);
List<Employee> emps = query.getResultList();
```

Annotations do JPA

- Mapeamento de uma Tabela
- Annotation @Entity, informa ao JPA que uma determinada classe poderá ser manipulada por uma EntityManager,
  - Poderá ser persistida em um banco de dados definido pela PersistenceUnit.
- Por padrão, o JPA define o nome da tabela que será gerada no banco de dados em função do nome da classe.
  - Exemplo: se a classe se chamar Usuario, a tabela ser a "usuario". Muitas vezes, a base de dados já existe, ou ainda, o DBA prefere utilizar nomes personalizados para as tabelas, o que desconfiguraria o nome dos objetos em java, se o desenvolvedor optasse em seguir o padrão do DBA.

```
@Entity
@Table(name="TB_PESSOA")
public class Pessoa {
(...)
}
```

```
@Entity
@Table(name="TB_PESSOA",uniqueConstraints={@UniqueConstraint(columnNames="NAME"),
@UniqueConstraint(columnNames="CPF")})
public class Pessoa {
(...)
}
```

Annotations do JPA

- Annotation @Column
- É possível mapear o nome da coluna no banco de dados com um nome diferente do nome do atributo

```
(...)
@Column(name="str_nome"
private String nomeDoFuncionario
(...)
```

- Definição da coluna
- Permite que o desenvolvedor faça a definição da coluna de forma manual, de acordo com seu banco de dados. Isso deve ser evitado, visto que o JPA vem para ser um framework independetemente do banco de dados utilizado.

```
(...)
@Column(name="START_DATE", columnDefinition="DATE
DEFAULT SYSDATE")
private java.sql.Date startDate;
(...)
```

- Define a coluna no banco de dados com o tipo DATE e que o valor default será através do SYSDATE.

Annotations do JPA

- length / Tamanho
- Define o tamanho da coluna. O número de caracteres ou dígitos que serão utilizados para armazenar o valor.

```
(...)
@Column(length=10)
private int codigo;

@Column(length=60)
private String nome;
(...)
```

- nullable / pode ser nulo?
- Informa se o valor pode (ou não) ser nulo (vazio). Caso nullable=true, então o valor pode ser nulo. Caso contrário, é de preenchimento obrigatório.

```
(...)
@Column(nullable=true)
private int idade;

@Column(length=60, nullable=false)
private String nome;
(...)
```

Annotations do JPA

- Tipos Enum
- É comum que seja armazenado campos do tipo "Status", "Fase do projeto", etc, que utilizam em seu conteúdo um tipo de Enum.
- Poderia haver uma tabela no banco de dados com este tipo, porém, muitas vezes não se faz necessário e a melhor estratégia é realmente utilizar uma coluna simples contendo esta informação.
- Neste segundo caso, utilizamos o recurso do java chamado Enumeration.

```
public enum UsuarioStatus {
ATIVO, AGUARDANDO_CONFIRMACAO, INATIVO
}
```

- Esta enum será utilizada no atributo status da entidade Usuario. A única preocupação que se deve ter é como esse enum será persistido na base de dados:
  - ORDINAL: será um numero sequencial, ou seja ATIVO = 0 e INATIVO = 2; ou,
  - String: será armazenado como String "ATIVO", "AGUARDANDO_CONFIRMACAO" e "INATIVO".

```
@Entity
public class Usuario {
   @Id
   private int id;

   @Enumerated(EnumType.STRING)
   private UsuarioStatus status;
}
```

Visão geral de mapeamento com JPA

- Relacionamento com valores "Únicos"
- Um relacionamento onde um dos lados (roles) possui a cardinalidade (multiplicidade) 1(um) é chamado de relacionamento com valor único (single-valued association). Estes relacionamentos podem ser:
  - Muitos para um (many-to-one); e,
  - Um para um (one-to-one).
- Muitos para um
- O relacionamento Muitos para Um (many-to-one) indica que, dependendo da direcionalidade, em A teremos uma lista de B e em B, pode-se ter uma referência única para A.
- O JPA estabelece este relacionamento utilizando a annotation @ManyToOne e, de acordo com a direcionalidade, também a @OneToMany.
- A annotation @ManyToOne deverá ser utilizada na Entidade que terá apenas uma instância da outra relacionada a ela.

```
public class Aluno {
(...)
        @ManyToOne
        private Turma turma;
(...)
}
```

Visão geral de mapeamento com JPA

```
public class Aluno {
(...)
        @OneToOne
        @JoinColumn(name="armario_id")
        private Armario armario;
(...)
}
```

- Utilizando colunas de Junção
- É possível informar ao JPA qual é o nome da coluna que será gerada para armazenar o código (ID) da outra Entidade. No caso de não informar, por exemplo, seria criado uma coluna turma_id dentro da tabela Aluno.

```
public class Aluno {
(...)
        @ManyToOne
        @JoinColumn(name="cod_turma")
        private Turma turma;
(...)
}
```

```
public class Aluno {
(...)
        @OneToOne
        @JoinColumn(name="armario_id")
        private Armario armario;
(...)
}
```

Visão geral de mapeamento com JPA

- Utilizando colunas de Junção

```
public class Aluno {
(...)
        @OneToOne
        @JoinColumn(name="armario_id")
        private Armario armario;
(...)
}
```

```
public class Armario {
(...)
        @OneToOne
        @JoinColumn(mappedBy="armario")
        private Aluno aluno;
(...)
}
```

Visão geral de mapeamento com JPA

- Associação Muitos para Muitos
- Já é conhecido que um relacionamento muitos-para-muitos (many-to-many) necessita de tabelas entidades no banco de dados para fazer as associações de duas entidades que podem ter muitas instâncias da outra, ou vice-versa.
- O JPA oferece este tipo de mapeamento e gera automaticamente esta entidade responsável pela junção das entidades Origem e Destino.
- A annotation utilizada é a @ManyToMany. Esta annotation deverá ser colocada primeiramente na entidade "dona" do relacionamento e, em seguida - e se for necessário -, na entidade que faz o relacionamento invesrso, definindo o parâmetro mappedBy.

```
@Entity
public class Aluno {
(...)
    @ManyToMany
    private List<Curso> cursos;
(...)
}
```

```
@Entity
public class Curso {
(...)
    @ManyToMany(mappedBy="cursos")
    private List<Aluno> alunos;
(...)
}
```

Visão geral de mapeamento com JPA

- Ajustando a Tabela de Junção
- Um relacionamento muitos para muitos resulta numa terceira tabela responsável pelo relacionamento das chaves primárias das entidades Origem e Destino.
- Na tabela resultante deste relacionamento serão encontrados somente duas colunas: KEY de origem e KEY de destino.
- No caso de desejar fazer ajustes nesta tabela resultante, é possível utilizando a annotation @JoinTable, na classe "dona" do relacionamento.

```
@Entity
public class Aluno{
(...)
    @ManyToMany
    @JoinTable(name="alunos_curso",
      joinColumns=@JoinColumn(name="ALUNO_ID"),
      inverseJoinColumns=@JoinColumn(name="CURSO_ID"))
    private List<Curso> cursos;
(...)
```