

Introdução à Java Persistence API – JPA

Após algum tempo de estouro do Hibernate, surge o Java Persistence API, sendo muito utilizado na comunidade para a camada de Persistência.

O JPA é um framework utilizado na camada de persistência para o desenvolvedor ter uma maior produtividade, com impacto principal num modo para controlar a persistência dentro de Java. Pela primeira vez, desenvolvedores tem um modo "padrão" para mapear nossos objetos para os do Banco de Dados. Persistência é uma abstração de alto-nível sobre JDBC.

O Java Persistence API - JPA define um caminho para mapear *Plain Old Java Objects POJOs* para um banco de dados, estes *POJOs* são chamados de *beans* de entidade. Beans de Entidades são como qualquer outra classe Java, exceto que este tem que ser mapeado usando *Java Persistence Metadata*, para um banco de dados.

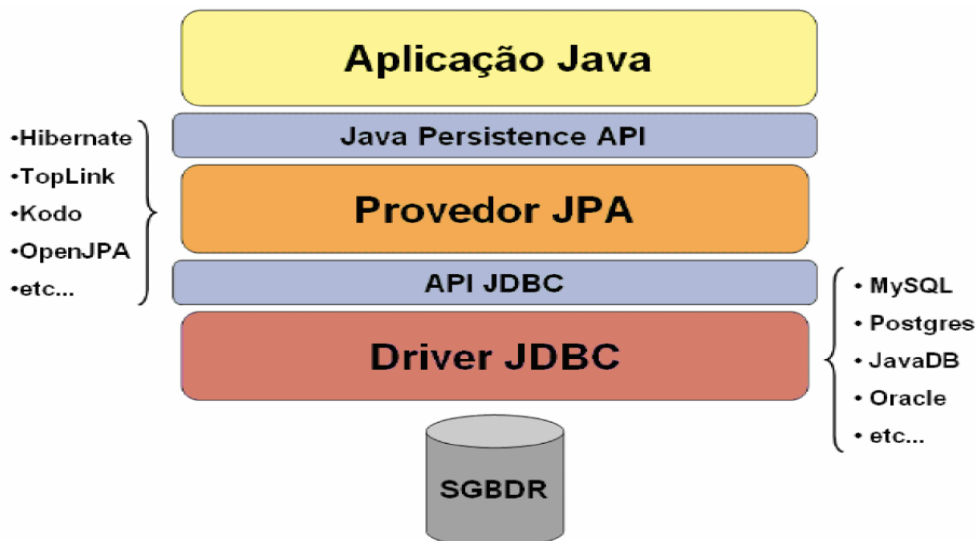
A nova *Java Persistence Specification* define mapeamento entre o objeto Java e o banco de dados utilizando ORM, de forma que Beans de entidade podem ser portados facilmente entre um fabricante a outro.

O que é ORM?

Em resumo, mapeamento objeto/relacional é automatizado (e transparente) persistência de objetos em aplicações Java para tabelas em um banco de dados relacional, usando metadata que descreve o mapeamento entre os objetos e o banco de dados.

Alguns Provedores JPA

Hibernate	http://jpa.hibernate.org
Toplink Essentials	http://oss.oracle.com/toplink-essentials-jpa.html
Open JPA	http://openjpa.apache.org



Objetos Persistentes, Transientes e *Detached*

Nas diversas aplicações existentes sempre que for necessário propagar o estado de um objeto que está em memória para o banco de dados ou vice-versa, há a necessidade de que a aplicação interaja com uma camada de persistência. Isto é feito, invocando o gerenciador de persistência e as interfaces de consultas do Hibernate. Quando interagindo com o mecanismo de persistência, é necessário para a aplicação ter conhecimento sobre os estados do ciclo de vida da persistência.

Em aplicações orientadas a objetos, a persistência permite que um objeto continue a existir mesmo após a destruição do processo que o criou. Na verdade, o que continua a existir é seu estado, já que pode ser armazenado em disco e então, no futuro, ser recriado em um novo objeto.

Em uma aplicação não há somente objetos persistentes, pode haver também objetos transientes.

Objetos transientes são aqueles que possuem um ciclo de vida limitado ao tempo de vida do processo que o instanciou. Em relação às classes persistentes, nem todas as suas instâncias possuem necessariamente um estado persistente. Elas também podem ter um estado transiente ou *detached*.

O Hibernate define estes três tipos de estados: persistentes, transientes e *detached*. Objetos com esses estados são definidos como a seguir:

- **Objetos Transientes:** são objetos que suas instâncias não estão nem estiveram associados a algum contexto persistente. Eles são instanciados, utilizados e após a sua destruição não podem ser reconstruídos automaticamente;
- **Objetos Persistentes:** são objetos que suas instâncias estão associadas a um contexto persistente, ou seja, tem uma identidade de banco de dados.
- **Objetos *detached*:** são objetos que tiveram suas instâncias associadas a um contexto persistente, mas que por algum motivo deixaram de ser associadas, por exemplo, por fechamento de sessão, finalização de sessão. São objetos em um estado intermediário, nem são transientes nem persistentes.

Todos as propriedades não transientes e não estáticas de um *entity bean* é considerado persistente, a menos que seja anotado como `@Transient`.

`@Basic` – permite declarar uma estratégia de recuperação de informação.

```
public transient int counter; //transient property

private String firstname; //persistent property

@Transient
String getLengthInMeter() { ... } //transient property
String getName() {... } // persistent property

@Basic
int getLength() { ... } // persistent property

@Basic(fetch = FetchType.LAZY)
String getDetailedComment() { ... } // persistent property

@Temporal(TemporalType.TIME)
java.util.Date getDepartureTime() { ... } // persistent property

@Enumerated(EnumType.STRING)
Starred getNote() { ... } //enum persisted as String in database
```

Atributos de Colunas

```
@Entity
public class Flight implements Serializable {
    ...
    @Column(updatable = false, name = "flight_name", nullable = false,
length=50)
    public String getName() { ... }

    @Column(
name="columnName"; (1)
boolean unique() default false; (2)
boolean nullable() default true; (3)
boolean insertable() default true; (4)
boolean updatable() default true; (5)
String columnDefinition() default ""; (6)
String table() default ""; (7)
int length() default 255; (8)
int precision() default 0; // decimal precision (9)
int scale() default 0; // decimal scale
```

Hibernate Annotation

Todas as ferramentas de mapeamento objeto relacional requerem um metadado que gerencia a transformação de dado de uma representação para outra.

No Hibernate 2.0 os metadados são declarados utilizando arquivos XML.

Pode-se utilizar XDoclet com anotações Javadoc com um compilador. O mesmo tipo de suporte a anotação está disponível no JDK padrão, porém mais poderoso e com suporte melhor às ferramentas.

A especificação EJB3 reconhece o interesse e o sucesso do paradigma transparente do mapeamento objeto/relacional.

Hibernate EntityManager implementa a interface de programa e as regras do ciclo de vida definido pela especificação de persistência EJB3 junto com Hibernate Annotations oferece uma solução completa de persistência EJB3 madura.

Mapeamento com EJB3/JPA Annotations.

Categoria de anotações

mapeamento lógico

Descrever o modelo, objeto, a associação entre classes

mapeamento físico

Descrever o schema físico, tabelas, colunas, índices.

Exemplo

```
import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
```

```

@Entity
@Table (name="estados")
public class Estado implements Serializable {
    Long ibge;
    @Id
    public Long getIbge() {
        return ibge;
    }
    public void setIbge (Long ibge) {
        this.ibge = ibge;
    }
}

```

Mapeamento com EJB3/JPA Annotations.

Categoria de anotações

mapeamento lógico

Descrever o modelo, objeto, a associação entre classes

mapeamento físico

Descrever o *schema* físico, tabelas, colunas, índices.

Exemplo

```

@Entity
@Table (name="estados")
public class Estado implements Serializable {
    Long ibge;

    @Id
    public Long getIbge {
        return ibge;
    }

    public void setIbge (Long ibge) {
        this.ibge = ibge;
    }
}

```

@Entity

declara que a classe como uma *entity bean* (classe de persistência POJO),

Especifica que uma classe é uma entidade

Uma entidade é um objeto que pode ser persistido

Representa uma tabela no banco de dados relacional

@Id declara o identificador do *entity bean*;

@Table define a tabela

Exemplo

```

@Table (name="tbl_sky",
        uniqueConstraints =
        {@UniqueConstraint (columnNames={"month", "day"})}
)

```

A restrição *unique* foi aplicada nas colunas *month*, *day*. O vetor *columnNames* refere-se ao nome

lógico da coluna.

@Column

Mapeia um atributo ou uma propriedade (*getter*) a um campo do banco de dados

Possui diversas opções de validação

Lança `javax.persistence.PersistenceException`

Versão do HibernateUtil para Annotations

```
import org.hibernate.*;
import org.hibernate.cfg.*;

public class HibernateUtil {
    private static final SessionFactory sessionFactory;
    static {
        try {
            sessionFactory = new
AnnotationConfiguration().configure().buildSessionFactory();
        } catch (Throwable ex) {
            // Log exception!
            throw new ExceptionInInitializerError(ex);
        }
    }
    public static Session getSession()
        throws HibernateException {
        return sessionFactory.openSession();
    }
    public static final ThreadLocal session = new ThreadLocal();

    public static Session currentSession() throws HibernateException {
        Session s = (Session) session.get();
        // Open a new Session, if this thread has none yet
        if (s == null) {
            s = sessionFactory.openSession();
            // Store it in the ThreadLocal variable
            session.set(s);
        }
        return s;
    }
    public static void closeSession() throws HibernateException {
        Session s = (Session) session.get();
        if (s != null)
            s.close();
        session.set(null);
    }
}
```

@Entity

declara que a classe como uma entity bean (classe de persistência POJO),

Especifica que uma classe é uma entidade

Uma entidade é um objeto que pode ser persistido

Representa uma tabela no banco de dados relacional

@Id

Declara o identificador do entity bean. Esta propriedade pode ser alterada pela aplicação ou gerada pelo Hibernate. Pode-se definir a estratégia de geração do identificador utilizando-se a anotação

@GeneratedValue.

- AUTO – Coluna identificadora, sequencia ou tabela dependendo do SGBD.
- TABLE – Tabela que hospeda do identificador.
- IDENTITY – Coluna identificadora.
- SEQUENCE – Sequencia.

Hibernate provê mais geradores de ID que a especificada pelo EJB3.

```
@Id
@GeneratedValue(strategy=GenerationType.SEQUENCE,
generator="SEQ_STORE")
public Integer getId() { ... }
```

O exemplo acima apresenta a sequencia geradora utilizando a configuração SEQ_STORE.

```
@Id
@GeneratedValue(strategy=GenerationType.IDENTITY)
public Long getId() { ... }
```

O exemplo acima usa a gerador Identity.

O gerador AUTO é preferido pelas aplicações portáteis entre os SGBD. A configuração do gerador de identificador pode ser dividido por inúmeros mapeamentos de @ID utilizando-se os atributos. Existem inúmeras configurações para @SequenceGenerator e @TableGenerator.

```
@Entity
@Table(name="endereco_auto")
public class Endereco implements Serializable {
    @Id
    @GeneratedValue
    @Column(name = "codigo_endereco")
    private Long endereco_id;
    private String tipoLogradouro;
    private String logradouro;
    private String bairro;
    private String cep;
}
```

@Table define a tabela

Exemplo

```
@Table(name="tbl_sky",
uniqueConstraints =
{@UniqueConstraint(columnNames={"month", "day"})})
```

A restrição unique foi aplicada nas colunas month, day. O vetor columnnames refere-se ao nome lógico da coluna.

@Column

Mapeia um atributo ou uma propriedade (*getter*) a um campo do banco de dados

Possui diversas opções de validação

Lança *javax.persistence.PersistenceException*

Todas as propriedades não transientes e não estáticas de um *entity bean* é considerado *persistent*, a menos que seja anotado como *@Transient*.

@Basic – permite declarar uma estratégia de recuperação de informação.

```
public transient int counter; //transient property

private String firstname; //persistent property

@Transient
```

```
String getLengthInMeter() { ... } //transient property
String getName() {... } // persistent property

@Basic
int getLength() { ... } // persistent property

@Basic(fetch = FetchType.LAZY)
String getDetailedComment() { ... } // persistent property

@Temporal(TemporalType.TIME)
java.util.Date getDepartureTime() { ... } // persistent property

@Enumerated(EnumType.STRING)
Starred getNote() { ... } //enum persisted as String in database
```

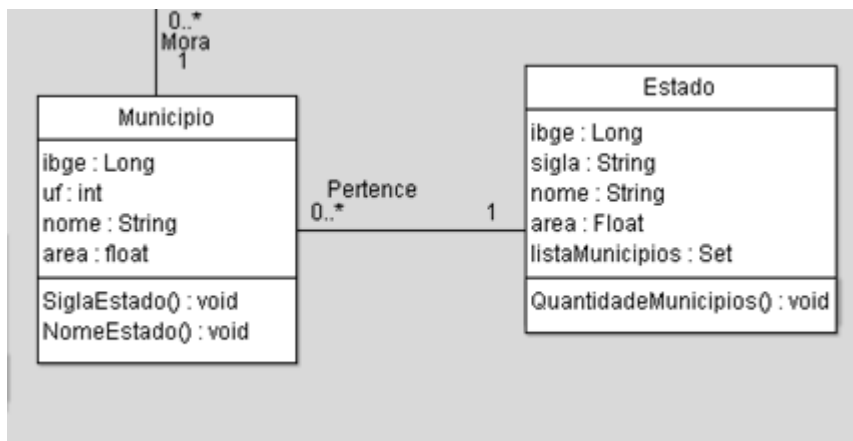
Atributos de Colunas

```
@Entity
public class Flight implements Serializable {
    ...
    @Column(uptdatable = false, name = "flight_name", nullable = false,
length=50)
    public String getName() { ... }

    @Column(
name="columnName"; (1)
boolean unique() default false; (2)
boolean nullable() default true; (3)
boolean insertable() default true; (4)
boolean uptdatable() default true; (5)
String columnDefinition() default ""; (6)
String table() default ""; (7)
int length() default 255; (8)
int precision() default 0; // decimal precision (9)
int scale() default 0; // decimal scale
```

Collection, List, Map, Set

Exemplo de mapeamento Estado x Município



```
@Entity
@Table (name="estados")
public class Estado implements Serializable {
```

```

    @OneToMany (mappedBy="estado")
    @OrderBy("nome")
    private Collection<Municipio> municipio ;
    public List getMunicipio() {
        return (List) municipio;
    }
    @Id
    Long ibge;
    private String nome;
    private Float area;
    ...
}

```

```

@Entity
@Table (name="municipios")
public class Municipio implements Serializable {
    @Id
    private Long ibge;
    private String nome;
    private Float area;

    @JoinColumn (name="uf")
    @ManyToOne
    private Estado estado;
    public Estado getEstado() {
        return estado;
    }
    ...
}

```

É possível declarar um componente embutido dentro da entidade. Classes componentes devem ser anotadas dentro da classe com a anotação `@Embeddable`. É possível sobrescrever o mapeamento da coluna de um objeto embutido para uma entidade particular utilizando as anotações `@Embedded` e `@AttributeOverride`.

(lazy, eager, and prefetching)

A associação pode ser bidirectional. Em uma relação bidirecional, um dos lados e somente um tem que ser o proprietário. O proprietário é responsável pela atualização da coluna de associação. Para declarar o lado como não responsável pela relação, o atributo `mappedBy` é utilizado. `mappedBy` refere-se ao nome da propriedade da associação do lado do proprietário.