JPA e Hibernate

Porque JPA? É uma questão fundamental para muitos desenvolvedores Java. Por que eu preciso saber como usar essa API quando as ferramentas de mapeamento Objeto-Relacional como o Hibernate e Toplink estão disponíveis? A resposta é que JPA não é uma nova tecnologia, mas sim, que tenha colhido as melhores ideias de tecnologias de persistência existentes, como Hibernate, TopLink, e JDO. O resultado é uma especificação padronizada que ajuda a construir uma camada de persistência que é independente de qualquer provedor de persistência particular. Java Persistence API baseia-se nas principais ideias dos frameworks de persistência principais e APIs como Hibernate, TopLink da Oracle e Java Data Objects (JDO), e assim como na primeira EJB persistência gerenciada por recipiente.

Versão 1.0

Apostila destinada ao curso com carga horária de 32 (trinta e duas) horas

Sumário

1. Java Persistence API (JPA)

A Java Persistence API oferece aos desenvolvedores Java a instalação de um objeto (ou mapeamento) relacional de gerenciamento de dados relacionais em aplicações Java. Java Persistence consiste em quatro áreas:

- A Java Persistence API
- A linguagem de consulta
- A Java Persistence API Criteria
- Objeto / Metadados de mapeamento relacional

Comparativo entre os modelos:

Desenvolvedores Java que precisam armazenar e recuperar dados persistentes possuem várias opções disponíveis: serialização, JDBC, JDO, Ferramentas Proprietárias Mapeamento Objeto-Relacional (ORM), Bancos de Dados Orientado a Objetos (ODB) e Beans de Entidade (EJB 2). Porquê usar outro framework de persistência? A resposta a esta questão é que, com a exceção de JDO, cada uma das soluções de persistência acima mencionados tem sérias limitações. JPA ultrapassa estas limitações, como ilustrado pela tabela abaixo:

Suporte	Serialização	JDBC	ORM	ODB	EJB 2	JDO	JPA
Objetos Java	Sim	Não	Sim	Sim	Sim	Sim	Sim
Conceitos avançados de OO	Sim	Não	Sim	Sim	Não	Sim	Sim
Integridade Transacional	Não	Sim	Sim	Sim	Sim	Sim	Sim
Concorrência	Não	Sim	Sim	Sim	Sim	Sim	Sim
Conjunto Largo de Dados	Não	Sim	Sim	Sim	Sim	Sim	Sim
Existência de Esquemas	Não	Sim	Sim	Não	Sim	Sim	Sim
Relacional e Não-Relacional	Não	Não	Não	Não	Sim	Sim	Não
Consultas	Não	Sim	Sim	Sim	Sim	Sim	Sim
Padrões Estritos / Portabilidade	Sim	Não	Não	Não	Sim	Sim	Sim
Simplicidade	Sim	Sim	Sim	Sim	Não	Sim	Sim

Características da APP

JPA é um *framework* leve baseado em POJO para mapeamento objeto-relacional. Anotações da linguagem Java ou metadados XML do descritor de implantação é usado para o mapeamento entre os objetos Java e um banco de dados relacional. Permite que a linguagem de consulta SQL funcione tanto de maneira estática como em consultas dinâmicas. Permite também que o uso da API persistência conectável. Java Persistence API são depende, principalmente, anotações de metadados. API inclui:

- Java Persistence API
- Anotações de metadados
- Java Persistence Query Language

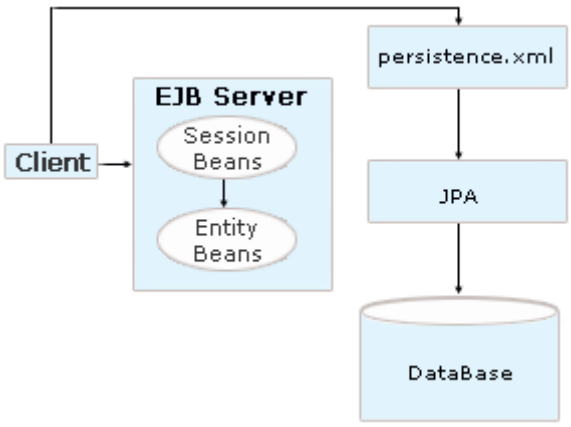
Vantagens da APP

JPA se aproveita das melhores ideias de tecnologias como a persistência TopLink, JDO e Hibernate. É compatível com o ambiente Java SE, bem como Java EE e permite aos tirar vantagens da API de persistência padrão.

Persistência de dados não é tão fácil para a maioria das aplicações empresariais, porque para isso necessitam do acesso ao banco de dados relacional, como por exemplo, **Oracle 10g**. É de sua responsabilidade atualizar e recuperar o banco de dados e escrever o código usando SQL e JDBC. Enquanto vários *frameworks* objeto-relacional (OR) como JBoss Hibernate e OracleTopLink fazem com que a persistência se torne mais simples e popular. Permitem que o desenvolvedor Java seja livre de escrever código JDBC e se concentre apenas na lógica de negócios. No EJB 2.x, era necessário Gerenciar a Persistência do Recipiente (CMP – *Container Manage Persistence*) para resolver os desafios de persistência, que nem sempre isso era completamente bem sucedido.

A camada de aplicação de persistência pode ser desenvolvida de várias maneiras, mas a plataforma Java não segue nenhum padrão que deve ser utilizado tanto pela plataforma Java EE e Java SE. A **JPA** parte da especificação EJB 3.0 (JSR-220) e realiza a persistência padrão API para a plataforma Java. Fornecedores de mapeamento O/R como o Hibernate e TopLink, bem como fornecedores JDO e outros fornecedores de servidores líderes de aplicação estão recebendo a JSR-220.

Processo de trabalho de uma aplicação EJB usando JPA:

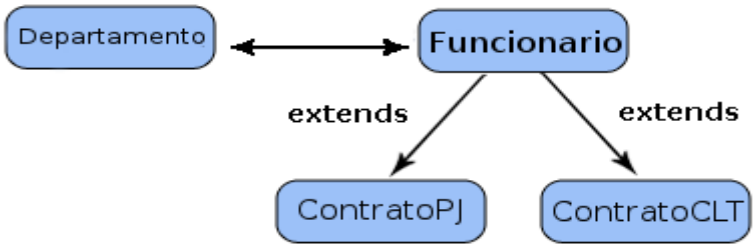


Modelo de Domínio

Ao desenvolver um aplicativo corporativo, o primeiro projeto do modelo de objeto de domínio

necessário para manter os registros no banco de dados. Modelo de domínio representa os objetos de persistência ou entidades no banco de dados. Uma entidade representa uma coleção de registros (Tabela). Uma entidade pode ser uma pessoa, lugar ou qualquer coisa sobre a qual deseja-se armazenar os dados no banco de dados. Um modelo de domínio rico inclui as características de todo o comportamento orientado a objetos como herança, polimorfismo e muitos mais.

Ao desenvolver um aplicativo corporativo, primeiro projeto do modelo de objeto de domínio para manter os dados no banco de dados, então, projetar o esquema de banco de dados com a ajuda do Administrador do Banco de Dados (DBA). A seguinte figura ilustra uma relação bi direcional de “um para muitos” entre a entidade **Funcionário** e **Departamento**. As entidades **ContratoPJ** e **ContratoCLT** são herdadas da entidade **Funcionário**.



Os princípios da JPA e *framework* de mapeamento O-R: Cada estrutura de mapeamento OR, como Oracle TopLink fornece três instalações:

- Define uma forma declarativa conhecida como mapeamento O-R de metadados para realizar o mapeamento O-R. A maioria do quadro de usar XML para armazenar metadados do mapeamento O-R.
- Uma API é necessário para manipular como para efetuar CRUD (CRUD significa criar, ler, atualizar e excluir) operações. A API permite persistir, remover, atualizar ou recuperar o objeto do banco de dados. O *framework* O-R executa operações usando a API e o mapeamento O-R de metadados em seu nome.
- O uso de uma linguagem de consulta para recuperar objetos a partir do banco de dados é a maneira apropriada, uma vez que as declarações impróprias SQL podem resultar em retardar o desempenho da operação de realização, no banco de dados. Uma linguagem de consulta permite recuperar as entidades do banco de dados e poupá-lo de escrever as instruções SQL SELECT.

JPA fornece um modo padrão de usar a persistência, fornecendo um mecanismo padrão ou mapeamento, uma maneira de estender EJB-QL para recuperar entidades e uma API *EntityManager* para executar operações CRUD.

JPA fornece POJO (Plain Old Java Object) padrão e mapeamento objeto relacional (ou mapeamento) para persistência de dados entre aplicações. Persistência, que trata de armazenamento e recuperação de dados do aplicativo, agora pode ser programado com a JPA a partir de EJB 3.0 como resultado da JSR 220. Um dos grandes benefícios da JPA é sua independência.

2. Design Patterns para Persistência

Acesso a dados é um assunto popular entre os desenvolvedores. Sem dúvida existem muitas opiniões sobre tecnologias específicas de acesso a dados e *frameworks* de persistência, respondendo as seguintes perguntas:

- Qual é a melhor maneira de usar essas ferramentas em seu projeto?
- Que critérios deve-se usar para selecionar a ferramenta certa para seu projeto?
- O que é necessário conhecer conceitualmente sobre essas ferramentas antes de usar?
- Se existe muito tempo para escrever sua própria ferramenta de persistência?

A resposta para todas estas perguntas é examinar os Design Patterns (Padrões de Projeto) subjacentes da persistência.

Modelos de Domínio

Quando se pensa qual o caminho para estruturar e representar a **Lógica de Negócios** em seu sistema, possuímos uma gama de escolhas. A partir da descrição formal, um **Modelo de Domínio** é um modelo de objeto do domínio que incorpora tanto o **comportamento** como os **dados**.

Por exemplo, o projeto atual envolve Customer Relationship Management (CRM). Os objetos de entidade contêm dados e implementa regras de negócios envolvendo esses dados. Um modelo de domínio pode variar em um modelo “anêmico” que é simplesmente um conjunto de estruturas de dados para um modelo “muito rico” que guarda zelosamente os dados brutos por trás uma interface restrita (Desenvolvimento Domain-Driven). Onde o seu modelo de domínio cai nessa faixa é em grande parte uma questão de quão complicada a lógica de negócios em seu sistema realmente é e como os relatórios prevalente ou a entrada de dados estão em seus requisitos de sistema.

Antes de começar, vamos rever as duas principais formas de perceber o papel do código de acesso de banco de dados e dados em seu sistema:

- O banco de dados é a pedra angular da aplicação e um ativo importante do negócio. O código de acesso a dados e até mesmo o código do aplicativo ou serviço são simplesmente mecanismos para conectar o banco de dados com o mundo exterior.
- Os objetos de negócios na camada intermediária e a interface de usuário ou camada de serviço são a aplicação e o banco de dados é um meio confiável para manter o estado dos objetos de negócios entre as sessões.

Normalmente que se está trabalhando com objetos de entidade na camada intermediária. De uma forma ou de outra, provavelmente se está fazendo Mapeamento Objeto Relacional (O/RM) para mapear os dados das entidades empresariais para as tabelas do banco de dados e vice-versa. Pode-se realizar este trabalho manualmente, porém é mais provável

com as ferramentas de algum tipo. Essas ferramentas normalmente são grandes e podem potencialmente salvar um monte de tempo de desenvolvimento, mas existem algumas questões que se deve estar ciente, e é sempre bom entender como uma ferramenta funciona debaixo das cobertas.

Design Pattern – DAO

O padrão **Data Object Access** (DAO) é um mecanismo amplamente aceito para abstrair os detalhes de persistência em um aplicativo. A ideia é que em vez da lógica do domínio se comunicar diretamente com o banco de dados, sistema de arquivos, *Web Service*, ou qualquer mecanismo de persistência o aplicativo usa, a lógica do domínio fala com uma camada DAO. Esta camada DAO comunica com o sistema de persistência subjacente ou serviço.

A vantagem da camada DAO é quando se deseja alterar o mecanismo de persistência subjacente é necessário somente mudar a camada de DAO, e não todos os lugares da lógica de domínio. A camada DAO geralmente consiste de um conjunto menor de classes, que o número de aulas de lógica de domínio que o utiliza. Ao necessitar alterar o que acontece por trás da camada DAO, a operação é um pouco menor. É também uma operação um pouco mais controlada, já que é possível procurar por todas as classes DAO, e verifique se eles são alterados para usar o novo mecanismo de persistência.

Por esta encapsulação do mecanismo de persistência subjacente a trabalhar, é importante que não há detalhes do mecanismo de persistência subjacente vazar para fora da camada de DAO. Garantir isso é, no entanto, um pouco de um desafio, como eu te mostrar no texto ao lado nesta trilha.

Design Pattern – PathProxy

É um padrão de projeto para persistência de relações complexas, sem ocupar o seu banco de dados. Use este padrão para resolver problemas de muitas entidades cujas inter-relações são complexas e exigem conhecimento de outros relacionamentos. Criação de objetos explícitos para representar estes tipos de relações torna-se onerosa.

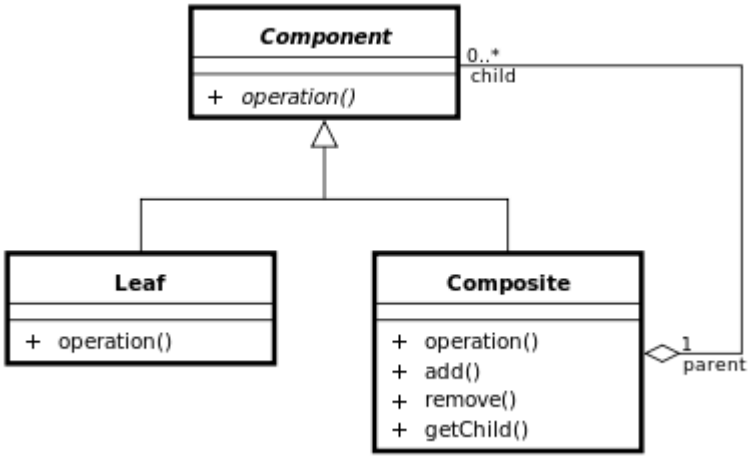
Isto é especialmente verdadeiro se os objetos devem ser persistentes, criando uma proliferação de tabelas de banco de dados. Se o projeto do sistema exige um número de classes, cuja única ou principal função é a de modelar as relações entre outros objetos.

Usar **PathProxy** é mais complicado do que usar objetos simples para representar as relações, por isso deve considerar a sua situação em particular. Se este possui poucos relacionamentos para armazenar e não são muito complicados, **PathProxy** pode não ser a escolha certa. Por outro lado, uma vez que se alcança um certo nível de complexidade de relacionamento, usando **PathProxy** simplifica o seu projeto do sistema global. Ser capaz de

reutilizar o mesmo mecanismo e outra vez é também uma enorme economia de tempo.

Design Pattern – Composite

É consideradoum padrão de particionamento. Descreve que um grupo de objetos serão tratados da mesma forma como uma única instância de um objeto. A intenção deste padrão é "compor" objetos em estruturas de árvore para representar as hierarquias parte todo. Implementar este padrão permite que os clientes tratem objetos individuais e composições de maneira uniforme.



Interface Component

- é a abstração de todos os componentes.
- declara a interface para os objetos na composição.
- define opcionalmente uma interface para acessar a classe pai de um componente na estrutura recursiva, e implementá-lo se isso for apropriado.

Classe Leaf

- representa objetos na composição.
- implementa todos os métodos de componentes.

Classe Composite

- representa um componente e seus compostos (componente com filhos).
- implementa os métodos para manipular os filhos.
- implementa todos os métodos de componentes, geralmente, delegando-os a seus filhos.

3. Mapeamento Objeto Relacional

Caso necessitemos de implementar puramente Orientação a Objetos em um Sistema, devemos decidir qual paradigma de banco de dados utilizaremos para o seu sistema. Atualmente temos as seguintes opções:

- Sistemas de Bancos de Dados Orientados a Objetos (SGBDOO)
- Mapeamento Objeto Relacional sobre um banco de dados relacional, ou uma camada de acesso a Banco de Dados Relacional, o que levará a uma camada de objeto chamado de Representação de Negócio.

Ao mapear objetos para bancos de dados relacionais o lugar para começar é com os atributos de dados de uma classe. Um atributo será mapeado para zero ou mais colunas em um banco de dados relacional. Nem todos os atributos são persistentes, alguns são usados para cálculos temporários. Por exemplo, um objeto de “Aluno” pode ter um atributo “média” que é necessário dentro da aplicação, mas não é armazenado na base de dados, porque é calculado dinamicamente. Como alguns atributos de um objeto são de direito próprio, um objeto “Cliente” tem um objeto da classe “Endereço” como um atributo - isso realmente reflete uma associação entre as duas classes que provavelmente precisam ser mapeadas, e os atributos da classe “Endereço” precisam ser mapeados. O importante é que esta é uma definição recursiva: Em algum momento o atributo será mapeado para zero ou mais colunas.

Conceitos Chaves

Mapeamento. O ato de determinar como os objetos e seus relacionamentos são mantidos em armazenamento de dados permanente, neste caso, bancos de dados relacionais.

Mapping. A definição de como a propriedade de um objeto ou um relacionamento é mantido no armazenamento permanente.

Propriedade. Um atributo de dados, quer implementado como um atributo físico, tais como, “Nome”, ou como um atributo virtual implementado através de uma operação tal como o método **obterTotal()**, que retorna o total de uma ordem.

Mapeamento da Propriedade. Um mapeamento que descreve como manter a propriedade de um objeto.

Relação de Mapeamento. Um mapeamento que descreve como manter um relacionamento (associação, agregação ou composição) entre dois ou mais objetos.

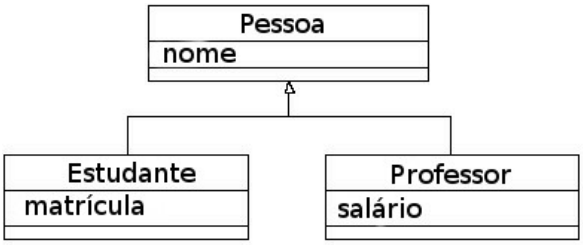
Atributos de mapeamento para colunas

Um atributo de classe será mapeado para zero ou um número de colunas em um banco de dados relacional. É importante lembrar que nem todos os atributos são persistentes. Por exemplo, uma classe “NotaFiscal” pode ter um atributo “total” que é usado por suas instâncias para efeito de cálculo, mas que não são salvos no banco de dados.

Além disso, alguns atributos de objetos são objetos de direito próprio, por exemplo, um objeto da classe “Curso” tem uma instância de “LivroDidático” como um atributo, que mapeia as várias colunas no banco de dados (na verdade, as chances são de que a classe “LivroDidático” será mapeado para um ou mais tabelas em seu próprio direito).

O importante é que esta é uma definição recursiva: em algum momento o atributo será mapeado para zero ou mais colunas. É também possível que vários atributos pode mapear para uma única coluna de uma mesa. Por exemplo, uma classe que representa um CEP podem ter três atributos numéricos, uma representando cada uma das secções de um código total, enquanto que o “CodigoPostal” pode ser armazenada como uma única coluna em uma tabela de endereços.

O conceito de herança joga em várias reviravoltas interessantes ao salvar objetos em um banco de dados relacional. A questão basicamente se resume a descobrir como organizar os atributos herdados dentro de seu modelo de persistência. A modo de resolver este desafio pode ter um grande impacto em seu projeto do sistema. Há três soluções fundamentais para o mapeamento de herança em um banco de dados relacional, e entendê-las tais como discutir os *trade-offs* para mapear o **Diagrama de Classes** apresentado na seguinte figura:



Classes de mapeamento para as tabelas

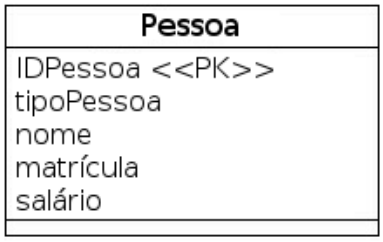
Classes são mapeadas para tabelas, embora muitas vezes não diretamente. Exceto com bancos de dados muito simples, quase nunca teremos um mapeamento “um para um” de classes para tabelas. Vamos discutir algumas estratégias para a implementação de estruturas de herança a um banco de dados relacional:

- Utilizar uma entidade de dados para uma hierarquia de classe
- Utilizar uma entidade de dados por classe concreta
- Utilizar uma entidade de dados por classe

Utilizar uma entidade de dados para uma hierarquia de classe

Com essa abordagem, mapeamos uma hierarquia de classe em uma entidade, onde todos os atributos de todas as classes na hierarquia são armazenadas. A próxima figura representa esse modelo de persistência para o **Diagrama de Classes**. Observe que a coluna “IDPessoa” foi introduzida como chave primária da tabela, utilizaremos **OID** (*Object ID*) como identificadores, também conhecidos como **Chaves Substitutas**, em todas as soluções, de modo a ser coerente e ter uma melhor abordagem para atribuição de chaves

para as entidades.



As vantagens desta abordagem é ser muito simples, o polimorfismo é suportado quando ocorre uma troca de papéis da entidade *Pessoa*, para os relatórios *ad hoc* (listagem realizada para fins específicos de um grupo pequeno de usuários, que normalmente criam seus próprios relatórios) também é muito fácil com esta abordagem, pois todos os dados pessoais necessários são encontrados em uma única **Entidade**.

As desvantagens são:

- Cada vez que um novo atributo é adicionado em qualquer lugar na hierarquia de classe um novo atributo deve ser adicionado à tabela. Isso aumenta o acoplamento dentro da hierarquia de classes - se for cometido um erro ao adicionar um único atributo, que pode afetar todas as classes dentro da hierarquia, para além das subclasses de qualquer classe tem o novo atributo.
- Potencialmente desperdiça muito espaço no banco de dados. Observamos que foi necessário adicionar a coluna “tipoObjeto” para indicar se a linha representa um *Estudante*, um *Professor*, ou qualquer outro tipo de *Pessoa*. Isso funciona bem quando alguém tem um papel único, mas rapidamente se decompõe se possuem múltiplos papéis (por exemplo, a *Pessoa* é um *Estudante* e um *Professor*).

Utilizar uma entidade de dados por classe concreta

Com esta abordagem, cada entidade inclui os atributos e os atributos herdados da classe que representa. A próxima figura representa esse modelo de persistência para o **Diagrama de Classes**. Existem dados das entidades correspondentes de ambas classes *Estudante* e *Professor*, porque são uma extensão, mas não para a classe *Pessoa* porque é a *superclasse* (indicada pelo nome). A cada uma das entidades foi atribuído uma chave primária própria **IDEstudante** e **IDProfessor** respectivamente.

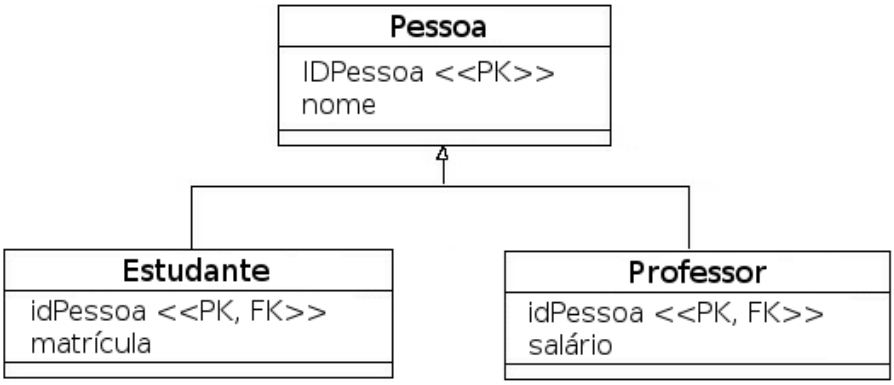
A maior vantagem desta abordagem é que ainda é bastante fácil de executar os relatórios *ad hoc*, uma vez que todos os dados necessários estão sobre uma única classe é armazenada em uma única entidade.

As desvantagens são:

- Ao modificar uma classe deve-se modificar a entidade e a entidade de qualquer uma de suas subclasses. Por exemplo, ao adicionar a altura e peso para a classe *Pessoa*, é necessário atualizar em ambas as tabelas.
- Sempre que um objeto muda seu papel – por exemplo, um *Estudante* ser contratado como um *Professor* – é necessário copiar todos os dados para a entidade apropriada e atribuir um novo **OID**.
- É difícil suportar múltiplos papéis e ainda manter uma integridade dos dados. Por exemplo, onde armazenaríamos o nome de alguém que é ao mesmo tempo é um *Estudante* e um *Professor*?

Utilizar uma entidade de dados por classe

Com essa abordagem, criamos uma entidade por classe, os atributos que são o **OID** e os atributos são específicos para essa classe. A próxima figura representa esse modelo de persistência para o **Diagrama de Classes**. Notamos que o campo **IDPessoa** é utilizado como chave primária para as três entidades. A coluna **IDPessoa** nas entidades *Professor* e *Estudante* é atribuído como dois estereótipos, algo que não é permitido para a UML (*Unified Modeling Language*).



A principal vantagem dessa abordagem é que está de acordo com os conceitos Orientados a Objeto. Suporta o polimorfismo muito bem de modo que temos registros nas tabelas apropriadas para cada papel que um objeto pode ter. É também muito fácil de modificar as *superclasses* e adicionar subclasses novas, porque você só precisa modificar ou adicionar uma tabela.

As desvantagens são:

- A existência de muitas tabelas no banco de dados - uma para cada classe, de fato (mais tabelas para manter relacionamentos).
- Mais tempo para ler e gravar dados utilizando esta técnica pois é necessário acessar várias tabelas. Este problema pode ser aliviado ao organizar o banco de dados, colocando cada tabela dentro de uma hierarquia de classes em diferentes físicas da unidade de disco (Isso pressupõe que cada um dos chefes da unidade de disco

operar de forma independente).

- Relatórios *ad hoc* em seu banco de dados é difícil, a menos que seja adicionado visões para simular as tabelas desejadas.

Mapeando Associações, Agregação e Composição

Não se deve mapear somente os objetos no banco de dados, como também mapear as relações que o objeto está envolvido como possam ser restaurados em uma data posterior. Existem quatro tipos de relações que um objeto pode ser envolvidos: Herança, Associação, Agregação e Composição. Para mapear essas relações de forma eficaz, devemos entender as diferenças entre elas, como implementar relacionamentos em geral, e como implementar especificamente os relacionamentos “muitos para muitos”.

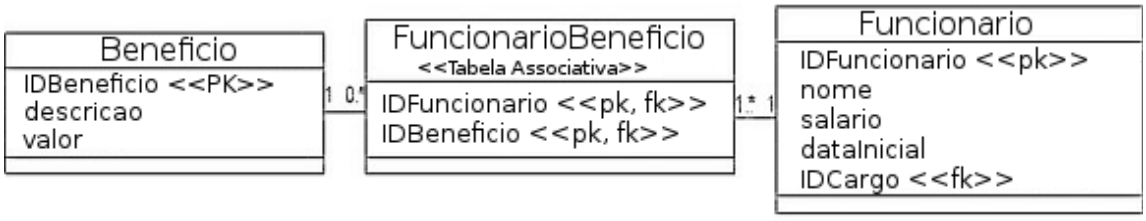
De uma perspectiva do banco de dados, a única diferença entre as relações de Associação, Agregação e Composição é quão firmemente os objetos são ligados um ao outro. Com a Agregação e Composição, tudo o que realizamos no banco de dados é necessário a ser realizado para as partes, enquanto que com a Associação este não é o caso.

Relações de aplicação em bancos de dados relacionais

Relações em bases de dados relacionais são mantidas através da utilização de chaves estrangeiras. Uma chave externa é um ou mais atributos de dados que aparecem em uma entidade que pode ser parte de, ou é coincidente com, a chave de outra entidade. As chaves estrangeiras permitem relacionar uma linha em uma entidade com uma linha em outra. Para implementar relacionamentos “um para um” e “um para muitos” que só tem que incluir a chave de uma entidade em outra entidade.

Implementar relacionamentos “muitos para muitos”

É necessário o conceito de uma tabela associativa, uma entidade de dados, cujo único propósito é manter uma associação entre duas ou mais tabelas em um banco de dados relacional. Na figura seguinte veremos uma relação de “muitos para muitos” entre as entidades *Funcionário* e *Benefício*.



Na bases de dados relacionais os atributos estão contidos numa tabela associativa e são tradicionalmente a combinação das chaves nas tabelas envolvidas na relação. O nome de uma tabela associativa é tipicamente a combinação dos nomes das tabelas que associa ou o nome da associação que implementa.

4. JPQL

Java Persistence Query Language (JPQL) é uma plataforma independente de linguagem de consulta orientada a objeto, foi definida como parte da especificação Java Persistence API. JPQL é usado para fazer consultas com entidades armazenados em um banco de dados relacional. É fortemente inspirada pelo SQL e suas consultas assemelham as consultas SQL na sintaxe, mas operam em objetos de entidade JPA ao invés de diretamente com as tabelas de banco de dados.

JPQL é usado para definir as pesquisas nas entidades persistentes e são independentes do mecanismo usado para armazenar essas entidades. Como tal, JPQL é "portátil", e não restrito a nenhum armazenamento de dados particular. JPQL é uma extensão de EJB-QL (*Enterprise JavaBeans Query Language*), acrescentando as operações como excluir e atualizar, operações de junção, agregações, projeções, e subconsultas. Além disso, JPQL pode ser declarado estaticamente em metadados, ou podem ser dinamicamente construído em código.

Estrutura de consulta JPQL

A sintaxe da linguagem JPQL é muito semelhante à sintaxe SQL. Ter um SQL como sintaxe em consultas JPA é uma vantagem importante, pois o SQL é uma linguagem de consulta muito poderosa e muitos desenvolvedores já estão familiarizados.

A principal diferença entre SQL e JPQL é que o resultado de uma consulta SQL são registros e campos, ao passo que o resultado de uma consulta JPQL classes e objetos. Por exemplo, uma consulta JPQL pode recuperar e retornar objetos de entidade, em vez de apenas os valores de campo a partir de tabelas de banco de dados, como acontece com SQL. Isso faz objecto JPQL mais orientado amigável e fácil de usar em Java.

Tal como acontece com SQL, uma consulta JPQL SELECT também é composto por até 6 cláusulas no seguinte formato:

```
SELECT ... FROM ...
[WHERE ...]
[GROUP BY ... [HAVING ...]]
[ORDER BY ...]
```

As duas primeiras cláusulas, SELECT e FROM são necessários em cada consulta de recuperação (atualizar e excluir têm uma forma ligeiramente diferente). As outras cláusulas JPQL, onde, GROUP BY, HAVING e ORDER BY são opcionais.

A estrutura JPQL para excluir e modificar é mais simples:

```
DELETE FROM ... [WHERE ...]
UPDATE ... SET ... [WHERE ...]
```

Além de algumas poucas exceções, JPQL diferencia maiúsculas de minúsculas. Palavras-chave JPQL, por exemplo, pode aparecer nos comandos quer no caso superior (por exemplo, **SELECT**) ou em letras minúsculas (por exemplo, **select**).

As poucas exceções em que JPQL diferencia maiúsculas de minúsculas incluem elementos de origem principalmente Java, como nomes de classes de entidade e campos persistentes, que são sensíveis ao caso. Além disso, strings também são maiúsculas de minúsculas (por exemplo, "**ORM**" e "**orm**" são valores diferentes).

Uma consulta minima JPQL

A consulta seguinte recupera todos os objetos de **País** no banco de dados:

```
SELECT p FROM Pais AS p
```

Porque SELECT e FROM são obrigatórias, isto demonstra uma consulta mínima JPQL.

A cláusula FROM declara uma ou mais variáveis de consulta (também conhecidas como variáveis de identificação). Variáveis de consulta são semelhantes às variáveis de laço em linguagens de programação. Cada variável de consulta representa a iteração sobre os objetos no banco de dados. Uma variável de consulta está ligado a uma classe de entidade é referido como uma variável de intervalo.

Variáveis de intervalo definir a iteração sobre todos os objetos de banco de dados de uma classe de entidade de ligação e suas classes descendentes. Na consulta anterior, **p** é uma variável de intervalo que é vinculado à classe da entidade **País** e define a iteração sobre todos os objetos no banco de dados. A cláusula SELECT define os resultados da consulta. A consulta acima simplesmente retorna todos os objetos do campo a partir da iteração do intervalo variável **p**, que neste caso é efetivamente todos os objetos do campo no banco de dados.

Outros exemplos

Retornar todas as pessoas que moram no DF (Distrito Federal), GO (Goiás) ou BA (Bahia):

```
SELECT p FROM Pessoa p WHERE p.endereco.uf IN ("DF","GO", "BA")
```

Retornar objetos de **Pessoa** que possuem o estado civil “casado” ou “divorciado”:

```
SELECT p FROM Pessoa p JOIN p.estadocivil e WHERE e.descricao IN
("casado","divorciado")
```

Retornar objetos de **Pessoa** por uma determinada lista de enumerações:

```
SELECT p FROM Pessoa WHERE p.enum = x25.com.tutorial.enum.value
```

Retornar objetos de **Autor** por ordem alfabética de Nome

```
SELECT a FROM Autor a ORDER BY a.nome
```

Retornar objetos de **Autor** que publicaram suas obras pela editora XYZ:

```
SELECT DISTINCT a FROM Autor a INNER JOIN a.livro l WHERE l.editora.nome = 'XYZ'
```

Retornar a moeda e a população dos objetos de **País** que estão no continente Europeu:

```
SELECT p.moeda, SUM(p.populacao) FROM Pais p WHERE 'Europa' MEMBER OF p.continente
GROUP BY p.moeda HAVING COUNT(p) > 1
```

5. Consultas: Criteria

A especificação da JPA 2.0 define uma nova API para as consultas dinâmicas através da construção de uma instância da classe **javax.persistence.CriteriaQuery** baseada em objeto, em vez de uma abordagem baseada em strings usado em JPQL (*Java Persistence Query Language*). Esta capacidade de definição de consulta dinâmica é referida como **Criteria API**, e baseia-se no esquema persistente das entidades, seus objetos incorporados e suas relações.

A sintaxe é projetada para construir uma árvore de comando cujos os nós representam os elementos de consulta semântica, tais como projeções, predicados condicionais de cláusula WHERE ou GROUP BY elementos.

Construindo uma consulta Criteria

A interface **CriteriaBuilder** é o padrão *Abstract Factory* para **CriteriaQuery**. Um **CriteriaBuilder** é obtido a partir de qualquer um **EntityManagerFactory** ou um **EntityManager** como por exemplo:

```
EntityManager em = factory.createEntityManager();
CriteriaBuilder queryBuilder = em.getCriteriaBuilder();
CriteriaQuery qdef = queryBuilder.createQuery();
```

O primeiro passo na construção de uma definição de consulta é a especificação de um objeto de **Root**. Este objeto é responsável por especificar os objetos de domínio em que a consulta é avaliada, sendo uma instância da interface **Root <T>**. É adicionada a uma **CriteriaQuery** pela chamada ao método **addRoot()**.

```
Root<Cliente> cliente = qdef.from(Cliente.class);
```

Um domínio de consulta pode ser refinado para unir outros objetos de domínio. Por exemplo, para a definição acima operar sobre os Clientes e suas Ordens de Serviço, podemos utilizar o metodo **join()**:

```
Root<OrdemServico> os = cliente.join(cliente.get(Cliente_.ordens));
```

onde **Cliente_.ordens** representa um campo de classe metamodelo para o Cliente. Essas classes de metamodelo são geradas durante a compilação, processando a anotação persistente no código fonte de **Cliente.java**.

A condição de uma definição de consulta é definido através do método **where()** onde o argumento designa um predicado condicional. Predicados condicionais são frequentemente compostos de um ou mais comparações entre os valores dos atributos dos objetos de domínio e algumas variáveis.

Por exemplo, para selecionar o Cliente cujo nome é "Fernando Anselmo" e tem pedidos que ainda não foram entregues, podemos construir o predicado e configurá-lo para a definição da consulta como:

```
qdef.where(cliente.get(Cliente_.nome).equal("Fernando Anselmo")
.and(os.get(OrdemServico_.status).equal(OrdemStatus.ENTREGADO).not()));
```

O método **select()** define o resultado da consulta. Se deixado sem especificação, o resultado da seleção é assumido como sendo um objeto do domínio raiz. No entanto, podemos especificar as projeções selecionadas explicitamente como uma lista:

```
qdef.select(cliente.get(Cliente_.name), os.get(OrdemServico_.status));
```

Um atributo de um objeto de domínio também pode ser especificado através da navegação via **get()**. O atributo se refere a uma propriedade válida persistente do objeto de domínio recebe, no entanto, a validação que não seja executada durante a construção da definição da consulta. Toda validação é adiada até que a consulta seja realmente executada. Por outro lado, usando o metamodelo para o caminho da navegação reforça a verificação de tipo.

6. Validação e Integridade

Com a implementação da **JSR-303** podemos utilizar a estrutura de validação no lado do servidor (*EJB Session Bean*) e no *Managed Bean* das aplicações **Web JSF**. Isto evita a perde de tempo na tomada das Regras de Negócio, que não necessitarão mais de validadores nos componentes **JSF UI**.

Para tornar isso mais interessante podemos combinar com o **Apache MyFaces Extensions Validator** (**ExtVal**). Esta estrutura pode ser usada com a JSF (existe uma biblioteca genérica e um especial para Trinidad) e suporta a validação através do uso dos **Bean Validation**. **ExtVal** também tem alguns recursos extras.

- Tipo seguro para o grupo de validação
- Validação do modelo
- Validação da gravidade consciente
- Validação do cliente
- Ordenar mensagens de violação
- Apoio a injeção de dependência para os validadores de restrição
- Fonte de restrição mapeada (por exemplo, usar DTO com BV)
- Suporte a anotação @Valid

A validação de dados é uma tarefa comum que ocorre em todas as camadas de uma aplicação, incluindo persistência. JPA 2.0 oferece um suporte para a **API Bean Validation** para que a validação de dados possa ser realizada em tempo de execução.

A **API Bean Validation** fornece uma validação transparente em todas as tecnologias da plataforma Java EE e Java SE. Além de JPA 2.0, estas tecnologias incluem JavaServer Faces (JSF) 2.0 e Java EE Connector Architecture (JCA) 1.6.

Há três conceitos fundamentais para **Bean Validation**: restrições, manipulação de violação de restrição e do validador. Ao executar aplicações em um ambiente integrado como APP, não há necessidade de interagir diretamente com o *Validador*.

Restrições de validação são anotações ou código XML que são adicionados a uma aula de campo, ou método de um componente **JavaBeans**. As restrições podem ser construídas ou definido pelo usuário. São utilizados para as definições de restrição regulares e para a composição de restrições. As restrições internas são definidas pela especificação da **Bean Validation** e estão disponíveis com cada provedor de validação.

Grupos de validação

Bean Validation usa grupos de validação para determinar o tipo de validação e quando a validação ocorre. Não existem interfaces especiais para implementar ou anotações para criar um grupo de validação. Um grupo de validação é denotada por uma definição de

classe.

Quando no uso de grupos, utilizar interfaces simples. Usando uma interface simples torna mais utilizável grupos de validação em vários ambientes. Considerando que, se uma definição de classe ou entidade é usado como um grupo de validação, pode poluir o modelo de objeto de outro aplicativo, trazendo em classes de domínio e lógica que não realiza qualquer sentido para a aplicação.

Por padrão, se um grupo de validação ou vários grupos não é especificado em uma restrição individual, que é validado utilizando o grupo **javax.validation.groups.Default**. Criar um grupo personalizado é tão simples como criar uma definição de uma nova interface.

7. SQL Nativo

JPA pode executar consultas SQL simples. Esse tipo de Consulta é denominada JPA Native Query e possui as seguintes propriedades:

- Uma Native Query retorna um único valor escalar
- Retorna todos os valores persistentes de classe de entidade especificada.

É interessante notar que não estamos limitado a JPQL ao definir que as consultas sejam firmadas com a API Query. Pode-se surpreender ao saber que a API EntityManager oferece métodos para criar instâncias de consulta para a execução de comandos SQL nativos.

Aqui está um exemplo simples de uma consulta SQL dinâmico nativo:

A. Entidade: Estudante

```
CREATE TABLE `estudante` (
 `id` int(11) NOT NULL auto_increment,
 `nome` varchar(100) NOT NULL,
 `curso` varchar(10) NOT NULL,
 PRIMARY KEY (`id`)
)
```

Modelo Classe: Estudante.java

```
package x25.com.tutorial;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.Table;

@Entity
@Table(name="estudante")
public class Student {

 @Id
 @GeneratedValue
 private int id;
 @Column(name="nome", length=100, nullable=false)
 private String nome;
 @Column(name="curso", length=10, nullable=false)
 private String curso;

 public int getId() {
  return id;
 }
 public void setId(int id) {
  this.id = id;
 }
```

```
 public String getNome() {
  return nome;
 }
 public void setNome(String nome) {
  this.nome = nome;
 }

 public String getCurso() {
  return curso;
 }
 public void setCurso(String curso) {
  this.curso = curso;
 }
}
```

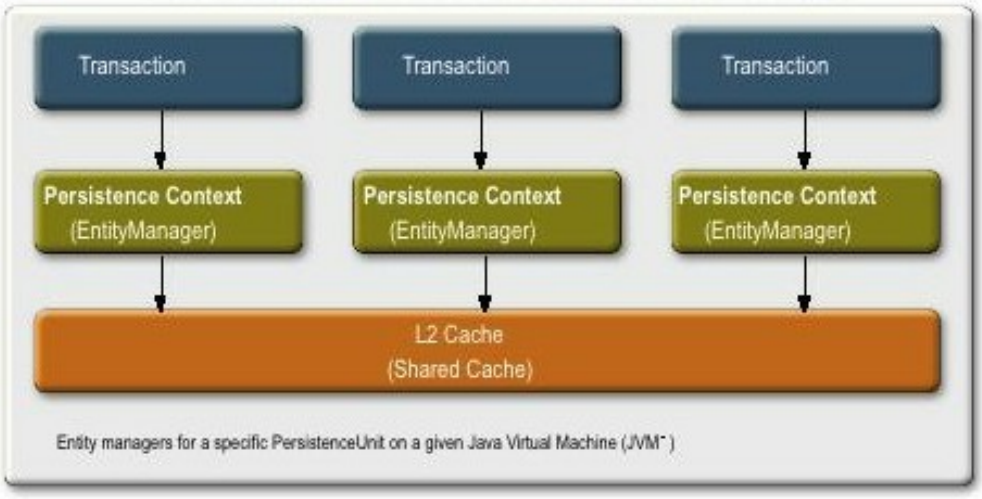
Classe Principal: JPANativeQuery.java

```
package x25.com.tutorial;

import java.util.Iterator;
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;
import javax.persistence.Query;

public class JPANativeQuery {

 public static void main(String[] args) {
  EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpa");
  EntityManager em = emf.createEntityManager();
  try {
   EntityTransaction entr = em.getTransaction();
   entr.begin();
   Query query = em.createNativeQuery("SELECT nome FROM estudante");
   List stList = query.getResultList();
   Iterator stIterator = stList.iterator();
   while (stIterator.hasNext()){
    System.out.println("Nome: " + stIterator.next());
   }
   entr.commit();
  } finally {
   em.close();
  }
 }
}
```

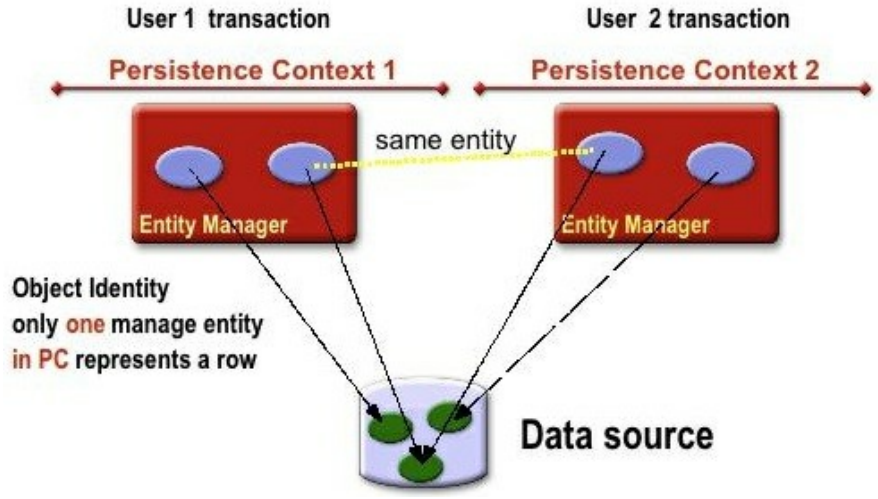
A coisa mais importante a compreender sobre as consultas SQL nativas é que são criadas com métodos *EntityManager* é que eles, como consultas JPQL, retornam instâncias da entidade, ao invés de registros da tabela de banco de dados.

8. Recursos Avançados: Cache e Locks

JPA possui um cache opcional de dados de objetos persistentes que opera no nível da classe *EntityManagerFactory*. Esse cache é projetado para aumentar significativamente o desempenho, permanecendo em total conformidade com o padrão JPA. Isto significa que ligar a opção de cache transparente pode aumentar o desempenho de sua aplicação, sem alterações em seu código.



O Gerenciador de Entidade garante que dentro de um contexto de persistência única, para qualquer linha de banco de dados particular, haverá uma instância do objeto. No entanto, a mesma entidade pode ser controlada na transação de outro usuário, então deve-se usar um bloqueio **otimista** ou **pessimista**.



O cache de dados do JPA não é relacionado ao cache da *EntityManager* ditada pela especificação da JPA. A especificação da JPA possui um comportamento para o cache do *EntityManager* que visa garantir isolamento de transação, quando operando em objetos persistentes. É projetado para proporcionar aumentos significativos de desempenho sobre a

operação cacheless, garantindo que o comportamento será idêntico tanto na operação de cache habilitado e cacheless.

Concorrência Otimista

O bloqueio otimista permite processar transações simultaneamente, mas detecta e evita colisões, isso funciona melhor para aplicações onde a maioria das transações concorrentes não entram em conflito. O bloqueio JPA otimista permite a qualquer pessoa ler e atualizar uma entidade, no entanto, uma verificação de versão é feito após confirmação e uma exceção é lançada se a versão foi atualizada no banco de dados uma vez que a entidade foi lida. Deve existir a anotação com **@Version**, conforme mostrado abaixo:

```
public class Funcionario {
    @ID int id;
@Version int versao;
```

O atributo versão é incrementado com uma confirmação bem-sucedida. O atributo versão pode ser um tipo *int*, *short*, *long* ou *timestamp*. Isto resulta no seguinte SQL:

```
UPDATE Employee SET ..., version = version + 1
 WHERE id = ? AND version = readVersion
```

As vantagens do bloqueio otimista é que não há bloqueios na base de dados e são realizadas para dar uma melhor escalabilidade. As desvantagens são que o usuário ou aplicação deve atualizar e tentar novamente com as atualizações que falharam.

Concorrência pessimista

O bloqueio pessimista promove um “lock” na linha do banco de dados quando os registros são lidos, é equivalente a um comando:

```
SELECT ... FOR UPDATE [NOWAIT]
```

O bloqueio pessimista assegura que as operações não atualizam a mesma entidade, ao mesmo tempo, que pode simplificar código de aplicação, mas limita o acesso simultâneo aos dados que podem causar uma mau escalabilidade e bloqueios. É melhor para aplicações com maior risco de discórdia entre transações concorrentes.

Os *trade-offs* são utilizados para segurar bloqueios maiores, os riscos de escalabilidade ruim e impasses. Quanto mais tarde for executado um “lock”, maior o risco de dados desatualizados, que pode causar uma exceção do bloqueio otimista, se a entidade foi atualizado após a leitura, mas antes de bloquear.

9. Hibernate

Hibernate, criado por Gavin King, conhecido como o objeto e melhor dominada ferramenta de Persistência Objeto Relacional (ORM) para desenvolvedores Java. Fornece muitas maneiras elegantes e inovadoras para simplifica a tarefa de movimentação de dados relacional em Java.

JPA 2 define critérios de consulta que podem ser construídas de uma maneira fortemente tipada, usando os objetos do metamodelo para fornecer uma segurança de tipos. Em vez de usar o arquivo específico de configuração do **Hibernate**, *hibernate.cfg.xml*, JPA, define um processo de inicialização diferente, que usa seu próprio arquivo de configuração chamado *persistence.xml*. Como isso funciona, **bootstrapping** é definida pela especificação **JPA**. Em ambientes Java o provedor de persistência (**Hibernate**, neste caso) é necessário para localizar todos os arquivos de configuração do JPA por pesquisa classpath do nome do recurso META-INF/persistence.xml.

```
<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="org.hibernate.tutorial.jpa">
  ...
 </persistence-unit>
</persistence>
```

O arquivo **persistence.xml** fornece um nome único para cada unidade de persistência. Este nome é feito para os aplicativos referência a configuração ao obter uma referência *javax.persistence.EntityManagerFactory*.

Aqui as variedades *javax.persistence-prefixed* são usadas quando possível. Para os restantes especificações do **Hibernate** os nomes de parâmetros de configuração notam que estão agora prefixado.

```
protected void setUp() throws Exception {
 entityManagerFactory =
  Persistence.createEntityManagerFactory("org.hibernate.tutorial.jpa");
}
```

Este código obtém um objeto de *javax.persistence.EntityManagerFactory*. Observe a utilização de org.hibernate.tutorial.jpa como o nome unidade de persistência, que corresponde a partir do exemplo "persistence.xml". Exemplo para salvar:

```
EntityManager entityManager = entityManagerFactory.createEntityManager();
entityManager.getTransaction().begin();
entityManager.persist(new Event("Este é um primeiro evento", new Date()));
entityManager.persist(new Event("O próximo evento", new Date()));
entityManager.getTransaction().commit();
entityManager.close();
```

Aqui usamos um *javax.persistence.EntityManager* em oposição a um *org.hibernate.Session*. JPA chama o método **persist()**, em vez de **insert()**. Obter uma listagem:

```
entityManager = entityManagerFactory.createEntityManager();
entityManager.getTransaction().begin();
List<Event> result = entityManager.createQuery("para o Evento", Event.class)
 .getResultList();
for (Event event: result) {
 System.out.println("Evento (" + event.getDate() + ") :" + event.getTitle());
}
entityManager.getTransaction().commit();
entityManager.close();
```

10. Erros Comuns

Para os dados relacionais de persistência do **Hibernate** é certamente a solução mais popular no mercado. Tem sido em torno de anos e é provavelmente sido usado em milhares de projetos. Com a sua mais recente versão, mesmo em conformidade com a especificação JPA.

Então, com **Hibernate** oferecendo tudo por que utilizar outra coisa? Bem, pensamos que o Hibernate e a JPA não são um par simplesmente tão perfeito quanto parece. Para explicar o porquê, queremos oferecer-lhe algumas razões que achamos que pode estar errado.

Dados modelo de definição: O problema com os metadados

Para trabalhar cada solução de persistência de dados relacionais precisamos entender o modelo de dados subjacente. **Hibernate** oferece duas formas para definir o modelo de dados: arquivos de mapeamento XML e anotações. Anotações só recentemente foram introduzidas para simplificar o mapeamento objeto relacional e há uma abundância de razões pelas quais as anotações são claramente a melhor solução sobre mapeamento XML. Assim não vamos considerar o mapeamento XML mais longe e concentrar-se em anotações. No entanto, tudo que é dito aqui anotações sobre aplica-se a XML de mapeamento de uma forma similar.

Os metadados fornecidos com anotações é exigido pelo **Hibernate** para saber onde e como persistir objetos no banco de dados. Esta informação, contudo, não pode apenas ser útil para o **Hibernate**, mas também para a lógica do aplicativo e assim pode-se querer acessar a partir do seu código em vez de redundante fornecer as mesmas informações novamente. Um bom exemplo é, o comprimento máximo de um campo de texto ou se ou não um campo é obrigatório. Em uma interface de usuário, por exemplo, podemos necessitar desta informação para a exibição de um controle de entrada de formulário ou para validação.

Com as anotações é possível acessar essas informações de seu código, mas uma propriedade de anotação particular não pode ser diretamente relacionado a partir do código.

Dados definição do objeto: A mania Setter e Getter

Além de metadados também precisamos de um lugar para armazenar e acessar os dados. Para **Hibernate** e **JPA** este é um **JavaBean** ou **POJO** que está equipado com um campo de membro, bem como um absorvente e um método no padrão SET para cada uma das colunas da tabela correspondente. Para os modelos de dados grandes, isso significa lotes de linhas de código. Ferramentas do **Hibernate** pode ser usado para gerar automaticamente todo este código usando engenharia reversa. No entanto, para projetos grandes e maduros podemos executar o problema que depois de ter alterado manualmente um **bean** ou código de mapeamento - e manter essa mudança - ferramentas automáticas são problemáticas.

Então, muitas vezes todo este código, incluindo os metadados são mantidos à mão. Ainda pior, uma vez que estes objetos são geralmente utilizados como objetos de transferência de

dados a fim de preencher objetos de negócios, você vai encontrar filas intermináveis de código onde os valores de propriedade são copiados de um objeto Java para outro. Então, qual é o ponto em que todos esses métodos GET e SET em primeiro lugar?

Com os **Dynamic Beans** só existe um GET e SET genérico para cada entidade que ambos já foram implementadas. A quantidade de classes podem ficar na mesma, como também recomendou a criação de uma classe de objeto de dados individuais para cada entidade do banco de dados - embora isso não é necessário quando se utiliza um objeto **DBRecord** genérico.

Isso é recomendável por duas razões:

- Um novo tipo de segurança, desde que queira que o seu código interno conte com determinadas entidades.
- É provável que, ao crescer o projeto, precisamos substituir novos métodos existentes implementá-los.

Mas mesmo assim, devido à ausência de todos esses campos de membros e sua correspondente e métodos GET e SET vai acabar com muito menos código para manter. Ainda assim podemos adicionar GET e SET especiais para as colunas individuais se for necessário ou conveniente.

Usar a identidade do aplicativo pode causar um comportamento estranho

A causa mais comum de problemas como estes é quando usando a identidade do aplicativo é a falha para a classe identidade do aplicativo para substituir os métodos corretamente os métodos **equals()** e **hashCode()** para que os objetos de identidade com equivalentes valores de chave primária são considerados iguais.

Erros de conexão ao retornar instâncias persistentes de um Session Bean

Uma causa comum desse problema é devido ao contêiner EJB, serialização nos casos que são retornados do EJB. O processo de serialização acontece em um momento no ciclo de vida do EJB, onde o atual status da transação está indefinida.

A serialização pode resultar em relações descarregados sendo percorridos, é tentado obter uma conexão JDBC para realizar a travessia, e o servidor de aplicativo pode então proibir o acesso de conexão devido a um status de transação inválida. A solução mais simples para isso é fazer o objeto ser devolvido de forma transitória, ou retornar uma instância individual, ou executar manualmente a serialização antes de retornar a partir do método EJB.

Relações se perdem após um Commit

O problema pode ser a definição de uma relação bidirecional, mas não definir ambos os

lados da relação. Kodo não executa nenhuma "mágica" para manter relações consistentes por padrão, o aplicativo deve sempre garantir que o modelo de objeto Java é consistente. É possível, no entanto, configura-lo para gerir as relações bidirecionais.

Considerações Finais

De modo a otimizar a JPA recomendamos a leitura do artigo que pode ser obtido no seguinte endereço: http://java-persistence-performance.blogspot.com.br/2011/06/how-to-improve-jpa-performance-by-1825.html