FACET – Faculdade de Ciências Exatas e Tecnológicas

FASA – Faculdade Santo Agostinho  
Disciplina: Linguagem de Programação IV  
Professor: Petrônio Candido de Lima e Silva  
Equipe: Mariana Mayumi Ferreira Shibuya  
 Jeferson Silveira Santos de Andrade

Tema: JPA - Java Persistense API

**Parte 1: ORM e JPA**

**01)ORM - O que é mapeamento objeto/relacional? Como funciona? Quais os  
principais ORM do mercado?**

É uma técnica de desenvolvimento utilizada para reduzir a impedância da programação orientada aos objetos utilizando bancos de dados relacionais. As tabelas do banco de dados são representadas através de classes e os registros de cada tabela são representados como instancias das classes correspondentes.

Com essa técnica, o programador irá usar uma interface de programação simples que faz todo o trabalho de persistência. Os principais ORM do mercado são o Hiberbate(apache) , Toplink (Oracle), Struts e o Spring.

**02)O que é o padrão JPA? O que ele descreve? De onde surgiu?**

É um padrão da Java para persistência que deve ser implementada por frameworks que queiram seguir o padrão. A JPA define um meio de mapeamento objeto-relacional para objetos Java Simples e Comuns (POJOs), denominados beans de entidade. Diversos frameworks de mapeamento objeto/relacional como o Hibernate implementaram a JPA. Também gerencia o desenvolvimento de entidades do Modelo Relacional usando a plataforma nativa Java SE e Java EE.

**03)Quais as principais implementações do JPA?**

**Hibernate:**

Foi criado por desenvolvedores Java, espalhados ao redor do mundo, e liderado por Gavin King, posteriomente, Jboss Inc. (empresa comprada pela Red Hat). É um frameworks para o mapeamento objeto-relacional escrito na linguagem Java, mas também é disponível em .Net como o nome Nhibernate. Este programa facilita o mapeamento dos atributos entre uma base tradicional de dados relacionais e o modelo objeto de uma aplicação, mediante o uso de arquivos (XML) para estabelecer esta relação. Hibernate é um software livre de código aberto distribuído com a licença LGPL.

O objetivo de Hibernate é diminuir a complexidade entre os programas Java, baseado no modelo orientado objeto, que precisam trabalhar com um bando de dados do modelo relacional (presente na maioria dos SGBDs). Em especial, no desenvolvimento de consultas e atualizações dos dados.

Sua principal característica é a transformação de classes em Java para tabelas de dados (e dos tipos de dados Java para os da SQL). O Hibernate gera as chamadas SQL e libera o desenvolvedor do trabalho manual da conversão dos dados resultante, mantendo o programa portável para quaisquer bancos de dados SQL, porém causando um pequeno aumento no tempo de execução.

**Toplink:**

Toplink foi desenvolvido originalmente pelos povos do objeto dentroSmalltalk nos 1990s. O “ALTO” em Toplink é um acrônimo para os povos do objeto, e o nome capitalized originalmente como “Toplink”. Em 1996-1998 uma versão de Java do produto foi adicionada à linha de produto, nomeada Toplink para Java. Após a aquisição comum dos povos do objeto em abril 2000 perto Sistemas de BEA e Webgain, A linha de produto de Toplink transformou-se a propriedade de Webgain.

Em 2002, Toplink foi adquirido perto Oracle Corporatione continua a ser tornado como um membro do Middleware da fusão de Oraclefamília dos produtos.

Em 2006, Oracle doou o código de fonte do produto de Toplink e dos recursos do desenvolvimento à fonte aberta Sun Microsystems java.net Glassfish projeto. Este projeto foi nomeado fundamentos de TopLink e era a execução da referência de Java EE EJB 3.0 JPA.

Em 2007, Oracle doou o código de fonte do produto de TopLink e dos recursos do desenvolvimento à fonte aberta Fundação do eclipse EclipseLink projeto.

Em março 2008, a fundação do eclipse, anunciadaesse Sun Microsystems selecionou o projeto de EclipseLink como a execução da referência para o JPA 2.0, padrão de JSR 317 para suceder fundamentos de TopLink.

O TopLink é um dos possíveis provedores de persistência para a JPA. Ele provê alta performance e escalabilidade, produtividade para o desenvolvedor e flexibilidade. O TopLink trabalha em conjunto com qualquer banco de dados, incluindo banco de dados não relacionais, e com qualquer servidor de aplicação.

O TopLink provê também recursos como:

* Um rico framework que provê opções de fácil uso para formular queries dinâmicas e sofisticadas. Onde os desenvolvedores podem definir queries usando qualquer expressão, como EJB QL, SQL e Stored Procedures.
* Um framework de transação que provê uma transação a nível de objetos. O framework de transação suporta gerenciamento direto das transações, e também gerenciamento externo das transações através de JTA/JTS.
* Um sistema avançado de cache que melhora a performance do sistema, ao não requerer que os dados mais recentemente ou mais frequentemente acessados sejam recuperados do banco de dados, mas sim da memória do servidor.

**EclipseLink**

Os líderes da indústria da Borland, IBM, MERANT, QNX Software Systems, Rational Software, Red Hat, SuSE, TogetherSoft WebGain se juntaram e formaram a primeira eclipse.org Conselho de Comissários Desportivos, em novembro de 2001. Até o final de 2003, este consórcio inicial tinha crescido para mais de 80 membros.

Em 2 de fevereiro de 2004 o Conselho de Regentes Eclipse anunciou a reorganização Eclipse em uma organização sem fins lucrativos corporação. Originalmente um consórcio que se formou quando a IBM lançou a Plataforma Eclipse Open Source, Eclipse tornou-se um órgão independente que irá conduzir a evolução da plataforma de vantagem para os prestadores de ofertas de desenvolvimento de software e usuários finais. Todo o código fonte e tecnologia fornecidos e desenvolvido por esta comunidade em rápido crescimento é disponibilizada a título gratuito através da Eclipse Public License.

Eclipse é uma comunidade de fonte aberta, cujos projetos são focados em construir uma plataforma de desenvolvimento aberta extensível composto por quadros, ferramentas e tempos de execução para a construção, implantação e gerenciamento de software em todo o ciclo de vida. A Eclipse Foundation é uma organização sem fins lucrativos, apoiada sobre as sociedades membros que hospeda os projetos Eclipse e ajuda a cultivar tanto a comunidade open source e um ecossistema de produtos e serviços complementares.

O projeto Eclipse foi originalmente criado pela IBM em Novembro de 2001 e apoiado por um consórcio de fornecedores de software. A Eclipse Foundation foi criada em janeiro de 2004 como um independente sem fins lucrativos para a empresa para atuar como o mordomo da comunidade Eclipse. A organização independente sem fins lucrativos para a empresa foi criada para permitir um fornecedor neutro e comunidade, aberta e transparente a ser estabelecida em torno do Eclipse. Hoje, a comunidade Eclipse é composta por indivíduos e organizações a partir de uma seção transversal da indústria de software.

A Eclipse Foundation é financiada por contribuições anuais dos nossos membros e regido por um Conselho de Administração. Estratégico desenvolvedores e consumidores Estratégica lugares cativos nesta Câmara, assim como representantes eleitos pelo add-in Provedores e committers Open Source. A Fundação emprega um em tempo integral equipe de profissionais para prestação de serviços à comunidade, mas não empregar os desenvolvedores de código aberto, committers chamado, o que realmente trabalhar nos projetos do Eclipse. Committers Eclipse são normalmente empregadas por organizações ou é desenvolvedores independentes, que doam seu tempo para trabalhar em um projeto open source.

**OpenJPA**

**Kodo**, a Objetos dos dados de Java a execução, foi desenvolvida originalmente por SolarMetric, Inc em 2001. Sistemas de BEA SolarMetric adquirido em 2005, onde Kodo foi expandido para ser uma execução de ambos os JDO (JSR 12) e JPA (JSR 220) especificações. Em 2006, BEA doou uma parte grande do código de fonte de Kodo à Fundação do software de Apache sob o nome **OpenJPA**. OpenJPA será o motor do persistence do núcleo de Usuário de BEA Weblogic, IBM WebSphere, e Geronimo Usuário da aplicação. Em maio 2007, em OpenJPA graduado do incubador a um projeto top-level e a um Sun também passado Jogo da compatibilidade da tecnologia compliant com o Persistence API de Java.

OpenJPA Apache é um projeto de persistência em Java da Apache Software Foundation , que pode ser utilizado como um stand-alone POJO camada de persistência ou integrado em qualquer contêiner Java EE compliant e muitos outros quadros leves, como Tomcat e Primavera.

As versões 1.x (1.2.2 é a última) são uma produção pronta, rico em recursos compatíveis implementação da Java Persistence API (JPA) 1.0 parte da JSR-220 Enterprise Beans Java 3.0 da especificação, que passam a APP dom. Kit de Compatibilidade 1.0b Tecnologia.

As versões 2.x (2.1.0 é a última) são uma produção compatível, a execução do pronto -317 Java Persistence JSR 2,0 especificação, o que é compatível com a especificação do JPA 1,0 e passa o Sol JPA 2.0 Technology Compatibility Kit.

A próxima versão 2.2 é baseado fora do fluxo de liberação 2.1.Xe irá conter algumas funcionalidades adicionais (TBD).

**04)Como se configura o JPA? (persistence.xml)**

*<?xml version="1.0" encoding="UTF-8"?>*

*<persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence">*

*<persistence-unit name="manager1" transaction-type="JTA">*

*<provider>org.hibernate.ejb.HibernatePersistence</provider>*

*<jta-data-source>java:/DefaultDS</jta-data-source>*

*<mapping-file>ormap.xml</mapping-file>*

*<jar-file>MyApp.jar</jar-file>*

*<class>org.acme.Employee</class>*

*<class>org.acme.Person</class>*

*<class>org.acme.Address</class>*

*<shared-cache-mode>ENABLE\_SELECTIVE</shared-cache-mode>*

*<validation-mode>CALLBACK</validation-mode>*

*<properties>*

*<property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect" />*

*<property name="hibernate.connection.url" value="jdbc:mysql://localhost:3306/database" />*

*<property name="hibernate.connection.driver\_class" value="com.mysql.jdbc.Driver" />*

*<property name="hibernate.connection.password" value="" />*

*<property name="hibernate.connection.username" value="root" />*

*<property name="hibernate.hbm2ddl.auto" value="create-drop" />*

*</properties>*

*</persistence-unit>*

*</persistence>*

Para utilizar outros Bancos de Dados, basta alterar a url de conexão e o driver utilizado, por exemplo, no caso do Oracle:

*<property name="hibernate.dialect" value="org.hibernate.dialect.OracleDialect" />*

*<property name="hibernate.connection.url" value="jdbc:oracle:thin:@localhost:1521:DB" />*

*<property name="hibernate.connection.driver\_class" value="oracle.jdbc.OracleDriver" />*

Para o HSQLDB:

*<property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect"/>*

*<property name="hibernate.connection.url" value="jdbc:hsqldb:." />*

*<property name="hibernate.connection.driver\_class" value="org.hsqldb.jdbcDriver" />*

Em password e username você deve utilizar os dados para acessar o seu BD.

**Name:** (Atributo) Todo gerente de entidade deve ter um nome.

**Transaction-type:** (Atributo) tipo de transação usado. Ou JTA ou RESOURCE\_LOCAL (padrão em um ambiente JTA JavaEE e RESOURCE\_LOCAL em um ambiente de JavaSE). Quando um datasource JTA é usado, o padrão é JTA, se não datasource JTA é usado, RESOURCE\_LOCAL é usado.

**Provider:** É um nome de classe totalmente qualificado do provedor de persistência EJB. Isto é necessário quando você estiver usando implementações de vários fornecedores de persistência EJB.

**Jta-data-source, non-jta-data-source:** É o JNDI do local onde o javax.sql.DataSource está situado. Ao ser executado sem um DataSource JNDI disponível, você deve especificar conexões JDBC com as propriedades especificas do Hibernate.

**Mapping-file:** O elemento de classe especifica uns arquivos de mapeamento EJB3 XML compatível que irá mapear. O arquivo deve estar no classpath. De acordo com a especificação do EJB3, Hibernate EntityManager irá tentar carregar o arquivo de mapeamento localizado no arquivo jar em Meta\_INF/orm.xml. É claro que qualquer arquivo de mapeamento explícito serão carregados também. Por uma questão de fato, você pode fornecer qualquer arquivo XML no ie elemento de mapeamento de arquivo ou arquivos hbm ou descritor de implantação EJB3.

**Jar-file:** Os elementos jar especifica uma jar para analisar. Todas as classes devidademente anotado e todos os pacotes de parte dos arquivos hbm.xml deste arquivo jar será adicionada à configuração da unidade de persistência. Pode-se alternativamente apontar para um diretório (Isto é especialmente útil quando em seu ambiente de teste, o arquivo persistence.xml não está sob o mesmo diretório raiz o que o seu modelo de domínio).

**Class:** O elemento de classe especifica um nome de classe totalmente qualificado que irá mapear. Por padrão todas as classes devidamente anotadas e todos os arquivos hbm.xml encontrada dentro do arquivo são adicionados à configuração da unidade de persistência. Você pode adicionar alguma entidade externa através do elemento de classe embora. Como uma extensão da especificação, você pode adicionar um nome de pacote na <class> elemento (por exemplo <class>org.hibernate.eg</class> ).

**Shared-cache-mode:** Por padrão, as necessidades são eleitas para cachê de segundo nível se anotado com @cacheale. Você pode, no entanto:

* ALL: Forçar cachê para todas as entidades.
* None: Desativa cachê para todas as entidades.
* Enable\_selective: Habilitar o cachê quando explicitamente marcada.
* Disable\_Selective: Habilita o cachê a menos que explicitamente marcado como @Cacheable(false) (não recomendado).

**Validation-mode:** Por padrão, Bean Validation (e Hibernate Validator) é ativado. Quando uma entidade é criada, atualizado (e, opcionalmente, excluída), é validada antes de ser enviado para o banco de dados. O esquema do banco de dados gerado pelo Hibernate também reflete as restrições declaradas na entidade.

Pode-se fazer um ajuste fino que, se necessário:

* AUTO: se validação Bean está presente no classpath, CALLBACK e DDL são ativados.
* CALLBACK: entidades são validadas na criação, atualização e exclusão. Se nenhum provedor de validação Bean está presente, é levantada uma exceção em tempo de inicialização.
* DDL: (no standart, ver abaixo): são entidades esquemas de banco de dados e são validados na atualização, criação e exclusão. Se nenhum provedor de validação Bem está presente, é levantada uma exceção em tempo de inicialização.
* NONE: Bean Validatin não e usado para todos.

Para usa adicione uma propriedade regular.

*<property name="javax.persistence.validation.mode">*

*ddl*

*</property>*

Com essa abordagem, pode-se misturar os modos ddl e retorno:

*<property name="javax.persistence.validation.mode">*

*ddl, callback*

*</property>*

**Properties:** O Elemento de propriedade é usado para especificar as propriedades específicas do fornecedor. Isto é onde você irá definir as configurações específicas do seu Hibernate. E onde você terá que especificar informações de conexão JDBC também.

**Parte 2: Mapeamentos Utilizando o JPA**

**01)O que são annotations? Como o JPA lê o mapeamento das classes?**

São uma forma especial de declaração de metadados que podem ser adicionadas ao código-fonte pelo programados. Proveem informações sobre o comportamento de um programa. São aplicáveis às classes, métodos, atributos e outros elementos de um programa, além disso, não tem nenhum efeito sobre a execução do código onde estão inseridas.  
No JPA é possível mapear classes de acordo com as tabelas já criadas no banco de dados. Existem ferramentas que podem auxiliar a criar a engenharia reversa disso. Caso prefira, também é possível mapear as classes e deixar o JPA criar as tabelas.

**02)Como se faz o mapeamento de classes -> tabelas? Dê exemplos.**

Um mapeamento típico de uma classe persistente irá mapear a classe para uma única tabela. Em JPA isso é definido através do quadro anotação ou <table> elemento XML. Se nenhuma anotação tabela está presente, a implementação JPA será automaticamente atribuir uma tabela para a classe, a APP nome da tabela padrão é o nome da classe com letras maiúsculas; cada atributo da classe será armazenado em uma coluna da tabela.

*... @ Entity*

*@ Table ( nome = "funcionário" )*

*do público da classe Empregado {*

*...*

*}*

**03) Como se faz o mapeamento de propriedades da classe -> campos em tabelas? Dê exemplos.**

A anotação @Column mapeia um campo ou propriedade persistidos para uma coluna da tabela.

Os parâmetros insetable e update são usados para o comportamento do controle da persistência. Se o parâmetro estiver marcado como false, o campo ou propriedade não será incluído ou atualizado pelo provedor de persistência. Esses dois parâmetros são geralmente úteis quando se trata de dados somente leitura, como chaves primárias geradas pelo banco de dados.

Eles podem ser definidos da seguinte forma:

*@Column(name="USER\_ID", insertable=false, updatable=false)*

*protected Long userId;*

A anotação @Column é opcional. Se omitida, o campo ou propriedade de persistência será salvo para a coluna da tabela que combine com o nome do campo ou propriedade.

Exemplo:

*public class Telefone implements Serializable, InterfacePojo {*

*private static final long serialVersionUID = 1L;*

*@Id*

*@GeneratedValue*

*@Column(name = "id")*

*private int id;*

*@Column(name = "ddd", nullable = false, length = 2)*

*private String ddd;*

*@Column(name = "numero", nullable = false, length = 8)*

*private String numero;*

*@Column(name = "nome", nullable = false, length = 40)*

*private String nome;*

**04)Como se faz o mapeamento de uma chave primária de uma tabela em uma classe? Dê exemplos.**

A JPA dá suporte ao mapeamento de chaves primárias:

* Simples (apenas um campo é chave primária)
* Compostas (vários campos são chave primária)

***Chave primária simples***

* Neste caso utilize a anotação @Id

***Chave primária composta***

* É necessário criar a classe que representa a chave primária:
* @Embeddable

O **@id** marca o atributo com identificador. Pode ser usada tanto baseada em um campo como em uma propriedade. Em um objeto de domínio pode ter somente uma anotação @Id.

|  |  |
| --- | --- |
| *@Entity*  *public class Category {*  *@Id*  *public Long id;*  *public String name;*  *public Date modificationDate;*  *public Category() {}*  *}* | *@Entity*  *public class Category {*  *...*  *protected Long id;*  *...*  *@Id*  *public Long getId() {*  *return this.id;*  *}*  *public void setId(Long id) {*  *this.id = id;*  *}*  *...*  *}* |

**05)O que são coleções(collections)? Quais as principais coleções utilizadas em Java?**

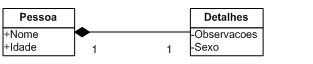
Coleções são estruturas de dados utilizadas para armazenar e manipular informações.

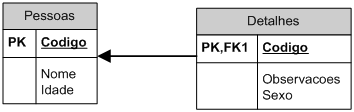
As principais coleções utilizadas em Java são:

* Conjunto (Set e SortedSet): uma coleção de elementos que não mantém uma ordem nem uma contagem dos elementos; cada elemento ou está no conjunto ou não (não há elementos repetidos).
* Lista (List): uma sequência de elementos; mantém dados a respeito de ambos, a ordem e a contagem.
* Fila (Queue): fila de elementos; modelo FIFO (First in, first out).
* Mapa (Map e SortedMap): uma associação entre chaves e valores; ele mantém um conjunto de chaves e mapeia cada chave para um único valor.

**06)Como se faz o mapeamento de relacionamentos um-para-um entre duas classes mapeadas? Dê exemplos.**

No mundo relacional um relacionamento um para um não é visto como uma boa prática de design, devendo ser utilizado apenas em casos especiais, conhecidos como particionamento horizontal ou vertical. Para este tipo de associação podemos adicionar uma chave estrangeira a uma das tabelas, mapeada para a chave estrangeira da outra tabela.

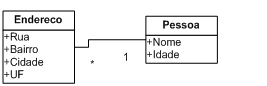




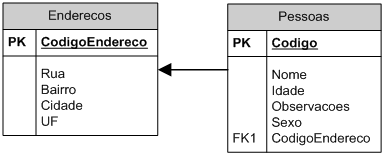
No exemplo acima é criada uma tabela para cada classe. Na tabela Pessoa é criada uma coluna Código como chave da tabela. Na tabela detalhes, uma coluna código é chave estrangeira para tabela de pessoas. Poderíamos ao invés disso ter criado um campo chave na tabela detalhes e em pessoas ter adicionado uma coluna como chave estrangeira. O resultado prático seria o mesmo, porém teríamos uma coluna a mais em pessoas, já detalhes é uma entidade fraca e por isso não poderíamos dispensar a criação de uma coluna chave em pessoas.

**07)Como se faz o mapeamento de relacionamentos um-para-muitos entre duas classes mapeadas? Dê exemplos.**

Para a associação um para muitos adicionamos uma chave estrangeira na tabela muitos.



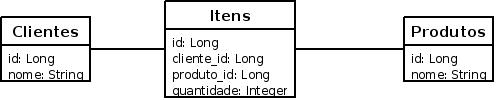
Observando o resultado abaixo, que a classe endereço é mapeada na forma da tabela Endereços, que recebe uma coluna CodigoEndereco para chave. A tabela pessoas recebe uma coluna de mesmo nome, sendo esta chave estrangeira para a tabela de Endereços.



**08)Como se faz o mapeamento de relacionamentos muitos-para-um entre duas classes mapeadas? Dê exemplos.**

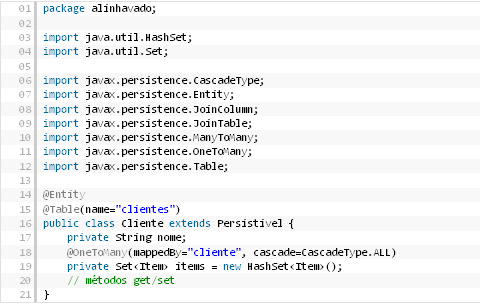
Digamos que precisamos saber exatamente qual a quantidade de um produto específico um cliente, nós precisaríamos fazer uma contagem dos produtos relacionados ao cliente específico, o que é possível, mas pouco pratico o melhor seria se o próprio relacionamento entre produtos e clientes já trouxesse esse relacionamento, dessa forma nós não precisaríamos ter produtos repetidos no relacionamento como também não seria necessário fazer contagens manuais, no próprio relacionamento a contagem já estaria feita. Vejamos como esse diagrama ficaria agora:

*Diagrama de exemplo com many-to-one*

[](http://alinhavado.files.wordpress.com/2008/01/many-to-one-example.jpeg)

Agora nós não temos apenas uma tabela que liga os dois objetos, mas uma entidade própria, que tem seus próprios atributos e representação dentro do sistema. O nosso item representa o relacionamento entre as tabelas clientes e produtos, além de conter informações que caracterizam o relacionamento, que no nosso caso é a quantidade de produtos que o cliente tem. A tabela de relacionamento “clientes\_produtos” não precisa mais existir, pois a nova tabela “itens” já faz o trabalho dela. Vejamos agora como ficariam os códigos para esse nosso novo modelo:

*Novo Cliente.java*



O nosso cliente agora não mais se relaciona diretamente com os produtos, agora ele se relaciona com os itens, que por fim vão ser o relacionamento com os produtos. E já que falamos neles, vejamos a nossa classe Item:

*Item.java*

É na classe Item que reside agora o nosso relacionamento, ela contém uma referência para um Cliente e também para um Produto, além de guardar a quantidade de produtos que esse Item representa. Vejamos o exemplo de código que mostra esse relacionamento sendo utilizado:

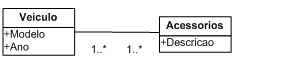
*Exemplo do relacionamento many-to-one*



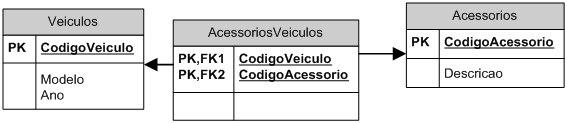
Como você pode perceber, as diferenças do código são pequenas, nós criamos um Cliente, um Produto e em vez de simplesmente relacionar os dois, nós criamos um novo objeto, o Item, que guarda uma referência para o Cliente e outra para o Produto, além disso ele também conta com uma propriedade, a quantidade. Seguindo no teste nós validamos que agora existem mais clientes e produtos que antes, além de ver se o item foi realmente relacionado ao cliente.

**09)Como se faz o mapeamento de relacionamentos muitos-para-muitos entre duas classes mapeadas? Dê exemplos.**

O modelo relacional não oferece suporte natural para associações muitos para muitos. Uma chave estrangeira não é suficiente para identificar a associação. Como não podemos ter campos muito valorados, precisamos criar uma tabela de ligação. Tecnicamente, uma tabela de ligação é formada pelas chaves primárias das tabelas relacionadas.



No mapeamento exemplificado abaixo, cada classe foi mapeada para uma tabela, adicionando-se uma coluna para chave. Em seguida, foi criada uma terceira tabela, contendo unicamente duas chaves estrangeiras que são também as chaves primárias de ambas as tabelas.



**Parte 3: O Gerenciador de Entidades**

**01)O que é o Entitymanager? Para que ele serve?  “Dê exemplos.”**

É o serviço central para todas as ações de persistência, sendo responsável pelas operações de persistência de objetos. O Entitymanager é uma classe que se interage com o banco de dados, assim o Entitymanager administra o O/R que o mapeia entre uma classe de entidade e uma fonte de dados subjacente. O Entitymanager provê APIs para criar consultas, buscando objetos, sincronizando objetos, e inserindo objetos no banco de dados. Também pode prover caching e pode administrar a interação entre uma entidade e serviços transacionais em um ambiente Java EE como JTA. O Entitymanager é firmemente integrado com Java EE e EJB, mas não é limitado a estes ambientes;  pode ser usado em programas de Java SE.

Exemplo de EntityManager:

*Public class ClienteRepository{*

*Private EntityManager manager;*

*Private EntityManagerFactory factory;*

*Private EntityTransaction transaction;*

*Public ClienteRepository(){*

*factory=Persistence.CreateEntityManagerFactory (“teste\_jpa”);*

*manager=factory.CreateEntityManager();*

*public void Save (Cliente obj) throws Exception{*

*try{*

*transaction=manager.getTransaction();*

*transaction.begin();*

*manager.persist(obj);*

*transaction.commit ();*

*}*

*Catch(Exception ex){*

*Transaction.rollback;*

*Throw ex;*

*}*

*}*

*}*

**02)Como se faz consultas no JPA? O que é o objeto Query?**

As consultas são feitas através do objeto query. Objeto Query é um padrão de projeto que visa libertar o programador de conhecer e/ou usar uma linguagem de pesquisa de dados como SQL. Martin Flowler define *Query Object* como uma especialização do padrão *interpreter* que constrói frases SQL de pesquisa com base numa estrutura de objetos. Exemplo de implementação desde conceito são os Criteria do Hibernate.

Existem dois tipos principais de consulta em JPA, consultas nomeadas e consultas dinâmicas. Uma consulta nomeada é usada para uma consulta estática que será usada muitas vezes no aplicativo. A vantagem de uma consulta nomeada é que ela pode ser definida uma vez, em um lugar, e reutilizada no aplicativo. A maioria dos provedores JPA também compila consultas nomeadas, para que eles sejam mais otimizados do que as consultas dinâmicas que normalmente devem ser compilados toda vez que forem executados. Consultas nomeadas são definidas através do NamedQuery e NamedQueries; consultas nomeadas são acessadas através do EntityManager.createNamedQuery.

As consultas dinâmicas são normalmente utilizadas quando a consulta depende do contexto. Por exemplo, dependendo de quais itens do formulário de consulta for preenchida, a consulta pode ter diferentes parâmetros. As consultas dinâmicas são também úteis para consultas incomuns, ou protótipos. As consultas dinâmicas são acessadas através do EntityManager.createQuery.  
  
**03)O que é JPQL? Dê exemplos.**

Java Persistence Query Language, que permite a construção de consultas usando uma linguagem mais próxima dos objetos e ao mesmo tempo é uma linguagem similar ao SQL. A JPQL é bem definida e flexível, tornando possível escrever quase todas as consultas que podem ser escritas em SQL, ela ate prove alguns comandos particulares de alguns gerenciadores de banco de dados.

Uma das melhorias sobre o SQL é a facilidade de expressar restrições sobre os relacionamentos entre as entidades nas consultas. Antes feito por meio dos comandos “INNER JOIN”, “OUTER JOIN” etc, o acesso a entidades relacionadas pode ser realizada simplesmente invocando a variável que representa este relacionamento. Por exemplo:

*@NamedQuery(  
                   name = "usersFromConditionGreaterEquals",  
                   query = "select v.user from CellValue v "  
                   + "where v.cell.matrix = ? and v.cell.knowledge = ? and v.cell.ability = ? and v.grade >= ? "),*

No exemplo acima, especificamos "select v.user from CellValue v". Isto quer dizer que queremos os usuários referenciados pelas instâncias persistidas da entidade CellValue e demos à entidade o apelido "v". Depois especificamos "where v.cell.matrix = ? and v.cell.knowledge = ? and v.cell.ability = ? and v.grade >= ?". Isto significa que queremos filtrar as instâncias da entidade CellValue. Os filtros aplicados são:

* "v.cell.matrix = ?" - a matriz referenciada pela célula referenciada pelas instâncias de CellValue deve ser igual à determinada referência passada como parâmetro
* "v.cell.knowledge = ?" - a área de conhecimento referenciada pela célula referenciada pelas instâncias de CellValue deve ser igual à determinada referência passada como parâmetro
* "v.cell.ability = ?" - a habilidade referenciada pela célula referenciada pelas instâncias de CellValue deve ser igual à determinada referência passada como parâmetro
* "v.grade >= ?" - a grau de competência referenciado pela célula referenciada pelas instâncias de CellValue deve ser maior ou igual à determinada referência passada como parâmetro

**04)O que é Criteria? Dê exemplos.**

É uma API orientada a objeto para busca, manipula objetos e suas propriedades tendo como resultado a coleção de objetos Java. Uma opção para o uso de Critérias para montar estruturas de consultas através de chamadas de métodos.

|  |  |  |
| --- | --- | --- |
| SQL | HQL | Criteria |
| “Select \*  from CLIENTE” | “from “ +  Cliente.class.getName(); | Criteria criteria = session.createCriteria  (Cliente.class); |

Outro Exemplo:

*List cats = session.createCriteria(Cat.class)*

*.add( Restrictions.like("name", "Iz%") )*

*.add( Restrictions.gt( "weight", new Float(minWeight) ) )*

*.addOrder( Order.asc("age") )*

*.list();*

Onde:

* Add: adicionar uma restrição para restringir os resultados a serem recuperados.
* Add order: Dê uma ordering para o conjunto de resultados.
* List: Obtenha os resultados.

**Construindo um CriteriaQuery**

A interface CriteriaBuilder é a fábrica de CriteriaQuery. A CriteriaBuilder é obtida a partir de qualquer um EntityManagerFactory ou um EntityManager como segue:

*EntityManager in = ... ; ;*

*QueryBuilder CriteriaBuilder em.getCriteriaBuilder = ();*

*qdef CriteriaQuery queryBuilder.createQuery = ();*

O primeiro passo na construção de uma definição de consulta é a especificação de raízes consulta. Consulta raízes especificar os objetos de domínio no qual a consulta é avaliada. Consulta de raiz é uma instância da interface <T> Raiz. Uma raiz de consulta é adicionada a um CriteriaQuery por addRoot(Class c) método.

*Raiz cliente <Customer> = qdef.from (Customer.class);*

Um domínio de consulta pode ser refinada por se juntar aos objetos de domínio. Por exemplo, para a definição da consulta acima para operar sobre os clientes e seus pedidos, use join(String attribute) :

*Raiz fim <Order> = customer.join (customer.get (Customer\_.orders));*

Onde Customer\_.orders representar um campo de classe do metamodelo canônica para o Cliente. Essas classes de metamodelo canônicos são gerados durante a compilação pelo processamento da anotação persistente no código-fonte do Customer.java.

A condição de uma definição de consulta é definido através de where(Predicate p) onde o argumento designa um predicado condicional. Predicados condicionais são frequentemente compostas 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 *"John Doe"* e tem pedidos que ainda não foram entregues, você pode construir o predicado e configurá-lo para a definição da consulta como:

*qdef.where (customer.get (Customer\_.name). iguais ("John Doe")*

*. E (order.get (Order\_.status). Iguais (OrderStatus.DELIVERED). ()));*

O select() método define o resultado da consulta. Se não especificado, a projeção de seleção é assumida como sendo o objeto do domínio raiz. No entanto, você pode especificar as projeções selecionadas explicitamente como uma lista:

*qdef.select (customer.get (Customer\_.name), order.get(Order\_.status));*

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

**Referências Bibliográficas**

<http://www.guj.com.br/content/articles/hibernate/intruducao_hibernate3_guj.pdf>

<http://www.hibernate.org/docs><http://www.hibernate.org/>

<http://pt.wikipedia.org/wiki/Hibernate>

<http://www3.ulbra-to.br/eventos/encoinfo/2009/Anais/Persistencia_de_Dados_em_Java_com_JPA_e_Toplink.pdf>

<http://en.wikipedia.org/wiki/TopLink>

<http://www.oracle.com/technetwork/middleware/toplink/overview/index.html>

<http://www.eclipse.org/eclipselink/>

<http://wiki.eclipse.org/EclipseLink/Examples/JPA>

<http://openjpa.apache.org/documentation.html>

http://openjpa.apache.org/builds/2.0.1/apache-openjpa/docs/jpa\_overview\_criteria.html

http://code.google.com/intl/pt-BR/appengine/docs/python/datastore/queryclass.html

http://www.lucasallan.com/2009/12/27/hibernate-utilizando-a-api-criteria/

http://www.slideshare.net/rodrigocasca/jpa

http://docs.jboss.org/hibernate/annotations/3.5/reference/en/html/entity.html

http://linu.com.br/papers/paper020.htm

<http://www.devmedia.com.br/articles/viewcomp.asp?comp=5206>

http://www.docstoc.com/docs/29667025/JPA-Passo-a-Passo