**Tranzactii si JPA**

O tranzactie este un mecanism ce manipuleaza un grup de operatii ca si cum ar fi o singura operatie. Intr-o trnzactie toate operatiile se executa sau niciuna. Operatiile implicate intr-o tranzactie se pot baza pe mai multe baze de date.

Un exemplu tipic de utilizare a unei tranzactii este urmatorul: o aplicatie client face o cerere de serviciu ce poate implica operatii multiple de citire si scriere pe o baza de date. Daca orice invocare este esuata, orice stare ce este scrisa trebuie facuta rolled back.

Concret, fie un transfer interbancar ce se executa prin intermediul unei aplicatii. Operatia de transfer cere serverului sa faca urmatoarele invocari:

* Metoda debit dintr-un cont al primei banci
* Metoda credit al altui cont al celei de-a doua banci

Daca invocarea pe cea de-a doua banca esueaza, aplicatia trebuie sa faca roll back contului din prima banca, la valoarea de dinaintea apelului.

O tranzactie este formal definita ca o multime de proprietati incluse in acronimul ACID:

* Atomicitate: o tranzactie este efectuata integral sau deloc. In caz de esec operatiile si procedurile se considera neefectuate si toate datele revin la starea anterioara tranzactiei
* Consistenta: o tranzactie transforma un sistem dintr-o stare consistenta intr-o alta stare consistenta
* Izolare: fiecare tranzactie se efectueaza independent de alte tranzactii ce se efectueaza in acelasi timp
* Durabilitate: tranzactiile efectuate cu succes devin permanente chiar daca sistemul esueaza

O aplicatie trebuie sa converseze cu un obiect pe care il gestioneaza si trebuie sa faca multiple invocari pe instanta obiectului. Conversatia poate fi caracterizata de unul sau mai multe dintre urmatoarele:

* Datele sunt cache-uite in memorie sau scrise in baza de date in timpul sau dupa fiecare invocare succesiva
* Datele sunt scrise in baza de date la sfarsitul conversatiei
* Aplicatia client cere ca obiectul sa mentina un context in memory intre fiecare invocare; fiecare invocare succesiva foloseste datele ce sunt intretinute in memorie
* La sfarsitul conversatiei aplicatia client necesita capabilitatea de a opri toate operatiile de scriere asupra bazei de date aparute pe timpul sau la sfarsitul conversatiei

Daca tranzactia a fost executata cu succes modificarile pastrate in memorie sunt comise (persistate).

Operatiile pe entitate sunt tranzactionale, adica este necesar sa fie parte a unei tranzactii. Avem doua modele tranzactionale suportate de JPA:

* resource-local, reprezinta tranzactiile native suportate de driver-ele JDBC in unitatea de persistenta
* JTA, sunt parte a server-ului Java EE

Interfata EntityTransaction suporta tranzactii resource-local, si este obtinuta din EntityManager prin apelul metodei getTransaction(). Metode obisnuite ale unei tranzactii includ:

* begin(), pornesc un nou context tranzactional
* commit(), incheie contextul tranzactional curent si scrie orice modificari nescrise inca in baza de date
* rollback(), readuce tranzactia la starea initiala
* setRollbackOnly(), seteaza tranzactia actuala astfel incat la incheiere aceasta sa faca rollback. Una dintre intrebuintari este stergerea unei tranzactii care dureaza prea mult
* getRollbackOnly(), returneaza un boolean ce indica daca tranzactia curenta este marcata pentru rollback
* isActive(), returneaza un boolean ce indica daca resursa tranzactiei este in procesare

interfata EntityManager defineste metodele ce gestioneaza entitatile:

* persistenta unei entitati: preia o entitate ce nu are o reprezentare persistenta in baza de date si o stocheaza. Dupa persistenta entitatea este in starea managed. Orice alte urmatoare modificari pe acesta entitate sunt de asemenea persistate
* gasirea unei entitati: daca entitatea este localizata in contextul persistent aceasta este returnata de metoda find si gestionata de entity manager
* stergerea unei entitati
* modificarea unei entitati: localizarea unei entitati existente si efectuarea modificarilor ce apoi sunt persistate

Aceste operatii formeaza CRUD (create, read, update, delete) pe baza de date.

Persistarea unei entitati este operatia de scriere a entitatii in baza de date. Este echivalenta clauzei INSERT din SQL.

**public** **class** Test {

**public** **static** **void** main(String[] args) {

EntityManagerFactory emf =

Persistence.*createEntityManagerFactory*("NIO2");

EntityManager em = emf.createEntityManager();

Broker brk = **new** Broker ();

brk.setBrokerId(3);

brk.setAddress("lunga 9");

brk.setBrokername("ion");

em.getTransaction().begin();

em.persist (brk);

em.getTransaction().commit();

em.close();

emf.close();

}

}

In cod, EntityManagerFactory este obtinut folosind clasa Persistence si numele configuratiei unitatii de persistenta. Din factory obtinem EntityManager-ul.

Pentru a localiza o entitate in baza de date, EntityManager-ul va localiza o linie in baza de date folosindu-se de cheia primara.

Metoda find() utilizeaza clasa entitate, trimisa ca prim argument, pentru a determina ce tip sa foloseasca pentru a cheia primara si pentru tipul returnat. Astfel, o extra conversie nu mai este necesara. Entitatea returnata se afla in starea managed. Daca nu exista entitatea ceruta se returneaza null. Cel de-al doilea argument al lui find() este Object. Metoda este echivalenta cu clauza SELECT din SQL.

Broker emp = em.find (Broker.**class**, 110);

Dupa ce o entitate se afla in starea managed putem face modificari obiectului entitate fara persistarea explicita a sa. Acest proces este echivalentul clauzei UPDATE in SQL. Pentru a updata o entitate ce nu este in starea managed va trebui sa folosim metoda merge() pentru a o aduce in aceasta stare.

em.getTransaction().begin();

emp.setBrokername(emp.getBrokername()+" Ionescu");

em.getTransaction().commit();

Stergerea unei entitati din baza de date este echivalenta cu clauza DELETE din SQL. Stergerea entitatii trebuie facuta pe o entitate in starea managed. Dupa remove() obiectul Java este in memorie.

em.getTransaction().begin();

em.remove(emp);

em.getTransaction().commit();

O entitate in starea managed necesita un context tranzactional. Exista situatii in care dorim sa „unmanage” o entitate pentru a face schimbarile inaintea de returnarea entitatii la starea managed.

In exemplu, entitatea a fost detached prin entity manager. Dupa aceasta operarea pe ea se face ca si pentru un obiect Java oarecare. Dupa ce am facut modificarile si au trecut validarile entitatea poate fi din nou facuta managed.

em.detach(emp);

emp.setBrokername("Jones");

// ... in some other part of the code

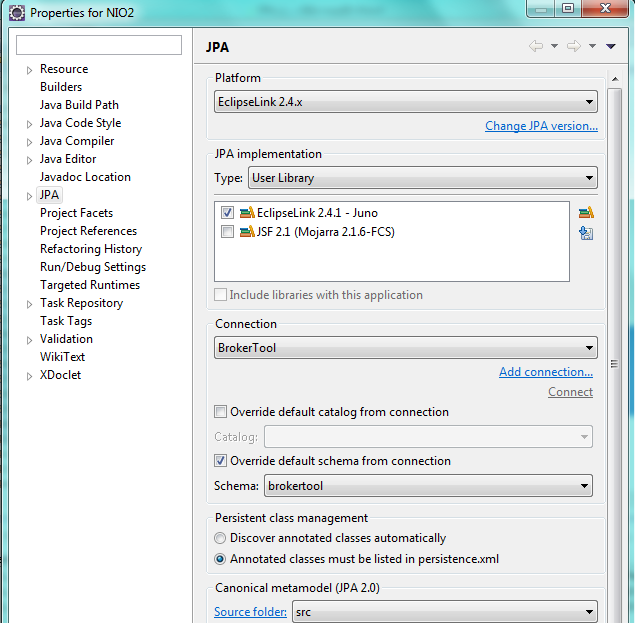
em.getTransaction().begin();

emp = em.merge(emp);

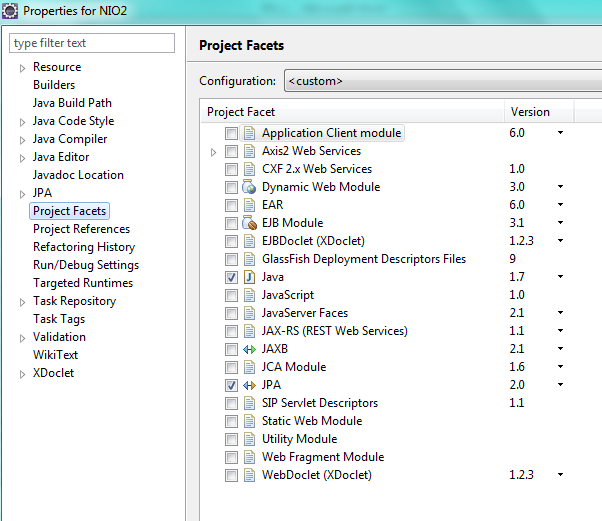
emp.setAddress(emp.getAddress() + "2");

em.getTransaction().commit();

Pentru a putea rula aplicatiile din acest curs va trebui sa creem un Java Project clasic si din Project>Properties sa setam valorile ca mai jos:



Facet-ul JPA apare in urma setarii:



Fisierul persistence.xml va trebui sa contina:

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

<persistence version=*"2.0"* 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"*>

<persistence-unit name=*"NIO2"* transaction-type=*"RESOURCE\_LOCAL"*>

<provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>

<class>model.Broker</class>

<properties>

<property name=*"javax.persistence.jdbc.url"* value=*"jdbc:mysql://localhost:3306/brokertool"*/>

<property name=*"javax.persistence.jdbc.user"* value=*"root"*/>

<property name=*"javax.persistence.jdbc.password"* value=*"root"*/>

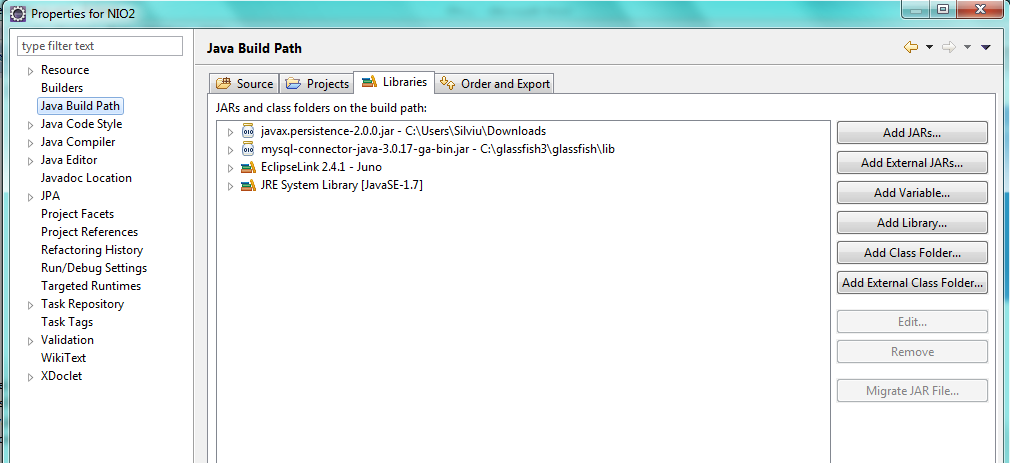
<property name=*"javax.persistence.jdbc.driver"* value=*"com.mysql.jdbc.Driver"*/>

</properties>

</persistence-unit>

</persistence>

Iar setarile pe proiect vor fi:



**Limbajul Java Persistence Query**

Defineste interogari ale entitatilor si starea lor persistenta. Limbajul de interogare utilizeaza scheme abstracte de persitenta a entitatilor, incluzand relationarea, pentru modelul de date si defineste operatori si expresii bazate pe modelul de date. Domeniul unei interogari cuprinde schemele abstracte ale entitatilor ce coexista in acelasi persistence unit. Query language utilizeaza sintaxa asemanatoare cu SQL pentru a selecta obiecte sau valori bazandu-se pe tipurile entitate ale schemei abstracte si a relationarii dintre ele.

Avem urmatorul dictionar de termeni frecvent utilizati in cele ce urmeaza:

* Abstract schema, reprezinta o abstractie a schemei persistente, adica entitatile persistente, starea lor si relationarile, peste care interogarile opereaza. Query language traduce interogarile, peste aceasta abstractizare a schemei persistente, in interogari ce sunt executate peste schema bazei de date de unde entitatile sunt mapate
* Abstract schema type, reprezinta tipul la care proprietatea persistenta a unei entitati se evalueaza in schema abstracta. Astfel, fiecare camp persistent sau proprietate dintr-o entitate are un camp de stare corespondent, de acelasi tip, in abstract schema. Tipul schemei abstracte al unei entitati este derivat dintr-o clasa entitate iar informatiile de metadate sunt furnizate de anotatii
* Backus-Naur Form (BNF), reprezinta o notatie ce descrie sintaxa unui limbaj de nivel inalt. Toate diagramele, in cele ce urmeaza vor fi BNF.
* Navigation, reprezinta traversarea unei relationari intr-o expresie query language. Operatorul de navigare este punctul
* Path expression, este o expresie ce navigheaza catre starea unei entitati sau camp al relatiei
* State field, un camp persistent al unei entitati
* Relationship field, reprezinta un camp persistent al unei relationari dintr-o entitate al carei tip este un tip in schema abstracta al entitatii relationate

**Crearea interogarilor utilizand JPQL**

Interogarile pot fi dinamice sau statice (anotate cu @NamedQuery sau @NamedNativeQuery)

Metodele EntityManager.createQuery() si EntityManager.createNamedQuery() sunt utilizate pentru a interoga datele utilizand JPQL.

Metoda createQuery() este utilizata pentru a crea interogari dinamice, ceea ce inseamna interogari definite direct in codul aplicatiei.

Metoda createNamedQuery() este utilizata pentru a crea interogari statice sau interogari definite in metadata utilizand anotatia javax.persistence.NamedQuery. Elementul name specifica numele interogarii, ce va fi folosit impreuna cu metoda createNamedQuery().

Parametrii denumiti (named parameter) sunt parametri ai interogarii prefixati de :. Parametrii denumiti intr-o interogare sunt legati de un argument prin metoda: javax.persistence.Query.setParameter(String name, Object value)

Parametrii denumiti sunt case-sensitive si pot fi folositi atat de interogarile dinamice cat si de cele statice.

Exemplu:

**public** **void** salut(String name) {

List<Customer> l= em.createQuery("SELECT c FROM Customer c WHERE c.name LIKE :custName").setParameter("custName", name)

.setMaxResults(10)

.getResultList();

**for** (Customer object: l) {

System.*out*.println(object.getName());

}

}

Parametrii pozitionali sunt prefixati de ? urmat de un numar ce identifica pozitia parametrului in interogare. Metoda

Query.setParameter(integer position, Object value)

este folosita pentru a seta valorile acestor parametri.

Parametrii de intrare sunt numerotati incepand cu 1, sunt case-sensitive si pot fi utilizati atat de interogarile dinamice cat si de cele statice.

Interogarile dinamice nu sunt altceva decat string-uri. Named query sunt statice si nemodificabile. Ele sunt mai eficiente deoarece furnizorul de persistenta poate translata string-ul JPQL in SQL de indata ce aplicatia porneste, in loc sa fie pornita de fiecare data cand interogarea este executata.

Exemplu pentru folosirea parametrilor pozitionali si de creare a interogarilor statice.

In directorul ejbModule>META-INF din PrimulEJB am editat fisierul orm.xml (deschis in eclipse si ales tabul Design). Click dreapta pe entity-mappings am ales Add Child si de acolo named-query. Atributul name are valoarea full iar query este SELECT c FROM Customer c WHERE c.name LIKE ?1. Am folosit aceasta in:

**public** **void** salut(String name) {

List<Customer> l= em.createNamedQuery("full")

.setParameter(1, name)

.getResultList();

**for** (Iterator<Customer> iterator = l.iterator(); iterator.hasNext();) {

Customer object = iterator.next();

System.*out*.println(object.getName());

}

}

Anotatia @NamedQuery poate fi plasata in definitia clasei entitatii. Anotatia defineste numele interogarii si textul acesteia.

@Entity

@Table(name = "EMPLOYEE")

@NamedQueries({

@NamedQuery(name = "Employee.findAll", query = "SELECT e FROM Employee e"),

@NamedQuery(name = "Employee.findByEmpId", query = "SELECT e FROM Employee e WHERE e.empId = :empId"),

@NamedQuery(name = "Employee.findByFirstname", query = "SELECT e FROM Employee e WHERE e.firstname = :firstname"),

@NamedQuery(name = "Employee.findByLastname", query = "SELECT e FROM Employee e WHERE e.lastname = :lastname"),

@NamedQuery(name = "Employee.findByBirthdate", query = "SELECT e FROM Employee e WHERE e.birthdate = :birthdate"),

@NamedQuery(name = "Employee.findBySalary", query = "SELECT e FROM Employee e WHERE e.salary = :salary")})

public class Employee implements Serializable {

Daca avem de definit mai mult de un named query intr-o clasa, le vom plasa intr-o anotatie @NamedQueries, ce accepta un sir de @NamedQuery

**Sintaxa Query Language**

JPQL suporta:

* Rezultatul poate fi de un singur tip sau de mai multe tipuri
* Sortare si grupare
* Functii agregat, expresii cu conditii si subinterogari
* Sintaxa cu jonctiuni
* Interogari ce permit stergeri sau update
* Captarea rezultatelor in clase nepersistente

Interfetele Query si TypedQuery pot fi folosite pentru a scrie interogari.

O interogare select are sase clauze: SELECT, FROM, WHERE, GROUP BY, HAVING si ORDER BY. Dintre acestea doar SELECT si FROM sunt obligatorii. Dam in continuare sintaxa BNF pentru o interogare select folosind query language:

QL\_statement ::= select\_clause from\_clause

[where\_clause][groupby\_clause][having\_clause][orderby\_clause]

Clauza SELECT defineste tipurile de obiecte sau valorile returnate de interogare.

Clauza FROM defineste domeniul interogarii prin declararea unuia sau mai multor identificatori de variabila, care pot fi referiti in clauzele SELECT si WHERE. Un identificator de variabila reprezinta unul dintre urmatoarele elemente:

* Numele schemei abstracte al unei entitati
* Un element al unei colectii de relationare
* Un element al unei relatii cu o singura valoare

Clauza WHERE este o expresie conditionala ce restrictioneaza obiectele sau valorile returnate de o cerere.

Clauza GROUP BY grupeaza rezultatele interogarii pe baza unui set de proprietati

Clauza HAVING este utilizata impreuna cu GROUP BY pentru a restrictiona rezultatele unei interogari in raport cu o expresie conditionala.

Clauza ORDER BY sorteaza obiecteel sau valorile returnate de o interogare.

Instructiunile de modificare si stergere furnizeaza operatii peste multimi de entitati. Aceste instructiuni au urmatoarea sintaxa:

update\_statement :: = update\_clause [where\_clause]

delete\_statement :: = delete\_clause [where\_clause]

Clauzele determina tipul entitatilor ce vor fi modificate sau sterse.

Exemple:

SELECT c

FROM Customer c

Returneaza toti clientii. Clauza FROM declara un identificator de variabila numit c, omitand cuvantul cheie AS. Daca acesta este inclus, clauza va fi scrisa ca FROM Customer AS c. Elementul Customer este numele schemei abstracte al entitatii Customer.

Pentru a obtine clientii distincti ce satisfac o anumita conditie:

SELECT DISTINCT c

FROM Customer c

WHERE c.name = ?1

Cuvantul rezervat DISTINCT elimina valorile duplicat. In aceasta interogare am folosit un parametru de intrare ?1.

Acelasi stil de interogare, dar folosind parametri denumiti este prezentata mai jos:

SELECT DISTINCT c

FROM Customer c

WHERE c.name = :name1 OR c.name = :name2

**Interogari ce navigheaza intre entitati relationate**

In query language o expresie poate naviga intre entitati relationate. Aceasta este prima diferenta intre JPQL si SQL. In acest caz interogarile navigheaza intre entitatile relationate in timp ce SQL jonctioneaza tabele.

Exemplu:

List<Customer> l= em.createQuery(

"SELECT c FROM Customer c , IN(c.savingsaccounts) s")

.setMaxResults(10)

.getResultList();

Returneaza toti clientii care au un cont de economii. Clauza FROM declara doi identificatori de variabila c si s. Variabila c reprezinta entitatea Customer, iar s entitatea relationata SavingsAccount. Ordinea declararii este stricta, intai c si apoi s. Cuvantul rezervat IN semnifica ca savingsaccounts este o colectie de entitati relationate. Expresia c.savingsaccounts navigheaza de la un Customer la SavingsAccount-ul relationat. Operatorul punct din expresie este operatorul de navigare. Clauza din exemplul anterior poate fi rescrisa echivalent:

SELECT c FROM Customer c WHERE c.savingsaccounts IS NOT EMPTY

Navigarea catre un camp al unei relationari se face utilizand clauza JOIN.

SELECT c FROM Customer c JOIN c.savingsaccounts s WHERE s.balance=6

In aceasta interogare returnam toti clientii care au in cont 6 unitati monetare.

In continuare dam un exemplu in care folosim un parametru de intrare:

SELECT c FROM Customer c , IN (c.savingsaccounts) AS s WHERE s.balance = :balance

Interogarea returneaza clientii care au un anumit nivel al economiilor, nivel introdus ca paarmetru.

Expresiile pot naviga catre entitati relationate si nu catre campuri persistente. Pentru a accesa un camp persistent o expresie utilizeaza punctul ca delimitator.

Observatie: O specificare de forma c.savingsaccounts.balance este ilegala pentru ca savingsaccounts este o colectie.

Orice clauza WHERE trebuie sa specifice o expresie conditionala. Vom da un exemplu de folosire a cuvantului cheie LIKE in construirea unei expresii conditionale:

SELECT c FROM Customer c WHERE c.name LIKE 'sil%'

Interogarea returneaza toti clientii al caror nume incepe cu sil. LIKE foloseste caractere wildcard pentru a cauta stringuri care se potrivesc unui anumit sablon. In cazul nostru am folosit %.

Expresia IS NULL poate fi folosita pentru a verifica daca o relationare a fost stabilita intre doua entitati.

Diagrame BNF in Java EE 6 Tutorial, paginile 585-617

**Folosirea Criteria API pentru a crea Queries**

Criteria API este folosit pentru a defini interogari ale entitatilor, cu ajutorul obiectelor. Criteria queries sunt definite folosind API Java, sunt typesafe si sunt portabile. Similar lui JPQL, Criteria API este bazat pe schema abstracta a entitatilor persistente, relationarii si obiectelor incluse. Criteria API permite dezvoltatorilor sa gaseasca, sa modifice si sa stearga entitatile persistente invocand operatii pe entitati prin JPA.

Criteria API:

* Standardizeaza multe dintre facilitatile de programare ce exista in produsele de persistenta
* Aplica best practices de programare ale modelelor
* Folosesc la maxim facilitatile de programare ale limbajului Java

Pasii de baza in vederea crearii unei interogari Criteria sunt:

1. utilizam instanta EntityManager pentru a crea un obiect CriteriaBuilder. Aceasta se obtine prin apelul metodei getCriteriaBuilder() din instanta EntityManager.
2. cream un obiect interogare prin crearea unei instante a interfetei CriteriaQuery. Acesta se obtine prin apelul metodei createQuery() din instanta CriteriaBuilder. Parametrul metodei este tipul returnat de interogare si este specificat pentru a crea o interogare typesafe. Acest obiect va fi modificat cu detaliile unei interogari concrete
3. setam query root-ul prin apelul metodei from() din obiectul CriteriaQuery
4. setam tipul rezultatului prin apelul metodei select() din CriteriaQuery
5. pregatim interogarea pentru executie prin crearea unei instante TypedQuery<T> specificand tipul rezultatului interogarii
6. executam interogarea prin apelul metodei getResultList() din obiectul TypedQuery<T>.

Pentru un obiect CriteriaQuery, entitatea radacina a interogarii, de la care pornesc toate navigarile se numeste query root. Aceasta este similara clauzei FROM dintr-o interogare JPQL.

Vom considera urmatoarea linie de cod care va returna rezultatele unei interogari simple:

List<Customer> l= em.createQuery("SELECT c FROM Customer c").getResultList();

Transpunem aceasta interogare intr-una echivalenta creata cu API-ul Criteria urmarind pasii descrisi anterior:

CriteriaBuilder cb = em.getCriteriaBuilder();

CriteriaQuery<Customer> cq = cb.createQuery(Customer.**class**);

Root<Customer> cust = cq.from(Customer.**class**);

cq.select(cust);

TypedQuery<Customer> q = em.createQuery(cq);

List<Customer> l = q.getResultList();

API-ul Metamodel este folosit pentru a crea un metamodel al entitatilor in unitatea de persistenta. Pentru fiecare clasa entitate dintr-un pachet, clasa metamodel este creata cu un nume identic, dar urmat de underscore. Atributele clasei metamodel corespund campurilor persistente sau proprietatilor clasei entitate.

Clasa metamodel si atributele sale sunt utilizate in interogarile Criteria pentru a referi clasele entitate gestionate si starea lor de persitenta si relationare.

Clasele metamodel ce corespund claselor entitate sunt de urmatorul tip: javax.persistence.metamodel.EntityType<T>. Clasele metamodel sunt de obicei generate de anotatii fie la rulare fie la dezvoltare. Dezvoltatorii ce utilizeaza interogari Criteria pot genera clase metamodel statice folosind apelul metodei getModel() pe obiectul query root sau obtinand o instanta a interfetei Metamodel si apoi trimitand tipul entitate metodei entity(). Ambele cazuri sunt prezentate mai jos;

CriteriaBuilder cb = em.getCriteriaBuilder();

CriteriaQuery cq = cb.createQuery(Customer.class);

Root<Customer> cust = cq.from(Customer.class);

EntityType<Customer> Customer\_ = cust.getModel();

sau

Metamodel m = em.getMetamodel();

EntityType<Customer> Customer\_ = m.entity(Customer.class);

Sintaxa de baza a interogarilor Criteria consta intr-o clauza SELECT, una FROM si o clauza optionala WHERE, similar cu JPQL. Interogarile Criteria seteaza aceste clauze prin utilizarea unor obiecte Java astfel incat, interogarile pot fi create typesafe.

Interfata javax.persistence.criteria.CriteriaBuilder este folosita pentru a construi:

* interogari Criteria
* selectii
* expresii
* predicate

Interogarile Criteria sunt construite prin obtinerea unei instante a interfetei: javax.persistence.criteria.CriteriaQuery. Obiectele CriteriaQuery definesc o interogare particulara ce navigheaza peste una sau mai multe entitati. CriteriaQuery este typesafe.

Pentru un anume obiect CriteriaQuery entitatea radacina a interogarii, adica entitatea de la care pornesc toate navigarile se numeste query root. Este similara, ca semnificatie, clauzei FROM din JPQL. Interogarile Criteria pot avea mai multe query roots, in cazul in care navigam din mai multe entitati.

CriteriaQuery<Customer> cq = cb.createQuery(Customer.class);

Root<Customer> cust1 = cq.from(Customer.class);

Root<Customer> cust2 = cq.from(Customer.class)

Putem obtine un Root si dintr-un EntityType.

Root<Customer> cust = cq.from(Customer \_);

Pentru interogari ce navigheaza catre entitati relationate interogarea trebuie sa definesca o jonctiune catre entitatea relationata prin apelul uneia dintre metodele From.join() din obiectul query root. Aceasta metoda este similara JOIN-ului din JPQL.

Destinatia jonctiunii utilizeaza clasa metamodel de tip EntityType<T> pentru a specifica campul persistent sau proprietatea entitatii jonctionate.

Metoda join() returneaza un obiect Join<X, Y> unde X este entitatea sursa si Y este entitatea destinatie.

Iata un exemplu de jonctionare intre entitatea Customer si SavingsAccount.

Metamodel m = em.getMetamodel();

EntityType<Customer> Customer\_ = m.entity(Customer.**class**);

CriteriaQuery<Customer> cq = cb.createQuery(Customer.**class**);

Root<Customer> cust = cq.from(Customer.**class**);

Join<Customer, SavingsAccount> x = (Join<Customer, SavingsAccount>) cust.join(Customer\_.getList("savingsaccounts"));

Jonctiunile pot fi inlantuite pentru a naviga catre entitatile destinatie fara a avea o instanta Join<X, Y> pentru fiecare jonctiune.

Obiectele Path sunt utilizate in clauzele SELECT si WHERE ale unei interogari Criteria si pot fi entitati query root, entitati join sau alte obiecte. Metoda Path.get() este folosita pentru a naviga catre atributele unei entitati a unei interogari. Argumentul metodei este atributul corespondent din clasa metamodel. Atributul poate fi single-valued, specificat prin anotatia @SingularAttribute in clasa metamodel sau un atribut collection-valued specificat printr-una dintre anotatiile @CollectionAttribute, @SetAttribute, @ListAttribute sau @MapAttribute.

Vom da un exemplu prin care am selectat un camp al unei entitati folosind un metamodel.

**public** List<AllCustomer> salut(String name) {

CriteriaBuilder cb = em.getCriteriaBuilder();

CriteriaQuery<Customer> cq = cb.createQuery(Customer.**class**);

Root<Customer> cust = cq.from(Customer.**class**);

Metamodel m = em.getMetamodel();

EntityType<Customer> Customer\_ = m.entity(Customer.**class**);

SingularAttribute<Customer, String> custName = Customer\_.getDeclaredSingularAttribute("name", String.**class**);

cust.get(custName);

TypedQuery<Customer> q = em.createQuery(cq);

List<Customer> l = q.getResultList();

List<AllCustomer> lac =**new** ArrayList<AllCustomer>();

**for** (Iterator<Customer> iterator = l.iterator(); iterator.hasNext();) {

Customer customer = (Customer) iterator.next();

lac.add(conv.fromEntity(customer));

}

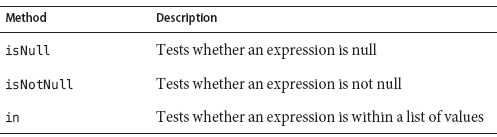
**return** lac;

}

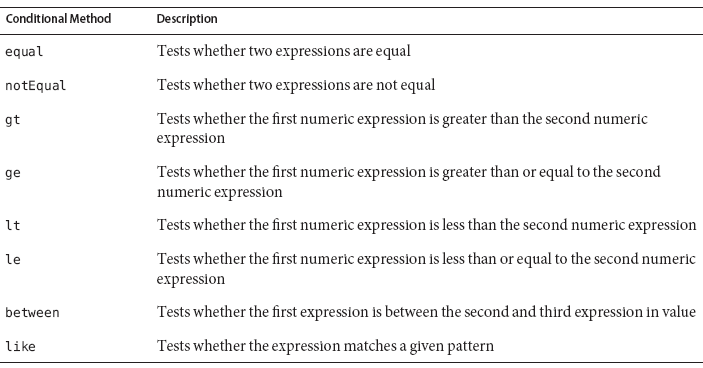
Rezultatele unei interogari pot fi restrictionate in acord cu conditiile setate de apelul metodei CriteriaQuery.where(). Apelul acestei metode este analog cu clauza WHERE din JPQL.

Metoda where() evalueaza instantele interfetei Expression pentru a restrictiona rezultatele in acord cu conditiile expresiilor. Instantele Expression sunt create utilizand metodele definite in interfetele Expression si CriteriaBuilder.

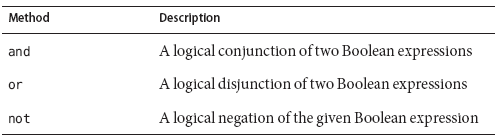
Obiectele Expression sunt folosite in clauzele SELECT, WHERE sau HAVING. Metodele conditionale utilizate in obiectele Expression sunt descrise in tabelul urmator:



In interfata CriteriaBuilder avem si alte metode pentru crearea expresiilor. Aceste metode corespund operatorilor si functtilor atritmetice, string, data, timp. Tabelul urmator ilustreaza aceste metode.



Predicatele formate din conditii multiple pot fi specificate utilizand urmatoarele metode din CriteriaBuilder.



Urmatorul exemplu urmareste crearea unui predicat si folosirea lui pentru a selecta din tabele customer acele nume care incep cu sil.

**public** List<AllCustomer> salut(String name) {

CriteriaBuilder cb = em.getCriteriaBuilder();

CriteriaQuery<Customer> cq = cb.createQuery(Customer.**class**);

Root<Customer> cust = cq.from(Customer.**class**);

Metamodel m = em.getMetamodel();

EntityType<Customer> Customer\_ = m.entity(Customer.**class**);

SingularAttribute<Customer, String> custName = Customer\_.getDeclaredSingularAttribute("name", String.**class**);

Path<String> pa=cust.get(custName);

Predicate p = cb.like(pa,"sil%");

cq=cq.where(p);

TypedQuery<Customer> q = em.createQuery(cq);

List<Customer> l = q.getResultList();

List<AllCustomer> lac =**new** ArrayList<AllCustomer>();

**for** (Iterator<Customer> iterator = l.iterator(); iterator.hasNext();) {

Customer customer = (Customer) iterator.next();

lac.add(conv.fromEntity(customer));

}

**return** lac;

}

Pentru interogari ce returneaza mai mult decat un singur rezultat avem uneori nevoie sa organizam iesirea. In interfata CriteriaQuery avem metoda orderBy() folosita pentru a ordona rezultatele in raport cu atributele entitatii. Argumentul metodei este un obiect Order, creat prin apelul uneia dintre metodele: CriteriaBuilder.asc() sau CriteriaBuilder.desc(). Prima dintre metode este folosita pentru a ordona rezultatele crescator, dupa expresia parametru, iar cea de-a doua pentru ordonarea descrescatoare. Metodei orderBy() ii putem trimite mai multe obiecte Order, precedenta fiind determinata de ordinea aparitiei in lista.

Metoda CriteriaQuery.groupBy() partitioneaza rezultatele interogarii in grupuri. Formarea acestor grupuri este data de argumentul metodei.

Metoda CriteriaQuery.having()este folosita impreuna cu groupBy() pentru a filtra grupurile. having() are ca parametru o expresie conditionala.

Pentru a pregati un query pentru executie cream un obiect TypedQuery<T> cu tipul rezultatului interogarii transmis de obiectul CriteriaQuery lui EntityManager.createQuery(). Interogarile sunt executate folosind setSingleResult() sau getResultList() pe obiectul TypedQuery<T>.

**String-Based Criteria API**

Reprezinta interogari ce folosesc stringuri in loc de obiectele metamodel pentru a specifica atribute atunci cand traversam o ierarhie de date. Interogarile string-based sunt construite similar interogarilor metamodel, pot fi statice sau dinamice.

Interogarile metamodel sunt recomandate in folosirea interogarilor. Avantajul interogarilor string-based este acela ca nu trebuie sa mai cream un metamodel. Marele dezavantaj este pierderea importantei proprietati de type safety, ceea ce va duce la erori la rulare (nedetectabila la rulare).

Pentru a crea o interogare string-based specificam numele atributelor claselor entitate direct ca stringuri, nu ca atribute ale clasei metamodel.

Spre exemplu avem urmatoarul apel simplificat:

cq.where(cb.equal(**cust.get("name")**, "ion"));

Si aceste interogari se executa similar cu cele de la Criteria API.