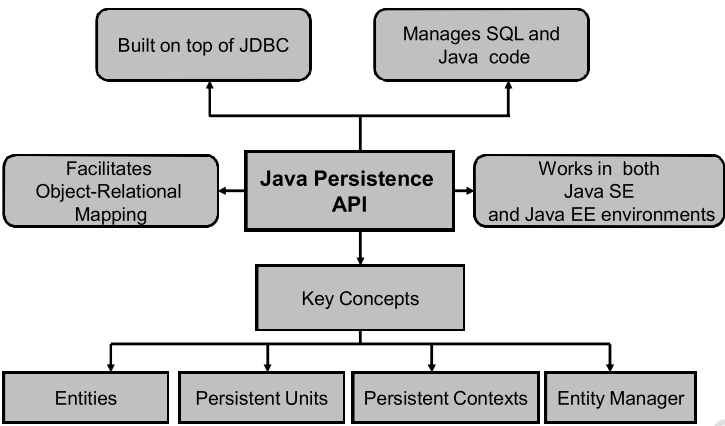
**JPA (Java Persistence API)**

JPA este un framework lightweight ce foloseste POJO (Plain Old Java Objects) pentru a persista obiecte Java ce reprezinta date relationale. JPA 1.0 a inceput ca parte a specificatiilor EJB 3.0 pentru a standardiza un model pentru ORM (object-relational mapping). JPA 2.0 (JSR-317) imbunatateste specificatiile originale.

POJO face parte, de asemenea, din specificatiile EJB 3.0. Orice obiect normal este un obiect POJO (altfel spus un obiect non-enterprise).

Beneficiile utilizarii JPA:

* Nu trebuie sa cream obiecte complexe de acces la date (DAO)
* API-ul este folosit pentru a gestiona tranzactii
* Codul de interactiune cu baza de date este standard, indiferent de vendorul bazei de date relationale
* Putem evita SQL si in schimb putem folosi un query language, orientat pe obiect
* Putem folosi JPA pentru persistenta aplicatiilor desktop

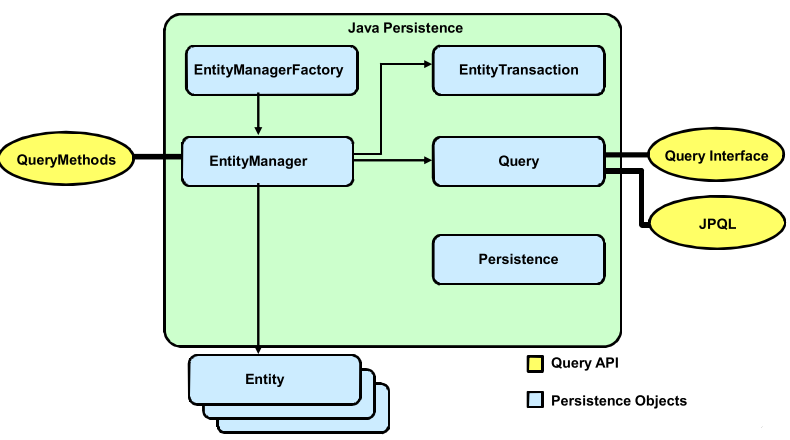


JDBC a fost primul mecanism pe care dezvoltatorii Java l-au folosit pentru a persista date. JPA este un framework de mapare ce pastreaza abilitatea de a manipula baza de date direct.

Conceptele fundamentale ale JPA sunt:

* Entitate, este utilizata pentru a reprezenta un tabel relational intr-un obiect Java
* Unitate de persistenta, defineste multimea tuturor claselor ce au legatura cu aplicatia si care sunt mapate unei singure baze de date
* Context persistent, este o multime de instanta de entitati in care este o unica instanta a entitatii pentru orice identificator de entitate persistenta
* Entity manager, face munca de creare, citire si scriere a entitatilor

**Componentele JPA**



Persistenta este mecanismul utilizat de aplicatii pentru a pastra datele, date ce altfel ar fi pierdute la inchiderea aplicatiei sau a calculatorului, intr-un context persistent, precum o baza de date.

JPA stabileste relationari statice ale modelului persistent, prin definirea componentelor entitate. API-ul defineste clasa entitate ca un echivalent al unei tabele din baza de date, pe partea de business a aplicatiei. O instanta a entitatii este definita ca un obiect, echivalent al unei linii din tabela bazei de date.

JPA stabileste relationarea dinamica a modelului persistent prin definirea unui obiect entity manager. Acesta este insarcinat cu sincronizarea datelor continute intr-o instanta entitate cu datele continute in linia echivalenta din baza de date. Spre exemplu, daca pe timpul executiei aplicatiei un camp al instantei entitate este modificat, entity managerul modifica linia echivalenta din baza de date.

JPA furnizeaza programatorilor Java facilitatea de mapare obiect/relatie, pentru a gestiona modelul relational al bazelor de date implicate in aplicatiile Java. Persistenta in Java consta din:

* JPA
* Limbajul de interogare (JPQL)
* Java Persistence Criteria API
* Metadatele de mapari obiect/relatie

In figura anterioara avem:

* Entity, obiectul persistent ce reprezinta o inregistrare din tabela bazei de date. Este un POJO cu anotatii
* Interfata EntityManager, furnizeaza API-ul de interactiune cu entitatea
* EntityManagerFactory este folosit pentru a crea o instanta de EntityManager.

Un obiect Java poate cuprinde date partiale dintr-o tabela sau date din mai multe tabele.

Framework-urile ORM gestioneaza maparea intre tabelele bazei de date relationale si obiectele Java astfel incat programatorii sa codeze cat mai putin. EclipseLink si Hibernate sunt exemple de soft-uri ORM.

Observatie: EclipseLink este o versiune continuata a TopLink, pe care Oracle a donat-o Eclipse-ului.

**Entitati**

Asadar, o entitate reprezinta o structura de date prin intermediul careia se asigura persitenta. O entitate este asociata unui tabel al unei baze de date relationale, iar fiecare instanta a entitatii corespunde unei linii din tabela. Campul unei clase entitate corespunde unei coloane din tabela bazei de date.

O entitate:

* Este un POJO creata utilizand cuvantul new
* Suporta mostenire si polimorfism
* Este serializata si poate fi utilizata ca un obiect detached

Caracteristicile JPA ale entitatilor includ:

* Persistenta, entitatile create pot fi persistate intr-o baza de date, dar ele sunt tratate ca obiecte
* Identitatea, fiecare entitate are un unic obiect identitate care este de obicei asociat unei chei in baza de date, de obicei cheia primara
* Tranzactii, JPA necesita un context tranzactional pentru operatii ce efectueaza schimbari in baza de date
* Granularitate, entitatile pot fi oricat de fine-grained sau coarse-grained precum doreste dezvoltatorul, dar reprezinta o singura linie dintr-un tabel

Starea de persistenta a unei entitati este reprezentata prin intermediul acestor campuri sau prin intermediul proprietatilor persistente (vezi in paragrafele urmatoare diferenta). Toate acestea folosesc anotatii pentru maparea entitatilor si datelelor modelului relational.

O clasa entitate trebuie sa indeplineasca urmatoarele cerinte:

* Clasa trebuie sa fie anotata cu anotatia javax.persistence.Entity
* Clasa trebuie sa aiba un constructor public sau protected, fara argumente, dar clasa mai poate avea si alti contructori
* Clasa nu trebuie declarata final, la fel metodele sau instantele clasei
* Daca o instanta a clasei este trimisa ca parametru al unei metode, precum in cazul interfetelor remote session bean, clasa trebuie sa implementeze interfata Serializable
* Clasele entitate pot extinde atat clase entitate cat si non-entitate, iar clasele non-entitate pot extinde clase entitate
* Variabilele instanta trebuie sa fie declarate private, protected sau package-friendly si pot fi accesate direct doar prin intermediul metodelor claselor entitate. Clientii trebuie sa acceseze starea entitatii prin accesori sau metode business
* Campul cheie primara sau metoda getter ce corespunde coloanei cheii primare va fi anotata cu @Id
* Clasele entitate nu definesc metoda finalize().

Setarile predefinite ale unei clase entitate pot fi suprascrise, astfel:

* Suprascrierea numelui tabelei bazei de date. In mod predefinit numele clasei entitate este numele tabelei (scris cu majuscula). Pentru a modifica acest comportament folosim anotatia @Table cu elementul name. Ex:

@Entity

@Table(name=”Cust”) //Cust este numele tabelei din baza de date

public class Client {

* Suprascrierea numelui coloanei tabelei bazei de date. In mod predefinit numele campului sau proprietatii clasei entitate este numele coloanei tabelei (scris cu litera mica). Pentru a modifica acest comportament folosim anotatia @Column cu elementul name. Ex:

@Column(name=”cname”)

private String clientName;

* Avem mai multe strategii de generare a cheii primare. Cel mai facil mod pentru a face aceasta este apelarea autogenerarii de catre container, folosind anotatia cu valoarea predefinita @GeneratedValue. Ex:

@Id

@GeneratedValue

private int id;

* Excluderea unui camp sau proprietate de la a fi facuta persistenta de catre container prin folosirea anotatiei @Transient

Starea persistenta a unei entitati poate fi accesata prin intermediul variabilelor instanta sau al proprietatilor. Proprietatile sau campurile clasei trebuie sa fie de urmatoarele tipuri Java:

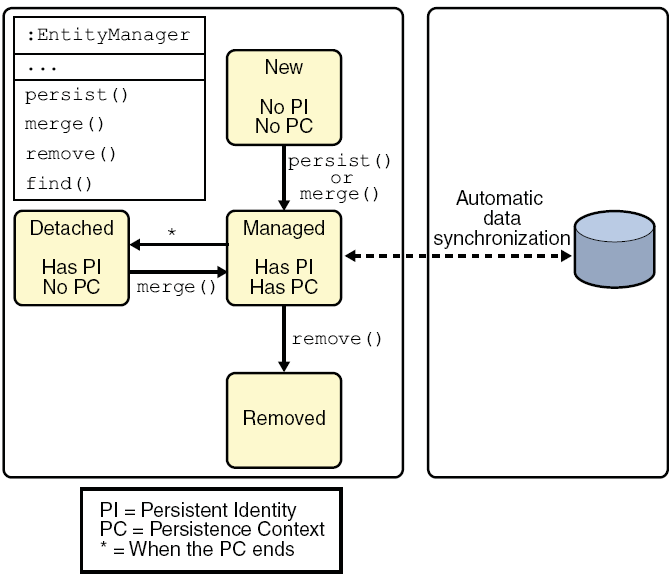
* Tipuri primitive
* String
* Alte tipuri serializabile cum ar fi:
  + Clasele wrapper ale tipurilor primitive
  + java.math.BigInteger
  + java.math.BigDecimal
  + java.util.Date
  + java.util.Calendar
  + java.sql.Date
  + java.sql.Time
  + java.sql.TimeStamp
  + tipuri utilizator serializabile
  + byte[]
  + Byte[]
  + char[]
  + Char[]
* Tipuri enumerare
* Alte entitati sau colectii de entitati
* Alte clase ce pot fi incorporate

Observatie: Putem folosi anotatia @Lob pentru a persista un camp sau o proprietate ca un obiect mare intr-o baza de date ce suporta acest tip de obiect. Acest tip poate fi caracter sau binar.

Starile din ciclul de viata al unei entitati sunt:

* New, se obtine in urma folosirii operatotului new. In acest caz nu este nicio linie din tabela in corespondenta cu cheia primara a entitatii, din nivelul persistent asociat
* Managed, exista o linie in corespondenta din tabela si datele sunt tinute sincronizate, de catre persistence provider, cu datele din entitate
* Detached, la fel ca mai sus doar ca datele din entitate nu sunt tinute sincronizate
* Removed, reprezinta o stergere in asteptare a datelor, din linia corespondenta din baza de date

Acest proces este ilustrat grafic dupa cum urmeaza:



Entity managerul gestioneaza instantele entitatilor pe parcursul ciclului de viata. Fiecare entity manager este asociat unui context persistent. Un context persistent este o multime de instante in care pentru fiecare persistent entity identity avem o singura instanta entitate. Persistent entity identity este o valoare unica folosita de furnizorul de persistenta pentru a mapa instanta entitatii cu linia corespunzatoare din tabela.

Printre alte operatii ale entity managerului avem:

* flush(), forteaza sincronizarile bazei de cate cu entitatile, in contextul persistent
* refresh(), reface instantele entitate din contextul persistent
* find(), cauta o instanta entitate prin executarea unei interogari pe cheia primara in baza de date
* contains(), returneaza true daca instanta entitatii se afla in contextul persistent. Aceasta inseamna ca instanta entitatii este “managed”

Contextul persistent poate avea urmatoarele domenii:

* domeniul transaction, care se intinde pe durata unei tranzactii
* domeniul extended, care se intinde pe durata mai multor tranzactii.

Contextul persistent nu se creaza direct de catre utilizator, ci se foloseste cel pe care containerul il aloca entity managerului cand este creat. Folosirea unui context persistent transaction duce la obtinerea unui entity manager transaction. Acelasi lucru se intampla si pentru extended. Ex:

@PersistenceContext(type=Extended)

private EntityManager entityManager;

Putem efectua urmatoarele operatii asupra unei tabele folosind instante entitate, intr-un domeniu transaction:

* crea o noua intrare. Pentru aceasta trebuie sa urmam pasii:
  + sa cream o noua instanta a clasei entitate ce corespunde tabelei
  + sa folosim metoda persist() a obiectului entity manager pentru a crea o intrare in tabel
* sterge o intrare. Pentru aceasta trebuie sa apelam metoda remove() a entity managerului, ce are ca argument o instanta managed
* cauta o intrare. Pentru aceasta utilizam metoda find() a entity managerului. Metoda returneaza o instanta managed
* modifica o intrare. Pentru aceasta trebuie sa urmam pasii:
  + trebuie sa folosim un setter pentru a modifica valoarea unui camp sau proprietati a instantei entitate
  + daca instanta entitate este in starea managed atunci acest pas nu mai este necesar, altfel va trebui sa apelam merge() din entity manager avand ca atribut instanta detached. Ex:

// item este o instanta detached; item1 este o instanta managed

Item item1 = entityManager.merge(item);

item1.setProp(newProp);

sau

item.setProp(newprop); // item este instanta detached

Item item1 = entityManager.merge(item); // item1 este managed

In continuare descriem aceleasi operatii intr-un context extended. Principalele diferente intre un entity manager cu domeniul transaction si unul cu domeniul extended sunt:

* in entity manager de domeniu transaction toate instantele managed devin detached dupa ce tranzactia se incheie
* in entity manager de domeniu extended toate instantele managed raman managed pe tot timpul de existenta al entity managerului

Pentru a desfasura clasele entitate trebuie sa cream o unitate de persistenta. O unitate de persistenta este o grupare logica a tuturor elementelor necesare containerului pentru a gestiona persistenta unei aplicatii. Printre aceste elemente avem:

* toate clasele entitate managed, ce definesc ce urmeaza a fi persistat
* maparea, ce defineste legatura intre clasele managed si tabelele bazei de date
* entity manager factory si entity managers
* informatii de configurare pentru entity manager factory si entity managers
* un fisier persistence.xml

O unitate de persistenta nu este o unitate de desfasurare de sine statatoare.

Pentru desfasurarea pe un server componentele unitatii de persistenta trebuie sa fie plasate intr-un EAR, EJB-JAR, WAR sau intr-un JAR al aplicatiei client.

Ca locatie, componetele unitatii persistente sunt localizate astfel:

* toate clasele entitate in directoare ce mapeaza structura claselor managed. In cazul nostru in directorul ejbModule
* maparea, este data de anotatii in clasele entitate. Ca alternativa putem folosi descriptori XML
* persistence.xml, in directorul META-INF

Entitatile pot folosi campuri persistente, proprietati persistente sau o combinatie a celor doua. Daca anotatiile de mapare sunt aplicate variabilelor de instanta ale entitatii, atunci entitatea utilizeaza campuri persistente. Daca anotatiile sunt aplicate getterilor, atunci entitatea foloseste proprietati persistente.

Toate campurile neanotate cu javax.persistence.Transient sau nemarcate ca tranzitorii vor fi persistente in baza de date. Anotarea de mapare trebuie sa insoteasca variabilele instanta.

Daca entitatea foloseste proprietati persistente, atunci entitatea trebuie sa urmareasca conventia componentelor JavaBean. Intr-un JavaBean proprietatile folosesc getteri si setteri ce au numele format cu numele atributelor clasei. Pentru proprietatea proprietate, vom avea metodele getProprietate() si setProprietate(), cu o singura exceptie daca proprietatea este boolean atunci getterul va fi isProprietate().

Anotatiile de mapare pentru proprietatile persistente vor fi aplicate getterilor. Acestea nu pot fi aplicate campurilor sau proprietatilor anotate @Transient sau marcate tranzitoriu.

Daca campurile sau proprietatile persistente sunt colectii, atunci acestea trebuie sa fie de unul dintre urmatoarele tipuri colectie Java:

* java.util.Collection
* java.util.Set
* java.util.List
* java.util.Map

In general, toate aceste interfete sunt generice.

Daca un camp sau proprietate a unei entitati consta dintr-o colectie va fi anotata cu javax.persistence.ElementCollection. Aceasta anotatie are doua elemente targetClass si fetch. Primul specifica numele clasei si este utilizat optional daca campul sau proprietatea este definita folosind tipurile generice. Al doilea este la randul sau optional si este folosit pentru a specifica daca colectia este returnata lazily sau eagerly. Specificarea celor doua valori se face folosind constantele javax.persistence.FetchType.LAZY sau EAGER. Predefinit este lazily.

Fie urmatorul exemplu:

@Entity

public class Persoana {

...

@ElementCollection(fetch=EAGER)

protected Set<String> nume = new HashSet();

...

}

Am definit o entitate ce are un camp persistent, nume, ce reprezinta o colectie de String-uri ce vor fi aduse eagerly. Elementul targetClass al anotatiei nu este necesar pentru ca folosim tipul generic.

Una dintre colectiile frecvent folosite este java.util.Map. Un Map consta dintr-o cheie si o valoare. Cand folosim Map trebuie sa tinem cont de urmatoarele reguli:

* cheia sau valoarea pot fi de un tip de baza Java, o clasa incorporata sau o entitate
* cand valoarea este o clasa incorporata sau un tip de baza folosim anotatia @ElementCollection
* daca valoarea este o entitate folosim anotatia @OneToMany sau @ManyToMany
* tipul Map se foloseste doar intr-una dintre partile relationarii bidirectionale

Cand tipul cheii unui Map este un tip de baza Java vom folosi anotatia javax.persistence.MapKeyColumn pentru a seta maparea cheii. Predefinit, atributul name al acestei anotatii este PROPERTY-NAME\_KEY. Spre exemplu, daca campul relationat referit este image, atributul name predefinit este IMAGE\_KEY.

Daca tipul cheii unui Map este entitate, utilizam anotatia javax.persistence.MapKeyJoinColumn. Daca avem nevoie de mai multe coloane pentru a seta maparea vom folosi anotatia javax.persistence.MapKeyJoinColumns pentru a include multiple anotatii javax.persistence.MapKeyJoinColumn. Daca anotatia javax.persistence.MapKeyJoinColumn este prezenta atunci numele coloanei mapate este in mod predefinit PROPERTY-NAME\_KEY. Spre exemplu, daca campul relationat referit este angajat, atributul name predefinit este ANGAJAT\_KEY.

Daca nu folosim tipurile generice Java in campul relationat, atunci va trebui ca clasa cheie sa fie setata explicit prin anotatia javax.persistence.MapKeyClass.

Daca cheia unui Map este cheie primara sau in cazul valorii unui camp persistent/proprietate al unui Map atunci folosim anotatia javax.persistence.MapKey. Cele doua anotatii javax.persistence.MapKey si javax.persistence.MapKeyClass nu pot fi folosite simultan.

Daca valoarea unui Map este un tip de baza Java sau o clasa incorporata, va fi mapata ca o colectie. Daca nu folosim tipul generic, elementul targetClass al anotatiei @ElementCollection trebuie sa fie setat la tipul valorii.

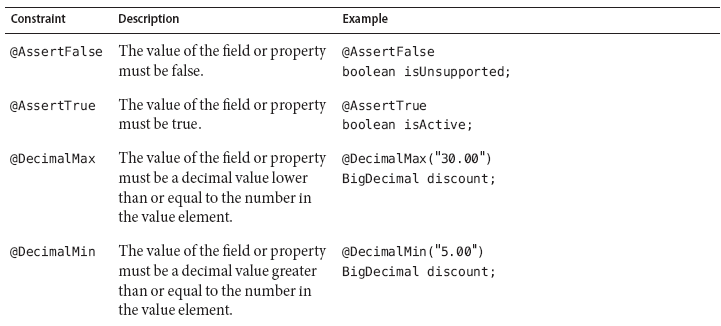
Daca valoarea lui Map este o entitate si parte a unei relationari M:N sau 1:N unidirectionale, atunci va fi mapata unui tabel jonctionat. O relationare unidirectionala 1:N ce utilizeaza Map trebuie sa fie mapata folosind anotatia @JoinColumn.

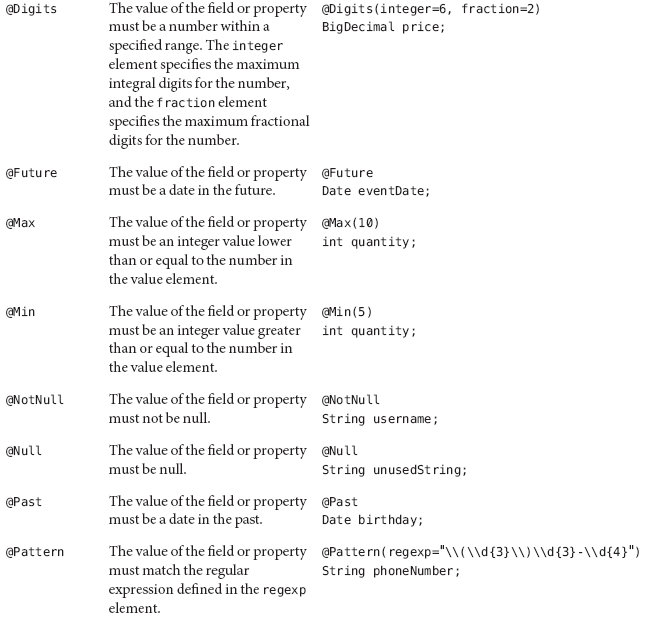
Daca entitatea este parte a unei relationari bidirectionale 1:N/M:N, ea va fi mapata in tabelul entitatii ce reprezinta valoarea Map-ului. Daca nu folosim tipuri generice atunci atributul targetEntity al anotatiilor @OneToMany si @ManyToMany va trebui setat la tipul valorii.

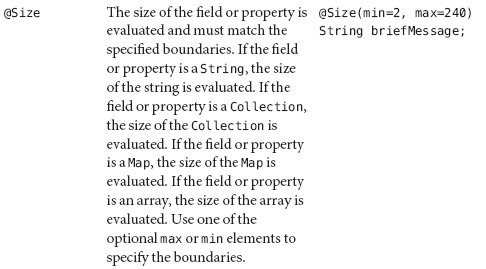
Java furnizeaza un mecanism pentru validarea datelor aplicatiei.

Bean Validation este integrat in containerele EE. Mecanismul de validare poate fi aplicat claselor entitate persistente, claselor incorporate si superclaselor mapate. Java face validari automate entitatilor cu campuri persistente sau proprietati anotate pentru validare, imediat dupa evenimentele ciclului de viata si anume PrePersistent, PreUpdate si PreRemove.

Java furnizeaza anotatii predefinite pentru validare dar da posibilitatea si utilizatorului sa-si creeze propriile avalidari. Fiecare constrangere folosita pentru validare este asociata cu cel putin o clasa validator ce valideaza valoarea campului sau proprietatuu supuse constrangerii. Constrangerile de validare se aplica prin anotatii campurilor sau getterilor proprietatilor.







Toate constrangerile predefinite, definite in tabelul anterior, au in corespondenta anotatia: NumeConstrangere.List pentru gruparea mai multor constrangeri de acelasi tip asociate aceluiasi camp sau proprietate. Spre exemplu:

@Pattern.List({

@Pattern(regexp="..."),

@Pattern(regexp="...")

})

**Cheile primare in entitati**

Fiecare entitate are un obiect unic pe post de identificator, cunoscut sub numele de cheie primara. Cheia primara poate fi simpla sau compusa. Cheile primare simple utilizeaza anotatia javax.persistence.Id pentru a marca campul cheie primara. Cheile primare compuse, adica formate din mai multe atribute, trebuie definite intr-o clasa cheie primara. Cheile primare compuse sunt notate folosind anotatiile javax.persistence.EmbeddedId si javax.persistence.IdClass.

Cheia primara sau proprietatea sau campul unei chei primare compuse trebuie sa fie de unul dintre urmatoarele tipuri Java:

* tip primitiv
* tip wrapper al tipului primitiv
* java.lang.String
* java.util.Date
* java.sql.Date
* java.math.BigDecimal
* java.math.BigInteger

Tipurile reale nu trebuie a fi folosite ca tipuri pentru cheile primare. Daca folosim chei primare generate atunci singurele tipuri folosite sunt cele intregi.

Furnizorul de persistenta este responsabil de modul de generare a cheilor:

* AUTO, se alege strategia cea mai buna pentru generarea cheii
* TABLE, se foloseste tabela bazei de date pentru generarea cheii
* SEQUENCE, IDENTITY, se foloseste o secventa sau coloana identitate

O clasa primary key trebuie sa indeplineasca urmatoarele cerinte:

* modificatorul de acces al clasei trebuie sa fie public
* proprietatile trebuie sa public
* clasa trebuie sa aiba un constructor public predefinit
* clasa trebuie sa implementeze metodele hashCode() si equals()
* clasa trebuie sa fie serializabila
* cheia primara compusa trebuie sa fie reprezentata si mapata mai multor campuri sau proprietati ale clasei entitate sau trebuie sa fie reprezentata si mapata ca o clasa incorporata
* daca clasa este mapata mai multor campuri sau proprietati, numele si tipul campurilor cheie primara sau al proprietatilor cheii primare din clasa cheie primara trebuie sa corespunda celor din clasa entitate

**Relationarea entitatilor**

Primul pas in modelarea asocierii la nivelul datelor este definirea relationarii ca o multime de proprietati. Aceste proprietati pot fi apoi utilizate pentru a implementa relationarea obiectelor prin clasele entitate.

Urmatoarele patru proprietati sunt folosite pentru a descrie asocierea intre obiecte:

* Cardinalitatea unei relatii, care specifica numarul de relationari intre doua entitati relationate, poate fi de urmatoarele tipuri: unu-unu, unu-multi, multi-unu, multi-multi.
* Directia unei relatii, care determina navigabilitatea si vizibilitatea, poate fi:
  + Bidirectionala, fiecare entitate poate vedea cealalta entitate din relatie. Putem astfel include cod in oricare dintre entitati pentru a naviga catre cealalta entitate in vederea obtinerii de informatii si servicii de la aceasta. In JPA natura bidirectionala a unei relatii este specificata prin utilizarea unei anotatii de cardinalitate in ambele clase entitate situate de cele doua parti ale relatiei
  + Unidirectionala, doar o singura entitate dintre cele doua implicate in relatie poate vedea cealalta entitate. Natura unidirectionala a relatiei este specificata prin folosirea uneia dintre anotatiile de cardinalitate intr-una dintre entitati.
* Proprietarul relatiei, specifica partea de owning a relationarii, care la randul sau contine maparea fizica. Cealalta parte a relatiei se numeste inverse side.
* Tipul de propagare a operatiei (cascading type), se refera la propagarea efectului unei operatii entitatilor asociate. Se exprima prin termenii: All, Persist, Merge, Remove, Refresh, None

Intr-o relatie bidirectionala fiecare entitate are un camp sau o proprietate ce refera la cealalta entitate din relatie. Prin acest camp codul entitatii poate accesa obiectul relationat.

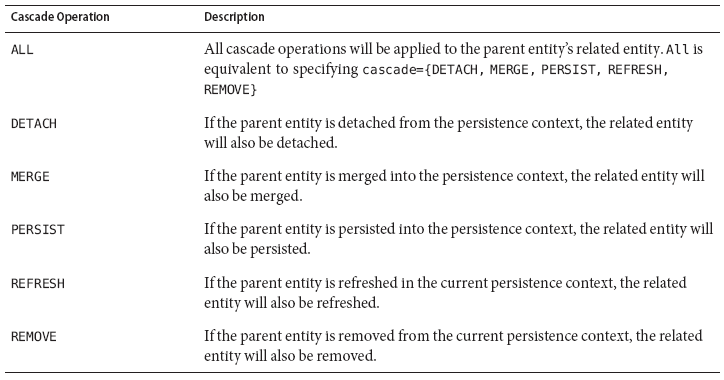
Relatiile bidirectionale trebuie sa urmareasca regulile:

* inverse side trebuie sa refere catre owning side utilizand elementul mappedBy al uneia dintre anotatiile: @OneToOne, @OneToMany sau @ManyToMany. Valoarea acestui element este campul sau proprietatea din entitatea ce joaca rol de owner al relatiei
* partea “multi” a unei relatii multi-unu, intr-o relationare bidirectionala, nu trebuie sa defineasca elementul mappedBy. Aceasta parte este considerata intotdeauna owner
* intr-o relatie unu-unu bidirectioanala owning side corespunde partii ce contine cheia straina corespunzatoare
* intr-o relatie multi-multi ambele parti pot fi owning side

Intr-o relatie unidirectionala doar o singura entitate are un camp sau o proprietate ce refera la cealalta entitate.

Directionalitatea este necesara pentru a putea naviga intre relatii.

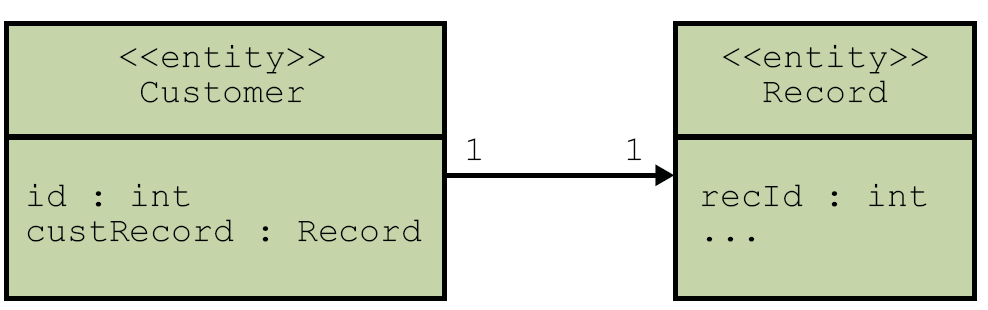
Entitatile ce utilizeaza relatiile au deseori dependente relativ la existenta celeilalte entitati din relatie. Tipul enumerare javax.persistence.CascadeType defineste operatiile in cascada, ca valoare a elementului cascade al anotatiilor de relationare. Valorile acestei enumerari sunt date in tabelul urmator:



Cand entitatea destinatie dintr-o relatie unu-unu sau unu-multi este stearsa din relatie, este deseori recomandabil sa stergem si entitatea. Asemenea entitati se numesc orfane si atributul orphanRemoval poate fi folosit pentru a specifica entitatile orfane ce ar trebui sterse. Acest atribut se foloseste in anotatiile @OneToOne si @OneToMany si are o valoare booleana, implicit false.

**Implementarea asocierii unidirectionale unu-unu**

O astfel de asociere poate fi cea din schema de mai jos:



Procesul de realizare a acestei asocieri intre doua clase entitate urmeaza pasii:

1. definirea celor doua clase entitate
2. identificarea clasei owner
3. in acea entitate cream un camp sau o proprietate pentru a reprezenta relationarea entitatii cu entitatea tinta (target)
4. anotarea acestuia prin @OneToOne
5. pentru ca este o asociere unidirectionala nu este necesara o alta modificare in entitatea tinta

Portiunea de cod implicata in aceasta relationare este data mai jos:

@Entity

public class Customer {

@Id

private int id;

@OneToOne

private Record custRecord;

..//

}

Respectiv, clasa target:

@Entity

public class Record {

@Id

private int recId;

..//

}

Strategia ilustrata anterior este cea generica. Ea poate fi modificata astfel:

* putem suprascrie numele predefinit al coloanei de jonctiune prin adaugarea anotatiei @JoinColumn anotatiei @OneToOne
* daca cele doua tabele impart aceleasi valori ale cheii primare putem folosi anotatia @PrimaryKeyJoinColumn in loc de @JoinColumn
* daca cele doua tabele sunt jonctionate utilizand multiple coloane vom folosi anotatia @JoinColumns in loc de @JoinColumn

**Implementarea asocierii bidirectionale unu-unu**

Pe baza exemplului anterior com crea o proprietate si in target prin care putem vedea entitatea proprietar. Pentru aceasta pina la pasul 4, inclusiv, procedam ca mai inainte. Vom modifica pasul 5 si adauga pasul 6:

1. in entitatea inversa cream un camp sau o proprietate ce reprezinta relationarea cu entitatea owning
2. anotam aceasta proprietate prin @OneToOne si setam valoarea atributul mappedBy la valoarea campului sau proprietatii din entitatea owning

rezultatul este descris de codul urmator:

@Entity

public class Record {

@Id

private int recId;

@OneToOne(mappedBy="custRecord")

private Customer customer;

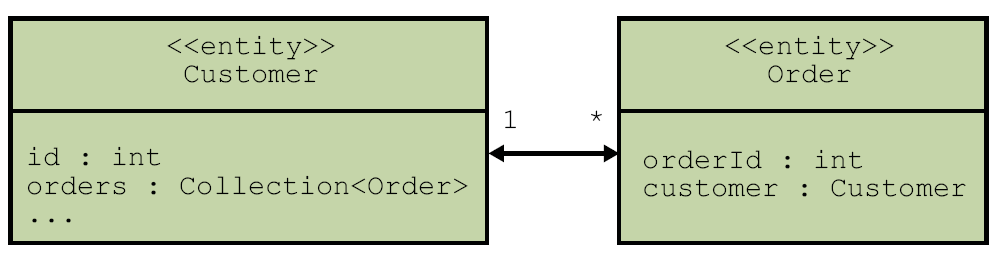
..//

}

Si in acest caz comportamentul predefinit poate fi modificat prin folosirea anotatiilor @JoinColumns, @JoinColumn sau @PrimaryKeyJoinColumn.

**Implementarea asocierii bidirectionale unu-la-multi/multi-la-unu**

Putem exemplifica aceasta asociere prin:



Observatie: in acest tip de asociere entitatea owning este intotdeauna entitatea din partea multi a relatiei.

Urmatorii pasi descriu procesul de definire a acestui tip de asocieriere intre doua clase entitate:

1. definirea celor doua clase entitate
2. identificarea clasei owner (cea de pe partea “multi”)
3. in acea entitate cream un camp sau o proprietate pentru a reprezenta relationarea acestei entitati cu o entitatea tinta
4. anotarea acestuia prin @ManyToOne
5. in cealalata entitate a asocierii cream o proprietate sau un camp ce reprezinta relationarea
6. anotarea acestuia prin @OneToMany
7. Setam valoarea atributului mappedBy la valoarea proprietatii sau campului din entitatea owning

Un exemplu de codare este dat in cele ce urmeaza:

@Entity

public class Customer {

@Id

private int id;

@OneToMany(mappedBy=”customer”)

private Collection <Order> orders;

..//

}

@Entity

public class Order {

@Id

private int orderId;

@ManyToOne

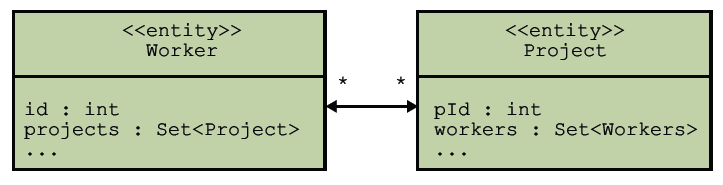
private Customer customer;

..//

}

**Implementarea asocierii bidirectionale multi-la-multi**

Putem exemplifica aceasta asociere prin:



Urmatorii pasi descriu procesul de definire a acestui tip de asocieriere intre doua clase entitate:

1. definirea celor doua clase entitate
2. identificarea clasei owner
3. in acea entitate cream un camp sau o proprietate pentru a reprezenta relationarea acestei entitati cu o entitatea tinta
4. anotarea acestuia prin @ManyToMany
5. in cealalata entitate a asocierii cream o proprietate sau un camp ce reprezinta relationarea
6. anotarea acestuia prin @ManyToMany
7. Setam valoarea atributului mappedBy la valoarea proprietatii sau campului din entitatea owning

Un exemplu de codare este dat in cele ce urmeaza:

@Entity

public class Worker {

@Id

private int id;

@ManyToMany

private Set <Project> projects;

..//

}

@Entity

public class Project {

@Id

private int pId

@ManyToMany(mappedBy=”projects”)

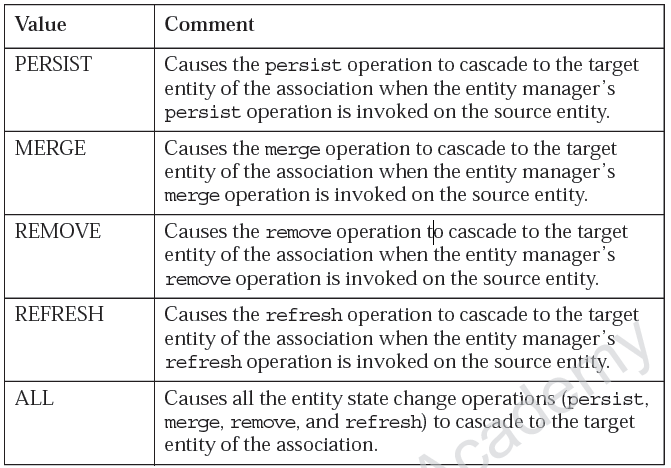
private Set <Worker> workers;

..//

}

Anotatiile pentru maparea asocierii au anumite atribute optionale. Printre acestea:

* fetch, care seteaza instructiuni catre furnizorul de persistenta referitoare la momentul incarcarii in memorie a elementelor de asociere. Valorile posibile sunt
  + EAGER, indica furnizorului de persistenta sa incarce tinta asocierii in momentul in care entitatea este incarcata din baza de date
  + LAZY, indica furnizorului de persistenta faptul ca incarcarea tintei asocierii sa se faca in momentul accesului
* cascade, specifica momentul din ciclul de viata al unei entitati in care au loc modificari asupra relationarilor. In mod predefinit nu apar modificari. Alte valori posibile sunt indicate in urmatorul tabel



**Clase incorporate in entitati**

Clasele incorporate sunt utilizate pentru a reprezenta starea unei entitati, dar nu sunt persistente. Instantele claselor incorporate folosesc identitatea entitatii careia ii apartin. Intr-o entitate putem avea clase incorporate singulare sau colectii ale acestora.

Clasele incorporate folosesc aceleasi reguli ca si clasele entitate, dar sunt anotate cu javax.persistence.Embeddable in loc de @Entity.

Entitatile ce folosesc clasele incorporate ca parte a starii de persistenta anoteaza, optional, campurile sau proprietatile incorporate cu javax.persistence.Embedded.

La randul lor clasele incorporate pot folosi alte clase incorporate pentru a-si defini starea. Clasele incorporate pot contine relationari catre alte entitati sau colectii de entitati.

Exemplu:

@Embeddable

public class DataNasterii {

int zi;

String luna;

int an;

}

Si clasa ce o contine:

@Entity

public class Persoana {

@Id

protected long id

String nume;

@Embedded

DataNasterii dataNasterii;

...

}

**Mostenirea entitatilor**

Entitatile suporta mostenirea, asocierea polimorfica si interogarile polimorfice. Clasele entitate pot extinde clase non-entiate, iar clasele non-entitate pot extinde clasele entitate. Clasele entitate pot fi abstracte.

O clasa abstracta poate fi declarata entitate prin anotarea cu @Entity. Entitatile abstracte sunt ca entitatile obisnuite, dar nu pot fi instantiate.

Entitatile abstracte pot fi interogate ca niste entitati obisnuite. Daca intr-o interogare apare o clasa abstracta, atunci interogarea opereaza pe toate subclasele ce extind entitatea abstracta.

Entitatile pot mosteni superclase ce contin o stare persistenta si informatii mapate, dar nu sunt entitati. In acest caz superclasele nu sunt decorate cu anotatia @Entity si nu sunt mapate, ca o entitate, de catre Java persistence provider. Aceste superclase se folosesc de obicei atunci cand avem stari sau informatii mapate in comun de catre mai multe entitati. Ele se numesc superclase mapate (mapped superclasses) si sunt anotate cu javax.persistence.MappedSuperclass.

Superclasele mapate nu pot fi interogate si nu pot fi folosite in operatiile din EntityQuery sau Query. Nu pot fi, de asemenea, relationate, dar pot fi abstracte. Ele nu au niciun tabel in corespondenta in baza de date.

Pe linga entitati si superclase mapate, enitatile pot avea si clase non-entitate pe post de superclase. Si acestea pot fi abstracte. Starea unei superclase non-entitate nu este persistenta si orice stare mostenita, dintr-o superclasa non-entitate, de catre o entitate nu este persistenta. Aceste superclase nu pot fi utilizate in operatii ale EntityQuery sau Query. De asemenea, nu pot fi relationate.

Putem configura modul in care JPA mapeaza entitatile mostenite cu baza de date prin decorarea clasei radacina a ierarhiei cu javax.persistence.Inheritance. avem posibilitatea sa alegem dintre:

* O singura tabela pe intreaga ierarhie de clase
* O tabela pe fiecare entitate concreta
* Strategia „join”, in care campurile sau proprietatile unei subclase sunt mapate unei tabele diferite dacat cea a clasei parinte

Strategiile pot fi configurate prin setarea valoarii elementului strategy din anotatia @Inheritance cu o valoare din enumerarea:

public enum InheritanceType {

SINGLE\_TABLE,

JOINED,

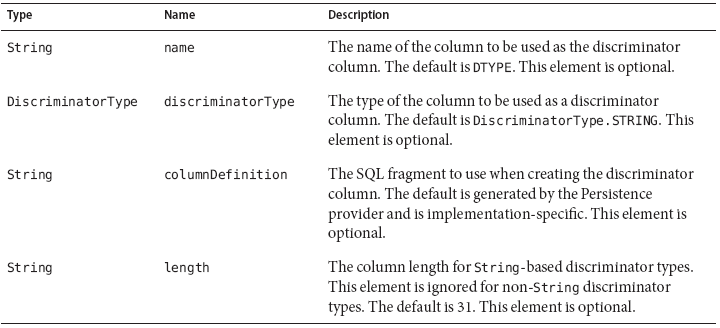
TABLE\_PER\_CLASS

};

Valoarea implicita este SINGLE\_TABLE.

Discutam pe rand fiecare dintre strategii:

* Un singur tabel, adica toate clasele din ierarhie sunt mapate unui singur tabel din baza de date. Acest tabel are o coloana discriminator ce contine o valoare ce identifica subclasa caruia ii corespunde instanta liniei curente. Coloana discriminator poate fi specificata prin utilizarea anotatiei javax.persistence.DiscriminatorColumn in radacina ierarhiei de clase entitate. Elementele acestei anotatii sunt ilustrate in tabelul urmator



Tipul enumerare javax.persistence.DiscriminatorType este folosit pentru a seta tipul coloanei discriminator in baza de date. Valorea elementului discriminatorType, a anotatiei @DiscriminatorColumn poate fi, asadar:

public enum DiscriminatorType {

STRING,

CHAR,

INTEGER

};

Daca anotatia @DiscriminatorColumn nu este explicit specificata in entitatea radacina atunci valoarea predefinita este String, iar numele predefinit al coloanei este DTYPE.

Anotatia javax.persistence.DiscriminatorValue poate fi folosita pentru a seta valoarea introdusa in coloana discriminator, pentru fiecare entitate din ierarhia de clase. Doar clasele din ierarhie pot fi decorate cu aceasta anotatie. Daca anotatia nu este specificata intr-o entitate, JPA va furniza o valoare, dependenta de implementare, predefinita. Daca valoarea elementului discriminatorType al anotatiei @DiscriminatorColumn este String, atunci valoarea din tabela este numele entitatii. Aceasta strategie suporta polimorfismul atat la nivelul relatiilor cat si al interogarilor.

* Un tabel pentru fiecare clasa, adica fiecare clasa este mapata unei tabele separate din baza de date. Toate campurile sau proprietatile din clasa, inclusiv campurile sau proprietatile mostenite sunt mapate ca coloane in tabelul clasei, in baza de date. In general, aceasta strategie nu suporta polimorfismul la nivelul relationarilor iar interogarile polimorfice au la baza clauza UNION. In mod predefinit GlassFish-ul nu suporta aceasta strategie
* Subclasa jonctionata, adica clasa radacina este mapata unui tabel din baza de date iar fiecare subclasa are mapat un tabel separat, care contine doar acele campuri specifice subclasei. Asadar, tabelul subclasei nu contine coloane pentru campurile sau proprietatile mostenite. In plus, tabelul subclasei are o coloana sau mai multe coloane ce reprezinta cheia primara, ce este cheie straina pentru cheia primare a tabelei superclasa. Aceasta strategie furnizeaza suport pentru relatii polimorfice, dar sunt necesare una sau mai multe jonctiuni atunci cand instantiem subclasele entitate. Acelasi situatie se intalneste si in cazul interogarilor. Unii dintre furnizorii de JPA, incluzand serverul GlassFish, impun o coloana discriminator ce corespunde entitatii radacina. Daca nu folosim crearea automata a tabelei trebuie sa ne asiguram ca tabela este setata corect pentru coloana discriminator predefinita sau folosim anotatia @DiscriminatorColumn pentru a defini o astfel de coloana.

**Gestiunea entitatilor**

Entitatile sunt gestionate printr-un entity manager reprezentat de o instanta a interfetei javax.persistence.EntityManager. Fiecare entity manager este asociat unui context persistent, adica instante ale unora dintre entitatilor ce exista mapate dintr-o baza de date. Contextul persistent defineste domeniul sub care instantele entitatilor sunt create, salvate sau sterse. EntityManager defineste metodele utilizate pentru a interactiona cu contextul persistent.

In cazul unui entity manager gestionat de container, o instanta a EntityManager este automat propagata de catre container catre toate componentele aplicatiei ce utilizeaza instanta in cadrul unei tranzactii JTA.

Tranzactiile JTA implica deseori apeluri peste mai multe componente. Pentru a executa o tranzactie JTA aceste componente au nevoie de acces catre un singur context persistent. Aceasta apare atunci cand EntityManager este injectat in componentele aplicatiei, prin anotatia javax.persistence.PersistenceContext. Acest context persistent este propagat automat impreuna cu tanzactia curenta JTA, iar referintele lui EntityManager sunt mapate aceluiasi context persistent ca si tranzactia. Prin propagarea automata a contextului persistent componenetele aplicatiei nu au nevoie sa-si trimita referintele catre instantele EntityManager in cadrul unei tranzactii. Containerul Java EE gestioneaza ciclul de viata entity managerilor gestionati de container.

Obtinerea unei instante se face prin constructia sintactica:

@PersistenceContext

EntityManager em;

In cazul unui entity manager gestionat de aplicatie, contextul persistent nu este propagat catre componentele aplicatiei, iar ciclul de viata al instantelor EntityManager este gestionat de aplicatie. Acest gen de management al entitatilor este utilizat atunci cand aplicatiile au nevoie sa acceseze un context ce nu este propagat cu tranzactia JTA de-a lungul instantelor EntityManager. In acest caz EntityManager creaza un nou context persistent, izolat. O alta utilizare a acestui tip de gestiune este atunci cand nu putem injecta direct instante ale EntityManager pentru ca acestea nu sunt thread-safe. In schimb EntityManagerFactory este.

Aplicatiile creaza instante ale EntityManager folosind metoda createEntityManager() din javax.persistence.EntityManagerFactory. Obtinerea unei instante a EntityManagerFactory se face prin injectarea sa in componenta aplicatiei utilizand anotatia javax.persistence.PersistenceUnit.

@PersistenceUnit

EntityManagerFactory emf;

EntityManager em = emf.createEntityManager();

Gestiunea entitatilor de catre aplicatie nu propaga automat contextul tranzactiei. Aplicatiile trebuie sa ceara accesul de la managerul de tranzactie si sa adauge informatiile de demarcatie ale tranzactiei atunci cand executa operatiile entitatii. Interfata javax.transaction.UserTransaction defineste metodele pentru inceputul, commit-ul sau roll back-ul unei tranzactii. Injectam o instanta a interfetei UserTransaction prin crearea unei variabile de instanta anotate cu @Resource.

@Resource

UserTransaction utx;

Pentru a incepe o tranzactie apelam metoda begin() din UserTransaction. Cand toate operatiile pe entitati sunt incheiate apelam commit() din aceeasi interfata. Metoda rollback(), din aceeasi interfata, ne permite revenirea la starea de dinaintea inceperii tranzactiei.

Iata un cadru clasic de folosire a gestiunii de catre aplicatie a entity managerului:

@PersistenceContext

EntityManagerFactory emf;

EntityManager em;

@Resource

UserTransaction utx;

...

em = emf.createEntityManager();

try {

utx.begin();

em.persist(Entitatea1);

em.merge(Entitatea2);

em.remove(Entitatea3);

utx.commit();

} catch (Exception e) {

utx.rollback();

}

**Operatii uzuale folosind EntityManager-ul**

Gasirea entitatilor se face folosind metoda find() din EntityManager ce are doua atribute, entitatea in care se face cautarea si valoarea cautata.

O instanta a unei entitati se poate afla in una dintre urmatoarele stari:

* New, in care instantele entitatii nu au inca o identitate persistenta si nu sunt asociate inca unui context persistent
* Managed, au o identitate persistenta si sunt asociate unui context persistent
* Detached, au o identitate persistenta dar nu au fost inca asociate unui context persistent
* Removed, au o identitate persistenta si sunt asociate unui context persistent, dar sunt programate pentru stergere din baza de date

Noile instante entitate devin managed si deci persistente fie prin invocarea metodei persist() sau printr-o operatie in cascada persist, invocata de entitatile relationate ale entitatii curente, care au elementul cascade setat la valoarea PERSIST sau ALL in anotatia relationarii. Aceasta inseamna ca datele entitatii sunt stocate in baza de date cand tranzactia asociata cu operatia persist este incheiata. Daca entitatea este deja managed atunci operatia persist este ignorata. Daca operatia persist este apelata unei instante removed, entitatea devine managed. Daca entitatea este detached atunci operatia va arunca un IllegalArgumentException si tranzactia va esua. Operatia de persistenta va fi propagata tuturor entitatilor relationate daca acestea au elementul cascade setat corespunzator.

Stergerea entitatilor se face prin apelul metodei remove() sau printr-o operatie de stergere in cascada, invocata din entitatile relationate ce au elementul cascade setat la valoarea REMOVE sau ALL. Daca metoda remove() este invocata pe o entiate aflata in starea new, operatia este ignorata. Daca este invocata pe o entitate avand starea detached se va arunca un IllegalArgumentException si tranzactia va esua. Daca va fi invocata pe o entitate removed, operatia va fi ignorata.

Starea entitatilor persistente este sincronizata cu baza de date atunci cand tranzactia in care se afla aceste entitati este comisa. Daca gestionam o relatie bidirectionala atunci datele vor fi persistente bazandu-ne pe partea owning a relatiei. Pentru a forta sincronizarea entitatilor cu baza de date invocam metoda flush() in instanta EntityManager.

**Unitati de persistenta**

O unitate de persistenta (persistence unit) defineste multimea tuturor claselor entitate gestionate de instantele EntityManager intr-o aplicatie. Unitatile de persistenta sunt definite de fisierul de configurare persistence.xml. Un exemplu de astfel de fisier este cel creat in aplicatia din cursul anterior:

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

<persistence version=*"1.0"* 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\_1\_0.xsd"*>

<persistence-unit name=*"PrimulEJB"*>

<jta-data-source>jdbc/account</jta-data-source>

<class>entitati.Customer</class>

<class>entitati.SavingsAccount</class>

</persistence-unit>

</persistence>

Fisierul descrie o unitate persistenta numita PrimulEJB ce foloseste o sursa de date JTA jdbc/account. Elementele class specifica clasele persitente si anume: clase entitate, clase incorporate si superclase mapate.

Unitatile persistente pot fi impachetate intr-un fisier WAR sau EJB JAR, sau pot constitui un JAR ce este inclus intr-un WAR sau EAR.

**Folosirea JPA intr-o aplicatie Java SE**

1. Cream baza de date pe serverul de baze de date
2. Adaugam bibliotecile EclipseLink si conectorul la BD
3. Definim entitatile in aplicatie
4. Cream persistence.xml si-l configuram: definim o unitate de persistenta si un tip de tranzactie, furnizam nume claselor entitate, definim proprietatile conexiunii JDBC
5. Cream instante ale EntityManagerFactory si EntityManager
6. Scriem cod pentru a efectua persistenta entitatilor folosind instanta entity manager