

Java Persistence API

Imre Gábor

Q.B224

gabor@aut.bme.hu



Automatizálási és
Alkalmazott
Informatikai Tanszék

Tartalom

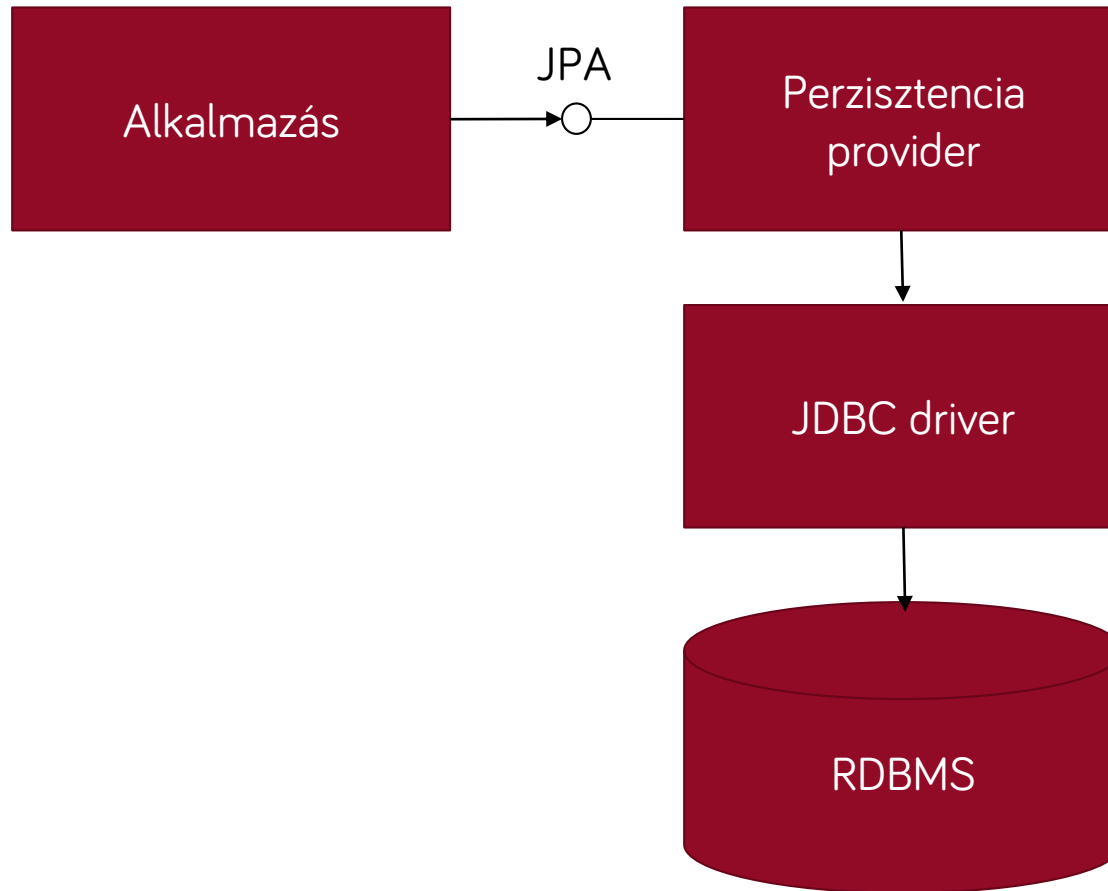
- Általános jellemzők
- O-R leképezés annotációkkal
- A perzisztenciakontextus
 - > Entitások életrajza
 - > Adatbázis szinkronizáció
 - > Lekérdezések
- Criteria API
- Natív lekérdezések
- Öröklés
- Entitások közti kapcsolatok

Általános jellemzők

Általános jellemzők

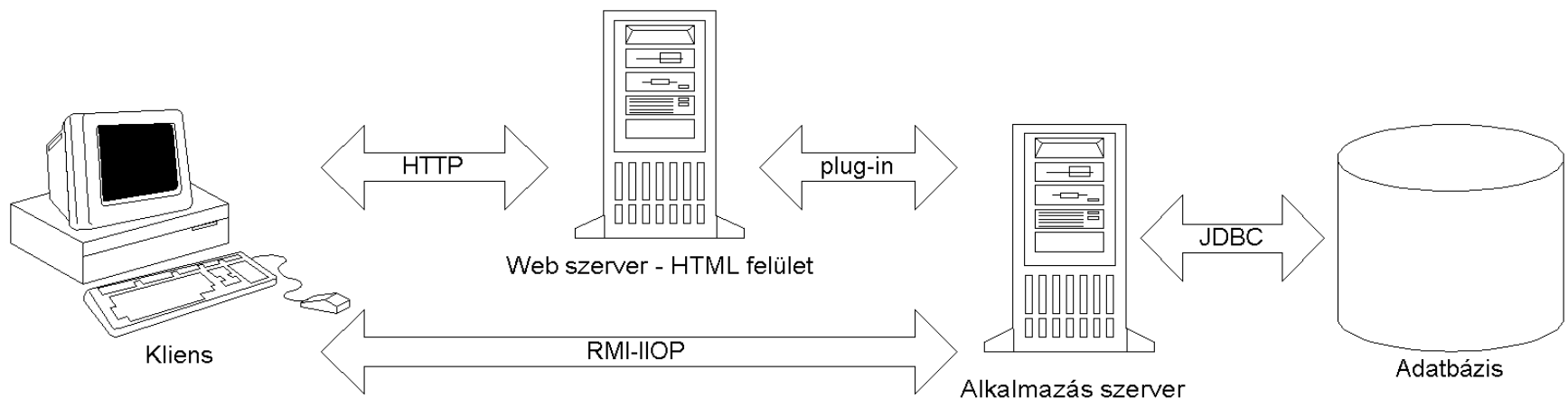
- Szabványos ORM API a Java világban
- Utolsó verzió Java Persistence API néven: **JPA 2.2**
- Java EE → Jakarta EE átnevezés után Jakarta Persistence API, ott **JPA 3.1** a legfrissebb
- Csak interfészeket specifikál → több lehetséges implementáció (perzisztencia provider – P.P.)
 - > Pl. *Hibernate, EclipseLink, OpenJPA*
- JPA entitás: olyan osztály, melynek példányait relációs adatbázisban perzisztensen tárolja a JPA
- A **javax.persistence**, majd **jakarta.persistence** csomag tartalmazza

JPA architektúra



Java Enterprise Edition

- A JPA Java SE-ben is használható, de ott a P.P. osztálykönyvtárait nekünk kell hozzáadni
- A Java EE 5 óta minden alkalmazáserver tartalmaz JPA implementációt
 - > Java EE: Java Enterprise Edition, vállalati méretű alkalmazások server oldali fejlesztését támogató Java kiadás
 - > Háromrétegű architektúra
 - > További Java EE technológiák (EJB, JTA) megkönnyítik a JPA használatát
- Java EE 8 után Oracle → Eclipse átadás, név innenől Jakarta EE



O-R leképezés annotációkkal

O-R leképezés annotációkkal

- JPA entitás kötelező tulajdonságai:
 - > No-arg konstruktor
 - > **@Entity** annotáció
 - > Elsődleges kulcs attribútum: **@Id** annotációval megjelölve
 - Az elsődleges kulcs értéke akár generáltatható: lásd **@GeneratedValue**
 - Az elsődleges kulcs lehet összetett: lásd **@IdClass** vagy **@EmbeddedId**
- A perzisztens attribútumok getterek/setterek formájában érhetők el
- A P.P. elérheti közvetlenül az adatmezőket, ha azokat annotáljuk és nem a gettereket (mező vs. property alapú elérés)
 - > JPA 2.0 óta egy osztályon belül is változtatható: pl. **@Access(FIELD)** az osztályra, **@Access(PROPERTY)** bizonyos getterekre

Entitás példa

@Entity

```
public class Employee {  
    @Id  
    private Integer id;  
    private String name;  
    private Date birthDate;  
  
    // ... getterek, setterek  
}
```

Az O-R leképezés testreszabása

- Az O-R leképezés részleteinek megadása (táblanév, oszlopnevek) opcionális: by default az osztály és attribútumnévvel egyezik meg
- Osztályra: **@Table(name="MyTable")**
- Attribútumokra vagy getterekre:
@Column(name="MyColumn")
 - > Ezeknek egyéb paramétereik is vannak, pl. schema, catalog, nullable, length, ...
- Az annotációk helyett xml fájl is használható, de ritkán használatos

Entitás attribútumok típusai

- Primitív típusok és wrappereik
- String, char[], Character[]
- BigInteger, BigDecimal
- java.util.Date, java.util.Calendar, java.sql.Date, java.sql.Time, java.sql.Timestamp, Java 8 java.time típusai
 - > DB-ben dátum, idő vagy mindkettő? → **@Temporal(DATE/TIME/TIMESTAMP)**
- byte[], Byte[]
 - > **@Lob**
- Enum
 - > DB-ben szám vagy szöveg? → **@Enumerated(ORDINAL/STRING)**
- Más entitás, más entitások vagy nem-entitások gyűjteménye
 - > Lásd *Entitások közti kapcsolatok*
- Beágyazott osztályok
- Ha egy entitás attribútumot nem akarunk DB-ben tárolni → **@Transient**

Beágyazott osztály

- Olyan osztály, ami önmagában nem él perzisztens entitásként, csak egy perzisztens entitás példányhoz kapcsolódva, pl.

@Embeddable

```
public class EmploymentPeriod {  
    Date startDate;  
    Date endDate;  
    // ... getterek, setterek  
}
```

- Lehetséges felhasználása egy entitáson belül:

@Entity public class Employee {

```
//...
```

@Embedded

@AttributeOverrides({

```
    @AttributeOverride(name="startDate", column=@Column("EMP_START")),
```

```
    @AttributeOverride(name="endDate", column=@Column("EMP_END"))
```

```
})
```

```
private EmploymentPeriod empPeriod;
```

```
}
```

- JPA 2.0 óta egymásba ágyazhatók

Konverterek

- JPA 2.1 óta használhatók
- DB oszlop értéke és az attribútum között konvertálnak
- Típus alapján automatikusan, vagy egyes attribútumokhoz kötve

@Converter(autoApply=false) //false a default

public class WeightConverter implements

AttributeConverter<Double, Double> {

public Double convertToDatabaseColumn(Double pounds) {

return pounds / 2.2046;

}

public Double convertToEntityAttribute(Double kilograms) {

return kilograms * 2.2046;

}

}

Konverter bekapcsolása

@Entity

```
public class Part {
```

```
    @Id Integer partId;
```

```
    String name;
```

```
    ...
```

```
    @Convert(converter=WeightConverter.class)
```

```
    Double shippingWeight;
```

```
    ...
```

```
}
```

Persistence unit

- A JPA ún. persistence unit-okat (P.U.) kezel
- P.U.: entitások olyan halmaza, melyeket ugyanabban a DB-ben tárolunk
- A persistence unit-o(ka)t a persistence.xml fájlban kell definiálni, pl.

```
<persistence>
```

```
  <persistence-unit name="MyPU">
```

```
    <jta-data-source>jdbc/MyDB</jta-data-source>
```

```
    <class>com.xy.Employee</class>
```

```
    <class>com.xy.Company</class>
```

```
  </persistence-unit>
```

```
</persistence>
```

A P. U. neve. Több is lehet belőle, ha pl. több DB-t használ az alkalmazás

Az adatbázis JNDI neve

Az entitás osztályok. Java EE környezetben nem kötelező felsorolni őket

- Helye: META-INF könyvtár az entitások class fájljait tömörítő jar fájlban

JNDI

- Java Naming and Directory Interface
- Egységes Java API, amelyen keresztül elvileg tetszőleges névfeloldási (és directory) szolgáltatás elérhető
- A névszolgáltatás lehetőséget nyújt, hogy valamilyen néven valamilyen objektumot regisztráljunk, később pedig név alapján megkeressük
- J2SE 1.3-tól, *javax.naming* package alatt
- Java EE alkalmazásszerverekben mindig van névszolgáltatás, tipikus használata:
 - > Komponensek név alapján érhetik el egymást
 - > Külső erőforrásokat (pl. adatbázis, üzenetsor, SMTP szerver) név alapján érhetünk el

Adatbázis elérése Java EE környezetben

- A **DriverManager.getConnection()** hátrányai:
 - > A kapcsolat minden részletét ismerni kell a fejlesztőnek (DB gyártó, szerver, port, DB név)
 - > A megszerzett kapcsolatot nem tudjuk hatékonyan újrahasználni
- Helyette: **DataSource** interfész
 - > A fejlesztő JNDI névre hivatkozva keresi meg
 - > Az üzemeltető feladata a szerverben az adott JNDI névhez beregisztrálni a DB elérés részleteit
 - > Connection poolingot valósíthat meg, vagyis a tőle szerzett kapcsolat bezárása nem zárja be fizikailag a kapcsolatot

Tipikus JDBC kód Java EE környezetben

@Stateless

```
public class MyBean {
```

```
    @Resource(lookup="mydb")
```

```
    DataSource ds;
```

```
    public void myMethod() {
```

```
        try(Connection conn = ds.getConnection()) {
```

```
            ...
```

```
        } catch(Exception e){...}
```

```
    }
```

```
}
```

- A **@Resource** hatására az EJB-konténer végzi el a JNDI keresést (függőséginjektálás), ha nekünk kellene:

```
DataSource ds = (DataSource) new InitialContext().lookup("mydb");
```

- Az alkalmazáserver konfigurációs fájljában: driver osztály, user/pass és JDBC URL megadva a mydb névhez

A perzisztenciakontextus

Perzisztenciakontextus

- A perzisztencia provider által kezelt, memóriában lévő entitások egy halmaza (a továbbiakban P.C.)
- Ezen keresztül kezeljük az entitásokat, ez a kapcsolat a memóriabeli entitások és az adatbázis között
- API szinten az EntityManager interfészen érhető el, pl.

```
EntityManagerFactory emf =  
Persistence.createEntityManagerFactory("MyPU");  
EntityManager em = emf.createEntityManager();  
em.getTransaction().begin();  
em.persist(new Employee(12345, "Gabor"));  
em.getTransaction().commit();  
em.close();  
emf.close();
```

Perzisztenciakontextus

- Egy EntityManager referencián keresztül egy perzisztenciakontextust érünk el, egy P.C-on belül minden entitás egy példányban fordulhat elő
- Kihívást jelent az EntityManager élelciklusának kezelése:
 - > Mikor nyissuk meg?
 - > Ha megnyitottuk, mikor zárjuk be?
 - Ha áthívunk másik metódusba, praktikus lenne abban ugyanazt az EntityManager referenciát látni, ezért fölösleges még becsukni → legyen tagváltozó
 - De ha tagváltozó, és másik osztály metódusába hívunk át, jó lenne ott is ugyanazt látni → legyen metódus bemenő paraméter?
- Megoldás: függőséginjektálás Java EE (EJB) vagy Spring segítségével

Menedzselt perzisztenciakontextus

@Stateless

```
class PersonService {  
    @PersistenceContext  
    EntityManager em;  
  
    public void createEmployee{  
        em.persist(new Employee(12345, "Gabor"));  
    }  
}
```

- Működés feltétele: a PersonService-t ne közvetlenül new-val példányosítsuk, hanem valamilyen konténertől kérjünk példányt, ami a konstruktor híváson kívül elvégzi az em tagváltozó inicializálását is
- Az EJB (@Stateless) és a Spring (@Service) is képes az ilyen, ún. menedzselt perzisztenciakontextus kezelésére

Menedzselte perzisztenciakontextus

- Az EntityManagerFactory
 - > csak az alkalmazás indulásakor jön létre, egy példányban
- Az EntityManager
 - > a tranzakció elején jön létre
 - > tranzakció végén záródik be
 - > ha ugyanabban a tranzakcióban más osztályban is van injektált EntityManager → ugyanazt a perzisztenciakontextust fogják látni

EntityManager

- Ezen az interfészen keresztül kezeljük az entitásokat
- 3 típusú metódus:
 - > entitások életciklusának kezelése
 - > adatbázis szinkronizáció
 - > entitások keresése

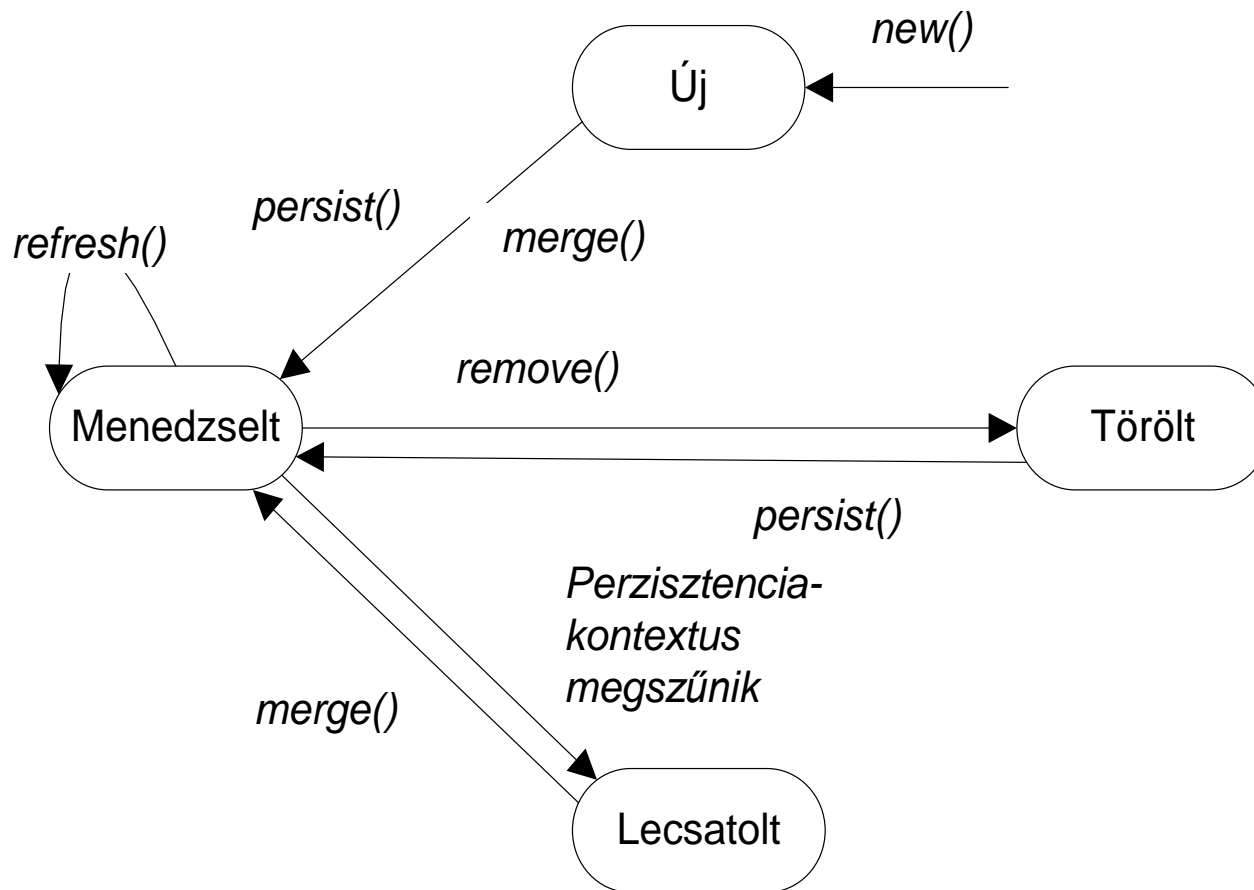
A perzisztenciakontextus

Entitások élelciklusa

Entitások állapotai

- new: new-val létrehozva kerül ide, csak a memóriában létezik, a módosítások nem mennek adatbázisba
- managed: létezik az adatbázisban, és hozzátartozik egy perzisztenciakontextushoz. A P.C.-n hívott flush() metódussal (ami automatikusan meghívódik tranzakció commitkor) beíródnak a módosítások az adatbázisba
- detached: adatbázisban megvan, de nem tartozik perzisztenciakontextushoz
- removed: még perzisztenciakontextushoz tartozik, de már ki van jelölve, hogy törölve lesz az adatbázisból tranzakció végén

Entitások életrajza



Entitások élelciklusa

- Új entitás menedzseltté tétele:
 - > **persist()**: ha elsődleges kulcs ütközés van, kivétel
 - > **merge()**: ha elsődleges kulcs ütközés van, SQL UPDATE, ha nincs, INSERT
- A **merge()** *visszatérési értéke* a menedzselte entitás példány!
- Entitás lecsatolódása:
 - > A P.C. kiürítésével: **em.clear()**;
 - > A P.C. bezárásával: **em.close()**;
 - > Az entitás sorosításakor (pl. ha távoli metódushíváson megy át)
 - > JPA 2.0 óta akár egyedileg: **em.detach(entity)**;

A perzisztenciakontextus

Adatbázis szinkronizáció

Adatbázis szinkronizáció

- A P.P. az **EntityManager** 2 metódusa segítségével szinkronizál az adatbázis felé/felől:
 - > **flush()**: beírja DB-be a teljes PC összes módosítását
 - ritkán hívjuk explicit módon, mert a tranzakció commit automatikusan meghívja
 - > **refresh(entity)**: beolvassa DB-ből a változtatásokat egy entitásra
- **EntityManager.setFlushMode()**:
 - > **AUTO**-ra állítva (ez a default) minden lekérdezés előtt flush történik, hogy a query a legfrissebb állapotban hajtsódjon végre
 - > **COMMIT**-ra állítva a tranzakció végén van csak flush
 - jobb teljesítmény érhető el
 - a tranzakción belüli módosítások nem befolyásolják a lekérdezés eredményét → zavaró lehet

A perzisztenciakontextus

Lekérdezések

Lekérdezések

- Többféle, mindegyik az EntityManager-en keresztül
- Keresés elsődleges kulcs alapján:
 - > `<T> T find(Class<T> entityClass, Object primaryKey)`
- Lekérdezés teljesen dinamikusan:
 - > JPQL nyelven: `public Query createQuery(String jpqlString)`
 - SQL-hez hasonló nyelv, de entitás példányokkal tér vissza
 - PL. `"SELECT e from Employee e WHERE e.name = :name"`
 - > natív SQL: `public Query createNativeQuery(String sqlString)`
- Statikusan definiált, névvel azonosítható lekérdezés:
 - > `public Query createNamedQuery(String nameOfQuery)`
 - > a lekérdezés `@NamedQueries`-ben van definiálva az entitás osztálynál

```
@NamedQueries({  
    @NamedQuery(name="Employee.findAll",  
        query="SELECT e FROM Employee e")  
})
```

```
@Entity
```

```
public class Employee { ...}
```


Lekérdezések

- **Query** fontosabb metódusai:
 - > **setParameter**: név vagy index alapján
 - > Lapozás támogatására: **setMaxResult**, **setFirstResult**
 - > Tényleges lefuttatás:
 - **getSingleResult** (kivétel, ha nem pontosan egy találat)
 - **getResultList** (üres lista, ha nincs találat)
 - **executeUpdate** (UPDATE, DELETE esetén)
- Néhány fontosabb JPQL lehetőség
 - > többes törlés, módosítás (nem kell megtalálni a törlendőket, és egyesével törölni/módosítani őket)
 - > JOIN, GROUP BY, HAVING, subquery
 - > paraméterek ?1, ?2 vagy :paraméterNév formában
 - > projekció (csak bizonyos attribútumokat adjon vissza, Object[]-ek formájában)
 - > új objektum létrehozás selectben: a visszaadott oszlopokat egyből valamilyen általunk definiált objektumként kapjuk vissza

JPQL példák

- **SELECT o FROM Order o WHERE o.shippingAddress.state = 'CA'**
 - > azok az Order entitások, amelyeket Californiába kell kiszállítani
- **SELECT DISTINCT o FROM Order o JOIN o.lineItems l WHERE l.shipped = FALSE**
 - > azok az Order entitások, amelyek tartalmazznak olyan tételeket, amelyek még nincsenek kiszállítva

Criteria API

Criteria API

- A JPA 2.0 vezette be, a deklaratív, string alapú JPQL objektumorientált, típusbiztos alternatívája lekérdezések előállításához, pl.:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
```

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

```
Root<Employee> emp = cq.from(Employee.class);
```

```
cq.select(emp);
```

```
cq.where(cb.equal(emp.get("lastName"), "Smith"));
```

```
TypedQuery<Employee> query = em.createQuery(cq);
```

```
List <Employee> rows = query.getResultList();
```

Criteria API

- Látható, hogy több kód, mint a JPQL esetén, de cserébe típusbiztos (ez a verzió még nem teljesen)
- Ez a megoldás még tartalmaz string alapú navigációt ("lastName"), de ez is kiküszöbölhető az ún. Metamodel API segítségével
- A perzisztencia egység metamodelje metaadatokat tartalmaz az entitásokról és beágyazott osztályokról
- Általában az annotációk feldolgozásával tudja generálni a perzisztencia provider segédeszköze, de kézzel is megírható, pl.

Criteria API

@Entity

```
public class Employee {  
    @Id Long id;  
    String firstName;  
    String lastName;  
    Department dept;  
}
```

@StaticMetamodel(Employee.class)

```
public class Employee_ {  
    public static volatile SingularAttribute<Employee, Long> id;  
    public static volatile SingularAttribute<Employee, String> firstName;  
    public static volatile SingularAttribute<Employee, String> lastName;  
    public static volatile SingularAttribute<Employee, Department> dept;  
}
```

Criteria API

- Az ún. kanonikus metamodel, a hordozhatóság érdekében célszerű ilyet generálni/írni. Jellemzői:
 - > osztálynév után _
 - > public static volatile attribútumok, **SingularAttribute** vagy **Collection/List/Map/SetAttribute** típusúak, megfelelő generikus típussal
 - > az attribútumok nevei azonosak

Criteria API

- Metamodel segítségével teljesen típusbiztos lehet a lekérdezés, pl.

```
CriteriaQuery<Employee> cq =  
    cb.createQuery(Employee.class);  
  
Root<Employee> emp = cq.from(Employee.class);  
cq.select(emp);  
cq.where(cb.equal(  
    emp.get(Employee_.lastName), "Smith"));  
  
TypedQuery<Employee> query = em.createQuery(cq);  
List<Employee> rows = query.getResultList();
```


Criteria API

- Egy összetettebb példa:

```
CriteriaQuery<Vendor> q = cb.createQuery(Vendor.class);
```

```
Root<Employee> emp = q.from(Employee.class);
```

```
Join<ContactInfo, Phone> phone =
```

```
    emp.join(Employee_.contactInfo).join(ContactInfo_.phones);
```

```
q.where(cb.equal(emp.get(Employee_.contactInfo)
```

```
    .get(ContactInfo_.address)
```

```
    .get(Address_.zipcode),
```

```
    "95054"))
```

```
.select(phone.get(Phone_.vendor));
```

- Az ekvivalens JPQL:

```
SELECT p.vendor
```

```
FROM Employee e JOIN e.contactInfo.phones p
```

```
WHERE e.contactInfo.address.zipcode = '95054'
```

Natív lekérdezések

Natív lekérdezések

- Bizonyos esetekben nem elég a JPQL/Criteria API, pl.
 - > UNION
 - > DB specifikus lehetőségek
- Létrehozásuk:
 - > **em.createNativeQuery(nativeSqlString, entityClass)**
 - > **em.createNamedQuery(queryName, entityClass)**
 - **@NamedNativeQuery(**
 - name="complexQuery",**
 - query="SELECT USER.* FROM USER_ AS USER WHERE ID = ?",**
 - resultClass=User.class)**
 - > Criteria API nincs hozzá!
- Létrehozás után ugyanúgy Query objektumot kapunk, mint JPQL vagy Criteria queryknél

Natív lekérdezések eredményének leképezése Java objektumokra

- Egy natív lekérdezésen a **getResultList()** by default **Object[]**-ök listáját adja vissza, ahol a tömb elemei a szelektált oszlopokban lévő értékek
- Triviális eset: entitást szeretnénk visszakapni: lásd az előző dia példáját
- Egyéb esetekre: entitás osztályokon elhelyezhető **@SqlResultSetMapping** annotáció, pl.

```
@SqlResultSetMapping(name="JediResult", classes = {
```

```
    @ConstructorResult(
```

```
        targetClass = Jedi.class,
```

```
        columns = {@ColumnResult(name="name"),
```

```
                @ColumnResult(name="age")})
```

```
    })
```

```
    ...
```

```
Query query = em.createNativeQuery("SELECT name,age FROM jedis_table", "JediResult");
```

```
List<Jedi> samples = query.getResultList();
```

- Fontos: az annotációt egy entitásra tesszük, de a query eredménye nem feltétlen entitás!
- Természetesen a Jedi osztályban szükséges a Jedi(String name, int age) konstruktor

Öröklés

Öröklés leképezése JPA-ban

- Az öröklési hierarchia kialakítása
 - > **extends** kulcsszó
 - > **@Entity** minden osztályra
 - > **@Id** attribútum csak a legfelső ősből
- A legfelső őosztályon annotációval adjuk meg a választott O-R leképezési módot:
 - > **@Inheritance(strategy = SINGLE_TABLE)** → Egy közös táblába
 - > **@Inheritance(strategy = JOINED)** → Összes osztály leképezése táblába
 - > **@Inheritance(strategy = TABLE_PER_CLASS)** → Valós osztályok leképezése táblába
 - Ezt nem kötelező támogatni a P.P.-nek, mert nehézkes a polimorfizmus megvalósítása

Diszkriminátor oszlop

- Csak **TABLE_PER_CLASS** esetben határozza meg a tábla a Java típust
- **SINGLE_TABLE** és **JOINED** esetben plusz egy oszlop szükséges a típus tárolásához → diszkriminátor oszlop
- Az oszlop neve by default **DTYPE**
 - > Testreszabás a legfelső űsben:
@DiscriminatorColumn(name="mycolumn")
- A beleírt érték by default az entitás neve
 - > Testreszabás az egyes entitásokon:
@DiscriminatorValue("mytype")

Egyéb öröklési lehetőségek

- Entitás származhat nem-entitásból
 - > Kérdés: a leszármazott entitásokban perzisztensek legyenek-e a nem-entitás ősből örökölt attribútumok?
 - By default nem
 - Ha **@MappedSuperClass** annotációt teszünk a nem-entitás ősre, akkor viszont igen. De a nem-entitás ősnek így sem lesz külön tábla, nem szerepelhet lekérdezésben, nem kezelhető EntityManagerrel!
- Nem entitás származhat entitásból
 - > Nem lesz neki tábla, nem szereplhet lekérdezésben, nem végezhető vele művelet az EntityManager-en keresztül
- Entitás lehet absztrakt
 - > nem példányosodhat, de le lehet képezni táblába, lehet rá lekérdezést írni

Entitások közti kapcsolatok

Kapcsolatok leképezése

- A kapcsolatot reprezentáló tagváltozó típusa:
 - > Entitás
 - > **Collection, Set, List** vagy **Map**
- A kapcsolatot annotálni kell kardinalitás alapján:
 - > **@OneToOne** (DB-ben idegen kulcs)
 - > **@OneToMany** (DB-ben idegen kulcs)
 - > **@ManyToOne** (DB-ben idegen kulcs)
 - > **@ManyToMany** (DB-ben kapcsoló tábla)
- További lehetséges annotációk:
 - > **@JoinColumn** (mellőzése esetén van default)
 - > **@JoinTable** (több-többes esetben, de itt is van default, ha nem tesszük ki)
 - > **@OrderBy** (List kapcsolat esetén)
 - > **@MapKey** (Map kapcsolat esetén)

Kapcsolatok leképezése

- Irány szerint:
 - > Egyirányú
 - > Kétirányú
 - A két irányt a fejlesztő tartja konzisztensen!!!!
 - A két irány összerendelése szükséges, pl.
- Tulajdonos oldalon (Employee):
 - @ManyToOne**
 - @JoinColumn(name="company_id")**
 - private Company company;**
- Másik oldalon (Company):
 - @OneToMany(mappedBy="company")**
 - private Collection<Employee> employees;**
- A kapcsolatnak mindig egy tulajdonos oldala van, amelyikre a másik oldal mappedBy paraméterrel hivatkozik

Collection típusú mezők nem-entitás elemekkel

- JPA 2.0-tól egy entitás Collection attribútumában az elemek már lehetnek nem-entitások is: alap típusok, vagy beágyazott osztályok
- Külön táblába mennek a collection elemei, idegen kulcs oszlop hivatkozik a szülő entitásra
- De a collection táblája nem önálló entitás, nincs pl. elsődleges kulcsa sem
- **@ElementCollection** és **@CollectionTable**-vel szabható testre

@ElementCollection példa

```
public enum FeatureType { AC, CRUISE, PWR, BLUETOOTH, TV, ... }
```

@Embeddable

```
public class ServiceVisit {  
    @Temporal(DATE)  
    Date serviceDate;  
    String workDesc;  
    int cost;  
}
```

@Entity

```
public class Vehicle {  
    ...
```

@ElementCollection példa

@Entity

```
public class Vehicle {
```

```
    @Id int vin;
```

```
    @ElementCollection
```

```
    @CollectionTable(name="VEH_OPTNS")
```

```
    Set<FeatureType> optionalFeatures;
```

```
    @ElementCollection
```

```
    @CollectionTable(name="VEH_SVC")
```

```
    @OrderBy("serviceDate")
```

```
    List<ServiceVisit> serviceHistory;
```

```
    ...
```

```
}
```

Entitások közti kapcsolatok

Kapcsolatok finomhangolása

Cascade

- Mind a 4 kapcsolatdefiniáló annotációhoz megadható egy cascade elem, pl.

```
@OneToMany(cascade={
```

```
    CascadeType.PERSIST, CascadeType.MERGE
```

```
});
```

- Lehetséges értékek: PERSIST, MERGE, REMOVE, REFRESH, ALL
- Azt adja meg, milyen EntityManager műveletek hívódjanak meg a kapcsolódó entitásokra is
- Default: nincs cascade

Fetch

- Mind a 4 kapcsolatdefiniáló annotációhoz megadható egy fetch elem, pl. **@OneToMany(fetch=FetchType.LAZY)**
- Azt adja meg, hogy egy entitás betöltésekor betöltődjenek-e a kapcsolódó entitások is
- LAZY (lusta): nem töltődnek be, csak ha hivatkozunk rájuk → nem foglal memóriát, csak ha szükség van rá, de +1 lekérdezés (a P.P. figyelmen kívül hagyhatja)
- EAGER (mohó, ez a default, kivéve OneToMany és ManyToMany esetén): betöltődnek → gyorsabb, de több memóriát foglal (kötelező utasítás a P.P.-nek)
- Finomhangolási lehetőség:
 - > legyen LAZY, de azokban a lekérdezésekben, ahol tudjuk, hogy szükség lesz a kapcsolódókra, használjunk fetch join-t az JPQL-ben, pl.
 - > **SELECT c FROM Customer c LEFT JOIN FETCH c.orders**
- Oszlopokra is definiálható fetch, **@Basic** paramétereként (tipikusan **@Lob** esetén)

Fetch-hez kapcsolódó problémák

- Ha lusta betöltés miatt még nincsenek betöltve egy entitás kapcsolódó entitásai, és ilyenkor lecsatolódik → a lecsatolt állapotban nem lesznek elérhetőek ezek a kapcsolódó objektumok
- Megoldás: eager fetch, vagy lecsatolódás előtt a szükséges kapcsolatokra explicit getter hívás
- Ráadásul a lecsatolt, kapcsolatoktól megfosztott, kapcsolattulajdonos példány merge-elésekor a perzisztencia provider az adatbázisban is törölni fogja a kapcsolatot
- Megoldás: a lecsatolt példány merge-ölése helyett id alapján keresés, a módosítások átmásolása kézzel

Entitás gráfok

- Segítségükkel a lekérdezések fetch viselkedése definiálható JPA 2.1 óta
- **@NamedEntityGraph** vagy programozottan

```
@NamedEntityGraph(name="previewEmailEntityGraph",  
    attributeNodes={  
        @NamedAttributeNode("subject"),  
        @NamedAttributeNode("sender")  
    }  
)
```

@Entity

```
public class EmailMessage {  
    @Id String messageld;  
    String subject;  
    String body;  
    String sender;  
}
```

Entitás gráfok

- Referencia szerzése elnevezett gráfra:

```
EntityGraph<EmailMessage> eg =  
    em.getEntityGraph("previewEmailEntityGraph");
```

- Entitás gráf létrehozása programozottan:

```
EntityGraph<EmailMessage> eg =  
    em.createEntityGraph(EmailMessage.class);  
eg.addAttributeNodes("subject");  
eg.addAttributeNodes("sender");  
eg.addAttributeNodes("body");
```

Entitás gráfok

- Entitás gráf felhasználása a lekérdezésben:

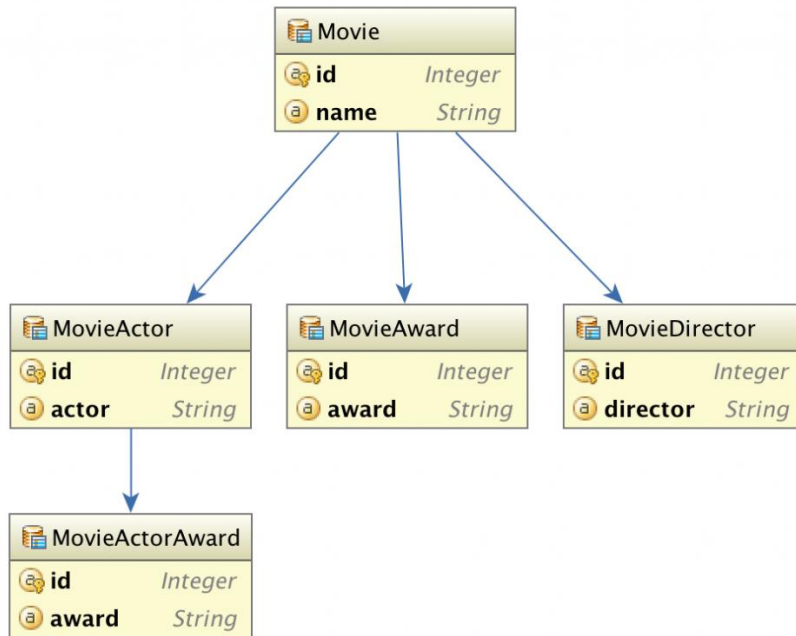
```
Properties props = new Properties();  
props.put("javax.persistence.loadgraph", eg);
```

```
EmailMessage message =
```

```
em.find(EmailMessage.class, id, props);
```

- Query hintek entitás gráf felhasználására
 - > **javax.persistence.fetchgraph**: csak a megadott mezők lesznek benne
 - > **javax.persistence.loadgraph**: a megadott mezőkön kívül a default gráf mezői is benne lesznek (az entitás EAGER betöltésű mezői)

Entitás gráfok



```
@NamedEntityGraph(  
    name = "movieWithActorsAndAwards",  
    attributeNodes = {  
        @NamedAttributeNode(  
            value = "movieActors",  
            subgraph = "movieActorsGraph")  
        },  
    subgraphs = {  
        @NamedSubgraph(  
            name = "movieActorsGraph",  
            attributeNodes = {  
                @NamedAttributeNode("movieActorAwards")  
            }  
        )  
    }  
)
```