Java Persistence API

JPA, entitások közötti kapcsolatok, JPQL

Simon Károly
simon.karoly@codespring.ro

JPA

- Objektumok állapotának tárolása adatbázis rendszerekben, a JDBC feletti (arra épülő) absztrakciós szint
- Specifikáció: Java Persistence API (JPA) (az EJB 3.0-tól különálló)
- JPA: standard eljárás definiálása a POJOk adatbázisba történő leképezésének
- Entitiy Beans – POJOk, amelyek a JPA meta-adatok (annotációs mechanizmus) segítségével le lesznek képezve egy adatbázisba. Az adatok mentése, betöltése, módosítása megtörténhet anélkül, hogy a fejlesztőnek ezzel kapcsolatos kódot kelljen írnia (pl. JDBC hozzáféréssel kapcsolatos kód nem szükséges)
- A JPA meghatároz egy lekérdező nyelvet is (a funkcionalitások azonosak az SQL nyelvek által biztosított funkcionalitásokkal, de Java objektumokkal dolgozhatunk, az objektumorientált szemléletmódnak megfelelően)
- Az új JPA specifikáció meghatároz egy teljes ORM leképezést, a komponensek hordozhatóak (a mechanizmus már nem függ gyártótól, vagy alkalmazásszerver típustól), sőt, hagyományos Java alkalmazásokon belül is használhatóak.

EntityManager

- Perzisztenciával kapcsolatos műveletek: központi szolgáltatás az EntityManager
- Az entitások egyszerű POJO-k és mindaddig ennek megfelelően viselkednek, ameddig az EntityManager-hez intézett explicit kéréseken keresztül nem kérjük állapotuk mentését, vagy más perzisztenciával kapcsolatos műveletet.
- Menedzselt (managed/attached) entitások: miután egy entitást hozzárendelünk (kapcsolunk) egy EntityManager-hez, a szolgáltatás szinkronizálja az állapotot az adatbázisba mentett állapottal.
- Nem menedzselt (unmanaged/detached) entitások: a "szétkapcsolás" után az EntityManager már nem követi az objektum állapotának változásait, nem menti az adatokat (az objektum egyszerű POJO-ként viselkedik, pl. lehetőség van a szerializálására és elküldésére a hálózaton keresztül). A későbbiekben (amikor újra menedzselt állapotba kerül) az EntityManager elvégzi a szükséges szinkronizációt (merging).
- Menedzselt entitások egy halmaza határoz meg egy perzisztencia környezetet (Persistence Context).

Persistence Context

- A perzisztencia környezetbe tartozó entity bean-eket az EntityManager kezeli, megvalósítja a perzisztenciával kapcsolatos műveleteket.
- Miután a környezet lezáródik, minden objektum "lekapcsolódik", a továbbiakban az EntityManager nem menedzseli ezeket, az állapotváltozások nem lesznek szinkronizálva az adatbázissal.
- Tranzakció-hatókörű kontextusok (transaction-scoped persistence context): a tranzakció ideje alatt érvényesek, ezután le lesznek zárva. Csak a @PersistenceContext annotációval befűzött (injection) EntityManager-ek által menedzselt (tehát a szerver által menedzselt) kontextusok lehetnek tranzakció-hatókörűek.
- Kiterjesztett kontextusok (extended persistence context): a tranzakció lezárása után is érvényesek, az objektumok továbbra is menedzseltek maradnak. Csak állapottal rendelkező SB-ekben használható (nyilván).
- Miután a kontextus megszűnik az entitások "lekapcsolódnak", és egyszerű POJO-ként használhatóak (pl. szerializálhatóak, hálózaton küldhetőek) → nincs szükség a Data Transfer Objects (DTO) minta alkalmazására (publikus metódus, amely az entitás tartalmát egy másik objektumba másolja).

Persistence Context - példák

- Transaction-scoped persistence context:

```
@PersistenceContext(unitName="titan") EntityManager entityManager;

@TransactionAttribute(REQUIRED)
public Customer someMethod() {
      Customer cust = entityManager.find(Customer.class, 1);
      cust.setName("new name");
      return cust;
}
```

- Extended peristence context:

```
Customer cust = null;
transaction.begin();
cust = extendedEntityManager.find(Customer.class, 1);
transaction.commit();
transaction.begin();
cust.setName("new name");
extendedEntityManager.flush();
transaction.commit();
```

Persistence Unit

- Adott adatbázisnak megfeleltetett (mapping) osztályok halmaza.
- Telepítésleíró: persistence.xml – egy vagy több perzisztencia egység meghatározása
- Elhelyezés: META-INF könyvtár, egy JAR (Java SE), EJB-JAR, WAR (/WEB-INF/lib), EAR (gyökér, vagy lib) állományon belül. A JAR állományon belül találhatóak a Persistence Unit osztályai (a megfelelő csomagokon belül).
- Opcionálisan megadható egy orm.xml telepítés-leíró állomány (szintén a META-INF könyvtáron belül), amelyben megadhatóak az osztály-adatbázis tábla megfeleltetések (ha nem az annotációs mechanizmust alkalmazzuk, vagy felül szeretnénk írni azt). Ez az állomány további mapping-et leíró xml állományokra hivatkozhat <mapping-file> elemek segítségével.
- Persistence Unit osztályainak meghatározása: explicit meghatározás az xml-en belül, vagy/és automatikus keresés a csomagokban. A persistence provider biztosítja ezt a szolgáltatást. Megkeresi a csomagban az annotációval ellátott osztályokat, illetve az orm.xml-ben meghatározott osztályokat, és csatolja a Unit-hoz a persistence.xml-ben explicit módon megadott osztályokat.

Persistence Unit – deployment descriptor

```
<persistence>
        <persistence-unit name = "titan">
                <jta-data-source>java:/OracleDS</jta-data-source>
                <class>com.titan.domain.Cabin</class>
                <properties>
                        <property-name = "org.hibernate.hbm2ddl">update</property>
                </properties>
        </persistence-unit>
</persistence>
```

- <persistence-unit> attribútumok: name (kötelező), transaction-type (opcionális). Alapértelmezett tranzakció-típus: JTA (Java EE esetében) (Java SE esetében RESOURCE_LOCAL – a javax.persistence.EntityTransaction API használata).
- belső elemek (opcionálisak):
  - <jta-data-source>, <non-jta-data-source> - az adatforrás azonosítója, általában a globális JNDI azonosító. Ha nincs meghatározva, a gyártó-specifikus alapértelmezett érték lesz használva.
  - <class> - explicit módon megadhatjuk a Persistence Unit osztályait. Az automatikus keresés (scanning) során beazonosított osztályokból álló Unit ki lesz egészítve ezekkel az osztályokkal. Az <exclude-unlisted-classes/> elem alkalmazásával az automatikus keresés kikapcsolható (ebben az esetben csak a meghatározott osztályokból fog állni a Unit).
  - <provider> - a javax.persistence.PersistenceProvider interfész egy gyártó-specifikus implementációjának meghatározása (az osztály teljes neve). Szintén használható a gyártó-specifikus alapértelmezett érték (és általában ezmegfelel).
  - <properties> - a persistence provider beállításai (provider-specifikus attribútumok).

EntityManager példány

- Lehetőségek: EntityManagerFactory alkalmazása, vagy injection (javasolt)
- Factory - Java SE:

```
EntityManagerFactory factory = Persistence.createEntityManagerFactory("titan");
…
factory.close();
```

- Factory - Java EE:

```
@PersistenceUnit(unitName="titan")
private EntityManagerFactory factory;
```

  A bezárás automatikus, a konténer feladata (ha manuálisan próbáljuk, kivételt kapunk)
- Példány létrehozása (factory alkalmazásával):

```
factory.createEntityManager();
```

- Ha JTA-t alkalmazunk meg kell hívnunk az **`EntityManager.joinTransaction()`** metódust (csak ha factory-t alkalmazunk)
- Példány létrehozása – injection (a javasolt eljárás):

```
@PersistenceContext(unitName="titan")
private EntityManager entityManager;
```

- Alapértelmezetten transaction-scoped, átállítható a type attribútummal (**`type="PersistentContextType.EXTENDED`**) (természetesen csak állapottal rendelkező session bean-ek esetében.
- A menedzsment (bezárás stb.) a konténer feladata.

EntityManager használata

```
package javax.persistence;
public interface EntityManager {
      public void persist(Object entity);
      public <T> T find(Class<T> entityClass, Object primary key);
      public <T> T getReference(Class<T> entityClass, Object primary key);
      public <T> T merge(T entity);
      public void remove(Object entity);
      public void lock(Object entity, LockModeType lockMode);

      public void refresh(Object entity);
      public boolean contains(Object entity);
      public void clear();

      public void joinTransaction();
      public void flush();
      public FlushModeType getFlushMode();
      public void setFlushModeType(FlushModeType flush);

      public Query createQuery(String queryString);
      public Query createNamedQuery(String name);
      public Query createNativeQuery(String sqlString);
      public Query createNativeQuery(String sqlString, String resultSetMapping);
      public Query createNativeQuery(String sqlString, Class resultClass);

      public Object getDelegate();
      public void close();
      public boolean isOpen();
}
```

EntityManager használata

- Beszúrás (insert):
  - ```
    Customer cust = new Customer();|
    cust.setName("Bill");
    …
    entityManager.persist(cust);
    ```
  - A konkrét insert művelet ezután a beállításoknak megfelelően lesz végrehajtva. Pl. ha tranzakción belül vagyunk, akkor lehet, hogy azonnal, lehet, hogy a tranzakció végén a beállított flushMode-nak megfelelően. Azonnali végrehajtást kérhetünk a tranzakción belül a flush() metódus meghívásával. Tranzakción kívül csak akkor hívhatjuk meg a persist metódust, ha EXTENDED kontextust használunk (egyébként TransactionRequiredException típusú kivételt kapunk).
  - Ha az entitás más objektumokkal is kapcsolatban áll, akkor a beállított CASCADE policy-nak megfelelően lehetséges, hogy ezeket is beszúrjuk az adatbázisba.
  - Lehetőség van automatikus kulcs (primary key) generálására
  - A metódus IllegalArgumentException-t dobhat, ha a paraméter típusa nem megfelelő.
    ```

EntityManager használata

- Keresés (find és getReference):
  - `Customer cust = entityManager.find(Customer.class, 2);`
  - ```
    Customer cust = null;
    try {
      cust = entityManager.getReference(Customer.class, 2);
    } catch (EntityNotFoundException ex) {
      …
    }
    ```
  - A find null értéket ad, ha nem találja az entity-t, a getReference kivételt dob
  - A második paraméter a kulcs, típusa Object (az auto-boxing mechanizmus jóvoltából szerepelhet a példában int)
  - A find lazy-loading-policy-t alkalmaz az állapot inicializálásakor
  - A metódusok tranzakción kívül is meghívhatóak, de az a visszatérített elem lekapcsolásával (detaching) jár (ha nem kiterjesztett a kontextus)
- Query-k (createQuery, createNamedQuery, createNativeQuery)
  - JPQL vagy natív query-k alkalmazása
  - ```
    Query query = entityManager.createQuery("from Customer c where id=2");
    Customer cust = (Customer) query.getSingleResult();
    ```

EntityManager használata

- Módosítás (update):
  - `Cabin cabin = entityManager.find(Cabin.class, id);`
    `cabin.setBedCount(noBeds);`
  - Az eljárás alkalmazható, ameddig az entitás menedzselt állapotban van
  - A konkrét módosítás a beállított flushType-nak megfelelően (vagy a flush meghívására azonnal) történik.
- Egybeolvasztás (merge)
  - Egy nem menedzselt (levállasztott/detached) entitás (pl. kilépés a tranzakció hatóköréből) állapotának módosítása után lehetőség van az egybeolvasztásra és visszacsatolásra: ezután a változások az adatbázisban is érvényesek lesznek. Az adatbázisban az adatok frissítése a beállított flushType-nak megfelelően fog megtörténni. A példa esetében a cabin a leválasztott (és módosított) objektum:
  - `Cabin copy = entityManager.merge(cabin);`
  - Ha az entityManager már menedzsel egy azonos ID-vel rendelkező Cabin entitást, akkor ennek állapotát megváltoztatja, és a merge egy erre mutató referenciát térít vissza.
  - Ha az entityManager nem menedzsel ilyen azonosítóval rendelkező entitást, akkor egy másolat fog készülni, és egy erre mutató referenciát térít vissza a merge. A másolat menedzselt lesz, de a cabin példány továbbra sem lesz az (megmarad lecsatoltnak).

EntityManager használata

- Törlés (remove)
  - `entityManager.remove(cabin);`
  - A törlés után a cabin entitás nem menedzselt (leválasztás)
  - A törlés a beállított flushModeType-nak megfelelően történik, ha más entitások is kapcsolatban vannak a törölt entitással, azok törlése a beállított CASCADING szabályoknak megfelelően történik.
  - Csak a hatókörön belül hívható (tranzakció/extended)
- Frissítés (refresh)
  - `entityManager.refresh(cabin);`
  - Az objektum állapotának szinkronizálása az adatbázisban tárolt adatokkal (az állapot frissítése).
  - Csak a hatókörön belül hívható, ha más entitásokkal is kapcsolatban áll, akkor azok frissítése a beállított CASCADING szabályoknak megfelelően történik
- contains() és clear()
  - A contains ellenőrzi, hogy a paraméterként kapott objektum jelenleg menedzselt állapotban van-e
  - A clear leválaszt minden entitást (a változtatások elvesztődhetnek, érdemes lehet flush-t hívni előtte)

EntityManager használata

- flush() és flushModeType
  - Az persist, merge és remove metódusok által végrehajtott változtatások nem lesznek azonnal érvényesek, csak amikor a rendszer flush-t hív.
  - Ez alapértelmezetten a kapcsolódó query végrehajtásakor történik, vagy a tranzakció végrehajtásakor (commit)
  - flushModeType: AUTO (alapértelmezett) vagy COMMIT (több query összevonható, így nincs annyi adatbázis művelet, és nincs sokáig foglalva az adatbázis kapcsolat)
  - Bármikor kérhetjük az azonnali végrehajtást a flush() metódus meghívásával
- Locking
  - Olvasási és írási műveletek zárolása (bővebben a tranzakciókezelésnél)
- getDelegate()
  - A persistence provider objektumra mutató referencia (az EntityManager interfészt implementáló osztály példánya) → a konkrét típusba alakítható, így meghívhatóak implementáció-függő metódusok.
- Resource Local Transaction
  - Ha nem JTA-t használunk (pl. Java SE alkalmazások esetében) az EntityManager getTransaction metódusának segítségével kérhetünk egy EntityTransaction objektumra mutató referenciát (a JPA biztosítja ezt a lehetőséget)
  - ```
    EntityTransaction transaction = manager.getTransaction();
    transaction.begin();
    manager.persist(cabin);
    transaction.commit();
    ```
    ```

Mapping - alapok

```
package com.titan.domain

import javax.persistence.*

@Entity
public class Customer implements java.io.Serializable {
      private long id;
      private String firstName;
      private String lastName;

      @Id
      public long getId() { return id; }
      public void setId(long id) { this.id = id; }

      public String getFirstName() { return firstName; }
      public void setFirstName(String firstName) {
            this.firstName = firstName;
      }

      public String getLastName() { return lastName; }
      public void setLastName(String lastName){
            this.lastName = lastName;
      }

}
```

Mapping - alapok

- Csak az @Entity és @Id annotációk használata kötelező. Ha nem használunk mást, a Persistence Provider feltételezi, hogy az adatbázis táblázat neve megegyezik az osztály nevével, az attribútumok nevei az oszlopok neveivel.
- Az Id annotáció az attribútumnál vagy a getter metódusnál alkalmazható. Ha a getter metódusnál használjuk, akkor a provider feltételezi, hogy a getter és setter metódussal rendelkező attribútumok részei a perzisztens állapotnak. Ugyanígy: ha attribútumnál alkalmazzuk, akkor feltételezi, hogy minden attribútum része az állapotnak. Ennek megfelelően a további annotációkat ugyanott kell alkalmazni, mint az Id-t
- XML mapping-et is alkalmazhatunk (a META-INF könyvtár orm.xml állományán, illetve az ebből <mapping-file> elemekkel hivatkozott más állományokban)
- ```
  <entity-mapping>
        <entity class="com.titan.domain.Customer" access="PROPERTY">
              <attributes>
                    <id name="id"/>
              </attributes>
        </entity>
  </entity-mappings>
  ```
- Megjegyzés: access: PROPERTY/FIELD
  ```

Mapping - alapok

```
package com.titan.domain

import javax.persistence.*

@Entity
@Table(name="CUSTOMER_TABLE")
public class Customer implements java.io.Serializable {
      private long id;
      private String firstName;
      private String lastName;

      @Id
      @Column(name="CUST_ID", nullable=false, columnDefinition="integer")
      public long getId() { return id; }
      public void setId(long id) { this.id = id; }

      @Column(name="FIRST_NAME", nullable=false, length=20)
      public String getFirstName() { return firstName; }
      public void setFirstName(String firstName) { this.firstName = firstName;
}

      @Column(name="LAST_NAME", nullable=false, length=20)
      public String getLastName() { return lastName; }
      public void setLastName(String lastName){ this.lastName = lastName; }

}
```

Mapping - alapok

- Ha nem megfelelőek az alapértelmezett nevek (osztály – tábla, attribútum – oszlop), használhatunk további annotációkat
- **@Table – táblázat**
  - **Attribútumok: name, catalog, schema, uniqueConstraints**
- **@Column – oszlop**
  - **Attribútumok: name, unique (default false), nullable (default true), insertable (default ture), updatable (default true), length (default 255), precision (default 0), scale (default 0), columnDefinition (oszlop típusát megadó DDL meghatározása), table.**
- **XML megfelelő:**

```
<entity-mapping>
        <entity class="com.titan.domain.Customer" access="PROPERTY">
                        <table name="CUSTOMER_TABLE />
                        <attributes>
                                        <id name="id">
                                                        <column name="CUST_ID" nullable="false"
                                                                        column-definition="integer" />
                                        </id>
                                        <basic name="firstName">
                                                        <column name="FIRST_NAME" nullable="false", length="20"/>
                                        </basic>
                                        <basic name="lastName">
                                                        <column name="LAST_NAME" nullable="false", length="20" />
                                        </basic>
                        </attributes>
        </entity>
</entity-mappings>
```

Id automatikus generálása

- ```
  @Id
  @GeneratedValue
  public long getId() { return id; }
  ```
- Vagy (XML):
  ```
  <id name="id">
        <generated-value strategy="AUTO" />
  </id>
  ```
- AUTO: automatikus generálás (használható IDENTITY is – speciális oszloptípus elsődleges kulcsok generálására, amit több adatbázis-kezelő rendszer támogat)
- Table generator stratégia: TABLE – a felhasználó által meghatározott speciális táblázat alkalmazása a kulcsgeneráláshoz.
  - A @TableGenerator annotáció segítségével létrehozzuk a táblának megfelelő osztályt (ebben lesznek tárolva a kulcsoknak megfelelő számlálók) (kulcs azonosító, érték párok)
  - Attribútumok: name (a generátor neve, amire az Id.generator attribútum hivatkozik), catalog, schema, pkColumnName (a kulcsot meghatározó oszlop neve), valueColumnName (a számlálónak megfeleltetett Id oszlop neve), pkColumnValue (a számláló értéke), allocationSize (default 50 – mennyivel növekedjen a számláló, ha a provider lekérdez egy következő értéket → cache-elési lehetőség), uniqueConstraints
  ```

Id automatikus generálása

- TableGenerator példa:
  - ```
    @TableGenerator(name="CUST_GENERATOR" table="GENERATOR_TABLE"
      pkColumnName="PRIMARY_KEY_COLUMN"
                          valueColumnName="VALUE_COLUMN"
    pkColumnValue="CUST_ID" allocationSize=10)
    @Id
    @GeneratedValue(strategy=GenerationType.TABLE, generator="CUST_GENERATOR")
    public long getId() { return id; }
    ```
  - Vagy (XML):
    ```
    <table-generator name="CUST_GENERATOR" table="GENERATOR_TABLE" pk-column-
    name="PRIMARY_KEY_COLUMN"
      value-column-name="VALUE_COLUMN" pk-column-value="CUST_ID" allocation-
    size="10" />
    <attributes>
      <id name="id"> <generated-value strategy="TABLE"
    generator="CUST_GENERATOR" /> </id>
    </attributes>
    ```
-
    ```

Id automatikus generálása

- Ha az RDBMS támogatja, lehetőség Sequence stratégia alkalmazására: szekvenciális kulcsgenerálás egy speciális táblázat segítségével. Példa:
  - ```
    @Entity
    @Table(name="CUSTOMER_TABLE")
    @SequenceGenerator(name="CUSTOMER_SEQUENCE" sequenceName="CUST_SEQ")
    …
    @Id
    @GeneratedValue(strategy=GenerationType.SEQUENCE,
    generator="CUSTOMER_SEQUENCE") …
    ```
  - **Vagy (XML):**
    ```
    <sequence-generator name="CUSTOMER_SEQUENCE" sequence-name="CUST_SEQ"
    initial-value="0"
      allocation-size=50 />
    <attributes>
      <id name="id"> <generated-value strategy="SEQUENCE"
    generator="CUSTOMER_SEQUENCE" /> </id>
    </attributes>
    ```
  - Attribútumok: name (hogyan hivatkozik az Id annotáció a generátorra), sequenceName (milyen szekvencia táblázat lesz használva), initialValue (az elsőként használt érték), allocationSize (mennyivel lesz növelve az érték hozzáféréskor)
  - Mind a TABLE, mind a SEQUENCE stratégiák esetében deklarálható a generátor a @Table annotáció után is (vagy az Id előtt)
    ```

Kulcs osztályok és összetett kulcsok

```
package com.titan.domain;

public class CustomerPK implements java.io.Serializable {
      private String lastName;
      private long ssn;

      public CustomerPK() {  }
      public CustomerPK(String lastName, long ssn) {…}

      //getters and setters
      …

      public boolean equals(Object obj) {
            if (obj == this) return true;
            if (!(obj instanceOf CustomerPk)) return false;
            CustomerPk pk = CustomerPK(obj);
            if (!lastName.equals(pk.lastName)) return false;
            if (ssn != pk.ssn) return false;
            return true;
      }

      public int hashCode() {
            return lastName.hashCode() + (int) ssn;
      }
}
```

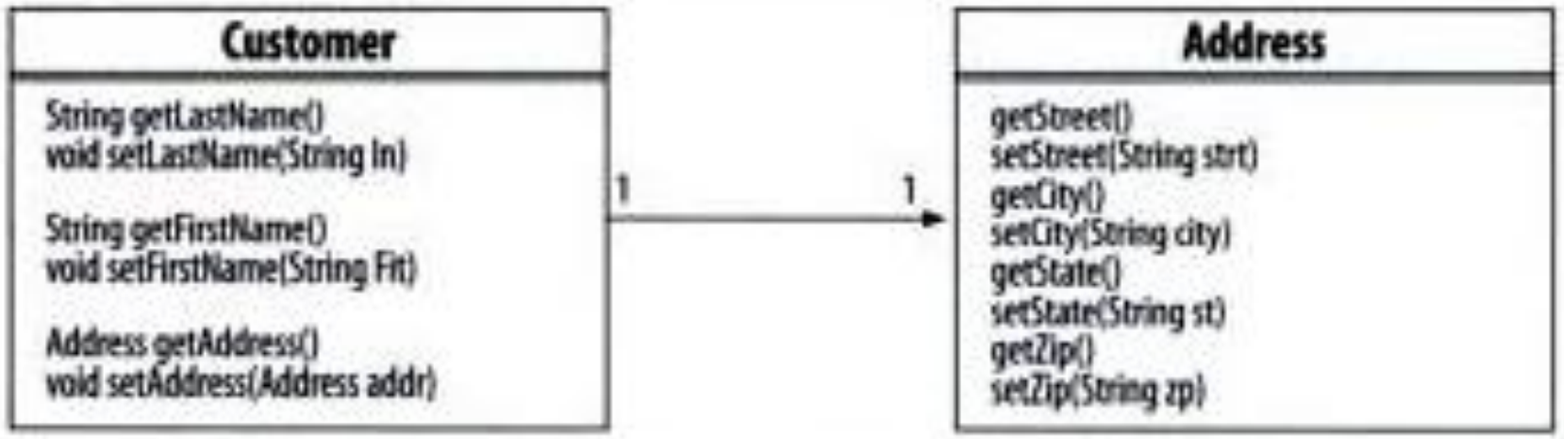
Entitások közötti kapcsolatok

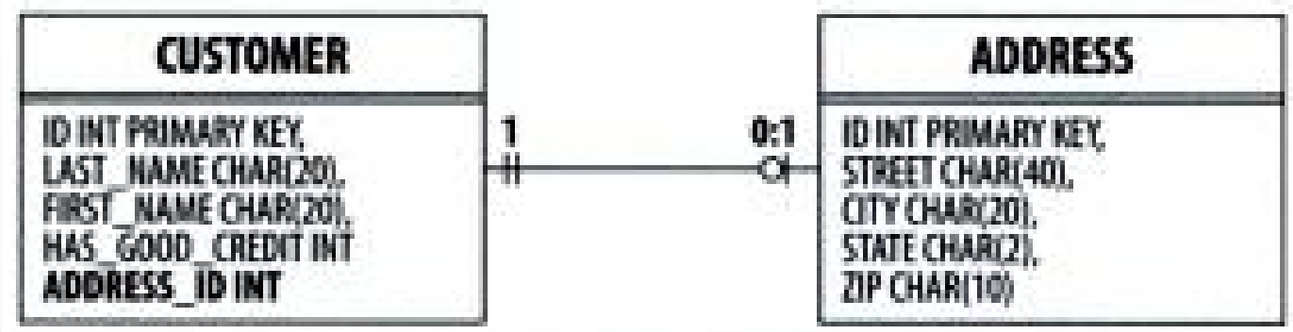
Simon Károly
simon.karoly@codespring.ro

Kapcsolatok típusai

- Egy az egyhez, egyirányú (one-to-one unidirectional): pl. Customer és Address
- Egy az egyhez, kétirányú (one-to-one bidirectional): pl. Customer és CreditCard
- Egy a többhöz, egyirányú (one-to-many unidirectional): pl. Customer és PhoneNumber
- Egy a többhöz, kétirányú (one-to-many bidirectional): pl. Cruise és Reservation
- Több az egyhez, egyirányú (many-to-one unidirectional): pl. Cruise és Ship
- Több a többhöz, egyirányú (many-to-many unidirectional): pl. Reservation és Cabin
- Több a többhöz, kétirányú (many-to-many bidirectional): pl. Cruise és Customer

One-to-One Unidirectional





- A Customer rendelkezik Address referenciával (az Address nem rendelkezik Customer referenciával)
- A CUSTOMER táblázat tartalmaz egy külső kulcsot az ADDRESS táblázathoz (az ADDRESS táblázat nem tartalmaz külső kulcsot)

One-to-One Unidirectional

- ```
  package com.titan.domain;
  @Entity
  public class Customer implements java.io.Serializable {
        …
        private Address address;
        …
        @OneToOne(cascade={CascadeType.ALL})
        @JoinColumn (name="ADDRESS_ID")
        public Address getAddress( ) { return address; }
        public void setAddress(Address address) { this.address = address; }
        …
  }
  ```

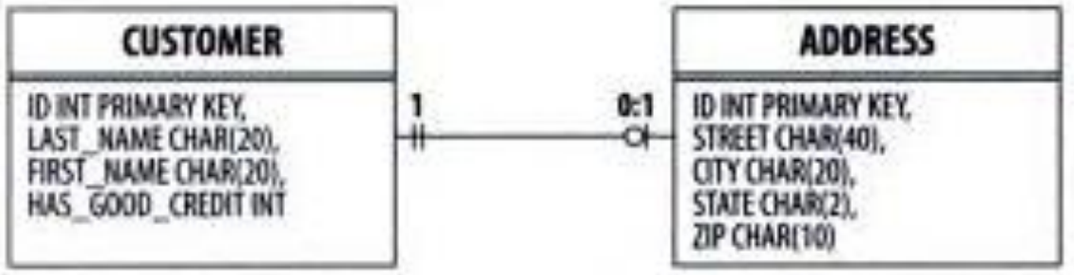
- ```
  <entity-mappings>
        <entity class="com.titan.domain.Customer" access="PROPERTY">
                  <attributes>
                            <id name="id"> <generated-value/> </id>
                            <one-to-one name="address"
  targetEntity="com.titan.domain.Address"
        fetch="LAZY" optional="true">
                                      <cascade>ALL</cascade>
                                      <join-column name="ADDRESS_ID"/>
                            </one-to-one>
                  </attributes>
        </entity>
  </entity-mappings>
  ```

One-to-One Unidirectional

- JoinColumn annotáció: meghatározza a külső kulcsnak megfelelő oszlopot, amely a másik táblázat elsődleges kulcsára hivatkozik. Ha valamelyik másik oszlopra akarunk hivatkozni (nem a másik tábla elsődleges kulcsára), akkor használhatjuk a referencedColumnName attribútumot.
- Ha összetett kulcsra akarunk hivatkozni, akkor a JoinColumns annotációt alkalmazhatjuk. A value attribútum JoinColumn típusú elemeket tartalmaz (JoinColumn típusú tömb)
- OneToOne annotáció:
  - targetEntity: a hivatkozott entitásnak megfelelő osztály meghatározása. Általában nem szükséges, mert a típus alapján a Persistence Provider meg tudja határozni.
  - fetch: lazy (lusta) vagy eager (buzgó) betöltés
  - optional: lehet-e null (true/false)
  - cascade: a CASCADING stratégia meghatározása (később tárgyaljuk)
  - mappedBy: kétirányú kapcsolat esetében alkalmazzuk (későbbi példa)
- Primary-key join columns: néhány esetben a kapcsolatban álló entitások adatait rögzítő táblázatok esetében az elsődleges kulcsok azonosak (ugyanazokat a kulcsokat használjuk), nincs külön oszlop
a külső kulcsnak. Ilyenkor a PrimaryKeyJoinColumn annotáció alkalmazható.

One-to-One Unidirectional

- PrimaryKeyJoinColumn:
  - name: az elsődleges kulcs oszlop neve
  - referencedColumnName: a hivatkozott oszlop neve (ha nem adjuk meg az elsődleges kulcsnak megfelelő).
  - columnDefinition: amennyiben a Provider generálja a sémát az SQL típus meghatározása
  - A PrimaryKeyJoinColumns annotáció szintén alkalmazható.




```
package com.titan.domain;
@Entity
public class Customer implements java.io.Serializable {
        …
        private Address homeAddress;
        …
        @OneToOne(cascade={CascadeType.ALL})
        @PrimaryKeyJoinColumn
        public Address getAddress( ) { return homeAddress; }
        public void setAddress(Address address) { this.homeAddress = address; }
        …
}
```

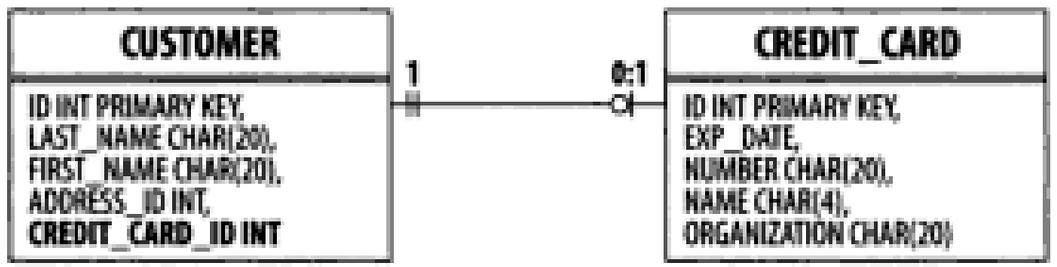
```
<entity-mappings>
        <entity class="com.titan.domain.Customer" access="PROPERTY">
                <attributes>
                        <id name="id"> <generated-value/> </id>
                        <one-to-one name="address" targetEntity="com.titan.domain.Address"
                                fetch="LAZY" optional="true">
                                <cascade-all/>
                                <primary-key-join-column/>
                        </one-to-one>
                </attributes>
        </entity>
</entity-mappings>
```

One-to-One Unidirectional

- Default relationship mapping: ha a Persistence Provider támogatja az automatikus séma-generálást, akkor nem szükséges a JoinColumn vagy PrimaryKeyJoinColumn annotációk alkalmazása. A Provider automatikusan legenerálja a táblákat és megfeleltetéseket (alapértelmezetten a generált külső kulcs oszlop neve: a hivatkozott entitás neve + "_" + az elsődleges kulcs oszlop neve).
- ```
  package com.titan.domain;
  @Entity
  public class Customer implements java.io.Serializable {
        ...
        private Address address;
        ...
        @OneToOne public Address getAddress( ) { return homeAddress; }
        public void setAddress(Address address) { this.homeAddress = address; }
        …
  }
  ```
- ```
  CREATE TABLE CUSTOMER
  (
        ID INT PRIMARY KEY NOT NULL,
        address_id INT,
        ...
  )
  ALTER TABLE CUSTOMER ADD CONSTRAINT customerREFaddress
        FOREIGN KEY (address_id) REFERENCES ADDRESS (id);
  ```

One-to-One Bidirectional

- A Customer tartalmaz CreditCard referenciát, és a CreditCard is tartalmaz Customer referenciát.
- A relációs adatbázis modellben nincs megfelelője a "kapcsolat irányának", így ebben az esetben is ugyanazt a sémát alkalmazzuk, mint az egyirányú kapcsolat esetében: a CUSTOMER táblázat fog rendelkezni egy külső kulcs oszloppal, a kulcsok a CREDIT_CARD táblázat elsődleges kulcsára hivatkoznak.
- ```
  CREATE TABLE CREDIT_CARD
  (
        ID INT PRIMARY KEY NOT NULL,
        EXP_DATE DATE, NUMBER CHAR(20),
        NAME CHAR(40),
        ORGANIZATION CHAR(20)
  )
  ```
- ```
  CREATE TABLE CUSTOMER
  (
        ID INT PRIMARY KEY NOT NULL,
        LAST_NAME CHAR(20),
        FIRST_NAME CHAR(20),
        ADDRESS_ID INT,
        CREDIT_CARD_ID INT
  )
  ```



One-to-One Bidirectional

- A OneToOne annotáció mappedBy attribútumát alkalmazzuk: meghatározza, hogy melyik osztály (esetünkben Customer) melyik mezője (esetünkben creditCard) az alapja a megfeleltetésnek.
- ```
  @Entity
  public class CreditCard implements java.io.Serializable {
        private int id;
        private Date expiration;
        private String number;
        private String name;
        private String organization;
        private Customer customer;
        ...
        @OneToOne(mappedBy="creditCard")
        public Customer getCustomer( ) { return this.customer; }
        public void setCustomer(Customer customer) { this.customer = customer; }   ...
  }
  ```
- ```
  @Entity public class Customer implements java.io.Serializable {
        private CreditCard creditCard;
        ...
        @OneToOne (cascade={CascadeType.ALL})
        @JoinColumn(name="CREDIT_CARD_ID")
        public CreditCard getCreditCard( ) { return creditCard; }
        public void setCreditCard(CreditCard card) { this.creditCard = card; }
        ...
  }
  ```

One-to-One Bidirectional

- A kétirányú kapcsolat beállítása:

```
Customer cust = new Customer( );
CreditCard card = new CreditCard( );
cust.setCreditCard(card);
card.setCustomer(cust);
entityManager.persist(cust);
```

- Két oldal: owning side (Customer), inverse side (CreditCard)
- Mielőtt a CreditCard példányt egy másik Customer-hez rendeljük hozzá, meg kell hívnunk az első Customer objektumra a setCreditCard metódust, null értéket adva át neki.

```
Customer newCust = em.find(Customer.class, newCustId);
CreditCard card = oldCustomer.getCreditCard( );
oldCustomer.setCreditCard(null);
newCust.setCreditCard(card);
```

- Mindkét oldalon be kell állítanunk az értékeket, hogy a változtatás érvényes legyen.
- Ha egy Customer lemondja a Credit kártyáját:

```
Customer cust = em.find(Customer.class, id);
em.remove(cust.getCreditCard( ));
cust.setCreditCard(null);
```

One-to-One Bidirectional

```
<entity-mappings>
      <entity class="com.titan.domain.Customer" access="PROPERTY">
            <attributes>
                  <id name="id"> <generated-value/> </id>
                  <one-to-one name="creditCard"
                        target-entity="com.titan.domain.CreditCard"
fetch="LAZY">
                        <cascade-all/>
                        <join-column name="CREDIT_CARD_ID"/>
                  </one-to-one>
            </attributes>
      </entity>
      <entity class="com.titan.domain.CreditCard" access="PROPERTY">
            <attributes>
                  <id name="id"> <generated-value/> </id>
                  <one-to-one name="customer" target-entity=
                        "com.titan.domain.Customer" mapped-
by="creditCard"/>
            </attributes>
      </entity>
</entity-mappings>
```

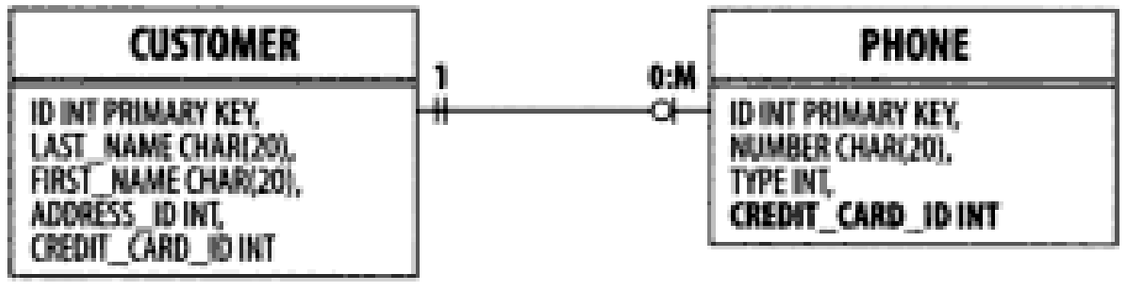
One-to-One bidirectional

- Default relationship mapping:
- ```
  package com.titan.domain;
  @Entity
  public class Customer implements java.io.Serializable {
        ...
        private CreditCard creditCard;
        ...
        @OneToOne
        public CreditCard getCreditCard( ) { return homeAddress; }
        ...
  }
  ```
- ```
  @Entity
  public class CreditCard implements java.io.Serializable {
        ...
        private Customer customer;
        ...
        @OneToOne(mappedBy="creditCard")
        public Customer getCustomer( ) { return this.customer; }
        ...
  }
  ```
- ```
  CREATE TABLE CUSTOMER
  (
        ID INT PRIMARY KEY NOT NULL,
        creditCard_id INT,
        ...
  )
  ALTER TABLE CUSTOMER ADD CONSTRAINT customerREFcreditcard
        FOREIGN KEY (creditCard_id) REFERENCES CREDITCARD (id);
  ```
  ```

One-to-Many Unidirectional

- Gyűjtemények alkalmazása. Példa: Customer és Phone (egy kliens, több telefonszám)
- Több lehetőség a megvalósításra. A PHONE táblázat tartalmazhat egy külső kulcsot, amely a Customer táblára hivatkozik. A gyakorlatban általában Join táblázatot alkalmazunk.
- Reverse-pointer mechanizmus alkalmazásának lehetősége: különbség az adatbázis séma és az OO modell között (az adatbázisban a PHONE mutat a CUSTOMER-re, míg a modellben ez fordítva van). A Persistence Provider kezeli a helyzetet.

```
CREATE TABLE PHONE
(
        ID INT PRIMARY KEY NOT NULL,
        NUMBER CHAR(20),
        TYPE INT,
        CUSTOMER_ID INT
)
```




```
@Entity
public class Customer implements java.io.Serializable {
        ...
        private Collection<Phone> phoneNumbers = new ArrayList<Phone>( );
        ...
        @OneToMany(cascade={CascadeType.ALL})
        @JoinColumn (name="CUSTOMER_ID")
        public Collection<Phone> getPhoneNumbers( ) { return phoneNumbers; }
        public void setPhoneNumbers(Collection<Phone> phones) {
                this.phoneNumbers = phones;
        }
}
```

One-to-Many Unidirectional

- Az OneToMany annotáció: egy a többhöz reláció, gyűjtemények alkalmazása
- Amennyiben nem használunk generikus típusokat, alkalmaznunk kell a OneToMany annotáció targetEntity attribútumát
- A Phone osztály:

```
package com.titan.domain;
import javax.persistence.*;
@Entity public class Phone implements java.io.Serializable {
        private int id;
        private String number;
        private int type;

        // required default constructor
        public Phone( ) {}
        public Phone(String number, int type) {
                this.number = number;
                this.type = type;
        }
        @Id
        @GeneratedValue
        public int getId( ) { return id; }
        public void setId(int id) { this.id = id; }

        public String getNumber( ) { return number; }
        public void setNumber(String number) { this.number = number; }

        public int getType( ) { return type; }
        public void setType(int type) { this.type = type; }
}
```

One-to-Many Unidirectional

- Telefonszámok hozzáadása:

```
Customer cust = entityManager.find(Customer.class, pk);
Phone phone = new Phone("617-333-3333", 5);
cust.getPhones( ).add(phone);
```

- Owning side: Customer → a Phone példánynak megfelelő bejegyzés létrejön az adatbázisban (a Provider automatikusan létrehozza)
- Telefonszám eltávolításánál el kell távolítani az objektumot a gyűjteményből, és a bejegyzést az adatbázisból:

```
cust.getPhones( ).remove(phone);
entityManager.remove(phone);
```

- XML mapping:

```
<entity-mappings>
      <entity class="com.titan.domain.Customer" access="PROPERTY">
            <attributes>
                  <id name="id"> <generated-value/> </id>
                  <one-to-many name="phones"
targetEntity="com.titan.domain.Phone">
                        <cascade-all/>
                        <join-column name="CUSTOMER_ID"/>
                  </one-to-many>
            </attributes>
      </entity>
</entity-mappings>
```

One-to-Many Unidirectional

- Join table alkalmazása:

```
create table CUSTOMER_PHONE
(
        CUSTOMER_ID int not null,
        PHONE_ID int not null unique
)
```

-
```
@Entity
public class Customer implements java.io.Serializable {
        ...
        private Collection<Phone> phoneNumbers;
        ...
        @OneToMany(cascade={CascadeType.ALL})
        @JoinTable(name="CUSTOMER_PHONE",
                        joinColumns={@JoinColumn(name="CUSTOMER_ID")},
                        inverseJoinColumns={@JoinColumn(name="PHONE_ID")})
        public Collection<Phone> getPhoneNumbers( ) { return phoneNumbers; }
        public void setPhoneNumbers(Collection<Phone> phones) {
                this.phoneNumbers = phones;
        }
}
```

- A JoinTable annotáció:
  - joinColumns attribútum: a külső kulcs megfeleltetése az "owning side" elsődleges kulcsával
  - inverseJoinColumns attribútum: a "non-owning side"-nak megfelelő megfeleltetés.
  - Ha valamelyik oldalon összetett kulcsot alkalmazunk, egyszerűen ki kell egészítenünk a tömböt további JoinColumn elemekkel

One-to-Many Unidirectional

- Join table alkalmazása, XML mapping:

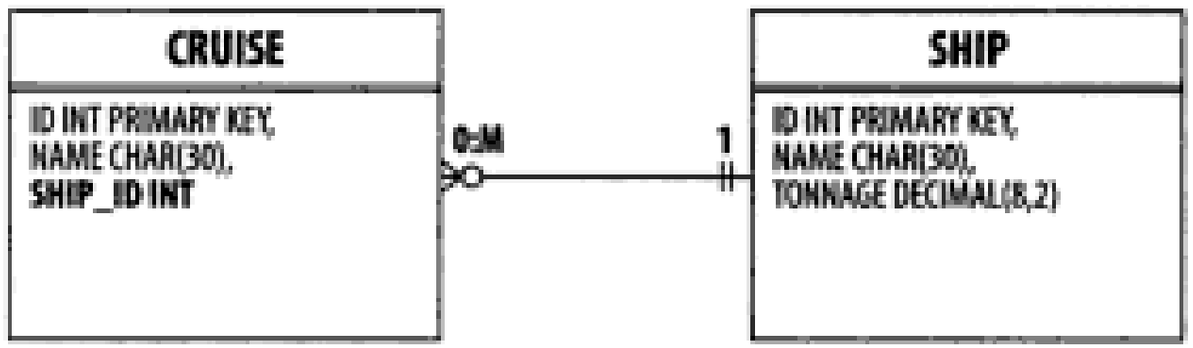
```
<entity-mappings>
      <entity class="com.titan.domain.Customer" access="PROPERTY">
                  <attributes>
                              <id name="id"> <generated-value/> </id>
                              <one-to-many name="phones"
targetEntity="com.titan.domain.Phone">
                                          <cascade-all/>
                                          <join-table name="CUSTOMER_PHONE">
                                                  <join-column name="CUSTOMER_ID"/>
                                                  <inverse-join-column
name="PHONE_ID"/>
                                          </join-table>
                              </one-to-many>
                  </attributes>
      </entity>
</entity-mappings>
```

One-to-Many Unidirectional

- Default relationship mapping:
- ```
  package com.titan.domain;
  @Entity public class Customer implements java.io.Serializable {
        ...
        private Collection<Phone> phoneNumbers = new ArrayList<Phone>( );
        ...
        @OneToMany
        public Collection<Phone> getPhoneNumbers( ) { return phoneNumbers; }
        ...
  }
  ```
- ```
  CREATE TABLE CUSTOMER_PHONE ( CUSTOMER_id INT, PHONE_id INT );
  ALTER TABLE CUSTOMER_PHONE
        ADD CONSTRAINT customer_phone_unique UNIQUE (PHONE_id);
  ALTER TABLE CUSTOMER_PHONE
        ADD CONSTRAINT customerREFphone FOREIGN KEY (CUSTOMER_id)
                REFERENCES CUSTOMER (id);
  ALTER TABLE CUSTOMER_PHONE
        ADD CONSTRAINT customerREFphone2 FOREIGN KEY (PHONE_id)
                REFERENCES PHONE (id);
  ```

Many-to-One Unidirectional

- Több különböző entitás tartalmaz egy adott másik entitásra mutató referenciát. Példa: több utazás (Cruise) történhet ugyanazzal az egy hajóval (Ship) (különböző időpontokban. A Cruise fog tartalmazni egy Ship referenciát, a Ship-nek nem kell tartalmaznia Cruise típusú referenciákat.
- ```
  CREATE TABLE SHIP
  (
        ID INT PRIMARY KEY NOT NULL,
        NAME CHAR(30),
        TONNAGE DECIMAL (8,2)
  )
  ```
- ```
  CREATE TABLE CRUISE
  (
        ID INT PRIMARY KEY NOT NULL,
        NAME CHAR(30),
        SHIP_ID INT
  )
  ```
- A ManyToOne annotáció alkalmazása.



Many-to-One Unidirectional

```
@Entity
public class Cruise implements java.io.Serializable {
      private int id;
      private String name;
      private Ship ship;

      // required default constructor
      public Cruise( ) {}
      public Cruise(String name, Ship ship) {
            this.name = name;
            this.ship = ship;
      }

      @Id
      @GeneratedValue
      public int getId( ) { return id; }
      public void setId(int id) { this.id = id; }

      public String getName( ) { return name; }
      public void setName(String name) { this.name = name; }

      @ManyToOne
      @JoinColumn (name="SHIP_ID")
      public Ship getShip( ) { return ship; }
      public void setShip(Ship ship) { this.ship = ship; }
}
```

Many-to-One Unidirectional

```
@Entity
public class Ship implements java.io.Serializable {
      private int id;
      private String name;
      private double tonnage;

      // required default constructor
      public Ship( ) {}

      public Ship(String name,double tonnage) {
            this.name = name;
            this.tonnage = tonnage;
      }

      @Id
      @GeneratedValue
      public int getId( ) { return id; }
      public void setId(int id) { this.id = id; }

      public String getName( ) { return name; }
      public void setName(String name) { this.name = name; }

      public double getTonnage( ) { return tonnage ; }
      public void setTonnage(double tonnage) { this.tonnage = tonnage ; }
}
```

Many-to-One Unidirectional

- XML mapping:

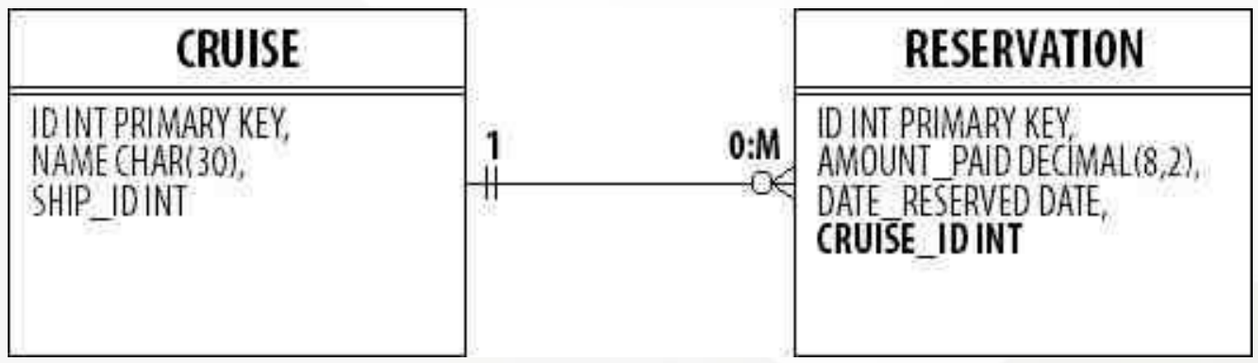
```
<entity-mappings>
      <entity class="com.titan.domain.Cruise" access="PROPERTY">
                <attributes>
                          <id name="id"> <generated-value/> </id>
                          <many-to-one name="ship" target-
entity="com.titan.domain.Ship"
                                    fetch="EAGER">
                                    <join-column name="SHIP_ID"/>
                          </many-to-one>
                </attributes>
      </entity>
      <entity class="com.titan.domain.Ship" access="PROPERTY">
                <attributes>
                          <id name="id"> <generated-value/> </id>
                </attributes>
      </entity>
</entity-mappings>
```

Many-to-One Unidirectional

- Default relationship mapping:
-
```
@Entity
public class Cruise implements java.io.Serializable {
        …
        @ManyToOne public Ship getShip( ) { return ship; }
        …
}
```
-
```
CREATE TABLE CRUISE
(
        ID INT PRIMARY KEY NOT NULL,
        ship_id INT,
        ...
)
ALTER TABLE CRUISE
        ADD CONSTRAINT cruiseREFship FOREIGN KEY (ship_id) REFERENCES SHIP (id);
```

One-to-Many Bidirectional

- Egy entitás rendelkezik egy Collection típusú referenciával. A gyűjtemény elemei más entitásokra mutató referenciák. Mindenik általuk mutatott objektum tartalmaz egy "visszamutató" (az első entitásra mutató) referenciát.
- Pl. egy Cruise entitás tartalmaz egy Reservation típusú objektumokra mutató referenciákat tartalmazó gyűjteményt (foglalások az illető hajóútra). A gyűjtemény minden eleme (Reservation objektumok) tartalmaz egy referenciát, amely az illető hajóútra (Cruise) mutat (melyik hajóútra vonatkozik az illető foglalás).
- Adatbázis séma: a RESERVATION táblázat tartalmaz egy külső kulcsot, amely a CRUISE táblázat elemeire hivatkozik. A CRUISE táblázat nem tartalmaz a RESERVATION táblázat elemeire hivatkozó külső kulcsot, de a Persistence Provider kezelni tudja a helyzetet.
-
```
CREATE TABLE RESERVATION
(
        ID INT PRIMARY KEY NOT NULL,
        AMOUNT_PAID DECIMAL (8,2),
        DATE_RESERVED DATE,
        CRUISE_ID INT
)
```



One-to-Many Bidirectional

- A ManyToOne annotáció alkalmazása a Reservation osztályban
-

```
@Entity
public class Reservation implements java.io.Serializable {
        private int id;
        private float amountPaid;
        private Date date;
        private Cruise cruise;
        public Reservation( ) {}
        public Reservation(Cruise cruise) { this.cruise = cruise; }

        @Id
        @GeneratedValue
        public int getId( ) { return id; }
        public void setId(int id) { this.id = id; }

        @Column(name="AMOUNT_PAID")
        public float getAmountPaid( ) { return amountPaid; }
        public void setAmountPaid(float amount) { amountPaid = amount; }
        @Column(name="DATE_RESERVED")
        public Date getDate( ) { return date; }
        public void setDate(Date date) { this.date = date; }

        @ManyToOne
        @JoinColumn(name="CRUISE_ID")
        public Cruise getCruise( ) { return cruise; }
        public void setCruise(Cruise cruise) { this.cruise = cruise ; }
}
```

One-to-Many Bidirectional

- OneToMany annotáció alkalmazása a Cruise osztályban, ahol a Reservation típusú gyűjtemény is található:
-
```
@Entity
public class Cruise implements java.io.Serializable {
        ...
        private Collection<Reservation> reservations =
                                                new ArrayList<Reservation>( );
        ...
        @OneToMany(mappedBy="cruise")
        public Collection<Reservation> getReservations( ) { return reservations; }
        public void setReservations(Collection<Reserveration> res) {
                this.reservations = res;
        }
}
```
- Owner side: a many-to-one oldal → mindig meg kell hívni a Reservation.setCruise metódust egy amikor hozzáadunk vagy törlünk egy Cruise esetében egy foglalást, ellenkező esetben nem lesz módosítva az adatbázis.
- Mindig állítsuk be mindkét oldalon a tulajdonságokat!
- A foglalások nem "átirányíthatóak", ha egy másik hajóútra akarunk helyet foglalni, előbb törölnünk kell az előző Reservation objektumot. Tehát a setCruise metódus null értékkel történő meghívása helyett egyszerűen töröljük a foglalást.

One-to-Many Bidirectional

- XML mapping:

```
<entity-mappings>
      <entity class="com.titan.domain.Cruise" access="PROPERTY">
                <attributes>
                          <id name="id"> <generated-value/> </id>
                          <one-to-many name="ship" target-entity=
                             "com.titan.domain.Reservation" fetch="LAZY"
mapped-by="cruise">
                          </one-to-many>
                </attributes>
      </entity>
      <entity class="com.titan.domain.Reservation" access="PROPERTY">
                <attributes>
                          <id name="id"> <generated-value/> </id>
                          <many-to-one name="cruise" target-
entity="com.titan.domain.Cruise"
                                    fetch="EAGER">
                                    <join-column name="CRUISE_ID"/>
                          </many-to-one>
                </attributes>
      </entity>
</entity-mappings>
```

One-to-Many Bidirectional

- Default relationship mapping:
-
```
@Entity
public class Reservation implements java.io.Serializable {
        …
        @ManyToOne public Cruise getCruise( ) { return cruise; }
        …
}
```
-
```
CREATE TABLE RESERVATION
(
        ID INT PRIMARY KEY NOT NULL,
        cruise_id INT,
        ...
)
ALTER TABLE RESERVATION ADD CONSTRAINT
        reservationREFcruise FOREIGN KEY (cruise_id) REFERENCES CRUISE (id);
```

Many-to-Many Bidirectional

- Több bean tartalmaz gyűjteményeket más bean-ekre mutató referenciákkal, és a gyűjteményeken belüli bean-ek szintén tartalmaznak egy-egy gyűjteményt "visszamutató" referenciákkal
- Példa: Customer – Reservation
- Séma: megfeleltetési táblázat (join table) alkalmazása
-
```
CREATE TABLE RESERVATION_CUSTOMER
(
      RESERVATION_ID INT,
      CUSTOMER_ID INT
)
```

-
```
@Entity
public class Reservation implements java.io.Serializable {
      ...
      private Set<Customer> customers = new HashSet<Customer>( );
      ...
      @ManyToMany
      @JoinTable (name="RESERVATION_CUSTOMER"),
                        joinColumns={@JoinColumn(name="RESERVATION_ID")},
                        inverseJoinColumns={@JoinColumn(name="CUSTOMER_ID")})
      public Set<Customer> getCustomers( ) { return customers; }
      public void setCustomers(Set customers);
      ...
}
```

Many-to-Many Bidirectional

- A ManyToMany annotáció alkalmazása
- Owning side: Reservation
  - JoinTable annotáció alkalmazása, joinColumns és inverseJoinColumns attribútumok meghatározása (ha nincs aktiválva az automatikus séma-generálás)
- Customer: Reservation gyűjtemény, ManyToMany annotáció (a mappedBy attribútum megadásával – a kapcsolatot meghatározó tulajdonság megadása)
-
```
@Entity
public class Customer implements java.io.Serializable {
        ...
        private Collection<Reservation> reservations =
                                        new ArrayList<Reservation>( );
        ...
        @ManyToMany(mappedBy="customers")
        public Collection<Reservation> getReservations( ) { return reservations;
}
        public void setReservations(Collection<Reservation> reservations) {
                this.reservations = reservations;
        }
        ...
}
```
- Owning side: Reservation → a kliens (foglaló) eltávolításának ezen az oldalon kell történnie:

```
Reservation reservation = em.find(Reservation.class, id);
reservation.getCustomers( ).remove(customer);
```

Many-to-Many Bidirectional

- XML mapping:

```
<entity-mappings>
      <entity class="com.titan.domain.Reservation" access="PROPERTY">
            <attributes>
                  <id name="id"> <generated-value/> </id>
                  <many-to-many name="customers" target-entity=
                        "com.titan.domain.Customer"
fetch="LAZY">
                        <join-table name="RESERVATION_CUSTOMER">
                              <join-column name="RESERVATION_ID"/>
                              <inverse-join-column
name="CUSTOMER_ID"/>
                        </join-table>
                  </many-to-many>
            </attributes>
      </entity>
      <entity class="com.titan.domain.Customer" access="PROPERTY">
            <attributes>
                  <id name="id"> <generated-value/> </id>
                  <many-to-many name="cruise" target-entity=
                        "com.titan.domain.Reservation"
                        fetch="LAZY" mapped-by="customers">
                  </many-to-many>
            </attributes>
      </entity>
</entity-mappings>
```

Many-to-Many Bidirectional

- Default relationship mapping:
- ```
  @Entity
  public class Reservation implements java.io.Serializable {
        ...
        private Set<Customer> customers = new HashSet<Customer>( );
        ...
        @ManyToMany public Set<Customer> getCustomers( ) { return customers; }
        public void setCustomers(Set customers);
        ...
  }
  ```
- ```
  CREATE TABLE RESERVATION_CUSTOMER
  (
        RESERVATION_id INT,
        CUSTOMER_id INT,
  )
  ALTER TABLE RESERVATION_CUSTOMER
        ADD CONSTRAINT reservationREFcustomer FOREIGN KEY (RESERVATION_id)
                REFERENCES RESERVATION (id);
  ALTER TABLE RESERVATION_CUSTOMER
        ADD CONSTRAINT reservationREFcustomer2 FOREIGN KEY (CUSTOMER_id)
                REFERENCES CUSTOMER (id);
  ```

Many-to-Many Unidirectional

- Több bean tartalmaz más bean-ekre mutató referenciákat tartalmazó gyűjteményeket, de a gyűjteményeken belüli bean-ek nem tartalmaznak "visszamutató" referenciákat.Pl. Reservation – Cabin
- Séma: join table (hasonlóan a one-to-many kapcsolattípushoz, csak egyediségre vonatkozó megkötés nélkül)
-
```
CREATE TABLE CABIN
(
        ID INT PRIMARY KEY NOT NULL,
        SHIP_ID INT, NAME CHAR(10),
        DECK_LEVEL INT, BED_COUNT INT
)
```
-
```
CREATE TABLE CABIN_RESERVATION
(
        RESERVATION_ID INT,
        CABIN_ID INT
)
```

-
```
@Entity
public class Reservation implements java.io.Serializable {
        ...
        @ManyToMany
        @JoinTable(name="CABIN_RESERVATION",
                                joinColumns={@JoinColumn(name="RESERVATION_ID")},
                                inverseJoinColumns={@JoinColumn(name="CABIN_ID")})
        public Set<Cabin> getCabins( ) { return cabins; }
        public void setCabins(Set<Cabin> cabins) { this.cabins = cabins; }
        ...
}
```

Many-to-Many Unidirectional

- A ManyToMany annotáció alkalmazása a Reservation (owning side) osztályon belül. Ugyanitt: a JoinTable annotáció alkalmazása a joinColumns és inverseJoinColumns attribútumok megadásával
-

```
@Entity
public class Cabin implements java.io.Serializable {
        private int id;
        private String name;
        private int bedCount;
        private int deckLevel;
        private Ship ship;

        @Id @GeneratedValue
        public int getId( ) { return id; }
        public void setId(int id) { this.id = id; }

        public String getName( ) { return name; }
        public void setName(String name) { this.name = name; }

        @Column(name="BED_COUNT")
        public int getBedCount( ) { return bedCount; }
        public void setBedCount(int count) { this.bedCount = count; }
        @Column(name="DECK_LEVEL")
        public int getDeckLevel( ) { return deckLevel; }
        public void setDeckLevel(int level) { this.deckLevel = level; }

        @ManyToOne
        @JoinColumn(name="SHIP_ID")
        public Ship getShip( ) { return ship; }
        public void setShip(Ship ship) { this.ship = ship; }
}
```

Many-to-Many Unidirectional

- XML mapping:

```
<entity-mappings>
	<entity class="com.titan.domain.Reservation" access="PROPERTY">
		<attributes>
			<id name="id"> <generated-value/> </id>
			<many-to-many name="cabins" target-
entity="com.titan.domain.Cabin"
fetch="LAZY">
<join-table name="CABIN_RESERVATION">
<join-column name="RESERVATION_ID"/>
<inverse-join-column name="CABIN_ID"/>
</join-table>
			</many-to-many>
		</attributes>
	</entity>
	<entity class="com.titan.domain.Cabin" access="PROPERTY">
		<attributes>
			<id name="id"> <generated-value/> </id>
			<many-to-one name="ship" target-
entity="com.titan.domain.Ship"
fetch="LAZY">
<join-column name="SHIP_ID"/>
			</many-to-one>
		</attributes>
	</entity>
</entity-mappings>
```

Many-to-Many Unidirectional

- Default relationship mapping:
- ```
  @Entity
  public class Reservation implements java.io.Serializable {
        ...
        @ManyToMany public Set<Cabin> getCabins( ) { return cabins; }
        public void setCabins(Set<Cabin> cabins) { this.cabins = cabins; }
        ...
  }
  ```
- ```
  CREATE TABLE RESERVATION_CABIN
  (
        RESERVATION_id INT,
        CABIN_id INT
  )
  ALTER TABLE RESERVATION_CABIN
        ADD CONSTRAINT reservationREFcabin FOREIGN KEY (RESERVATION_id)
                REFERENCES RESERVATION (id);
  ALTER TABLE RESERVATION_CABIN
        ADD CONSTRAINT reservationREFcabin2 FOREIGN KEY (CABIN_id)
                REFERENCES CABIN (id);
  ```

Gyűjtemény alapú megfeleltetések

- Eddig alkalmaztuk a Collection és Set típusokat
- Lehetőség van továbbá List és Map típusok alkalmazására
- A List alkalmazása lehetőséget ad rendezésre az OrderBy annotáció segítségével. Az annotáció value attribútuma lehetőséget ad a rendezés hogyanjának meghatározására EJB QL query segítségével (ha nem adjuk meg, a rendezés az elsődleges kulcs szerinti növekvő sorrendben történik).
- Példa: Reservation – Customer, a Reservation customer attribútuma a kliensek névsor szerint rendezett listája
- ```
  @Entity
  public class Reservation implements java.io.Serializable {
        ...
        private List<Customer> customers = new ArrayList<Customer>( );
        ...
        @ManyToMany
        @OrderBy ("lastName ASC")
        @JoinTable(name="RESERVATION_CUSTOMER"),
                  joinColumns={@JoinColumn(name="RESERVATION_ID")},
                  inverseJoinColumns={@JoinColumn(name="CUSTOMER_ID")})
        public List<Customer> getCustomers( ) { return customers; }
        public void setCustomers(Set customers);
        ...
  }
  ```
  ```

Gyűjtemény alapú megfeleltetések

- A "lastName ASC" karakterlánc alapján a Persistence Provider a vezetéknév szerinti növekvő sorrendben rendez. DESC is használható, és további attribútumok is megadhatóak. Pl. "lastName ASC, firstName ASC".
- XML mapping:

```
<entity-mappings>
      <entity class="com.titan.domain.Reservation" access="PROPERTY">
            <attributes>
                  <id name="id"> <generated-value/> </id>
                  <many-to-many name="customers" target-entity=
                              "com.titan.domain.Customer"
fetch="LAZY">
                        <order-by>lastName ASC</order-by>
                        <join-table name="RESERVATION_CUSTOMER">
                              <join-column name="RESERVATION_ID"/>
                              <inverse-join-column
name="CUSTOMER_ID"/>
                        </join-table>
                  </many-to-many>
            </attributes>
      </entity>
      ...
</entity-mappings>
```

Gyűjtemény alapú megfeleltetések

- Map alkalmazása: a kulcs egy megadott tulajdonság, az érték maga az entitás.
- A MapKey annotáció alkalmazása, a name attribútum határozza meg a tulajdonságot, amely a kulcsokat reprezentálja. Ha nem adjuk meg, az elsődleges kulcs lesz alkalmazva.
- Példa: Customer – Phone: a Map-en belül a telefonszámok (a Phone entitások number tulajdonságai) lesznek a kulcsok, a Phone entitások az értékek.
-

```
@Entity
public class Customer implements java.io.Serializable {
      ...
      private Map<String, Phone> phoneNumbers = new HashMap<String, Phone>( );
      ...
      @OneToMany(cascade={CascadeType.ALL})
      @JoinColumn(name="CUSTOMER_ID")
      @MapKey(name="number")
      public Map<String, Phone> getPhoneNumbers( ) { return phoneNumbers; }
      public void setPhoneNumbers(Map<String, Phone> phones) {
            this.phoneNumbers = phones;
      }
}
```

Gyűjtemény alapú megfeleltetések

- Map alkalmazása, XML mapping:

```
<entity-mappings>
      <entity class="com.titan.domain.Customer" access="PROPERTY">
                <attributes>
                          <id name="id"> <generated-value/> </id>
                          <one-to-many name="phoneNumbers" target-entity=
                                                    "com.titan.domain.Phone"
fetch="LAZY">
                                                    <cascade-all/>
                                                    <map-key name="number"/>
                                                    <join-column name="CUSTOMER_ID"/>
                                    </one-to-many>
                </attributes>
      </entity>
      ...
</entity-mappings>
```

Leválasztott entitások és FetchType

- Amikor egy entitást leválasztunk (detaching), nem biztos, hogy az állapota teljes mértékben inicializált. A kapcsolatok lehetnek "lazy"-ként jelölve.
- Minden kapcsolattípus esetében az annotáció rendelkezik egy fetch attribútummal, amely meghatározza, hogy a lekérdezésnél be legyen-e töltve a kapcsolatban álló entitás is.
- Amennyiben a FetchType.LAZY értéket állítjuk be, a kapcsolatban álló entitás nem lesz betöltve, mindaddig, amíg nem akarunk hozzáférni.
- Pl:

```
Customer customer = entityManager.find(Customer.class, id);
customer.getPhoneNumbers().size();
```

  A size() metódus meghívása eredményezi az adatok betöltését az adatbázisból.
- A betöltés a Persistence Provider feladata. Leválasztott entitások esetében a viselkedés Provider-specifikus. A legtöbb esetben a nem inicializált tulajdonságokra történő hivatkozás kivételt eredményez.

```
Cruise detachedCruise = ...;
try {
      int numReservations = detachedCruise.getReservations().size();
} catch (SomeVendorLazyInitializationException ex) { }
```

- Megjegyzés: az EJB QL lehetőséget ad FETCH JOIN alkalmazására:
  a query meghívásakor a kiválasztott kapcsolatok inicializálása.

Cascading

- Cascading: amikor az Entity Manager segítségével valamilyen perzisztenciával kapcsolatos műveletet végzünk el egy adott entitással, beállítható, hogy a kapcsolódó entitásokra is el legyen végezve ugyanaz a művelet.
- Példa: egy Customer mentésénél szeretnénk, ha a cím és telefonszámok is automatikusan el lennének mentve. A megfelelő (cascading) beállítások után ehhez csak össze kell kapcsolnunk az entitásokat, és meg kell hívnunk a persist metódust:

```
Customer cust = new Customer();
cust.setAddress(new Address());
cust.getPhoneNumbers().add(new Phone());
entityManager.persist(cust);
```

- A beállítás a OneToOne, OneToMany, MayToOne és ManyToMany annotációk cascade attribútumának segítségével történik.
- A cascade attribútum egy tömb, amely a végrehajtandó cascade műveleteket tartalmazza. A műveleteket CascadeType értékek segítségével adhatjuk meg. A lehetséges értékek: ALL, PERSIST, MERGE, REMOVE, REFRESH.

Cascading

- Az ALL minden műveletre vonatkozik, de külön is beállíthatóak a műveletek. Például, egy Customer esetében azt szeretnénk, hogy csak a mentésnél és törlésnél legyen végrehajtva a kapcsolódó cím mentése vagy törlése:
-
```
@Entity
public class Customer implements java.io.Serializable {
        …
        private Address homeAddress;
        …
        @OneToOne(cascade={CascadeType.PERSIST, CascadeType.REMOVE})
        @JoinColumn(name="ADDRESS_ID")
        public Address getAddress() { return homeAddress; }
        public void setAddress(Address address) { this.homeAddress = address; }
}
```
-
```
<entity-mappings>
        <entity class="com.titan.domain.Customer" access="PROPERTY">
                <attributes>
                        <id name="id"> <generated-value/> </id>
                        <one-to-one name="address"
targetEntity="com.titan.domain.Address"
                                fetch="LAZY" optional="true">
                                <cascade-persist/>
                                <cascade-remove/>
                                <primary-key-join-column/>
                        </one-to-one>
                </attributes>
        </entity>
</entity-mappings>
```

Cascading

- PERSIST: a kapcsolódó entitás mentése automatikus. Pl. nem kell külön mentenünk a lakcímet, a Customer mentésekor ezt a Provider automatikusan elvégzi (meghívva a megfelelő SQL insert utasítást). Amennyiben nincs beállítva, manuálisan kellene mentenünk.
- MERGE: ha be van állítva, akkor a szinkronizálásnál (lekapcsolt entitás visszacsatolása) nem kell meghívnunk a merge metódust a kapcsolódó entitásokra (a szinkronizálás automatikusan el lesz végezve azok esetében is). Ha a visszakapcsolás előtt új attribútumokat adtunk hozzá az objektumhoz (pl. egy új telefonszámot), és ezek még nem voltak mentve az adatbázisba, a visszacsatolásnál automatikusan el lesznek mentve (persist)
- REMOVE: törlésnél a kapcsolódó entitások is automatikusan törölve lesznek az adatbázisból
- REFRESH: a MERGE-hez hasonlóan. A refresh metódus meghívása automatikusan a kapcsolódó objektumok állapotának frissítését (szinkronizálás az adatbázissal) is eredményezi.
- **Figyelem** a használatnál: nem mindig jó ötlet. Pl. nem szeretnénk, ha egy foglalás (Reservation) törlésénél törölve lenne a kapcsolódó hajóút (Cruise), vagy kliens (Customer), stb. Mindig gondoljuk át, hogy szükséges-e, és csak akkor kapcsoljuk be, ha ebben biztosak vagyunk.

Öröklődés, JPQL, Entity Callbacks and Listeners

Simon Károly
simon.karoly@codespring.ro

Öröklődés

- A JPA specifikáció támogatja az entitások közötti származtatási kapcsolatokat, a polimorfizmust, és az ennek megfelelő query-k használatát.

- Példa:
  a Customer osztályunkat részévé tesszük egy hierarchiának,
  a Person alaposztályból származtatjuk, és a modellbe bevezetünk
  egy további Employee osztályt, melyet a Customer-ből
  származtatunk (például speciális kedvezményekben,
  árengedményekben szeretnénk részesíteni a cég alkalmazottait)

- A hierarchia leképezése a relációs adatbázisba három módon történhet:
  - Egy táblázat a osztályhierarchia számára (a single table per class hierarchy): a hierarchiával kapcsolatos minden információt ebben a táblában tárolunk
  - Egy táblázat minden osztálynak (a table per concrete class): minden osztály számára létre lesz hozva egy külön táblázat, amely tartalmazza az osztállyal kapcsolatos adatokat és a főosztálytól örökölt adatokat
  - Egy táblázat minden alosztálynak: minden osztály számára létre lesz hozva egy külön táblázat, és ez a táblázat kizárólag az osztállyal kapcsolatos adatokat tartalmazza (nem tartalmaz semmilyen főosztállyal, vagy származtatott osztállyal kapcsolatos információt)

Single Table per Class Hierarchy

- ```
  create table PERSON_HIERARCHY
  (
          id integer primary key not null,
          firstName varchar(255),
          lastName varchar(255),
          street varchar(255),
          city varchar(255),
          state varchar(255),
          zip varchar(255),
          employeeId integer,
          DISCRIMINATOR varchar(31) not null
  );
  ```

- ```
  @Entity
  @Table(name="PERSON_HIERARCHY") @Inheritance(strategy=InheritanceType.SINGLE_TABLE)
  @DiscriminatorColumn(name="DISCRIMINATOR", discriminatorType=DiscriminatorType.STRING)
  @DiscriminatorValue("PERSON")
  public class Person {
          private int id;
          private String firstName;
          private String lastName;

          @Id @GeneratedValue
          public int getId( ) { return id; }
          public void setId(int id) { this.id = id; }

          public String getFirstName( ) { return firstName; }
          public void setFirstName(String first) { this.firstName = first; }
          public String getLastName( ) { return lastName; }
          public void setLastName(String last) { this.lastName = last; }
  }
  ```

Single Table per Class Hierarchy

- Ha egy táblázatot használunk egy teljes hierarchia számára, annak a tulajdonságokon kívül tartalmaznia kell egy további oszlopot, amelynek segítségével különbséget tehetünk a különböző entitások között (**discriminator column**)
- A **@javax.persistence.Inheritence** annotációt használjuk a stratégia meghatározásához (jelezzük, hogy az entitások származtatási viszonyban állnak egymással). Az **InheritenceType** lehet **SINGLE_TABLE** (alapértelmezett), **JOINED**, vagy **TABLE_PER_CLASS**. Az annotációt, **csak** a gyökérosztályban kell alkalmazni (esetünkben Person). (Kivétel: a származtatott osztályok esetében változtatni akarjuk a stratégiát)
- A **@javax.persistence.DiscriminatorColumn** annotációt alkalmazzuk a különbségtétel lehetőségét biztosító oszlop jelölésére. A **name** attribútum határozza meg az oszlop nevét (alapértelmezetten DTYPE), a **discriminatorType** attribútum a diszkriminátor típusát. A DiscriminatorType lehet STRING (alapértelmezett), CHAR, vagy INTEGER. Ha az alapértelmezett értékeket használjuk, a teljes annotáció elhagyható.

Single Table per Class Hierarchy

- A **@javax.persistence.DiscriminatorValue** annotáció határozza meg az elkülönítő oszlop értékét (milyen érték jelenjen meg azoknak a soroknak az esetében, amelyek az illető osztály egy példányának adatait tárolják). Ha elhagyjuk, akkor a rendszer alapértelmezett értéket fog használni. Ez STRING típus esetében az entitás neve, CHAR vagy INTEGER típus esetében szolgáltató-specifikus érték.
- XML leíróállományt is alkalmazhatunk:

```
<entity-mappings>
      <entity class="com.titan.domain.Person">
                <inheritance strategy="SINGLE_TABLE"/>
                <discriminator-column name="DISCRIMINATOR" discriminator-
type="STRING"/>
                <discriminator-value>PERSON</discriminator-value>
                <attributes>
                          <id> <generated-value/> </id>
                </attributes>
      </entity>
      <entity class="com.titan.domain.Customer">
                <discriminator-value>CUST</discriminator-value>
      </entity>
      <entity class="com.titan.domain.Employee"/>
</entity-mappings>
```

Single Table per Class Hierarchy

- Előnyök:
  - Egyszerű implementálni és hatékony (a leghatákonyabb)
  - Egyetlen táblázatot kell adminisztrálni egy egész hierarchia esetében
  - A perzisztencia motornak nem kell költséges join, egyesítés (union), vagy kiválasztás (subselect) műveleteket végrehajtania egy entitás betöltésénél, vagy polimorfizmus alkalmazásának esetében a hierarchia bejárásánál.
- Hátrányok:
  - Minden oszlopnak támogatnia kell a null értékeket. Nem alkalmazható a NOT NULL megkötés.
  - A származtatott osztályok tulajdonságainak megfelelő oszlopok nem minden esetben lesznek kitöltve. A stratégia nem normalizált.

Table per Concrete Class

- ```
  create table Person
  (
        id integer primary key not null,
        firstName varchar(255),
        lastName varchar(255)
  );
  ```
- ```
  create table Customer
  (
        id integer primary key not null,
        firstName varchar(255),
        lastName varchar(255),
        street varchar(255),
        city varchar(255),
        state varchar(255),
        zip varchar(255)
  );
  ```
- ```
  create table Employee
  (
        id integer primary key not null,
        firstName varchar(255),
        lastName varchar(255),
        street varchar(255),
        city varchar(255),
        state varchar(255),
        zip varchar(255),
        employeeId integer
  );
  ```
  ```

Table per Concrete Class

- ```
  @Entity
  @Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
  public class Person { ... }
  ```
- ```
  @Entity
  public class Customer extends Person { ... }
  ```
- ```
  @Entity public class Employee extends Customer { ... }
  ```
- Csak az **Inheritence** annotáció szükséges, és csak a hierarchia gyökérosztályán belül. A stratégia típusa TABLE_PER_CLASS.
- Minden táblázat tartalmazza a hozzátartozó konkrét osztály minden tulajdonságát.
- XML leíróállomány:

```
<entity-mappings>
      <entity class="com.titan.domain.Person">
                <inheritance strategy="TABLE_PER_CLASS"/>
                <attributes>
                          <id> <generated-value/> </id>
                </attributes>
      </entity>
      <entity class="com.titan.domain.Customer"/>
      <entity class="com.titan.domain.Employee"/>
</entity-mappings>
```
```

Table per Concrete Class

- Előnyök:
  - Az oszlopok esetében használhatóak megszorítások (pl. NOT NULL).
  - Sok esetben könnyebb lehet alkalmazni egy már létező (legacy) adatbázis séma esetében
- Hátrányok:
  - Sok a redundáns oszlop: az alaposztály minden tulajdonsága újra megjelenik a származtatott osztályoknak megfelelő táblázatokban. A stratégia nem normalizált.
  - A szolgáltatónak több komplex háttérműveletet el kell végeznie, például polimorfizmus alkalmazásának esetében több queryt kell használnia a hierarchia bejárásához, vagy SQL UNION műveleteket (amely gyorsabb, de nem minden adatbázisrendszer támogatja). Mindenképpen a háttérműveletek szükségessége a hatékonyság kárára megy.
  - Következmény: alkalmazása csak indokolt esetben javasolt.

Table per Subclass

```
create table Person
(
      id integer primary key not null,
      firstName varchar(255),
      lastName varchar(255)
);
```

```
create table Customer
(
      id integer primary key not null,
      street varchar(255),
      city varchar(255),
      state varchar(255),
      zip varchar(255)
);
```

```
create table Employee
(
      EMP_PK integer primary key not null,
      employeeId integer
);
```

- Minden osztály részére létrehozunk egy külön táblázatot, de ebben az esetben csak az illető osztály tulajdonságait tároljuk ebben.
- Az entitások betöltésénél, vagy polimorfizmus alkalmazásakor a hierarchia bejárásánál a rendszer JOIN műveleteket fog végrehajtani. Minden táblázatnak kell rendelkeznie egy oszloppal, amely lehetővé teszi a JOIN művelet végrehajtását.
- Példánknál az entitásoknak megfelelő táblázatok esetében megosztott módon használjuk az elsődleges kulcs értékeket.

Table per Subclass

- ```
  @Entity
  @Inheritance(strategy=InheritanceType.JOINED)
  public class Person { ... }
  ```
- ```
  @Entity
  public class Customer extends Person { ... }
  ```
- ```
  @Entity
  @PrimaryKeyJoinColumn (name="EMP_PK")
  public class Employee extends Customer { ... }
  ```
- A **@PrimaryKeyJoinColumn** annotációt alkalmazzuk arra, hogy meghatározzuk, hogy melyik oszlop használható a JOIN művelethez. A **name** attribútum adja meg az illető oszlop nevét. Az alapértelmezett érték az alaposztály elsődleges kulcs oszlopának neve. A **referencedColumName** attribútum adja meg annak az oszlopnak a nevét, amelyet az alaposztály táblázatának esetében használunk a join műveletnél. Alapértelmezett az elsődleges kulcs oszlopának neve. Amennyiben a két osztály esetében az elsődleges kulcs neve azonos, az annotáció nem szükséges. Összetett kulcsok esetében a PrimaryKeyJoinColumns annotáció alkalmazható.
  ```

Table per Subclass

- XML leíróállomány:

```
<entity-mappings>
        <entity class="com.titan.domain.Person">
                <inheritance strategy="JOINED"/>
                <attributes>
                        <id> <generated-value/> </id>
                </attributes>
        </entity>
        <entity class="com.titan.domain.Customer"/>
        <entity class="com.titan.domain.Employee">
                <primary-key-join-column name="EMP_PK"/>
        </entity>
</entity-mappings>
```

- Előnyök:
  - Normalizált.
  - Alkalmazhatóak megszorítások.
  - Hatékonyabb a TABLE_PER_CLASS stratégiánál (amennyiben az SQL UNION műveleteket nem támogatja az adatbázis-kezelő rendszer)
- Hátrány:
  - Kevésbé hatékony, a SINGLE_TABLE stratégiával összehasonlítva

Megjegyzések, kiegészítések

- Kevert stratégiák: a JPA jelenlegi specifikációja a származtatási kapcsolatok esetében alkalmazható kevert stratégiákat (mixing strategies) opcionálisnak tekinti. A vonatkozó szabályok egy következő specifikációnak lehetnek részei.
- Nonentity Base Classes: lehetőség van arra, hogy alaposztályként ne entitást alkalmazzunk, hanem egy egyszerű osztályt (amelynek nem felel meg táblázat az adatbázisban). Az ilyen osztályok esetében a @javax.persistence.MappedSuperclass annotáció alkalmazható.
  - Példánkat módosíthatnánk olyan módon, hogy a Person osztály ne legyen entitás.
  - MappedSuperclass annotációval ellátott osztályokat elhelyezhetünk két entitás közé is a hierarchiában
  - Az annotáció alkalmazása ezekben az esetekben kötelező, mivel a Persistence Provider teljesen figyelmen kívül hagyja a nem annotált osztályokat.
  - Ha XML leíróállományt alkalmazunk a mapped-superclass elemet használhatjuk az entity-mapping elemen belül.

Query-k és JPQL

- JPA: EJB QL és native SQL alkalmazásának lehetősége
- JPQL: az SQL-hez hasonló deklaratív lekérdező nyelv, amely Java objektumokkal dolgozik (nem relációs sémával). A lekérdezéseknél az entitások tulajdonságaira és kapcsolataira hivatkozunk, nem a táblázatokban tárolt adatokra. A query végrehajtásakor a rendszer a metaadatok alapján natív SQL query(k)-be alakítja a kérést, és ezek továbbítódnak a JDBC driver-hez. Ilyen módon az JPQL adatbázisrendszertől független.
- Native queries: van, amikor az JPQL alkalmazása nem elégséges. Mivel rendszer-független, előfordulhat, hogy nem tudunk élni az egyes rendszerek által biztosított bizonyos hatékony megoldásokkal, illetve nem tudunk végrehajtani tárolt eljárásokat. Ezért: a JPA lehetőséget ad native query-k alkalmazására, ehhez megfelelő API-t biztosít.

Query API

- ```
  package javax.persistence;
  public interface Query {
          public List getResultList( );
          public Object getSingleResult( );
          public int executeUpdate( );
          public Query setMaxResults(int maxResult);
          public Query setFirstResult(int startPosition);
          public Query setHint(String hintName, Object value);
          public Query setParameter(String name, Object value);
          public Query setParameter(String name, Date value, TemporalType temporalType);
          public Query setParameter(String name, Calendar value, TemporalType temporalType);
          public Query setParameter(int position, Object value);
          public Query setParameter(int position, Date value, TemporalType temporalType);
          public Query setParameter(int position, Calendar value, TemporalType temporalType);
          public Query setFlushMode(FlushModeType flushMode);
  }
  ```
- ```
  package javax.persistence;
  public interface EntityManager {
          public Query createQuery(String ejbqlString);
          public Query createNamedQuery(String name);
          public Query createNativeQuery(String sqlString);
          public Query createNativeQuery(String sqlString, Class resultClass);
          public Query createNativeQuery(String sqlString, String resultSetMapping);
  }
  ```
- ```
  try {
          Query query = entityManager.creatQuery( "from Customer c where c.firstName='Bill'
                                                          and c.lastName='Burke'");
          Customer cust = (Customer)query.getSingleResult( );
  } catch (EntityNotFoundException notFound) {
  } catch (NonUniqueResultException nonUnique) {
  }
  ```
  ```

Query API

- Amennyiben a getSingleResult metódus nem talál eredményt, EntityNotFoundException típusú futási idejű kivételt kapunk. Amennyiben több eredményt kap (több Bill Burke létezik), NonUniqueResultException típusú futási idejű kivételt kapunk. Megoldás:

```
Query query = entityManager.creatQuery( "from Customer c where c.firstName='Bill'
                                                    and c.lastName='Burke'");
java.util.List bills = query.getResultList( );
```

- Paraméterek:
  - A JDBC API java.sql.PreparedStatement osztályához hasonlóan az JPQL lehetőséget ad paraméterek alklamazására, így lekérdezéseink többször végrehajthatóak különböző paraméterekkel. Két lehetőségünk van paraméterek megadására: név, vagy pozíció szerint
  - Név szerint (ez javasolt, mivel a kód áttekinthetőbbé válik):

```
public List findByName(String first, String last) {
  Query query = entityManager.createQuery( "from Customer c where c.firstName=:first
                                                    and c.lastName=:last");
  query.setParameter("first", first);
  query.setParameter("last", last);
  return query.getResultList( );
}
```

  - Pozíció szerint:

```
public List findByName(String first, String last) {
  Query query = entityManager.createQuery( "from Customer c where c.firstName=?1
                                                    and c.lastName=?2");
  query.setParameter(1, first);
  query.setParameter(2, last);
  return query.getResultList( );
}
```

Query API

- Date típusú paraméterek:
  - Date, vagy Calendar típusú paraméterek esetében a Query interfész megfelelő (speciális) metódusait kell alkalmaznunk. Egy Date vagy Calendar objektum lehet egy dátum, idő, vagy timestamp. A Query objektumnak meg kell mondanunk, hogy mi a paraméter pontos típusa. Ezt a TemporalType paraméter segítségével tehetjük meg, amely enum típusú, lehetséges értékei DATE (java.sql.Date), TIME (java.sql.Time), vagy TIMESTAMP (java.sql.TimeStamp).
- Hints:
  - Egyes szolgáltatók további funkcionalitásokat biztosítanak, amelyek a Query interfész setHint metódusűn keresztül érhetőek el. Pl. bizonyos alkalmazásszerverek esetében időhatárt (timeout értéket) szabhatunk meg a lekérdezésnek:

```
Query query = manager.createQuery("from Customer c");
query.setHint("org.hibernate.timeout", 1000);
```

    Az első paraméter a funkcionalitásnak megfelelő azonosító, a második egy tetszőleges objektum.
- FlushMode
  - Vannak helyzetek, amikor egy adott query esetében a lekérdezés végrehajtásának idejére egy adott flushing stratégiát szeretnénk beállítani. Ehhez az interfész biztosítja számunkra a setFlushMode metódust. Pl. tudathatjuk a Provider-rel, hogy nem szeretnénk engedélyezni automatikus flushing műveletek végrehajtását egy adott query lefutása előtt:

Query API

- Paging result (lapozás):
  - Megtörténhet, hogy egy lekérdezés nagyon sok objektumot térít vissza, és ez számunkra a feldolgozás szempontjából nem előnyös (pl. megjelenítés egy weboldalon egy listában, ahol az egyszerre megjeleníthető sorok száma korlátozott). A Query API által kínált megoldások:
  - Meghatározhatjuk a visszatérített eredmények maximális számát, és az első indexét:

```
public List getCustomers(int max, int index) {
  Query query = entityManager.createQuery("from Customer c");
  return query.setMaxResults(max).setFirstResult(index).getResultList( );
}
```

  - Lecsatolhatjuk az EntityManager által menedzselt entitásokat a clear metódus segítségével (előnyös lehet, a memóriakapacitással kapcsolatos problémák elkerüléséhez):

```
List results; int first = 0; int max = 10;
do {
  results = getCustomers(max, first);
  Iterator it = results.iterator( );
  while (it.hasNext( )) {
            Customer c = (Customer)it.next( );
            System.out.println(c.getFirstName() + " " + c.getLastName( ));
  }
  entityManager.clear( );
  first = first + results.getSize( );
} while (results.size( ) > 0);
```

JPQL

- Entitások absztrakt perzisztencia sémája: absztrakt sémanév, alaptulajdonságok, kapcsolatok tulajdonságai. Az EJB QL absztrakt sémaneveket használ a bean-ek azonosításához, alaptulajdonságokat az értékek meghatározásához, kapcsolat tulajdonságokat a hierarchián belüli navigáláshoz
- Absztrakt sémanevek:
  - Alapértelmezetten az entitás neve, vagy az Entity annotáció name attribútumának értéke
  - `@Entity public class Customer {...}`
    `entityManager.createQuery("SELECT c FROM Customer AS c");`
  - `@Entity(name="Cust") public class Customer {...}`
    `entityManager.createQuery("SELECT c FROM Cust AS c");`
- Egyszerű lekérdezések:
  - `SELECT OBJECT( c ) FROM Customer AS c`
  - A Customer az absztrakt sémanév, a c azonosító (alias).
  - Az alias nem lehet azonos a séma nevével, és nem case sensitive (pl. customer sem lehet)
  - Az AS és az OBJECT kulcsszavak használata opcionális, a fenti lekérdezés így is írható:
    `SELECT c FROM Customer c`
  - Nem használhatóak azonosítóként az EJB QL kulcsszavai, és az SQL kulcsszavak használata sem ajánlott (a későbbi verziókban le lehetnek foglalva) `(EJB QL kulcsszavak: SELECT, FROM, WHERE, UPDATE, DELETE, JOIN, OUTER, INNER, GROUP, BY, HAVING, FETCH, DISTINCT, OBJECT, NULL, TRUE, FALSE, NOT, AND, OR, BETWEEN, LIKE, IN, AS, UNKNOWN, EMPTY, MEMBER, OF, IS, AVG, MAX, MIN, SUM COUNT, ORDER ASC, DESC, MOD, UPPER, LOWER, TRIM, POSITION, CHARACTER_LENGTH, CHAR_LENGTH, BIT_LENGTH, CURRENT_TIME, CURRENT_DATE, CURRENT_TIMESTAMP, NEW)`

JPQL

- Entitások és kapcsolatok tulajdonságainak lekérdezése:
  - **`SELECT c.firstName, c.lastName FROM Customer AS c`**
  - Hivatkozás a queryn belül: az annotáció alkalmazásának megfelelően. Ha az annotációt közvetlenül az attribútumnál alkalmaztuk (pl. az id attribútumot annotáltuk), akkor a query-ben az attribútumok neveinek kell megjelennie. Ha a metódus(oka)t annotáltuk (pl. a getId metódusnál használtuk az annotációt), akkor a metódus nevéből az előtag elmarad, és a következő karakter kisbetűre alakul (pl. az id azonosítót használjuk a queryn belül). Amennyiben betartjuk a Java elnevezési konvenciókat, nincs különbség, de amennyiben eltérünk a konvencióktól, oda kell figyelnünk erre.
  - Amennyiben a lekérdezésnek több eredménye van, a getResultList metódust alkalmazzuk. Ha a lekérdezés több oszlopra vonatkozik, az eredmények egy Object tömb formájában lesznek visszatérítve:

```
Query query = manager.createQuery("SELECT c.firstName, c.lastName FROM Customer AS
c");
List results = query.getResultList();
Iterator it = results.iterator();
while (it.hasNext()) {
  Object[] result = (Object[]) it.next();
  String first = (String)result[0];
  String last = (String)result[1];
}
```

  - Kapcsolatok és navigáció:

```
SELECT c.creditCard FROM Customer AS c
```

  - Bármilyen mélységig lehetséges:

```
SELECT c.creditCard.creditCompany.address.city
FROM Customer AS c
```

JPQL

- Nem léphetünk "tovább" a perzisztens tulajdonságokon (csak entitások tulajdonságait kérdezhetjük le). Megjegyzés: lehetőség van az osztályok (amelyek nem entitások) beágyazására az @Embedded annotáció segítségével, és ebben az esetben lekérdezhetőek a tulajdonságok (továbbléphetünk)
- A több entitásra mutató (kollekció típusú) tulajdonságok nem kérdezhetőek le. Pl. nem helyes:
  `SELECT c.reservations.cruise FROM Customer AS c`
  (mert `a customer.getReservations( ).getCruise( );` sem helyes…)
  Megoldás: IN vagy JOIN operátorok alkalmazása

- Konstruktor kifejezések:
  - Lehetőségünk van konstruktorokat használni a SELECT műveleteken belül: egyszerű Java objektumokat (amelyek nem entitások) hozhatunk létre és átadhatunk a konstruktoroknak a select által visszatérített eredményeket (oszlopokat).
  - Pl.: a Customer lastName és firstName tulajdonságát össze szeretnénk vonni egy Name osztály segítségével (a Name példányok egyszerű Java objektumok lesznek, nem entitások):
    `SELECT new com.titan.domain.Name(c.firstName, c.lastName)FROM Customer c`

JPQL

- INNER JOIN
  - Ha kollekció alapú kapcsolatok esetében akarunk lekérdezéseket végrehajtani (az eredmények egy bean gyűjtemény elemei), az IN operátort kell alkalmaznunk.
  - Pl. egy Customer minden foglalását szeretnénk lekérdezni:
    ```
    SELECT r FROM Customer AS c, IN( c.reservations ) r
    ```
  - Az IN operátor a reservations gyűjtemény különálló elemeihez hozzárendeli az r azonosítót, így közvetlen módon hivatkozhatunk rájuk, és lekérdezhetjük őket EJB QL segítségével.
  - Az azonosító segítségével a "továbbnavigálás" is lehetséges:
    ```
    SELECT r.cruise FROM Customer AS c, IN( c.reservations ) r
    ```
  - A deklarált azonosítók értékelése a FROM kikötésen belül balról jobbra történik. Ha deklarálunk egy azonosítót, azt használhatjuk a FROM kikötés következő részein.
  - IN helyett INNER JOIN is alkalmazható (ez az SQL nyelvhez szokott programozóknak lehet természetesebb). Ezen kívül az INNER kulcsszó is opcionális (egyszerűen JOIN-t is írhatunk).
  - A megoldás természetesen bonyolultabb lekérdezésekre is lehetőséget ad:
    ```
    SELECT cbn.ship FROM Customer c
    INNER JOIN c.reservations r INNER JOIN r.cabins cbn
    ```

JPQL

- LEFT JOIN
  - Olyankor alkalmazhatjuk, amikor a lekérdezésnek nem biztos, hogy lesznek eredményei (a nem létező értékek null értékként fognak megjelenni a result set-ben)
  - Pl. le akarjuk kérdezni a Customerek listáját, és egy-egy Customer minden telefonszámát, de nem biztos, hogy minden Customer adott meg telefonszámot (ennek ellenére a nevét még le akarjuk kérdezni):

    ```
    SELECT c.firstName, c.lastName, p.number
    FROM Customer c LEFT JOIN c.phoneNumbers p
    ```

    (ha valaki nem adott meg telefonszámot, a neve mellett null érték fog megjelenni)
  - LEFT JOIN helyett LEFT OUTER JOIN is alkalmazható
- FETCH JOIN
  - Akkor is betölthetőek (preload) egy entitás kapcsolatai, ha a FetchType LAZY-re volt állítva.
  - Pl. egy Customer telefonszámai esetében:

    ```
    @OneToMany(fetch=FetchType.LAZY)
    public Collection<Phone> getPhones( ) { return phones; }
    ```
  - Lekérdezhetjük a Customer listát, és utána egy ciklussal bejárhatjuk, minden elemre meghívva a getPhoneNumbers metódust, de ebben az esetben a Provider-nek minden lekérdezésnél végre kell hajtania egy plusz query-t (N elem esetében N+1 lekérdezés lenne végrehajtva).
  - Jobb megoldás lehet:

    ```
    SELECT c FROM Customer c LEFT JOIN FETCH c.phones
    ```

JPQL

- DISTINCT
  - Pl. a foglalás esetében le akarjuk kérdezni az összes Customer-t, de amennyiben egy Customer-nek több foglalása is van, azt akarjuk, hogy csak egyszer jelenjen meg a listában:
    ```
    SELECT DISTINCT cust FROM Reservation AS res, IN (res.customers)
    cust
    ```
- WHERE – literálok alkalmazása
  - ```
    SELECT c FROM Customer AS c
    WHERE c.creditCard.creditCompany.name = 'Capital One'
    ```
  - Karakterláncokon kívül használhatunk számértékeket, boolean értékeket (TRUE/FALSE)
- WHERE – operátorok sorrendje
  - Navigáció (.), aritmetikai operátorok (unáris + és -, *, /, +, -), összehasonlítások (=, >, >=, <, <=, <>, LIKE, BETWEEN, IN, IS NULL, IS EMPTY, MEMBER OF), logikai operátorok (NOT, AND, OR)
- WHERE – aritmetikai operátorok
  - ```
    SELECT r FROM Reservation AS r WHERE (r.amountPaid * .01) > 300.00
    ```
  - A szabályok azonosak a Java programozási nyelv szabályaival (pl. ha double értéket szorzunk int-el az int először double-be lesz alakítva). String és boolean típus, valamint entitás nem használható (speciális függvények állnak rendelkezésünkre, pl. karakterláncok összefűzésére, stb.).

JPQL

- WHERE – logikai operátorok
  - Az AND és OR kiértékelése nem a Java-ban megszokott (&&, ||) módon történik. A specifikáció nem rögzít a kifejezés jobb oldalának kiértékelésével kapcsolatos szabályokat (pl. Java-ban a jobb oldal kiértékelése && operátor esetén csak akkor történik meg, ha a bal oldal true). A kiértékelés a natív SQL nyelvtől függ (amelybe a lekérdezés le lesz fordítva).
- WHERE – összehasonlítások
  - ```
    SELECT s FROM Ship AS s
    WHERE s.tonnage >= 80000.00 AND s.tonnage <= 130000.00
    ```
  - A specifikáció nem rögzít bizonyos szemantikai szabályokat. Pl. String-ek összehasonlításánál kis-/nagybetű, white space stb. Ezek figyelembevétele a natív SQL nyelvtől függ (mivel különbségek vannak, és az EJB QL-nek függetlennek kell maradnia)
- WHERE – egyenlőség
  - Vizsgálata azonos típusok között lehetséges, kivételt képeznek a számértékek (ahol pl. egy int összehasonlítható egy long értékkel, vagy a burkoló osztály egy példányának értékével, tehát egy Integer-rel is)
  - A String-ek vizsgálatát nem specifikálja (pl. nincs kikötve, hogy teljes karakterenkénti egyezés kell). Ez natív SQL nyelv-specifikus.
  - Azonos típusú entitások egyenlősége is vizsgálható, ez az elsődleges kulcs szerint történik.
    ```

Natív query-k

- Néhány esetben alkalmazásuk szükséges lehet (pl. ha el akarunk érni speciális adatbázis-specifikus funkcionalitásokat).
- Az EJB QL használatukat támogatja, az EntityManager interface biztosít erre metódsokat.
- Megjegyzés: a javax.sql.DataSource injektálásával közvetlenül hozzáférhetünk a JDBC kapcsolathoz is, de ebben az esetben változtatásaink nem fogják módosjtani a perzisztencia környezetet (persistence context).
- Egyszerű natív query:
  - ```
    Query query = manager.createNativeQuery(
      "SELECT p.phone_PK, p.phone_number, p.type FROM PHONE AS p",
    Phone.class );
    ```
  - Az entitással való megfeleltetés a metaadatok alapján történik (a második, entityClass paraméter alapján), tehát a result set-en belül visszatérített oszlopoknak teljesen meg kell felelniük az entitás O/R mapping-jének. Minden tulajdonságnak szerepelnie kell a lekérdezésben.
- Bonyolult natív query-k:
  - A Query createNativeQuery(String sql, String mappingName) metódus segítségével hozhatóak létre. Pl. több entitás visszatérítésénél alkalmazható
  - A második paraméter egy SqlResultSetMapping annotációnak felel meg.
    A megfeleltetés megadásához rendelkezésünkre állnak még az EntityResult, FieldResult és ColumnResult annotációk.
    ```

Natív query-k

- Több entitás lekérdezése:
  - ```
    @Entity
    @SqlResultSetMapping(name="customerAndCreditCardMapping",
                  entities={@EntityResult(entityClass=Customer.class),
                            @EntityResult(entityClass=CreditCard.class,
                                      fields={@FieldResult(name="id", column="CC_ID"),
                                              @FieldResult(name="number",
                                                      column="number")}
                            )})
    public class Customer {...}
    ```
  - ```
    // execution code
    {
      Query query = manager.createNativeQuery(
                "SELECT c.id, c.firstName, cc.id As CC_ID, cc.number"
                + "FROM CUST_TABLE c, CREDIT_CARD_TABLE cc"
                + "WHERE c.credit_card_id = cc.id",

      "customerAndCreditCardMapping");
    }
    ```
  - Az EntityField annotáció alkalmazásával oszlopokat feleltethetünk meg az entitások tulajdonságainak. A name attribútum határozza meg a tulajdonság nevét, a column az oszlop nevét. Példánk esetében csak a CreditCard entitásnál kell alkalmaznunk (mivel innen csak két tulajdonságot kérdezünk le). Mivel az elsődleges kulcsok nevei azonosak, lehetőséget kell teremtenünk a különbségtételre (cc.id AS CC_ID).

Natív query-k

- Több entitás lekérdezése, XML:
  - ```
    <entity-mappings>
      <sql-result-set-mapping name="customerAndCreditCardMapping">
                <entity-result entity-class="com.titan.domain.Customer"/>
                <entity-result entity-class="com.titan.domain.CreditCard"/>
                          <field-result name="id" column="CC_ID"/>
                          <field-result name="number" column="number"/>
                </entity-result>
      </sql-result-set-mapping>
    </entity-mappings>
    ```
- Skalár értékek és entitások lekérdezése:
  - Pl. hajóutak listájának és foglalások számának lekérdezése

```
@SqlResultSetMapping(name="reservationCount",
                entities=@EntityResult(name="com.titan.domain.Cruise",
                                       fields=@FieldResult(name="id", column="id")),
                columns=@ColumnResult(name="resCount"))
@Entity
public class Cruise {...}
```

  - ```
    Query query = manager.createNativeQuery(
      "SELECT c.id, count(Reservation.id) as resCount FROM Cruise c
      LEFT JOIN Reservation ON c.id = Reservation.CRUISE_ID
      GROUP BY c.id", "reservationCount");
    ```
  - ```
    <entity-mappings>
      <sql-result-set-mapping name="reservationCount">
                <entity-result entity-class="com.titan.domain.Cruise">
                          <field-result name="id" column="id"/>
                </entity-result>
                <column-result name="resCount"/>
      </sql-result-set-mapping>
    </entity-mappings>
    ```
    ```

Visszahívási mechanizmus és figyelők

- Entity Callbacks and Listeners
- Az EntityManager persist, merge, remove, find metódusai, valamint a query-k végrehajtása bizonyos életciklussal kapcsolatos események kiváltását eredményezi (pl. a persist metódus insert műveletet trigger-el)
- Bizonyos esetekben fontos, hogy entitásaink visszajelzést kapjanak ezekről az eseményekről (pl. naplózni szeretnénk az adatbázisban történő változásokat)
- Lehetőségek: callback metódusokat állíthatunk be az entitásokon belül, és entitás figyelőket (entity listeners) regisztrálhatunk.
- Callback események: az entitás életciklusának fázisait (javax.persistence) annotációk reprezentálják: **PrePersist** (a persist metódushívás pillanata), **PostPersist** (az insert művelet végre volt hajtva), **PostLoad** (az entitás betöltése után, find vagy getReference metódushívás hatására), **PreUpdate** (a szinkronizálás, pl. flush metódushívás, vagy commit előtt), **PostUpdate** (a szinkronizálás után), **PreRemove** (a remove metódus hívásánál), **PostRemove** (a törlés végrehajtása után)

Entity osztályok és visszahívás

- Callback metódusok beállítása az annotációk segítségével lehetséges. A metódus lehet public, private, protected, vagy package private. A visszatérített érték típusának void-nak kell lennie, a metódus nem dobhat ellenőrzött kivételt, és nem lehetnek argumentumai.
- ```
  @Entity
  public class Cabin {
        ...
        @PostPersist
        void afterInsert( ) {
              ...
        }

        @PostLoad
        void afterLoading( ) {
              ...
        }
  }
  ```
- ```
  <entity class="com.titan.domain.Cabin">
        <post-persist name="afterInsert"/>
        <post-load name="afterLoading"/>
  </entity>
  ```
- Egy adott esemény felléptekor entity manager meghívja ezeket a metódusokat.

Figyelők

- Az entity listenerek annotációk (vagy XML leíró) segítségével hozzárendelhetőek entitásokhoz. A figyelőkön belül metódusokat hozhatunk létre egyes események kezelésére. A metódusok visszatérített értékének típusa void, és egy Object típusú paraméterük van, amely az entitás példányra mutat. A metódusokat a callback mechanizmusnál leírt annotációkkal látjuk el.
- ```
  public class TitanAuditLogger {
        @PostPersist void postInsert(Object entity) {…}
        @PostLoad void postLoad(Object entity){…}
  }
  ```
- Az entitáshoz a figyelőt az EntityListeners annotációval rendelhetjük hozzá:

```
@Entity
@EntityListeners({TitanAuditLogger.class, EntityJmxNotifier.class})
public class Cabin {…}
```

- ```
  <entity class="com.titan.domain.Cabin>
        <entity-listeners>
              <entity-listener class="com.titan.listeners.TitanAuditLogger>
              </entity-listener>
              <entity-listener class="com.titan.listeners.EntityJmxNotifier>
                    <pre-persist name="beforeInsert" />
                    <post-load name="afterLoading" />
              </entity-listener>
        </entity-listeners>
  </entity>
  ```

Figyelők

- A figyelők értesítésének sorrendje azonos deklarációjuk sorrendjével. Az entitás callback metódusai a figyelők értesítése után lesznek meghívva.
- Alapértelmezett figyelőket adhatunk meg, amelyek a perzisztencia egység minden entitásához hozzá lesznek rendelve. Ez az <entity-listeners> elem közvetlenül az <entity-mappings> elemen belüli alkalmazásával lehetséges.
- Ha az alapértelmezett figyelőket ki szeretnénk kapcsolni egy adott osztály esetében, ezt az ExcludeDefaultListeners annotációval, vagy az <exclude-default-listeners /> elem segítségével tehetjük meg.
- Ha az entitások között származtatási viszony áll fent, és az alaposztályhoz figyelők vannak hozzárendelve, ezeket a származtatott osztályok "öröklik". Sorrend szempontjából előbb az alaposztály figyelői lesznek értesítve az eseményekről. A származtatott osztályokban az alaposztály figyelői "kikapcsolhatóak" az ExcludeSuperclassListeners annotációval, vagy az <exlude-superclass-listeners /> elemmel.