CODESPRING
OUR CODE YOUR SUCCESS
20 years

Codespring mentorprogram
Enterprise platformok es keretrendszerek

Let's win the race together!

Objektum-relációs leképezés

Jakarta Persistence API & Hibernate

Simon Károly, Sulyok Csaba
simon.karoly@codespring.ro, sulyok.csaba@codespring.ro

1. rész

JPA

DAO hierarchia mögötti intuició

- Az eddig tanulmányozott DAO hierarchia intuitíven diktál automatizálási lehetőségeket:
  - A modell osztályok struktúrája alapján a CRUD műveleteket megvalósító query-k generálhatóak
  - Ugyanígy a táblázatok struktúrája is (valamennyire) kikövetkeztethető
- Hátrányok az eddigi megközelítéssel:
  - Nem minden esetben migrálhatunk egyszerűen adatbázis-típusok között.
  - Mivel tényleges query-k vannak jelen a `Statement`-ekben, ezek lehet, hogy nem lesznek támogatottak minden elért RDBMS-ben.

OOP ↔ RDB(MS)

- Az objektum-orientált és relációs "látásmódok" között eltérések fedhetőek fel (*object-relational impedance mismatch*):
  - Öröklődés (inheritance)–támogatott az OOP-ben, de nem tiszta, hogy egy mappelt adaatbázis-rendszerben hogyan legyen kezelve.
  - Adatrejtés–az adatbázis-rendszerek nem támogatják a privát-publikus adattagokat.
  - Tranzakciók hiánya az OOP rendszerekben
  - Az OOP megengedi a komplex adattagokat (pl. másik entitásból álló kollekció)
  - Ezek által különböző sémákat kell karbantartanunk a 2 oldalon.
- Lehetőségek:
  - NoSQL / OODBMS
  - **Objektum-relációs leképezés** - object relational mapping (ORM) - keretrendszer
- *Java ORM keretrendszerek:*
  - Hibernate - leginkább közismert
  - EclipseLink - JPA referencia implementáció
  - OpenJPA, MyBatis, TopLink, stb.

JPA - Jakarta Persistence API

- A Jakarta Enterprise Edition (Jakarta EE, régen Java EE) csomaghoz tartozó **specifikáció** (egyedülállóan is használható)
- Egy általános és teljes objektum-relációs leképezést határoz meg
  - POJO osztályok entitásokká (táblákká) válnak, míg ennek példányai a sorok
- Hasonlóan az slf4j-hez, csak API, nem tartalmaz implementációt, így megkönnyíti a különböző implementációk között átjárást.
- Implementációi általában a JDBC-re épülnek.
- A JPA meghatároz egy lekérdező nyelvet is (JPQL). A funkcionalitások azonosak az SQL nyelvek által biztosított funkcionalitásokkal, de Java objektumokkal dolgozhatunk, az objektumorientált szemléletmódnak megfelelően, illetve könnyen migrálhatunk különböző típusú adatbázisok között.

JPA specifikáció

- A *JPA* önmagában nem tartalmaz implementációt, csak egy szabvány (JSR-220) API részét.
- Csak az API-ban definiált annotációkat, interfészeket és konfigurációs állományokat ajánlott használni.
  - JPA API egyedülálló:
    `'jakarta.persistence:jakarta.persistence-api:3.0.0'`
  - A Jakarta EE specifikáció magában foglalja:
    `'jakarta.platform:jakarta.jakartaee-api:9.1.0'`
- Ha alkalmazásunkat kitelepítjük egy alkalmazásszerverre, nem szükséges sem az API-t, sem az implementációt becsomagolni (**`provided`** függőség).
- Webszerverek ellenben nem nyújtanak JPA implementációt, így futási időben kell biztosítanunk egyet:
  - EclipseLink (referencia implementáció)
    `'org.eclipse.persistence:eclipselink:3.0.2'`
  - Hibernate (elterjedtebb)
    `'org.hibernate.orm:hibernate-core:6.2.0.Final'`

Entitások

- *Entity 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)
- Perzisztenciával kapcsolatos adatbázis-műveletek központi szolgáltatása 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.

Entitások

- Egy POJO osztály entitássá válik a következő feltételekkel:
  - Az `@Entity` annotáció megjelenik az osztályon–jelzi, hogy az osztály példányai menedzselhető állapotba kerülhetnek.
  - Létezik az osztálynak egy `@Id`-val annotált adattagja vagy gettere, amely a háttérben levő DB-tábla elsődleges kulcsát jelképezi. Az adattag lehet primitív vagy összetett típusú is (feltéve hogy szerializálható, s helyesen fölül van írva az `equals` metódusa).
- Mivel egy alkalmazásból több adatbázishoz is csatlakozhatunk, a fentiek nem elegendőek egy mappelt tábla létrehozásához/karbantartásához, mivel nem lenne egyértelmű egy többkapcsolatos rendszerben, hogy melyik DB-ben szeretnénk a mappelt táblát.
- Hogy egy kapcsolathoz hozzá tartozzon az entitás, deklarálni kell a kapcsolatnak megfelelő **persistence unit**ban.

`persistence.xml` - a telepítésleíró

- Persistence unit (perzisztenciai egység) - egy adott adatbázisnak megfeleltetett (*mapping*) osztályok halmaza.
- *Telepítésleíró:* `persistence.xml` - egy vagy több perzisztenciai egység meghatározása
- *Elhelyezés:* a classpath `META-INF` könyvtárában, így egy Gradle projekten belül konvencionálisan `src/main/resources/META-INF/persistence.xml`
- tartalmaz `<persistence-unit>` elemeket, belső elemek (opcionálisak):
  - `<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 ez megfelel).
  - `<properties>` - a persistence provider beállításai.

Adatbázis beállítások

Ha lokális tranzakciókezelést használunk (`RESOURCE_LOCAL`), az adatbázishoz történő csatlakozás paramétereit a Persistence Provider beállításaiban adjuk meg (`persistence.xml` → `<properties>`).

- `javax.persistence.jdbc.driver` - JDBC Driver osztályneve (csak futási időben kell `CLASSPATH`-en legyen)
- `javax.persistence.jdbc.url` - JDBC formátumú URL; megadja az adatbázis elérési címét
- `javax.persistence.jdbc.user` - adatbázis felhasználónév (opcionális)
- `javax.persistence.jdbc.password` - adatbázis jelszó (opcionális)

Adatbázis beállítások

A Persistence Provider létrehozhatja/törölheti az adatbázisban a megfelelő táblákat a következő beállításokkal:

- `javax.persistence.schema-generation.database.action` - beállítja a táblákkal az induláskor történő akciót;
  - lehetséges értékek: `none` (alapértelmezett), `create`, `drop`, `drop-and-create`
- `javax.persistence.schema-generation.create-source` - ha táblakészítés kéretik, honnan veszi a provider a táblaelkészítő SQL szkriptet; lehetséges értékek:
  - `metadata` (alapértelmezett) - az annotációk és konfigurációk alapján generálja
  - `script` (megadjuk manuálisan az SQL szkriptet)
  - kombináció: `metadata-then-script`, `script-then-metadata`
- `javax.persistence.schema-generation.create-script-source` - ha táblakészítés kéretik `script` segítségével, hol találjuk a szkriptet
- `javax.persistence.schema-generation.drop-source`
- `javax.persistence.schema-generation.drop-script-source`
- `javax.persistence.sql-load-script-source` - ha szeretnénk adattal populálni az adatbázist induláskor, megadjuk a megfelelő SQL szkript helyét

EntityManager példány

- Lehetőségek: **`EntityManagerFactory`** alkalmazása, vagy injection
- Factory használata:

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("puName");
EntityManager em = emf.createEntityManager();
...
em.close(); // vagy try-with-resources nyitáskor
emf.close();
```

- Az **`EntityManager`** pár alapvetőbb CRUD metódusa:
  - **`T find(Class<T> entityClass, Object id)`**
    lekéri az adott osztály adott ID-jú példányát
  - **`void persist(Object entity);`**
    menedzseltté teszi az adott objektumot (beszúrja, ha még nem létezik)
  - **`T merge(T entity);`**
    frissíti az objektumot (update), s visszatéríti a menedzselt összefésült változatot
  - **`void remove(Object entity);`**
    törli az adott entitást

Java SE + JPA példa: `build.gradle`

CODESPRING
OUR CODE YOUR SUCCESS
20 years

```
plugins {
    id 'java'
    id 'application'
}

repositories {
    mavenCentral()
}

dependencies {
    implementation group: 'org.slf4j', name: 'slf4j-api', version: '1.7.25'
    runtimeOnly group: 'ch.qos.logback', name:'logback-classic', version:'1.2.3'

    // JDBC drivert tartalmazó függőség továbbra is szükséges
    runtimeOnly group: 'com.h2database', name: 'h2', version: '1.4.200'
    // JPA API szükséges kompiláláskor
    implementation group: 'jakarta.persistence', name: 'jakarta.persistence-api', version: '3.1.0'

    // JPA implementáció (Hibernate) futáskor szükséges
    runtimeOnly group: 'org.hibernate.orm', name: 'hibernate-core', version: '6.2.0.Final'
}

application {
    mainClass = 'edu.codespring.jpaexample.JpaHibernateDemo'
}
```

Java SE + JPA példa: `persistence.xml`

CODESPRING
OUR CODE YOUR SUCCESS
20 years

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             version="3.1"
             xmlns="https://jakarta.ee/xml/ns/persistence"
             xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence
                                 https://jakarta.ee/xml/ns/persistence/persistence_3_1.xsd">

    <persistence-unit name="blogPu" transaction-type="RESOURCE_LOCAL">
        <!-- entitások listája -->
        <class>edu.codespring.blog.model.BlogPost</class>

        <!-- beállítások -->
        <properties>
            <!-- JDBC adatbázis-elérhetőség beállítása -->
            <property name="jakarta.persistence.jdbc.driver" value="org.h2.Driver"/>
            <property name="jakarta.persistence.jdbc.url" value="jdbc:h2:mem:test;DB_CLOSE_DELAY=-1"
            <property name="jakarta.persistence.jdbc.user" value="sa"/>
            <property name="jakarta.persistence.jdbc.password" value=""/>

            <!-- séma elkészítésével kapcsolatos beállítás -->
            <property name="jakarta.persistence.schema-generation.database.action" value="create"/>

            <!-- Hibernate-specifikus: SQL queryk kiírása -->
            <property name="hibernate.show_sql" value="true"/>
        </properties>
    </persistence-unit>
</persistence>
```

Java SE + JPA példa: entitás osztály

CODESPRING
OUR CODE YOUR SUCCESS
20 years

```java
@Entity
@Table(name = "blogpost")
public class BlogPost extends BaseEntity {

    @Column(nullable = false)
    private String title;
    private String author;
    @Column(length = 1024)
    private String content;
    @Temporal(value = TemporalType.TIMESTAMP)
    private Date date;

    // konstruktorok, getterek/setterek, toString...
}
```

Java SE + JPA példa: main osztály


```
// entity manager factory készítése
// egy persistence unithoz kapcsolódik - ez a persistence.xml-ben van definiálva
EntityManagerFactory entityManagerFactory = Persistence.createEntityManagerFactory("blogPu");
// entity manager példány kérése a factorytól
EntityManager entityManager = entityManagerFactory.createEntityManager();

// tranzakció létesítése
entityManager.getTransaction().begin();

// új entitások létrehozása = nem manage-elt objektumok manage-eltté tétele
LOG.info("Making new entities managed, i.e. inserting");
BlogPost entity1 = new BlogPost("Title 1", "Author 1", "Content 1", new Date());
BlogPost entity2 = new BlogPost("Title 2", "Author 2", "Content 2", new Date());
LOG.info("Entity1 before persist: {}", entity1);
entityManager.persist(entity1);
entityManager.persist(entity2);
LOG.info("Entity1 after persist: {}", entity1);

// az alábbi művelet nem csinál semmit, mivel a beadott bean már manage-elt
entityManager.persist(entity1);
LOG.info("Entity1 after second persist: {}", entity1);

// tranzakció lezárása
entityManager.getTransaction().commit();
LOG.info("");
// ...
```

Java SE + JPA példa: main osztály

CODESPRING
OUR CODE YOUR SUCCESS
20 years

```
// ...
// lekérdezés ID szerint
LOG.info("Selecting by ID");
BlogPost retrievedEntity = entityManager.find(BlogPost.class, 2L);
LOG.info("Retrieved entity: {}", retrievedEntity);

// nem létező ID lekérése = null
// figyelem a logokra: itt van select, míg a fentinél nincs
BlogPost retrievedNonExistentEntity = entityManager.find(BlogPost.class, 42L);
LOG.info("Retrieved non-existent entity: {}", retrievedNonExistentEntity);
```

SQL parancsok kiírása

- a háttérben meghívott SQL parancsok kiírhatóak egy **property** beállításával, de ezek nem részei a JPA standardnak
  - EclipseLink:

```
<property name="hibernate.show_sql" value="true"/>
```

  - Hibernate:

```
<property name="eclipselink.logging.level.sql" value="FINE"/>
```

- példa kimenetek Hibernate esetén:

```
Hibernate: create table db_blogpost (id bigint generated by default as identity, author varchar(255)
    content varchar(1024), date timestamp, title varchar(255) not null, primary key (id))
Hibernate: insert into db_blogpost (id, author, content, date, title) values (null, ?, ?, ?, ?)
Hibernate: select blogpost0_.id as id1_0_0_, blogpost0_.author as author2_0_0_,
    blogpost0_.content as content3_0_0_, blogpost0_.date as date4_0_0_,
    blogpost0_.title as title5_0_0_ from db_blogpost blogpost0_ where blogpost0_.id=?
```

Tranzakciók

- Bármilyen művelet, amely módosítja az entitások állapotát (beszúrás, törlés, módosítás), aktív tranzakción belül kell hogy történjen.
- A tranzakciók kezeléséért felelős Java EE API a **Java Transactions API (JTA)**. Egy enterprise alkalmazásszerver nyújt egy JTA implementációt, amely automatikusan kezeli a tranzakciókat minden `@Transactional`-lel annotált DAO metódusban.
- JTA híjján (`RESOURCE_LOCAL`), az `EntityManager getTransaction` metódusának segítségével manuálisan kell karbantartsuk a tranzakciókat:

```
try {
    entityManager.getTransaction().begin();
    // JPA actions ...
    entityManager.getTransaction().commit();
} catch (AppropriateException e) {
    entityManager.getTransaction().rollback();
}
```

Entitások

- *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*).

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 entitást, a **`getReference`** kivételt dob
- A második paraméter a kulcs, típusa **`Object`** (az auto-boxing mechanizmus jóvoltából szerepelhet a példában **`int`**)
- A **`find`** biztosan betölti az adatbázisból az objektumot, ha korábban nem tette már meg
- A **`getReference`** csak akkor téríti vissza a beállított objektumot, ha korábban be volt töltve, ellenkező esetben egy proxy-t épít fel, amelynek csak az ID-ja van beállítva

EntityManager használata

- Beszúrás (`persist`):

```
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 (több információ később).
  - Lehetőség van automatikus kulcs (primary key) *generálására*.
  - A metódus **`IllegalArgumentException`**-t dobhat, ha a paraméter típusa nem megfelelő.

EntityManager használata

- Módosítás:

```
Cabin cabin = entityManager.find(Cabin.class, id);
cabin.setBedCount(noBeds);
// no more action required
```

  - 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 entitás állapotának módosítása után lehetőség van az egybeolvasztásra és visszacsatolásra (DB is frissül). 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

EntityManager használata

- `contains(Object entity)` - ellenőrzi, hogy a paraméterként kapott objektum jelenleg menedzselt állapotban van-e
- `clear()` -leválaszt minden entitást (a változtatások elvesztődhetnek, érdemes lehet `flush()`-t hívni előtte)
- `flush()` és `flushModeType`
  - A `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

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`, `updatable` (default `true`)
    - `length` (default `255`)
    - `precision` (default `0`)
    - `scale` (default `0`)
    - `columnDefinition` (oszlop típusát megadó DDL meghatározása)
    - `table`

ID automatikus generálása


```
@Id
@GeneratedValue(strategy=GenerationType.AUTO)
private Long id;
```

- A **@GeneratedValue** által használt inkrementálási *stratégia* a **GenerationType** enumból veszi értékeit:
  - **AUTO** - default, automatikus választás (provider függő)
  - **IDENTITY** - elsődleges kulcs generálása egy egyedi oszlopban
  - **TABLE** - egy a felhasználó által meghatározott speciális táblázat alkalmazása a kulcsgeneráláshoz
  - **SEQUENCE** - elsődleges kulcs generálása egy adatbázisszekvenciában
- Mind a **TABLE**, mind a **SEQUENCE** stratégiák esetében deklarálható a generátor a **@Table** annotáció után is (vagy az **@Id** előtt)
- A JPA nem támogatja natívan a **UUID** bármely variánsának megfelelő elsődleges kulcsok automatikus generálását, de a Hibernate támogatja az **AUTO** generátor-stratégiával, s az EclipseLink is nyújt implementáció-specifikus megoldást.

ID automatikus generálása

CODESPRING
OUR CODE YOUR SUCCESS
20 years

- 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`** attribútuma 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`**

```
@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;
}
```

ID automatikus generálása

- Adatbázisszekvenciák esetén a **`@SequenceGenerator`** annotációval deklaráljuk a szekvenciát
- Az RDBMS muszáj támogassa a lehetőséget
- 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)

```
@Id
@SequenceGenerator(name="CUSTOMER_SEQUENCE",
                   sequenceName="CUST_SEQ",
                   initialValue=5,
                   allocationSize=50)
@GeneratedValue(strategy=GenerationType.SEQUENCE,
                generator="CUSTOMER_SEQUENCE") // ...
public Long getId() {
    return id;
}
```

Kulcsosztályok és összetett kulcsok

CODESPRING
OUR CODE YOUR SUCCESS
20 years

```
public class CustomerPK implements 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.getLastName()))
            return false;
        if (ssn != pk.getSsn())
            return false;
        return true;
    }

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

Gyakorlatok

BiblioSpring példaprojekt (Springgel):
`https://git.edu.codespring.ro/training-assets/BiblioSpring`

1. Módosítsuk a Gradle projektünket: adjuk hozzá a `JPA API`, illetve `hibernate` függőségeket. Figyeljünk a hatókörökre: melyik függőségre mikor van szükségünk.
2. Használjunk JPA annotációkat az entitás-hierarchia kiépítéséhez: `@Entity`, `@Table`, `@Column`, `@MappedSuperclass` `@Id`, `@GeneratedValue`.
3. Készítsük el a JPA beállításunkat a `persistence.xml` megírásával.
   - Az állomány vázát megkapjuk a GitLab példaprogramoknál.
   - Konfiguráljunk csatlakozást a korábban beállított adatbázisunkhoz.
   - Ürítsük le az adatbázisunkat (vagy használjunk újat) - a JPA automatiksan létrehozza a szükséges táblázatokat.
4. Készítsünk egy új, harmadik adatelérési réteget, amely `EntityManager` segítségével végzi a műveleteket. Implementáljuk a `findAll`, `findById`, `create` metódusokat a központi `BlogPost` entitás esetén.
5. Módosítsuk megfelelően a factory-t és teszteljük az alkalmazást.
6. **Otthoni:** Készítsük el az `update` és `delete` DAO metódusokat is.
7. **Otthoni:** Készítsük el a JPA DAO-t a további entitásunkra/entitásainkra.

2. rész

JPQL

Query-k és JPQL

- *JPQL - Java Persistence Query Language*
- Az SQL-hez hasonló deklaratív lekérdező nyelv, amely Java objektumokkal dolgozik (nem a 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 driverhez. Ilyen módon az JPQL adatbázisrendszertől **független**.
- *Natív* query-k is futtathatóak, amikor el szeretnénk egy-egy rendszer-specifikus funkcionalitást. Rendszerfüggetlenségért próbáljuk kerülni, pl. használjunk tárolt eljárásokat amennyiben támogatottak.

`EntityManager` használata JPQL-re


```
package javax.persistence;
public interface EntityManager {
    // ...
    public Query createQuery(String jpqlString);
    public TypedQuery<T> createQuery(String jpqlString, Class<T> resultClass);
    public Query createNamedQuery(String name);
    public TypedQuery<T> createNamedQuery(String name, Class<T> resultClass);

    public Query createNativeQuery(String sqlString);
    public Query createNativeQuery(String sqlString, Class resultClass);
    public Query createNativeQuery(String sqlString, String resultSetMapping);
    // ...
}
```

- A **`TypedQuery`** örökli a **`Query`**-t, s garantálja, hogy azon osztály példányai térnek vissza, amelyet a query stringben megadtunk $\Rightarrow$ a cast biztonságos. Ezen biztonsági okból mindig használjunk **`TypedQuery`**-t nem natív lekérdezéseknél.

Query API

Releváns metódusok a `Query` interfészből:

- `getResultList()` és `getResultStream()` - lefuttatja a query-t, majd lista vagy stream formájában visszaadja az eredményeket
- `getSingleResult()` - lefuttatja a query-t, s egyetlen eredménysort feltételez, amelyet visszaad
  - `0` eredmény ⇒ `NoResultException`
  - `>1` eredmény ⇒ `NonUniqueResultException`
- `executeUpdate()` - egyenértékű a JDBC `Statement` ugyanezen metódusával - ha nem lekérdezünk, hanem módosítunk
- `setParameter(key, value)` - a `Query`-k alapból `PreparedStatement`-ként működnek a háttérben, így paraméterezhetőek. A paraméterek kulcsának használhatunk mind számot, mind `String` nevet.

JPQL Query-k: példa


```
LOG.info("Selecting all");
TypedQuery<BlogPost> query = entityManager.createQuery("from BlogPost", BlogPost.class);
List<BlogPost> blogPosts = query.getResultList();
LOG.info("Retrieved entities: {}", blogPosts);
```

- JPQL query: **`from BlogPost`**
- Ha minden oszlopot szeretnénk lekérni, a **`select *`** prefix mellőzhető, így biztosan teljes **`BlogPost`** példányok épülhetnek fel.
- Az **entitás nevét** (alapértelmezetten az osztály neve) használjuk, még akkor is, ha a mappelt táblázatnak különbözik a neve.
- A **`TypedQuery`** garantálja a helyes típus visszatérését.

JPQL oszlopok

- Ha csak egy oszlopot szeretnénk lekérdezni, megadhatjuk annak típusát a **`TypedQuery`** generikus osztály paraméterének.

```
LOG.info("Selecting IDs");
TypedQuery<Long> idQuery = entityManager.createQuery("select id from BlogPost", Long.class);
List<Long> ids = idQuery.getResultList();
LOG.info("Retrieved IDs: {}", ids);
```

- Hivatkozás adattagokra 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.

JPQL oszlopok

- Ha a lekérdezés több oszlopra vonatkozik, az eredmény egy **`Object[]`** tömb formájában tér vissza:

```
LOG.info("Selecting multiple columns");
TypedQuery<Object[]> idAndAuthorQuery = entityManager.createQuery(
    "select id, author from BlogPost", Object[].class);
List<Object[]> idsAndAuthors = idAndAuthorQuery.getResultList();
for (Object[] row : idsAndAuthors) {
    LOG.info("Retrieved: id={}, author={}", row[0], row[1]);
}
```

- A táblanevek alias-olhatóak, pl.:
  `select bp.id, bp.author from BlogPost bp`
- Ez megkönnyítheti a **`JOIN`** műveleteket és komplexebb query-ket
- Kapcsolatosztályokba bármilyen mélységig lehetséges hivatkozni, pl.:
  `select c.creditCard.creditCompany.address.city from Customer c`

JPQL query paraméterek

- A JDBC API **`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):

```
LOG.info("Selecting by author with named parameter");
query = entityManager.createQuery("from BlogPost where author=:author", BlogPost.class);
query.setParameter("author", "Author 2");
blogPosts = query.getResultList();
LOG.info("Retrieved entities: {}", blogPosts);
```

- Pozíció szerint:

```
LOG.info("Selecting by author with numbered parameter");
query = entityManager.createQuery("from BlogPost where author=?1", BlogPost.class);
query.setParameter(1, "Author 1");
blogPosts = query.getResultList();
LOG.info("Retrieved entities: {}", blogPosts);
```

Named query

- Ha egy query egy bizonyos entitáshoz tartozik (ilyen típusú elemeket térít vissza), s gyakran van újrahasznosítva, akkor minden használatkor újra van kompilálva, amely nem optimális.
- Erre egy lehetséges megoldás a `@NamedQuery`-k használata
  - Egy névhez társít egy JPQL query-t–ez a query `PreparedStatement`-re fordul az entitás első betöltésekor a persistence unitba.
  - A **modellosztály**t ruházzuk fel `@NamedQuery` entitásokkal.
  - Több ugyanazon entitáshoz tartozó named query esetén lehetséges több ilyen annotációt tenni az osztályra, de az olvashatóságért ajánlott egyetlen `@NamedQueries` annotációt használni, amely egy tömb `@NamedQuery` annotációt fogad paraméterként.
  - Refaktorálási hibaküszöbölésért ajánlott a named query-k neveit konstansként tárolni maga a modellosztályban.
- Az `EntityManager createNamedQuery` metódusával kérhetünk `Query` vagy `TypedQuery` példányt. Noha ezeket többször lekérjük, a háttérben csak egyszer kompilálja a JPA a query-t.

Named query példa

- definició a modellosztályban:

```
@Entity
@NamedQueries({
        @NamedQuery(name = BlogPost.FIND_ALL, query = "from BlogPost"),
        @NamedQuery(name = BlogPost.FIND_BY_AUTHOR, query = "from BlogPost where author=:author"),
})
public class BlogPost extends BaseEntity {

    // konstansok a named query-k nevével
    public final static String FIND_ALL = "BlogPost.findAll";
    public final static String FIND_BY_AUTHOR = "BlogPost.findByAuthor";

    // ...
```

- használat a DAO osztályban:

```
LOG.info("Selecting by author using named query");
query = entityManager.createNamedQuery(BlogPost.FIND_BY_AUTHOR, BlogPost.class);
query.setParameter("author", "Author 1");
blogPosts = query.getResultList();
LOG.info("Retrieved entities: {}", blogPosts);
```

JPQL natív query-k

- Néhány esetben natív query-k alkalmazása szükséges lehet (pl. ha el akarunk érni speciális *adatbázis-specifikus* funkcionalitásokat).
- Az JPQL használatukat támogatja, az **`EntityManager`** interface biztosítja erre a **`createNativeQuery`** metódus variánsait:

```
LOG.info("Selecting by author using native query");
Query nativeQuery = entityManager.createNativeQuery(
    "select * from BlogPost where author=:author", BlogPost.class);
query.setParameter("author", "Author 1");
blogPosts = query.getResultList();
LOG.info("Retrieved entities: {}", blogPosts);
```

- Az entitással való megfeleltetés a metaadatok alapján történik (a második, **`resultClass`** paraméter alapján), tehát a **`ResultSet`**en belül visszatérített oszlopoknak teljesen meg kell felelniük az entitás O/R mappingjének. Minden tulajdonságnak szerepelnie kell a lekérdezésben.
- Alternatív esetben annotációk segítségével szükséges mappeljük az SQL **`ResultSet`**-eket a megadott (entitás) típusra. További leírás

Query API Lapozás

- Megtörténhet, hogy egy lekérdezés nagyon sok objektumot térít vissza, és ez számunkra a feldolgozás szempontjából nem előnyös (pl. RESTful webszolgáltatás paginationnel, megjelenítés egy weboldalon egy listában, ahol az egyszerre megjeleníthető sorok száma korlátozott).
- Ennek érdekében meghatározhatjuk a visszatérített eredmények maximális számát, és az első indexét, így egyszerűen implementálható a *pagination*:

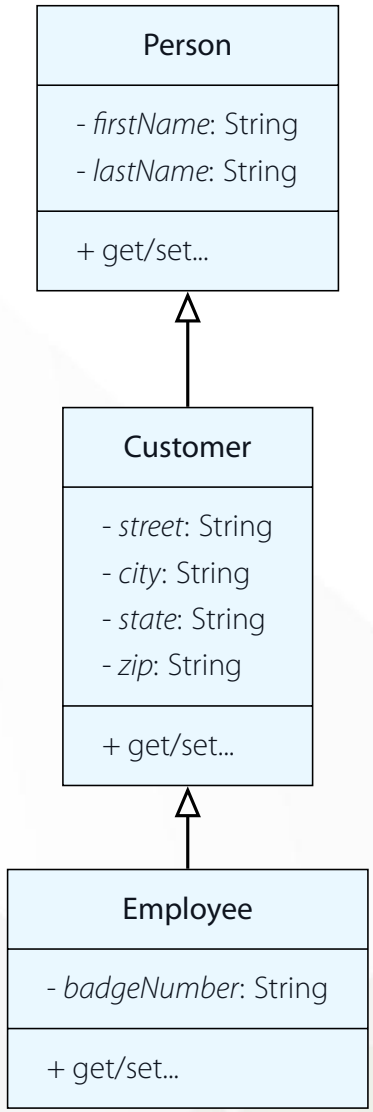
```
public List<Customer> getCustomers(int page, int perPage) {
    TypedQuery<Customer> query = entityManager.createQuery("from Customer", Customer.class);
    return query.setFirstResult(perPage * page).setMaxResults(perPage).getResultList();
}
```

3. rész

Öröklődés

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)



Öröklődés

- Ha egy szülő osztály absztrakt, vagy nem szeretnénk neki megfelelő táblát (*nonentity base class*), annotáljuk a `@MappedSuperclass`-szel.
  - `@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.
- A hierarchia leképezése a relációs adatbázisba három módon történhet:
  - Egy táblázat a osztályhierarchia számára (**single table per class hierarchy**): a hierarchiával kapcsolatos minden információt ebben a táblában tárolunk.
  - Egy táblázat minden osztálynak (**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 (**table per subclass**): 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


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

    @Id
    @GeneratedValue
    public Long getId() {
        return id;
    }
    // ...
}
```

```sql
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),
    badgeNumber varchar(255),
    DISCRIMINATOR varchar(31) not null
);
```

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**). Ennek megadhatjuk típusát, s értékét jelen osztály esetében.
- Az `@Inheritance` 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`).

Single Table per Class Hierarchy

- *Előnyök*:
  - Egyszerű implementálni, s a leghatékonyabb
  - 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.
- *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),
    badgeNumber varchar(255)
);
```

```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public class Person { ... }

@Entity
public class Customer extends Person { ... }

@Entity
public class Employee extends Customer { ... }
```

Table per Concrete Class

- 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.
- *Előnyök*:
  - Az oszlopok esetében használhatóak megszorítások (pl. `NOT NULL`).
  - Minden sor pontosan annyi információt tartalmaz, amennyi adattagot a leképezett objektum (nincsenek örökösen `null` oszlopok).
- *Hátrányok*:
  - Sok a redundáns oszlop: az alaposztály minden tulajdonsága újra megjelenik a származtatott osztályoknak megfelelő táblázatokban. A stratégia nem normalizált.
  - Kevésbé hatékony: A szolgáltatónak több komplex háttérműveletet el kell végeznie, például polimorfizmus alkalmazásának esetében több queryt kell használnia a hierarchia bejárásához.

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,
    badgeNumber varchar(255)
);
```

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class Person { ... }

@Entity
public class Customer extends Person { ... }

@Entity
public class Employee extends Customer { ... }
```

Table per Subclass

- Minden osztály kap táblázatot, de **csak** az illető osztály sajátos 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.
- *Előnyök*:
  - Normalizált.
  - Alkalmazhatóak megszorítások.
  - Hatékonyabb a `TABLE_PER_CLASS` stratégiánál, amennyiben az SQL `UNION` műveleteket nem támogatja az adatbázis-kezelő rendszer.
- *Hátrányok*:
  - `UNION`-t nem támogató rendszereknél nem hatékony.
  - Az elsődleges kulcsok minden táblában megjelennek.
  - Legkevésbé intuitív és átlátható - egy objektumhoz tartozó információ valójában több sor kombinációja.

4. rész

Entitások közötti kapcsolatok

JPA kapcsolatok

- `@Entity`-vel jelölt entitások közötti kapcsolatok megjelölhetőek annotációval, amely a kapcsolat számosságát jelképezi:
  - egy az egyhez:

```
public class Customer {
    // ...
    @OneToOne
    private Address address;
```

  - egy a többhöz:

```
public class BlogPost {
    // ...
    @OneToMany
    private List<Comment> comments;
```

  - több az egyhez:

```
public class Comment {
    // ...
    @ManyToOne
    private BlogPost blogPost;
```

  - több a többhöz:

```
public class Blog {
    // ...
    @ManyToMany
```

JPA kapcsolatok: irány

- Kapcsolatok lehetnek egyirányúak (*unidirectional*) vagy kétirányúak (*bidirectional*), aszerint hogy egy kapcsolat mindkét fél entitás osztályban megjelenik-e (pl. az előbbi esetben a **`Comment`** és **`BlogPost`** bidirekcionális kapcsolatban vannak).
- A kapcsolat iránya nem befolyásolja a háttérben létrejött adatbázis-sémát.
- Kétirányú helyzetekben szükséges beállítani egy-egy kapcsolatot mindkét irányból:

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

Gyűjtemény alapú megfeleltetések

- Gyűjtemények esetén (`@OneToMany` és `@ManyToMany`) a következő adattípusok alkalmazhatóak:
  - `Collection`
  - `Set`
  - `List` - lehetőséget ad egy vagy több attribútum szerinti rendezésre az `@OrderBy` annotáció segítségével

```
@Entity
public class Reservation implements Serializable {
    // ...
    @ManyToMany
    @OrderBy("lastName ASC")
    private List<Customer> customers = new ArrayList<Customer>();
```

  - `Map` - a kulcs egy megadott tulajdonság (alapértelmezetten az elsődleges kulcs, `@MapKey`-vel fölülírható), az érték maga az entitás

```
@Entity
public class Customer implements Serializable {
    // ...
    @OneToMany
    @MapKey(name="number")
    private Map<String, Phone> phoneNumbers = new HashMap<String, Phone>();
}
```

FetchType


```
public class BlogPost {
    // ...
    @OneToMany(fetchType=FetchType.EAGER)
    private List<Comment> comments;
```

- Minden kapcsolattípus esetében az annotáció rendelkezik egy `fetch` attribútummal, amely meghatározza, hogy a lekérdezésnél be legyen-e töltve a kapcsolatban álló entitás is.
  - `FetchType.LAZY`
    - A kapcsolatban álló entitás nem lesz betöltve mindaddig, amíg nem próbálunk hozzáférni.
    - Alapértelmezett minden kollekció típusnál (`@OneToMany` és `@ManyToMany`).

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

  - `FetchType.EAGER`
    - A kapcsolatban álló entitás együtt lesz betöltve a tulajdonos entitással.
    - Alapértelmezett minden atomikus kapcsolatnál (`@OneToOne` és `@ManyToOne`).

Cascading


```
public class BlogPost {
    // ...
    @OneToMany(cascade=CascadeType.ALL)
    private List<Comment> comments;

    @ManyToOne(cascade={ CascadeType.PERSIST, CascadeType.MERGE })
    private User author;
```

- *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.
- A beállítás a kapcsolatot jelző annotációk **`cascade`** attribútumának segítségével történik.
- A **`cascade`** attribútum egy **`CascadeType`** enum érték, vagy ennek egy tömbje. Itt listázzuk a végrehajtandó cascade műveleteket. A lehetséges értékek: **`ALL`**, **`PERSIST`**, **`MERGE`**, **`REMOVE`**, **`REFRESH`**.
- **Vigyázat** a használatával: 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.

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:
- `PERSIST`: a kapcsolódó entitás mentése automatikus. Pl. nem kell külön mentenünk a lakcímet, a `Customer` mentésekor ezt a Provider automatikusan elvégzi. 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.
- `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.

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 indít 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 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 `DELETE` végrehajtása után)

Entity osztályok és visszahívás

- Callback metódusok beállítása az annotációk segítségével lehetséges.
- A metódus lehet bármilyen hatáskörű.
- A visszatérített érték típusának **`void`**nak kell lennie, és nem lehetnek argumentumai.
- A metódus nem dobhat ellenőrzött kivételt.
- Egy adott esemény felléptekor az **`EntityManager`** meghívja ezeket a metódusokat.

```
@Entity
public class Cabin {
    // ...
    @PostPersist
    void afterInsert() {
    // ...
}
```

```
<entity class="edu.codespring.example.model.Cabin">
    <post-persist name="afterInsert"/>
    <post-load name="afterLoading"/>
</entity>
```

Figyelők

- Az *entity listenerek* annotációk (vagy XML leíró) segítségével hozzárendelhetőek entitásokhoz. A figyelőkön belül metódusokat hozhatunk létre egyes események kezelésére.
- Hasonlóan a callback metódusokhoz, a metódusok visszatérített értékének típusa **`void`**, nem lehetnek argumentumai és nem dobhat ellenőrzött kivételt.
- *Különbség:* egy **`Object`** típusú paraméterük van, amely az entitás példányra mutat. A metódusokat a callback mechanizmusnál leírt annotációkkal látjuk el.
- Az entitáshoz a figyelőt az **`@EntityListeners`** annotációval rendelhetjük hozzá.

```
public class TitanAuditLogger {
    @PostPersist
    void postInsert(Object entity) { ... }
    @PostLoad
    void postLoad(Object entity) { ... }
}

// ...
@Entity
@EntityListeners({ TitanAuditLogger.class,
                   OtherListenerClass.class })
public class Cabin { ... }
```

```
<entity class="edu.codespring.example.model.Cabin">
    <entity-listeners>
        <entity-listener class="...TitanAuditLogger">
        </entity-listener>
        <entity-listener class="...OtherListenerClass">
            <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 `<exclude-superclass-listeners />` elemmel.

Gyakorlatok

1. Készítsük el (ha még nem létezik) a `User` entitást, s helyezzük megfelelő kapcsolatba a `BlogPost` entitással. Használjunk unidirekcionális many-to-one kapcsolatot a `BlogPost` oldaláról.
2. Módosítsuk a megjelenítési rétegünket, hogy rajzolja ki a felhasználó információit is egy-egy `BlogPost` megjelenítésénél.
3. **Otthoni:** Egészítsük ki a rendszert még egy entitással, amely kapcsolatban áll egyikkel az eddigiek közül.
4. **Otthoni:** Módosítsuk a megjelenítési réteget, hogy kirajzolhassunk minden entitást.