Szakdolgozat

Kovács Lilla

Debrecen
2009

Debreceni Egyetem
Informatikai Kar

Szoftverfejlesztés Java környezetben

Témavezető:
Espák Miklós
Egyetemi tanársegéd

Készítette:
Kovács Lilla
Programtervező informatikus BSc

Debrecen
2009

Tartalomjegyzék

1. Bevezetés

A szoftverfejlesztés az informatikának azon területe, mely az utóbbi tíz évben rohamos fejlődésnek indult. Új módszerek, eszközök, irányzatok jelentek meg, ilyen például a komponens-technológia - melyben az újrafelhasználható komponensek speciális funkciókat írnak le, vagy hajtanak végre -, az aspektusorientált technológia, az egymással folyamatosan versengő Java és .NET technológiák, az agilis módszertanok.
A Java nyelv a hozzá kapcsolódó technológiákkal folyamatosan fejlődik. Az évek során kifejlesztettek néhány keretrendszert, melyek segítik az objektum-relációs leképezések és a perzisztencia kezelését. A dolgozatomban néhány keretrendszert, technológiát szeretnék bemutatni, többek között a Hibernate, Java Persistence API, EJBs, Object Relational Mapping, Java Data Object, TopLink használatát (elemek, kapcsolatok, lehetőségek stb.)

1.1. Témaválasztás indoklása

Szakdolgozatom elsődleges célja a tapasztalatszerzés és a tanulás volt. Tanulmányaim alatt inkább a Java alapjainak elsajátítása volt a cél, ezért döntöttem amellett, hogy egy számomra új dologgal ismerkedjek meg közelebbről. Bizonyos előismerettel rendelkezem a JSP terén, így a dolgozat mellett egy Java alapú webes alkalmazást készítettem, mely a Hibernate alapjainak elsajátításában nyújtott segítséget.
A dolgozatban igyekszem példákon keresztül szemléltetni a technológiák, keretrendszerek elemeinek működését, a könnyebb megértés érdekében.

2. A perzisztencia fogalma

A *perzisztencia* tulajdonképpen az objektumhierarchia egy részének adatbázisban való tárolását, illetve onnan visszatöltését jelenti. Megvalósítása meglehetősen munkaigényes. Ahhoz, hogy tudjuk, hogy melyik objektum változott meg, figyelemmel kell kísérni az objektumok mezőit. Szinte minden alkalmazásfejlesztés alapjául szolgál.

A *perzisztens adat* egy olyan információ, mely képes túlélni az őt létrehozó programot. A bonyolultabb programok nagy része használ perzisztens adatokat.

Könnyűsúlyú perzisztencia (lightweight persistence): perzisztens adatok tárolását, az adattárból való visszanyerést szolgálja, ezzel megkönnyítve a programozó munkáját. A szerializáció is egyfajta könnyűsúlyú perzisztencia, ahol a Java objektumok fájlba történő perzisztálása könnyedén megoldható.

2.1. Relációs adatbázisok

A *relációs adatbázisok* régóta működő, robosztus rendszerek, hatékonyan kezelik a perzisztens adatokat. Az adatok nagyon hosszú életűek, sokkal tovább maradnak „életben”, mint más alkalmazások. A relációs adatbázis kezelő rendszerek SQL-alapúak, ezért gyakran nevezzük őket SQL adatbázis kezelő rendszereknek. Az eszköz, mely a programozót segíti az adatbázisok által nyújtott szolgáltatások használatát, az adatbázis perzisztencia eszköz, más néven JDBC. Gyors elérést tesz lehetővé. A Java nyelv JDBC API-n keresztül támogatja az adatbázis-kezelést. A JDBC-t használó programokban módosításokat az SQL-nyelv eszközeivel hajthatunk végre (SELECT, INSERT stb.). Az adatbázishoz való kapcsolódás a Java nyelvből könnyedén megoldható, de a programozó feladata az, hogy a JDBC adatait objektumokká alakítsa át, viszont ez néhány problémához vezethet, például, ha egy mező neve megváltozik, akkor az ahhoz tartozó attribútumot a Java kódban is meg kell változtatni. A célunk olyan kódot készíteni, mely elmenti és kinyeri az objektumokat az adatbázisba/adatbázisból. Tehát azt mondhatjuk, hogy a relációs adatbázis és az SQL a perzisztencia kezelésére a legjobb megoldás.

2.2. ORM (Object Relational Mapping)

Az *ORM* az objektumok relációs adatbázisban való tárolása szolgál, mely metaadatokat használ az objektumok és az adatbázis közötti kapcsolat leírására – egyfajta adatkonverzió az objektumorientált nyelv és a relációs adatbázis között.

Metaadat: adat az adatról, azaz az adat használatára vonatkozó összes információ, tény együttese.

Az ilyen rendszerektől mit várunk el?

- egy API-t az objektumok CRUD műveleteire (`create / read / update / delete`)
- lekérdező nyelv / API az osztályokra és azok adattagjaira vonatkozóan
- egy keretrendszer a leképezések definiálására
- laza kapcsolatok felderítése, piszkos adatok ellenőrzése, egyéb optimalizáló funkció megvalósítása

Előnye, hogy legtöbbször csökkenti a programkód sorainak számát, ezért a rendszer sokkal hibatűrőbb, robosztusabb lesz – mivel kevesebb a sorok száma, a hibázási lehetőség is kisebb.

Az ORM fajtái:

- Pure relational (Tiszta reláció): az egész alkalmazás a relációs modell és az SQL műveletek szerint készül el. Kis alkalmazásoknál kiváló megoldás.
  Hátránya: nem hordozható, és nehezen karbantartható.
- Light object mapping (Könnyűsúlyú objektum leképezés): a tervezési minták elrejtik az SQL kódot. Az osztályok leképezése az adatbázisba manuálisan történik. Ideális kevés számú entitás esetén, vagy olyan alkalmazásoknál, melyek generikus nyelvi eszközökkel dolgoznak.
- Medium object mapping (Középsúlyú objektum leképezés): az objektum-modell köré tervezték. Az SQL kód előállítása fordítási vagy futási időben történik. A lekérdezést el lehet végezni valamilyen OO nyelven. Általában közepes méretű alkalmazásoknál használják, és fontos a hordozhatóság.
- Full object mapping (Teljes objektum leképezés): bonyolult objektummodellezés, beleértve a kompozíciót – egész-rész viszony, ahol a rész élettartama az egésztől függ, vagyis a rész soha nem élheti túl az egészet -, a polimorfizmust – alosztályban egy

átvett metódus implementációja megváltozik, de a specifikáció nem-, az öröklődést és a perzisztenciát. Egy perzisztens osztály nem öröklődik, és nem implementál semmilyen interfészt.

3. Java Data Objects (JDO)

A Java Data Objects strukturált Java objektumok adatbázisban való tárolását szolgálja. A JDO specifikáció egy magas szintű API-t használ az objektumperzisztencia és a relációs adatbázis-kezelés megvalósítására.

Tartalmaz:

- közvetlen, egyszerű I/O fájlkezelést
- JDBC-t: az összes funkcionalitásával. Itt a fejlesztő dolga a mezőkben lévő adatok tárolása, kezelése, ezért neki ismernie kell valamilyen lekérdező nyelvet (Például SQL-t).
- Enterprise Java Beaneket
- Bean-Managed Persistence entity beaneket (BMP): EJB rendszerben az entity beanek tartalmának hosszútávú tárolását a beanek maguk végzik. A beanek függetlenségre törekednek, és nincsenek közvetlen kapcsolatban más BMP beanekkel. BMP használata megköveteli az entitások létrehozására, törlésére, tárolására, keresésére vonatkozó metódusok megírását.
- Container-Managed Persistence entity beaneket (CMP): a beanek tartalmának hosszútávú rögzítését az őket befogadó konténer végzi. A CMP megengedi több beannek, hogy valamiféle kapcsolat legyen közöttük. Két modellel rendelkezik: lokális és távoli, ennek megfelelően lokális és távoli interfészeket használnak. Egy CMP modell közelebb áll egy Java modellhez, legfőképpen az enterprise beanek közötti paraméterátadásokra vonatkozóan.
- Java Persistence API-t

A BMP jobb teljesítményt, nagyobb flexibilitást – elérés bármilyen adatforrásból, például adatbázis, szövegfájlok stb. - és alkalmazásszerver függetlenséget nyújt, a CMP viszont könnyebb karbantarthatóságot és kényelmet biztosít.

A JDO előnyei:

- Kényelmes használat: a programozónak csak a domain modellre kell figyelnie, a perzisztencia-kezelés a JDO feladata.
- Hordozhatóság

- Adatbázis-függetlenség: különféle adattárolási módot megenged, beleértve a relációs adatbázist, XML vagy más, egyszerű szöveges fájlokat.
- Nagy teljesítmény: a JDO implementáció optimalizálja az adatelérést, így érhető el jobb teljesítmény.
- EJB kiegészítés: az EJB sajátosságai közül néhányat használ, ilyen például a távoli metódushívás, tranzakció-kezelés, biztonság - ugyanazt a domain modellt használva az egész alkalmazásra.

3.1. JDO vagy JDBC?

Igazából a JDO nem arra törekszik, hogy a JDBC helyébe lépjen. Elképzelhetjük úgy is, mintha a JDO egy réteg lenne a JBDC felett.
Mindkettőnek megvannak az előnyei és a hátrányai. A JDO elrejti a fejlesztő elől az `sql` utasításokat, így ez kényelmesebbé teszi a programozást, nincs szükség a nyelv mélyebb ismeretére. Ha nem szeretnénk lekérdezéseket írni, a lekérdezés eredményeképpen kapott eredményhalmazokat Java objektumokká alakítani, akkor a JDO a legjobb választás.
Viszont a JDBC a programozónak rugalmasságot biztosít, azáltal, hogy közvetlen hozzáférési/vezérlési lehetősége van az adatbázishoz, éppen ezért gyorsabb a működése, mint a JDO-nak.
A JDO lekérdező nyelve a JDOQL. A JDOQL inkább Java, mint SQL alapokon áll. Működése a következőképpen történik: a JDO kap egy sztringet JDOQL formában, és ezt a megfelelő SQL utasítássá konvertálja, majd ezen SQL utasítás által manipulálható(ak) az adatbázis megfelelő eleme(i).

3.2. JDO osztályok

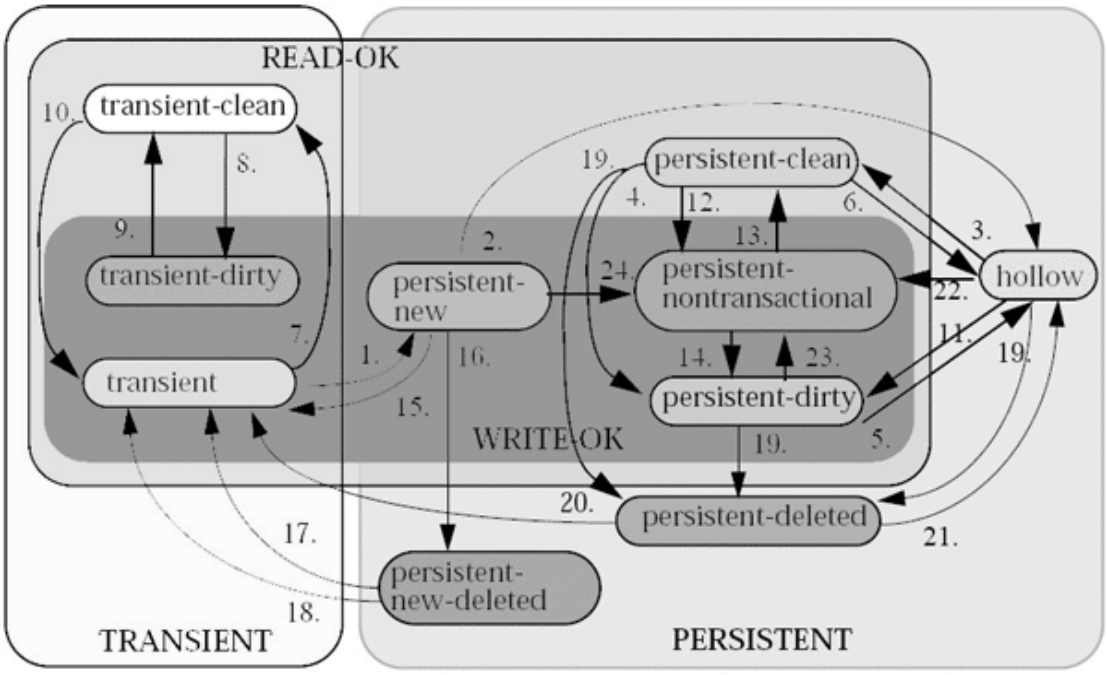
Háromféle osztályt különböztetünk meg:
Persistence-capable (perzisztenciára alkalmas): azon osztályok, melynek a példányait perzisztens módon tudjuk elmenteni valamilyen adattárolóba.

Persistence-aware (perzisztencia-tudatos): ezek az osztályok mapipulálják a persistence-capable osztályokat. A JDOHelper osztály szolgáltat olyan metódusokat, melyek egy persistence-capable osztály példányainak a perzisztens állapotát lekérdezi.
Normal (normál): nem perzisztens osztályok, sőt nincs is semmiféle fogalmuk a perzisztenciáról.

3.3. JDO példányok életciklusa

A JDO különböző objektum-állapotokat tart nyilván attól a perctől kezdve, hogy a példány létrejön, és egészen addig a percig él, amíg nem töröltük azt. Végül a Java Virtual Machine elvégzi a szemétgyűjtögetést. Az állapotátmenetekhez szükséges metódusokat a `PersistenceManager` osztály szolgáltatja – `makePersistent (Object object)`, `makeTransientAll (Collection c)`, `deletePersistent (Object object)`.

Az állapotváltozásokat a következő ábra mutatja:



A JDO specifikáció 10 állapotot definiál:

- Transient: a fejlesztő által írt konstruktorhívással létrehozott objektumok, melyek még nem perzisztensek. Még nincsen objektum azonosságuk
- Persistent-new: egy tranziens objektum a `makePersistent()` metódus hatására perzisztenssé válik. Ekkor más az objektumok rendelkeznek identitással.
- Persistent-dirty: minden olyan perzisztens objektum, mely megváltozott a jelenlegi tranzakció során.
- Hollow: egy speciális objektumállapot, azokat az objektumokat reprezentálja, melyek már benne vannak az adattárban, de még nincs értékük. Segít megbizonyosodni az objektum egyediségéről a tranzakciók között.
- Persistent-clean: a példány adatai a tárolóban vannak, és az aktuális tranzakció alatt nem változott meg az értéke.
- Persistent-deleted: a `deletePersistent()` metódushívás során előálló állapot. Egy persistent-deleted példány tranzienssé válik.
- Persistent-new-deleted: azok az objektumok, melyek egy tranzakción belül kerültek persistent-new állapotba és töröltük.
- Persistent-nontransactional: azon perzisztens objektumok, melyek értékeit már betöltöttük, de nem konzisztensek.
- Transient-clean: csak tranziens példány kerülhet ebbe az állapotba, akkor, ha a példány értéke nem változott meg a tranzakción belül
- Transient-dirty: ha egy transient-clean állapotban lévő példány értéke megváltozott, akkor kerül ebbe az állapotba.

Az első hét állapot kötelező módon jelen van, a másik három viszont opcionális, csak bizonyos állapotokból érhető el, amint azt a fenti ábra is mutatja.
Perzisztens osztályok megkövetelik a következőket: a mezők elérhetőek legyenek a JDO osztályok számára (`public`); egyes osztályok példánya nem támogatott (Például a `Thread, File, Socket` nem szerializálható).

A JDO kétféle interfészt definiál:

- JDO API (`javax.jdo` csomagban)
- JDO SPI (service provider interfész a `javax.jdo.spi` csomagban)

Egy alkalmazás két fontosabb interfészt használ:

- `PersistenceManagerFactory`: azt az elérési pontot reprezentálja a fejlesztő számára, amellyel a `PersistenceManager` osztály példányai elérhetőek – amiket a később módosítani, szerializálni lehet.
- `PersistenceManager`: a JDO alkalmazások komponensei számára ez az elsődleges interfész. Metódusokat szolgáltat az objektumoknak a perzisztencia kezelésére – úgymint perzisztens objektumokhoz való hozzáférés, törlés az adattárból; egy objektum módosítása; és ezzel együtt automatikus objektumállapot változás.

`PersistenceManagerFactory` és `PersistenceManager` elérése:

```
Properties props = new Properties(); //1
props.put(...); //2
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory(props); //3
PersistenceManager pm=pmf.getPersistenceManager(); //4
```

1. A Properties osztály a tulajdonságok perzisztens halmazát (kulcs-érték párok) reprezentálja.
2. A JDO és az adattár néhány sajátosságait adhatjuk meg a `put (…)` metódusban. Nézzünk meg ezek közül néhányat:
   (”`javax.jdo.option.ConnectionDriverName`”,”`com.mysql.jdbc.driver`”);
   //driver név
   (”`javax.jdo.option.ConnectionURL`”, ”`jdbc:mysql://localhost/jpox`”); //URL
   (”`javax.jdo.option.ConnectionUserName`”, ”`username`”); //kapcsolatnév
   (”`javax.jdo.option.ConnectionPassword`”, ”`password`”); //jelszó
3. A `PersistenceManagerFactory` létrehozása
4. A ’`pmf`’ factory `PersistenceManager` példánya is létrejön.

Ahhoz, hogy egy JDO alkalmazást futtatni tudjunk, a következő lépéseket kell végrehajtani:

1. Osztályok létrehozása, alapértelmezett, privát konstruktorral.

2. Metaadatok megírása, és azoknak a mezőknek és osztályoknak a kiválogatása, melyeket perzisztálni kell. Azokról a csomagokról is szolgáltat információt, melyekben perzisztens osztályok vannak.

   Egy metaadat fájl egy osztályhoz tartozik, és a neve megegyezik az osztály nevével, csak a kiterjesztése `.jdo`. Ezeket a fájlokat ugyanoda kell elhelyezni, mint ahol a `class` fájlok vannak.

   Példa egy egyszerű metaadat fájlra:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
  <package name="futok">
    <class name="Futo" identity-type="application">
    </class>
  </package>
</jdo>
```

   Az osztályokon belül megadhatjuk a mezők neveit és a kollekciókat, egy XML leírón belül pedig több osztályt is.

3. Az osztályok lefordítása a JDO Enhancer segítségével: bájtkód módosításokat végez a Java osztályokon, hogy persistent-capable osztályokká alakítsa. Ez alatt az idő alatt a `javax.jdo.PersistenceCapable` interfész hozzáadódik az osztályhoz. Tehát ha használni szeretnénk ennek az interfésznek a metódusait, nekünk nem kell implementálni a `PersistenceCapable`-t, hanem a fordítás után a JDO Enhancer-t kell futtatnunk, hogy elkészítse a bájtkód manipulációkat.

   Összességében csak a lefordított Java osztályokról és a JDO metaadat fájlról kell gondoskodnunk e lépés során.

4. Adatbázistáblák létrehozása; indexek, külső kulcsok definiálása. Vannak olyan JDO implementációk, melyek tartalmaznak séma-eszközöket: mindezeket automatikusan generálja a JDO leíróból.
5. Az alkalmazás futtatása.

4. Perzisztencia keretrendszerek

A *perzisztencia keretrendszereket* azon célból fejlesztették ki, hogy segítse a programozót az objektum relációs leképezések és az adat perzisztencia használatában. Nem mindig egyszerű azt eldöntenünk, hogy a feladatunk elkészítése során melyik megoldás lenne a legegyszerűbb, legmegfelelőbb. A továbbiakban bemutatásra kerülnek az egyes keretrendszerek, hogy mikor, melyiket érdemes használni, mik az előnyei, esetleg hátrányai.
Néhány ismertebb keretrendszer, amikről szó lesz a következőkben:

- Hibernate
- Entity Enterprise Java Beans
- Java Persistence API
- TopLink

4.1. Entity Enterprise Java Beans (EJBs)

Java kódok írása során az egyik legfontosabb tulajdonság a platformfüggetlenség. Az EJB-k alkalmazása mellett is teljesül a platformfüggetlenség, sőt implementáció-függetlenek is, ami azt jelenti, hogy bármely alkalmazásszerveren képesek futni.
Az EJB egy olyan komponens, mely a szerver oldalon jelenik meg, egy konténerben fut, szolgáltatásokat biztosít a bean számára (tranzakciókezelés, perzisztencia, biztonsági funkciók). A kliens az EJB-vel a konténeren keresztül kommunikál. Az EJB technológia ellát minket az üzleti logikához szükséges komponensekkel.
EJB konténer: az a környezet, mely közvetítő szerepet játszik a bean osztály és az EJB szerver között.
Az *üzleti logika*: a háromrétegű adatkezelő alkalmazások középső rétege, melynek szerepe az alkalmazás működési szabályainak, logikájának leírása. Adatokat mozgat és dolgoz fel a másik két réteg között. Az adatok tárolása, megjelenítése nem az ő feladata.
Egy beanhez kötelező módon tartozik egy interfész: home (saját) vagy remote (távoli). Mindkét interfész elérhető távoli kliensekről.

Home interface: a bean életciklusra vonatkozó metódusait tartalmazza, például új bean létrehozása, bean törlése stb.
Remote interface: az enterprise beanek azon üzleti metódusait definiálja, melyekhez a kliens hozzáfér. A `java.rmi` csomagban helyezkednek el.

Az EJB architektúra 3 féle beant különböztet meg:

- entity bean: a való világot modellezi. Az adatbázisban perzisztens adatokként jelennek meg. Az entity beant egyszerre több kliens is használhatja, van egyedi azonosítója. Ezen azonosító alapján az osztály egyedeit az EJB konténer egyértelműen meg tudja határozni.

  Az entity beaneket a Java Persistence API váltotta fel.
- session bean: az alkalmazást a kliens session bean metódusok hívásával érheti el. Egyszerre csak egy klienst szolgál ki. Ha a kliens megszűnik, vagy a szerver leáll, akkor a bean is megszűnik.

  A session beaneknek két fajtáját különböztetjük meg: stateless és stateful.
  - stateless (állapot nélküli): a kliens metódushívásai között nem őriz meg semmilyen állapotot. A metódus híváskor a bean minden példánya egyenértékű, ezért az EJB konténer kiválaszthatja a példányokat a kliensek számára. Ez az egyetlen olyan bean, mely webszolgáltatásokat implementálhat.
  - stateful (állapottal rendelkező): a kliens két metódushívása között megőrzi a példányváltozók állapotát.
- message-driven (eseményvezérelt) bean: JMS (Java Messaging Service) üzenetek kezelésére tervezték. Az eseményvezérelt bean a kliens üzenetek hatására működésbe lép. Amikor egy JMS üzenet célba ér, akkor az EJB konténer meghív egy message-driven beant, és átadja neki az üzenetet. Nincs semmilyen interfésze, mert a kliensek nem hívhatják közvetlenül a beaneket – ebben különbözik a másik két beantől. Csak bean osztálya létezik.

Az EJB előnye, hogy skálázható (további számítógépek hozzákapcsolása tulajdonképpen bármilyen mennyiségben), és hibatűrő. Viszont a használata nagyon sok tudást, és erőforrást igényel.

4.1.1. Java Naming and Directory Interface (JNDI)

A JNDI távoli objektumok elérésére szolgál, függetlenül attól, hogy milyen nyelven íródtak. Minden EJB konténernek része egy JNDI katalógus. A fájlrendszerek könyvtárszerkezetére hasonlít a JNDI adatstruktúrája.
Egy katalógus eléréséhez 3 dolog ismeretére van szükség:

- sémára, mely alatt a katalógus tárolja
- kontextusok listája (egy kontextus a könyvtárnévvel analóg)
- név: ami alatt a katalógus tárolja (egyenértékű a fájlnévvel)

Egy katalóguselemet a következő módon érhetünk el:

```
Context context = new InitialContext();
Object o = context.lookup(”java:comp/env/Bean”);
```

ahol a `java:comp/env/Bean` egy JNDI név. A `Context` osztály a `java.naming.*` csomagban helyezkedik el, és a JNDI egyik legfontosabb interfésze. A katalóguselemeknek típusa is lehet, sőt általában van is. A `lookup` metódus eredményét megfelelő típussá kell konvertálni, mivel az csak egy objektumreferenciát ad vissza.

4.2. TopLink

A TopLink egy olyan objektum-relációs perzisztens Java technológia, mely az objektumorientált rendszerek és a relációs adatbázisok közötti kapcsolatot teremti meg. Az Oracle Fusion Middleware alkalmazásfejlesztő eszközkészletének egy részét képezi. Gyors adatbázis elérést/kezelést tesz lehetővé.
A TopLink akkor ideális választás, ha a rendszerünk már tartalmaz valamilyen más, alapvető Oracle terméket.
Egyéb fontosabb tulajdonsága:

- Lekérdező nyelvekben gazdag: Query by Example, SQL, EJB QL.
- Gyorsítótárazás objektumazonosításra.

A TopLink egy szabadalmaztatott termék, ezért a fejlődése az Oracle fejlődésétől függ.

4.3. Java Persistence API

A JPA a POJO technológián alapul: öröklés, polimorfizmus, szabványos objektum/relációs leképezés, SQL nyelvű lekérdezések támogatása.
Ötletet merített a Hibernate, a TopLink, a JDO és az EJB keretrendszerekből. Így született meg a Java Persistence API, ami segíti a fejlesztőt a relációs adatbázis kezelését Java EE vagy Java SE alkalmazásokban.

3 lényeges részből áll:

- Entitások
- Persistence Unit (Perzisztencia Egység)
- Persistence Context (Perzisztencia Kontextus)

4.3.1. Entitások

Egyszerű Java osztályok, nincs olyan interfész, melyet kötelező módon implementálni kell. Annotációk segítik a programozót, és a lekérdezés perzisztencia-lekérdező nyelvvel történik. Az entitások egy API használatával érhetők el futási időben (Entitás Manager API).
Tulajdonságai:

- van saját vagy örökölt elsődleges kulcsuk
- lehet absztrakt is
- szerializálható

Alapvetően kétféle perzisztens osztályt különböztetünk meg: entity és embeddable (beágyazható). Az entity osztályok minden egyes példánya egyediként jelenik meg az adattárban. Egy perzisztens osztály példánynak nincs perzisztens identitása.
Entitást kétféleképpen kereshetünk:

- Azonosító alapján: `EntityManager` használatával, melyet a `@PersistenceContext` annotációval látunk el
- Kritérium alapján: valamiféle egyezőséget keresünk, ezt szolgálja a `Query`.

Beágyazható osztályoknál közvetlenül egyik sem szolgáltat eredményt.

Néhány követelmény a perzisztens osztályokhoz:

- Kötelező egy argumentum nélküli konstruktor `public` vagy `protected` láthatósággal.
- Sem egy entity osztály, sem a metódusa nem lehet `final`.
- Lennie kell egy vagy több azonosító mezőnek (elsődleges kulcs)
- A verzió mező használata nem kötelező, de ajánlott. Arra való, hogy egy entitáson belül ugyanannak az adatbáziselemnek az értékét konkurens módon ne lehessen megváltoztatni. Típusa valamilyen egész (`Long, int`) vagy `java.sql.Timestamp` lehet.
- Öröklődés: a perzisztens osztályok származtathatók perzisztens és nemperzisztens osztályokból, – kivétel néhány egyszerű osztály, mint például a `java.lang.Thread, java.net.Socket` - a nemperzisztens osztályok viszont perzisztens osztályokból kell, hogy származzanak. Az öröklődési hierarchiában minden perzisztens osztálynak azonos identitással kell rendelkeznie.
- A perzisztens mezők állapotait a JPA kezeli. Nem megengedett `static` és `final` mezők alkalmazása. Azonban tartalmaz beépített támogatást a legtöbb egyszerű típusra. Ezeket a típusokat három kategóriába sorolhatjuk: változó, állandó és relációs típus.

  Állandó típus: létrejötte után már nem módosítható. A JPA a következő típusokat támogatja: primitív típusok (`int, char, byte` stb.); a primitív típusok csomagoló osztályai (`Integer, Character, Byte` stb.); `java.lang.String; java.math.BigInteger; java.math.BigDecimal`.

  Változó típus: anélkül tudjuk változtatni egy mező értékét, hogy a mező új értéket kapna, azaz közvetlenül lehet manipulálni. Változó típusúak: `java.util.Date; java.util.Calendar; java.sql.Date; java.sql.Timestamp`; felsorolásos típus (`Enum`); entity és beágyazható típusok; `java.util.Collection; java.util.Set; java.util.List; java.util.Map`.

4.3.2. Entitások identitása

A Java kétféle objektum azonosságot különböztet meg: numerikus és kvalitatív. Numerikus egyezőség akkor áll fenn, ha a memóriában ugyanarra a példányra vonatkoznak. A kvalitatív azonosság az objektum egyenlőséget vizsgálja néhány, a felhasználó által definiált kritérium alapján. Ezzel szemben a JPA entity és perzisztens identitást különböztet meg. Az entity azt teszteli, hogy vajon két perzisztens objektum ugyanazt az állapotot reprezentálja az adattárban. Ha két azonos típusú entitás mezőértéke megegyezik, akkor a két entitás ugyanazt az állapotot képviseli az adatbázisban. Minden entitás azonosító mezőjének egyedinek kell lennie. A mező értéke lehet primitív, primitív csomagoló, sztring, dátum, `TimeStamp` vagy beágyazható.

Callback metódusok:
Gyakran szükség van különböző módosításra/pontosításra az objektumok életciklusának egyes állapotaiban. Ezt szolgálják a különféle callback metódusok. Minden perzisztens eseménynek van egy jelölője, amit a híváshoz kétféleképpen definiálhatunk, annotációt helyezünk el a metódus előtt, vagy egy XML fájlban megadjuk a metódusok listáját.
A szóban forgó metódus jelölők a következők:

- *PrePersist:* ezt az annotációt használjuk, ha elsődleges kulcsértéket akarunk tulajdonítani egy perzisztens objektumnak. Az XML leíró fájlban ezt a `pre-persist` elemmel jelöljük.
- *PostPersist:* egy objektum perzisztens állapotba kerülése előtt hívjuk meg. Például képernyőfrissítés egy sor hozzáadása, vagy törlése esetén. A `@PostPersist` az XML `post-persist` elemével egyenértékű.
- *PostLoad:* ezzel az annotációval ellátott metódust azután használjuk fel, miután az osztály minden `eager fetched` mezői betöltődtek az adatbázisból.
  Fetch: kétféle módban működik, EAGER (mohó) és LAZY (lusta). Azt mondja meg, hogy betöltésnél az adott irányban be kell-e tölteni a hivatkozott entitásokat. Eager fetch type esetén azonban előfordul, hogy egy entitás betöltésekor a vele kapcsolatban lévő összes entitást is behúzzuk vele.
- *PreUpdate:* azelőtt használjuk fel, mielőtt az objektumok új értékeit módosítanánk az adatbázisban (`flush`).

- *PostUpdate:* a `@PostUpdate` annotációval rendelkező metódusok az adatbázisban történt változások mentése után lépnek működésbe. Ez a művelet *cache* ürítésnél is fontos szerepet játszik.
- *PreRemove:* objektum törlése előtt használhatunk ezzel a jelölővel ellátott metódust, a perzisztens mezők elérése e metóduson belül lehetséges.
- *PostRemove:* az objektum már ki van jelölve törlésre.

Egy példa arra, hogyan deklaráljunk callback metódust:

```
@Entity
public class Futo{

        ...

        @OneToMany
        private List<Futam> futamCollection;

        @PreRemove
        public void logFutoDel (){
                ...
        }
}
```

A következő XML fájl ugyanazt jelenti, csak annotációk nélkül:

```
<entity class="Futo">
        <pre-remove> logFutoDel </pre-remove>
</entity>
```

4.3.3. Metaadatok

Minden perzisztens osztály mellé kell valamilyen metaadat. Ez a metaadat a következő célokat szolgálja:

1) Perzisztens osztályok azonosítása
2) Az alapértelmezett JPA viselkedés felüldefiniálása
3) Olyan információkat ad, melyeket egy perzisztens osztállyal nem tudunk egyszerűen kifejezni

A metaadat megadása történhet annotációk vagy XML leírók definiálásával, de használhatjuk a kettő keverékét is. Ha az XML metaadatot választjuk, akkor a fájlnak elérhetőnek kell lennie fejlesztési és futási időben is, és felderíthetőnek kell lennie a *provider* számára egy konfigurációs információnak:
Ezt szolgálja a META-INF könyvtárban elhelyezett persistence.xml fájl.

4.3.4. Osztályra vonatkozó metaadatok

1. Entitás: `@Entity` annotációval vagy XML leíróval vannak jelölve
2. Identity osztály: olyan osztályoknál van rá szükség, ahol összetett kulcsérték van, és a kulcsokat eggyé kell tenni. Ekkor használjuk a `@IdClass`-t, vagy XML-ben az ezzel egyenértékű *`id-class`*t.
3. A mapped ősosztály: nem entitás osztály. Példányait nem adhatjuk meg semmilyen lekérdezésben (sem az `EntityManager` sem a `Query` metódusaiban). Az annotáció a `MappedSuperclass`, az XML leíró pedig a `mapped-superclass`. A mapped szuperosztályok általában absztraktak.
4. Beágyazható osztály: a `@Embeddable` előzi meg az egyedet.

```
@Entity
public class Szemely{
        @Id
        int id;
        …
        @Embedded
        Cim cim;
}
@Embeddable
public class Cim{
        int ir_szam;
        String varos;
        String utca;
        …
}
```

4.3.5. Mező és property metaadatok

A JPA a perzisztens állapot elérésére két módot ajánl fel: *mező* és *property* alapú elérés.
Mező alapú elérés esetén az implementáció közvetlenül injektálja be az állapotot a mezőkbe, és vissza is adja a megváltozott állapotot. XML metaadat esetén az attribútum elérési típusát FIELD-re állítjuk, különben pedig csak meg kell jelölni a megfelelő annotációval az entitást. A *property típusú elérést* használva az adatok keresése és betöltése get és set metódusokon keresztül történik. Ez esetben az annotációt a get metódus előtt helyezzük el, és az XML-ben az entitás elérési attribútumát PROPERTY-ként tartjuk nyilván.
Minden osztály használhat akár mező, akár property alapú elérést, de egy osztályon belül a kettő nem keveredhet. Ha egy osztálynak van ősosztálya, akkor az alosztálynak ugyanazt a fajta elérést kell alkalmazni.

Típusai:

1) Transient: a nem perzisztens mezők `@Transient` annotációval vannak ellátva, az XML elem a `transient`. Nincs paramétere.
2) Id: `@Id`-vel jelöljük az azonosító mezőt, különben `id`-vel.
3) Generated Value: az `Id` annotáció mellett lehetőség van megadni, hogy az azonosító mezőt hogyan generáljuk. Ezzel az annotációval (`GeneratedValue`) egyedi azonosítót rendelhetünk a mezőkhöz, ehhez a következő két paramétert adhatjuk meg (vagy csak az egyiket, vagy mindkettőt egyszerre):
   - `GenerationType strategy`: egy `Enum` érték szemlélteti, hogy hogyan történjen a mező automatikus értékének létrehozása. Megjegyezném, hogy ezek az értékek csak egész típusúak lehetnek. Az `Enum`ok a következők:
     `GenerationType.AUTO`: definiál egy értéket. Ha nem adunk meg semmilyen típust, akkor ez az alapértelmezett.
     `GenerationType.IDENITY`: az adatbázis jelöli ki az értéket.
     `GenerationType.SEQUENCE`: az adatbázis felügyel egy számlálót, és ez alapján történik az elsődleges kulcsérték meghatározása.
     `GenerationType.TABLE`: az aktuális adattáblát használja fel az egyediség megbizonyosodására.

- *String generator*: a `@TableGenerator` egy elsődleges kulcs generátort definiál, melyre a nevével lehet hivatkozni. Több paraméter is megadható, de ezek közül csak a név megadása kötelező, aminek egyedinek kell lennie.

Ha a *`strategy`* be van állítva, de a *`generator`* nincs, akkor a JPA alapértelmezést használ (, azaz *`strategy=GenerationType.AUTO`* és *`generator=""`*).

A könnyebb megértés érdekében tekintsünk meg egy példát:

```
@Entity
public class Address {
		...
		@TableGenerator(
		name="addressGenerator", // a generátor név
		table="ID_GENERTOR", // a generált id-ket tároló tábla
			pkColumnName="GENERATOR_KEY", // a PK oszlop neve
			pkColumnValue="ADDRESS_ID", //a PK értéke a generátor táblában
		allocationSize=1) //az azonosítók értékének növelése
		@Id
		@GeneratedValue (strategy=TABLE, generator="addressGenerator")
		public int id;
		...
}
```

4) Embedded Id: egy entitást akkor látunk el `@EmbeddedId` jelöléssel, ha összetett kulccsal rendelkezik.
5) Version: a `@Version` szerepe, hogy a `merge` művelet idején és az optimista konkurenciavezérléskor biztosítsuk az entitás sértetlenségét.
6) Basic: az adatbázis oszlopainak az egyik legegyszerűbb leképezési módja a `@Basic`. Ezt az annotációt a következő típusú perzisztens mezőkre lehet alkalmazni: primitív típusok, primitív csomagolótípusok, `java.lang.String`, `java.math.BigDecimal`, `java.math.BigInteger`, `byte[]`, `Byte[]`, `char[]`, `Character[]`, `java.util.Calendar`, `java.util.Date`, `java.sql.Date`, `java.sql.Timestamp`, `Enum` és a `Serializable` típusok.

   Két paramétere lehet: `FetchType fetch` és `boolean optional`. A `FetchType` mód lehet LAZY vagy EAGER – a kettő közötti különbség már ismertetésre került korábban.

   A másik paraméter azt mondja meg, hogy a mező típus lehet-e `null` vagy sem.
7) Embedded: egy beágyazott mező az adatbázis rekordjának részeként jelenik meg. Beágyazható típusok jelölője lehet a `@Embedded`.

8) Order By: Ha szeretnénk a mezők valamilyen szintű rendezettségét elérni, akkor az `@OrderBy` annotációval érhetjük el. Lehet paraméter nélküli, ekkor elsődleges kulcs szerint rendez alapértelmezettként, de megadhatunk paramétert, ezen belül is azt, hogy növekvő (ASC) esetleg csökkenő (DESC) formában rendezze az adatokat – ha hiányzik az ASC vagy DESC, akkor növekvő sorrendben listáz. Több paramétert is megadhatunk, ebben az esetben először az első paraméter szerint rendez, majd a második szerint és így tovább.
9) Map Key: `OneToMany` és `ManyToOne` kapcsolatok esetén értelmezett. Az egymással kapcsolatban álló entitások formalizálják a `map` értékeit. A `MapKey` annotáció kijelöli azt a mezőt, melyet használni kíván kulcsként. A paraméter a `String name`, használata opcionális.
10) Perzisztens mezők:
    - A `static, transient, final` mezők nem perzisztensek.
    - primitív típusok, primitív csomagolók, `String, byte[], Byte[], char[], Character[], BigDecimal, BigInteger, Date, Calendar, Timestamp` és néhány `Serializable` típus perzisztens.
    - A beágyazható típusok perzisztensek.
    - Minden egyéb típus nemperzisztens.

4.3.6. Entitások relációja

A relációkat kétféleképpen tudjuk csoportba sorolni,

- számosság és
- a kapcsolat iránya szerint.

1) *Számosság:* egyik illetve másik oldalról hány egyed vesz részt a kapcsolatban. Négy különböző számosságot definiál a JPA: 1-1, 1-több, több-1 és a több-több. A használata csak a List, Collection, Set, Map típusoknál támogatott.
2) *Irány*: beszélhetünk egy- vagy kétirányú kapcsolatról. A kétirányú kapcsolat kezelése az alkalmazás feladata. Ebben az estben van egy tulajdonos és egy inverz oldal. A tulajdonos oldal egyenértékű az elsődleges kulccsal, az inverz oldalnak pedig a tulajdonos oldalra valamilyen módon hivatkoznia kell.

@OneToOne (1-1)

Egy `one-to-one` kapcsolatnál három estet különböztetünk meg: az asszociált entitásoknak ugyanaz az elsődleges kulcsa; egy külső kulcs tartozik valamelyik entitáshoz (a külső kulcsnak egyedinek kell lennie); létezik egy asszociációs tábla, mely tárolja a kapcsolatot a két entitás között.

Az első esetben közös kulcsot használva:

`@PrimaryKeyJoinColumn`: Ez az annotáció egy kulcs oszlopot specifikál, amit külső kulcsként használunk, hogy kapcsolódni tudjunk a másik táblához. Paraméterként megadhatjuk a tábla elsődleges kulcs oszlopának a nevét `(name="…")`, és a kapcsolandó tábla elsődleges kulcs oszlopát `(referencedColumnName="…")`. Persze ez nem kötelező, csak egy lehetőség.

Egy `one-to-one` kapcsolatot szemléltethetünk egy személy-személyigazolványszám közötti viszonnyal, ugyanis egy személyhez egy személyi szám tartozik és fordítva:

```
@Entity
public class Szemely{
        @Id
        public Long getId() { return id; }
        @OneToOne (cascade=CascadeType.ALL)
        @ PrimaryKeyJoinColumn
        public SzemIg getSzemIg() { return szemIg; }
        …
}

@Entity
public class SzemIg implements Serializable{
        @Id
        public Long getId() { … }
}
```

A másik esetben külső kulcson keresztül kapcsolódnak a táblák:

```
@Entity
public class Szemely implements Serializable {
        @OneToOne (cascade=CascadeType.ALL)
        @ JoinColumn (name=szemIg_fk)
        public SzemIg getSzemIg() { return szemIg; }
        …
}

@Entity
public class SzemIg{
        @OneToOne (mappedBy= ”szemIg”)
        public Szemely getSzemely() { … }
}
```

A Szemely tábla a SzemIg táblához külső kulcson keresztül kapcsolódik, ezt jelöli a `@JoinColumn`-ban megadott *szemIg_fk* név. A `JoinColumn` mindig a tulajdonos oldalon van definiálva. Emellett az inverz oldalon a `mappedBy` birtokolja a kapcsolatot.

A harmadik lehetőség az asszociációs tábla: a `@JoinTable` paramétereként megadható ennek a táblának a neve, egy `joinColumns`, mely a tulajdonos oldali tábla külső kulcsát jelöli, és egy `inverseJoinColumns`, ami az inverz tábla külső kulcsa.

```
@Entity
public class Szemely implements Serializable {
        @OneToOne (cascade = CascadeType.ALL)
        @ JoinTable (name = SzemelySzemIg,
                joinColumns = @JoinColumn (name = ”szemely_fk”)
                inversejoinColumns = @JoinColumn (name = ”szemIg_fk”))
        public SzemIg getSzemIg() { return szemIg; }
        …
}

@Entity
public class SzemIg{
        @OneToOne (mappedBy= ”szemIg”)
        public Szemely getSzemely() { … }
}
```

@ManyToOne (több-1)

A *@ManyToOne* reláció egy egyirányú kapcsolatot reprezentál egy másik entitással, feltüntetve a külső kulcsot (*JoinColumn*):

```
@Entity
public class Futam implements Serializable {
        …
        @JoinColumn (name = "FUTO_ID")
        @ManyToOne (optional = false)
        public  Futo getFuto() {
                return futo; }
        …
}
```

A many-to-one kapcsolatnak ugyanolyan paraméterei lehetnek, mint a one-to-one-nak, kivéve *a* mappedBy. A targetEntity paraméterrel leírhatjuk a cél entitás nevét. Általában nem szükséges megadni, csak ha egy interfészt visszatérési típusként akarunk használni (targetEntity = FutoImp.class).
Asszociációs táblán keresztül is le lehet képezni a kulcsokat, mint ahogyan azt láttuk a one-to-one reláció esetében.

@OneToMany (1-több)

Egy one-to-many kapcsolat lehet egy- vagy kétirányú.
Kétirányú kapcsolat:
A tulajdonos oldalon a one-to-many kapcsolat a következő módon van annotációval jelölve: *@OneToMany* (mappedBy = "…"). Egy példa arra, hogyan annotáljuk a propertyket, amikor a one-to-many kapcsolat az inverz oldalon van:

```
@Entity
public class Futo {
        @OneToMany (mappedBy = "futo")
        private Collection<Futam> getFutam() { …}
}
```

```
@Entity
public class Futam {
        @JoinColumn (name = "futo_fk")
        @ManyToOne
        public  Futo getFuto() {
                return futo; }
        …
}
```

A *Futo* osztály kétirányú `one-to-many` kapcsolatban van a *Futam* osztállyal, a `mappedBy` által megnevezett *futo* property-n keresztül. Ha viszont nem az inverz oldalon van a `one-to-many` kapcsolat, akkor a `mappedBy` paramétert el kell távolítanunk, és az inverz oldalon a `@JoinColumn insertable` és `updatable` paramétereit `false`-ra kell állítani. Ez a megoldás külön UPDATE-ek megírását vonja maga után.

Egyirányú kapcsolat:

Az egyik lehetőség egy külső kulcs létrehozása a tulajdonos oldalon:

```
@JoinColumn (name="futo_id")
```

De e helyett inkább *join table* használata javasolt. A `OneToOne` kapcsolatnál már láthattuk, hogy hogyan kell létrehozni egy *join table*-t, azonban ekkor egy kollekció annotálására alkalmazzuk.

Ha csak a `@OneToMany`-t adjuk meg a tulajdonos oldalon, akkor létrejön egy *join table*, alapértelmezett névvel (a két tábla neve között '_' szerepel).

@ManyToOne (több-több)

A `ManyToMany` reláció egy- vagy kétirányú kapcsolatot biztosít az egyedek egy kollekciójához. Az adatbázisban ezt a *join table* reprezentálja.

A `@JoinTable`:

- *name* paramétere specifikálja a kapcsolótábla nevét, ha viszont nincs megadva, akkor a tulajdonos és az inverz tábla nevéből áll össze a név.

- a kapcsolódó oszlopok tömbjét definiálja
- megmondja az inverz osztály oszlopainak tömbjét.

Az inverz oldalon a *mappedBy* argumentum hivatkozik a másik oldali property nevére, ha a kapcsolat kétirányú.

```
@Entity
public class Hallgato{
        @ManyToMany
        @JoinTable (name="HALLGATO_OKTATO",
        joinColumns = @JoinColumn (name="HALLG_ID", referencedColumnName="ID"),
        inverseJoinColumns = @JoinColumn (name="OKTATO_ID", referencedColumnName="ID")
        )
        public Set getOktatok() {
                return oktatok; }
        }
@Entity
public class Oktato{
        @ManyToMany (mappedBy = "oktatok")
        public Set getHallgatok() {
                return hallgatok; }
      }
```

4.3.7. Persistence Unit

- A csomagolás és a telepítés alapegysége.
- Entitások és a hozzájuk kapcsolódó osztályok halmaza.
- O/R mapping információk leírása: Java nyelvű annotációkkal és/ vagy XML fájlokkal
- Az entitások közötti relációk, lekérdezések hatáskörét definiálja
- persistence.xml: konfigurációs információ

4.3.8. Persistence Context

Egy perzisztencia unithoz tartozó entitás példányok halmaza, ezen keresztül kezeljük az entitások és az adatbázis közötti kapcsolatot.
A perzisztencia kontextus injektálás segítségével elérhető (például Session Bean-be):

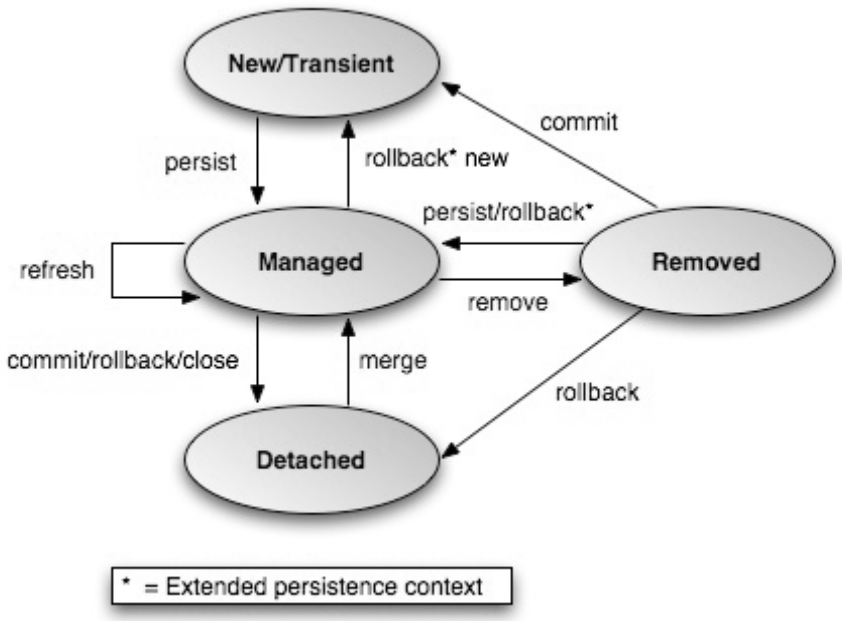
```
@PersistenceContext
private EntityManager em;
```

A `@PersistenceContext` néhány paramétere:

- unitName: ha több Unit van a persistence.xml-ben, akkor meg kell adni
- type: egy perzisztencia kontextus élettartama lehet az egész tranzakcióra kiterjedő (TRANSACTION) vagy több tranzakciót is átfogó (EXTENDED). Az élettartamot vagy a konténer vagy az alkalmazás menedzseli. A TRANSACTION az alapértelmezett.

Entitáskezelő API

A Persistence Context és az entitások életciklusának kezelésére szolgál
Az életciklus állapotait a következő ábra szemlélteti:



- New: létrejön egy új entitás, ekkor még nem perzisztens, és nincs menedzselt állapotban.
- Persist: menedzselt állapotba kerül az entitás. Az adatbázisba a legelső `flush` vagy `commit` esetén kerül. Ha detached állapotban hívjuk meg a `persist (Object o)` metódust, akkor `IllegalArgumentException`-t kapunk.
- Remove: entitás törlése, `commit` vagy `flush` után az adatbázisból is. Detached állapot esetén szintén `IllegalArgumentException`.
- Refresh: az entitás állapota frissül az adatbázisból. Ez a művelet hosszú futású tranzakcióknál hatékony, ugyanis fennáll az adatok elavulásának a veszélye.
- Detach: a `Persistence Context` törlése vagy befejeződése esetén az entitások leválnak a kontextusból.
- Merge: a `merge` metódus egy lekapcsolódott entitás állapotának másolatával tér vissza, és ez a másolat egy menedzselt entitás. Ennek állapota megegyezik az eredeti állapotával.
- Lock: zárolja a paraméterként megadott entitást a megadott módon. Két zárolási módot különböztetünk meg, ezek a `javax.persistence.LockModeType` enumjai: READ és WRITE.

  READ: más tranzakciók konkurens módon olvashatják ugyanazt az objektumot, de nem írhatják.

  WRITE: konkurens módon nem olvashat és írhat másik tranzakció egy objektumot.

4.3.9. Lekérdezések (Query)

- Statikus, névvel azonosítható lekérdezések: Java metaadattal vagy XML fájllal definiálható, ilyen például a `@NamedQuery`. Definíciója:

```
@NamedQuery (
        name="findAllFutoWithName",
        query="SELECT f FROM Futo f WHERE f.name LIKE :futoName"
)
```

A `name` és a `query` paramétereket kötelező megadni. Itt definiáljuk a lekérdezésünk feltételeit.

A használata:

```
@PersistenceContext
public EntityManager em;
 ...
futok = em.createNamedQuery("findAllFutoWithName")
            .setParameter("futoName", "Valaki")
            .getResultList();
```

Egy `EntityManager` példányt kell létrehozni, majd a `createNamedQuery` paramétereként megadjuk a `@NamedQuery`-ben meghatározott nevet. Az `EntityManager` interfész metódusai tartják a kapcsolatot és kommunikálnak a perzisztencia kontextussal.

- Dinamikus lekérdezések: Java Persistence Query Language vagy natív SQL nyelven

`public Query createQuery (String query)`: a `Query` objektumok arra valók, hogy megtaláljuk a keresett entitásokat valamilyen kritérium alapján. A metódus létrehoz egy `query` sztringet a JPQL-nek megfelelően.

Natív SQL: `public Query createNativeQuery (String sql)`: sql utasításokat adunk meg sztringként.

- Keresés elsődleges kulcs alapján:

```
public Futo keresFuto (Object id) {
        return em.find (Futo.class, id);
}
```

A `Query` fontosabb metódusai:

- `List getResultList()`: végrehajt egy *SELECT* utasítást, és visszatér az általa visszaadott eredménynek listájával (List).
- `Object getSingleResult()`: a *SELECT* utasítás egyetlen egy eredménnyel tér vissza
- `int executeUpdate()`: eleget tesz egy `update` vagy egy `delete` utasításnak. Az `int` típusú visszatérési érték azt jelzi, hogy hány entitás lett módosítva vagy törölve.

4.3.10. Java Persistence Query Language (JPQL)

Az SQL-hez hasonló, saját lekérdező nyelv. Az EJB QL módosítása néhány bővítéssel:

- törlést, módosítást nem kell egyesével elvégezni, létezik az ún. többes törlés/ módosítás
- GROUP BY, HAVING
- subquery-k (ALSELECT)
- projekciós lista (csak bizonyos attribútumokat adjon vissza)
- egy SELECT-ben visszaadott oszlopokat egy általunk definiált objektumként kaphatjuk meg.

4.3.11. Leszármaztatás

Az entitások származhatnak más entitásokból - akár konkrét, akár absztrakt -, közönséges Java osztályból és „mapped" ősosztályból (`@MappedSuperclass`).
A JPA a következő három öröklési módot támogatja:

- Table per Class: tábla tartozik minden konkrét osztályhoz, vagyis külön tábla szükséges minden altípushoz. Emellett minden tábla tartalmazza a szuperosztály attribútumait is. Normalizált megoldás, de a polimorfizmus támogatása nehézkes.
- Single Table: egy tábla egy öröklési hierarchiához tartozik, azaz a szuper- és alosztályok összes attribútuma egy táblába vannak leképezve, a példányok egy diszkriminátor által vannak megkülönböztetve. Előnye, hogy nem kell `join`, és a polimorfizmust is támogatja, viszont nagyobb osztályhierarchia esetén túlzsúfolt lesz a tábla. Az osztály hierarchia legfelső szintjén meg kell adni az osztálydefiníció előtt a következőket:

```
@Inheritance ( strategy=InheritanceType.SINGLE_TABLE )
@DiscriminatorColumn ( name="oszlop_név" )
```

  Továbbá minden osztály előtt a típusra utaló nevet:

```
@DiscriminatorValue ("osztálynév")
```

- Joined Subclass: külön tábla gyermekosztályonként. Az ősosztályban definiált oszlopok egy táblában, a gyermekosztályban definiált oszlopok pedig külön táblában vannak,

továbbá kell egy külső kulcs az ősre (`@PrimaryKeyJoinColumn`). Nincsenek fölösleges oszlopok, azonban sok `join` ronthatja az átláthatóságot. Az osztálydefiníciót a

```
@Inheritance ( strategy=InheritanceType.JOINED )
```

előzi meg.

4.4.Hibernate

A Hibernate az objektum perzisztenciát biztosító eszközrendszer Java és .NET alkalmazások esetén. Lehetővé teszi perzisztens osztályok létrehozását, melyek objektum-orientált módon működnek, vagyis megengedett az öröklődés, a polimorfizmus, a kompozíció, az asszociáció és a kollekciók használata.
A HQL (Hibernate Query Language) a Hibernate hordozható, saját lekérdező nyelve. A HQL az SQL nyelv kiterjesztése. A lekérdezéseket megfogalmazhatjuk mind HQL, mind SQL nyelven.
Továbbá a Hibernate rendelkezik a `Criteria` - mely a fejlesztőnek megadja azt a lehetőséget, hogy objektum-orientált módon készítse el a lekérdezéseit - és az `Example` keretrendszerrel.

Alrendszerei közül tekintsünk meg néhányat:

- Hibernate Core: Előnyei: SQL kód előállítás, megoldja helyettünk az objektum konverziót és a JDBC eredményhalmazok kezelését.
  Független keretrendszer, minden JavaEE alkalmazásszerveren működik. Egy egyszerű megoldás a perzisztencia megvalósítására, a hozzá tartozó egyszerű API-vel. A leképezett metaadatok XML fájlokban tárolódnak. Egy XML fájlban metaadatok segítségével definiálható az adatbázis tábla és a perzisztens objektum közötti kapcsolat.
- Hibernate Annotations: A JDK 5-től létezik. A Java kódban elhelyezett annotációk alkalmazásával az objektum relációs leképezés megvalósítása. Az egyszerű Hibernate XML mapping fájlok mellett egy kiegészítő megoldás. A metaadatok a kóddal együtt egy Java fájlba vannak egyesítve, azért, hogy a felhasználót segítsék a tábla struktúra megértésében. Ha a hordozhatóság fontos a számunkra, akkor a rendszer lehetőséget nyújt standard JPA, és fejlettebb Hibernate annotációk alkalmazására.
- Hibernate Entity Manager: implementálja a perzisztencia kezelő API-t (JPA), a perzisztencia lekérdező nyelvet (Java Persistence Query Language), a java perzisztencia objektum életciklusra vonatkozó szabályokat, a konfigurációt és csomagolást. Ezek mellett használható a Hibernate natív API, natív SQL, és a JDBC, ha szükséges.

- Hibernate Validator: adatbázis sémára vonatkozó megszorítások írhatók le, hogy mielőtt a Hibernate mentene egy változtatást, ellenőrzi az entitás „érvényességét”, azaz megfelelőek-e a módosítások. Ilyen lehet például a `@NotNull`, amikor is nem adtatunk meg null értéket, vagy a `@Email`, mikor ellenőrizzük az e-mail cím érvényességét stb.
- NHibernate: .NET platformra készült. A perzisztens .NET objektumok kezelését az XML-ben leírt metaadatok valósítják meg. A perzisztens osztályoknak nem szükséges semmilyen interfészt implementálni vagy öröklődnie valamilyen osztályból. Támogatja az OO paradigma használatát, nyílt forráskódú. Automatikus SQL generálás az objektumok betöltésére, mentésére.

4.4.1. A Hibernate felépítése

A Hibernate architektúrának három fő komponense van:

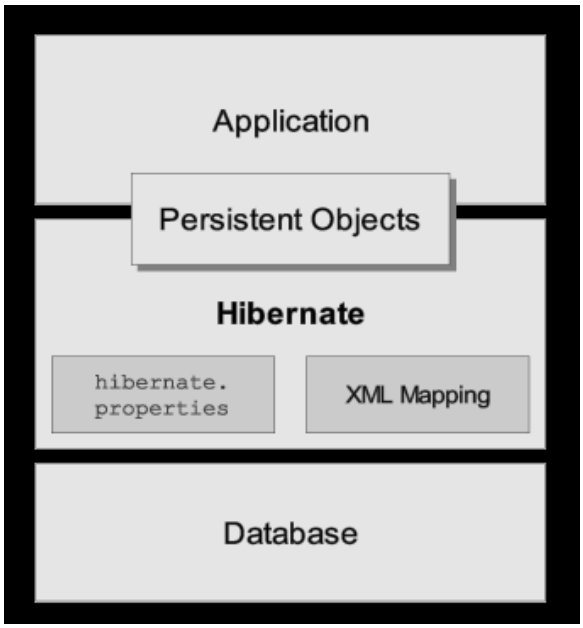
Kapcsolatok kezelése: cél a hatékony adatbázis kapcsolat kezelése, ugyanis egy adatbázis kapcsolat sok erőforrást igényel.

Tranzakciók kezelése: egy időben több adatbázis művelet is végrehajtódhat.

O/R mapping: adat reprezentációs leképezési technológia (objektum modellből relációs adatmodell). A Hibernate-nek ezt a részét használjuk a rekordod manipulálására az adatbázisban. (SELECT, INSERT, UPDATE, DELETE)

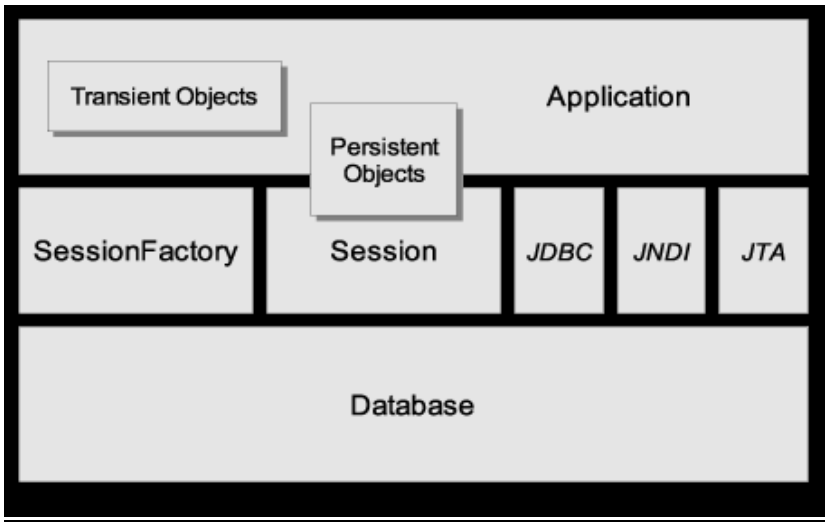
Az architektúra fajtái:

a) Magas szintű architektúra:

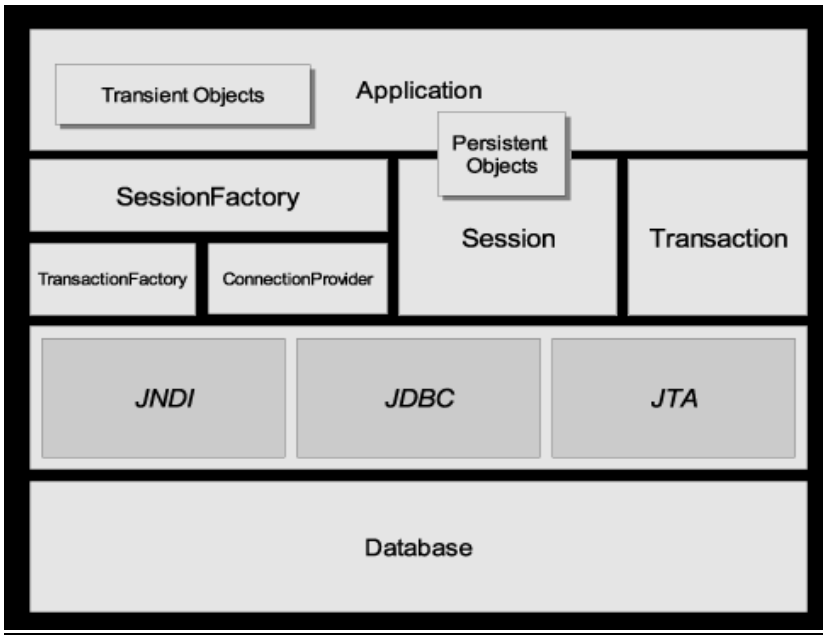


Az ábrán látható, hogy a Hibernate használ egy adatbázist, és konfigurációs adatokat**.** A Java osztályok a táblát reprezentálják az adatbázisban, és a példányszintű változók pedig a tábla sorait. A Hibernate különböző operációkat hajt végre az adatbázison a metaadatok és a konfigurációs beállítások segítségével, objektumokat ad át az alkalmazás számára, vagy az általa létrehozott objektumokat menti az adattárba. A lényeg, hogy a fizikai adatbázist elrejti előlünk.

b) „Lite" architektúra: gondoskodik saját JDBC kapcsolatról, és kezeli a tranzakciókat.



c) „Full cream” architektúra: mindhárom komponenst használja



Tekintsük meg közelebbről az ábrán látható objektumok definícióit:

- `org.hibernate.SessionFactory:` az adatbázis-leképezés tárolására kialakít egy szál-biztos, azaz állandó *cache*-t. `Session`-öket hoz létre, és kliens a `ConnectionProvider` felé. Nyilvántarthat egy másodlagos adat cache-t, melyek újrafelhasználhatóak a tranzakciók között.
- `org.hibernate.Session:` a kliens és a perzisztens tároló között egy párbeszédet reprezentáló rövid életű objektum. Mikor adatbázis módosításokat akarunk végezni, a `Session` objektumok létrehoznak egy `Transaction` típusú objektumot. Ezek az objektumok reprezentálják az adatbázison elvégzett munkát. A perzisztens objektumok számára fenntart egy cache-t.
- Perzisztens objektumok: rövid életű objektum, mely megvalósítja az üzleti funkciókat és a perzisztens állapotot. Ezek lehetnek Java Beanek vagy egyszerű Java objektumok, melyek egy `Session`-höz tartozhatnak csak. Ha a Session bezárul, az objektum elérhetővé válik az alkalmazás többi rétege számára.
- Tranziens objektumok: Egy perzisztens osztály azon példányai, melyek adott időben nincsenek kapcsolatban egy `Session`-nel sem. Ilyen például az az objektum, melyet egy olyan `Session` hozott létre, mely már lezárásra került, vagy amit az alkalmazás létrehozott és még nincs perzisztens állapotban.

- `org.hibernate.Transaction`: olyan objektum, melyet egy alkalmazás atomi feladatok specifikálására használ. Egy `Session`-ön belül több tranzakció is megadható, azonban egy nem véglegesített `Transaction` megléte javasolt.
- `org.hibernate.connection.ConnectionProvider`: a JDBC kapcsolatok kezeléséért felelős.
- `org.hibernate.TransactionFactory`: A `Transaction` példányok kezelését végzi. A felhasználó kiterjesztheti, újraimplementálhatja.
- Interfészek: a Hibernate egyéb interfészeket is biztosít a fejlesztő számára, ezeket az API tartalmazza.
- JTA (Java Transaction API): tranzakciókat megvalósító API Java alkalmazások és J2EE szerverek számára.

4.4.2. Konfiguráció

A Hibernate létrehozásának egyik fő célja az volt, hogy különböző környezetben használhassuk, éppen ezért sok konfigurációs paraméterrel rendelkezik. Az `org.hibernate.cfg.Configuration` egy példánya reprezentálja az alkalmazás beállításait. A `Configuration` egy példányán keresztül megadhatjuk a tulajdonságokat és a XML mapping dokumentációkat. Ha a mapping fájlok a classpath-ban vannak, akkor az `addResource()` metódusban megadhatjuk a fájl nevét, de általában jobb megoldás, ha meghatározzuk a class fájlokat, és hagyjuk, hogy a Hibernate keresse meg.
Ezen kívül különböző módokat kínál fel a Hibernate a konfigurációs beállítások megvalósítására:

- A `java.util.Properties` példányát átadjuk a `Configuration.setProperties()`-nek
- hibernate.properties: ez a legegyszerűbb megoldás. Abban az esetben, ha nem adunk meg konfigurációs fájlokra vonatkozó információkat, ezt a fájlt keresi a rendszer.
- A `java` parancs használatával
- hibernate.cfg.xml: `<property>` elemeket tartalmaz. Ez váltja fel a hibernate.properties konfigurációs állományt. A fejlesztő dönti el, hogy melyiket használja, akár mindkettőt is. A CLASSPATH-ban kell elhelyezni.

Fontosabb property-k:

- `hibernate.connection.driver_class`: az adatbázishoz tartozó JDBC driver.
- `hibernate.connection.url`: a kapcsolat URL-je, ezen keresztül érhető el az adatbázis.
- `hibernate.connection.username`: az adatbázishoz tartozó felhasználónév
- `hibernate.connection.password`: jelszó
- `hibernate.dialect`: az adatbázis dialektusa (`org.hibernate.dialect.DerbyDialect`)
- `hibernate.show_sql`: DML műveletek esetén megmutatja a használt SQL utasítást, ha értéke `true`.

Ha minden mapping illeszkedik az `org.hibernate.cfg.Configuration`-re, az alkalmazásnak el kell érnie egy factory-t egy `Session` példánynak.

```
SessionFactory sessions = cfg.buildSessionFactory();
```

Több `SessionFactory` használata is megengedett, ha több adatbázist használunk.
Egy Session létrehozása a következőképpen történik:

```
Session session = sessions.openSession();
```

Amint szükségünk lesz adatbázis elérésre, a JDBC kapcsolat elérhetővé válik. Abban az estben fog működni a kapcsolat, ha néhány tulajdonságot beállítunk.

4.4.3. Hibernate Query Language (HQL)

A Hibernate az SQL-hez hasonló, saját objektumorientált lekérdező nyelvvel rendelkezik.
Jellemzői:
Case-sensitive: nem különbözteti meg a kis- és nagybetűket, kivéve a Java osztályok és attribútumok nevei.
FROM záradék: hasonlít az SQL FROM-jához. Sokszor szükségünk lehet egy aliasnév meghatározására, mert lehet, hogy később hivatkozni szeretnénk rá.

```
from Osztalynev [as] aliasnev // az 'as' opcionális
```

Megadható továbbá JOIN (inner, left outer, right outer, full), valamint alkérdés.

SELECT záradék: SQL-szerű. A SELECT-ben adható meg, hogy mely attribútumokat szeretnénk lekérdezni. Abban az esetben, ha minden attribútumot vissza szeretnénk kapni, nem kell megadnunk a SELECT-t, hanem a lekérdezést egyszerűen a FROM kulcsszóval kezdjük. Összetett attribútum lekérdezése esetén az eredményt visszaadhatjuk `Object[ ]`-ként, List-ként:

```
SELECT new list (… [, ... ] ) FROM …
```

Java objektumként:

```
SELECT new Osztalynev (… [, ... ] FROM …
```

Map-ként (egy aliasnév és a hozzá tartozó kulcsérték):

```
SELECT new map (… , … ) FROM …
```

Összesítő függvények: A SELECT-ben használható az avg, sum, min, max, count stb. függvények, valamint használhatunk aritmetikai operátorokat, konkatenációt.

Polimorf leképezések: A FROM-ban megadhatók Java osztályok, interfészek. A lekérdezés minden olyan perzisztens osztály példányaival visszatér, mely kiterjeszti azt az osztályt, vagy kiterjeszti azt az interfészt. A

```
FROM Object o
```

lekérdezés minden perzisztens objektumot visszaad.

WHERE záradék: a záradékban használhatók azok a kifejezések, melyek SQL-ben (sztring konkatenáció, `case` kifejezések, matematikai/bináris/logikai operátorok, `between, is (not) null, current_time, cast , some, all`, névvel rendelkező paraméterek (`:name`), a Java `public static final` konstansa, SQL literálok stb.)

Egy speciális tulajdonság az id: az objektum egyedi azonosítójára hivatkozunk. Például:

```
FROM Person as p WHERE p.id=1
```

ORDER BY, GROUP BY

Subquery: HQL subquery-t csak SELECT és WHERE záradékban adhatunk meg.

Komponensek: SELECT, WHERE vagy ORDER BY utasításokban.

```
SELECT p.name FROM Person p
vagy
FROM Person p ORDER BY p.name
```

Itt a *name* a komponens.

DML utasítások, mint például UPDATE, INSERT stb.

Criteria lekérdezések:

Egy `Session` objektum által készíthető `org.hibernate.Criteria` példány egy leképezést reprezentál.

```
Criteria c=session.createCriteria(Osztalynev.class);
```

Az `org.hibernate.criterion.Restrictions` osztály különböző statikus metódusokat definiál abból a célból, hogy hozzájusson beépített `Criteria` típusokhoz, ezzel szűkítve az eredményhalmazt.

Például:

```
Restrictions.like (String property, Object value) // SQL LIKE operátorhoz hasonlít
Restrictions.or (Criterion c1, Criterion c2) // a c1 és c2 diszjunkciójával tér vissza
Restrictions.isNotNull (String property) //SQL-szerű „is not null” megszorítás
```

`org.hibernate.criterion.Order`: `Order.asc (”name”)` név szerinti rendezés növekvően.

Asszociáció: a `createCriteria()` metódust alkalmazva megszorítások adhatók a kapcsolódó entitások mellé.

```
session.createCriteria(”egyik_entitas”).add( Restrictions.like (… )).createCriteria(”masik_entitas”)
```

Ekkor egy új `Criteria` példány jön létre, mely a ”`masik_entitás`” kollekció példányaira hivatkozik.

Ha a `createCriteria()` helyett a `createAlias(”masik_entitas”,”aliasnev”)` metódust adjuk meg, akkor nem jön létre új példány.

Fetch módok: a `setFetchMode()` állítja be, hogy melyik osztály milyen módon kerüljön betöltásre:

```
.setFetchMode("egyik_entitas", FetchMode.EAGER)
.setFetchMode("masik_entitas", FetchMode.EAGER)
```

Osztályaink adatbázisba való leképezése kétféleképpen történhet:

1. Egy ún. mapping fájlba meg kell adni, hogy az egyes osztályoknak milyen tábla feleltethető meg, ezen belül a táblák kulcsainak, attribútumainak, a kollekciók a leképezését tartalmazza. *(Függelék I.2)*
2. Az egyszerű Java osztályokban annotációkat helyezünk el.

A Hibernate olyan osztályokat tud leképezni, melynek:

- adattagjai nem primitív típusúak
- van paraméter nélküli konstruktora
- az adattagokhoz léteznek get/set metódusok

4.4.4. Hibernate Annotations

Az XML fájlok mellett/helyett használhatunk annotációkat.
A Hibernate Annotations csomag a következőket tartalmazza:

- a szabványosított Java Persistence és az EJB 3.0 `O/R mapping` annotációkat (ami az előző részben ismertetése került)
- kiterjesztett Hibernate-specifikus annotációkat

Kiterjesztett Hibernate annotációk:

`@org.hibernate.annotations.Entity` néhány paramétere:

- `mutable`: az entitás csak olvasható, ha értéke `true`
- `dinamicUpdate`: ha `true`, akkor az SQL UPDATE futás időben jön létre és csak azokat az oszlopokat tartalmazza, melyek értéke megváltozott.
- `dinamicInsert`: ha `true`, akkor az SQL INSERT futás időben jön létre és csak azokat az oszlopokat tartalmazza, melyek értéke nem `null`.

- `selectBeforeUpdate`: A Hibernate csak akkor végzi el az UPDATE utasítást, ha biztos benne, hogy az objektum értéke megváltozott.

`@org.hibernate.annotations.Where`: egy WHERE záradékot definiál az osztály egyik példányának kinyerésére

`@org.hibernate.annotations.Check`: SQL-szerű `check` megszorítás

`@org.hibernate.annotations.Table`: kiegészítő információ arról, hogy a tábla elsődleges vagy másodlagos. A `@javax.persistence.Table` kiegészítése, nem a felüldefiniálása. Másodlagos táblára a következő elemeket lehet például használni:

- `fetch`: `FetchMode.JOIN`: a Hibernate `inner join`-t használ, hogy visszanyerjük a másodlagos táblát, amit egy osztály vagy egy szuperosztály határoz meg, és az `outer join`-t, amit egy alosztály határoz meg. Ez az alapértelmezett.
- `foreignKey`: az elsődleges táblára hivatkozó külső kulcsot definiál a másodlagos táblába.

`@org.hibernate.annotations.Immutable`: Az alkalmazás nem módosíthatja és nem törölheti ezt az entitást.

`@org.hibernate.annotations.GenericGenerator`: a Hibernate lehetőséget nyújt a fejlesztőnek saját azonosító generátor definiálását. Paraméterei: `name`, `strategy`. A `name` a generátor egyedi neve, a `strategy` pedig a generálás módja.

`@org.hibernate.annotations.Parent`: Beágyazható objektumnál definiálható, mely visszautal a tulajdonos elemre.

`@org.hibernate.annotations.TypeDef`: Saját típusdefiníció deklarálás. Osztály vagy csomagszinten helyezhető el.

`@org.hibernate.annotations.NotFound`: `@OneToOne`, `@OneToMany`, `@ManyToOne`, `@ManyToMany` annotációk előtt adható meg, hogy ne kelljen a kivételekkel foglalkozni.

`@org.hibernate.annotations.LazyToOne`: `lazy` állapotot definiál egy `OneToOne` vagy `ManyToOne` asszociációnak.

`@org.hibernate.annotations.LazyCollection`: `lazy` állapotot definiál egy kollekciónak.

`@org.hibernate.annotations.Fetch`: egy fetching stratégiát definiál az osztályok betöltésére.

`@org.hibernate.annotations.Sort`: a Javaban használatos `comparator`-hoz hasonlít. Saját `comparator` implementáció is definiálható, ekkor meg kell adni a `comparator` paramétert, és a kollekció típusa SortedSet vagy SortedMap lehet.
`@org.hibernate.annotations.IndexColumn`: megmondhatjuk, hogy melyik oszlop legyen a lista index eleme.
`@org.hibernate.annotations.Cascade`: A kapcsolatban álló entitások esetén megadja, hogy milyen műveletek hívódjanak meg. Ezt a `cascade` művelettel lehet konfigurálni. Például egy szülő objektumon alkalmazott `save()` és/vagy `delete()` műveletek végrehajtódnak a gyerek objektumon is. A CascadeType lehet `PERSIST, MERGE, REMOVE, REFRESH, DELETE` stb.
`@org.hibernate.annotations.FilterDef(s)`: Szűrőket definiálhatunk osztály vagy csomag szinten. A `FilterDef` két paramétere: `name` és `parameter`, ami a filter viselkedését adja meg.

4.4.5. Query-k

Lényegesen több paraméter adható meg, mint a standard JPA `query`-ben:

- `flushMode`: a lekérdezés `flush` módja (`Always, Auto, Commit, Manual`)
- `cacheable`: a lekérdezés `cache`-elhető vagy sem
- `timeout`: időtúllépés
- comment: az SQL lekérdezés mell komment adható
- `fetchSize`: a sorok száma egy lekérdezésre vonatkozóan
- `readOnly`: a `fetch` során megkapott eredmény csak olvasható vagy nemcsak olvasható

Az itt ismertetett annotációk csak egy kis részét képezik az API-nak, bővebb ismertető a http://docs.jboss.org/hibernate/stable/annotations/api/ oldalon érhető el.

5. A programról

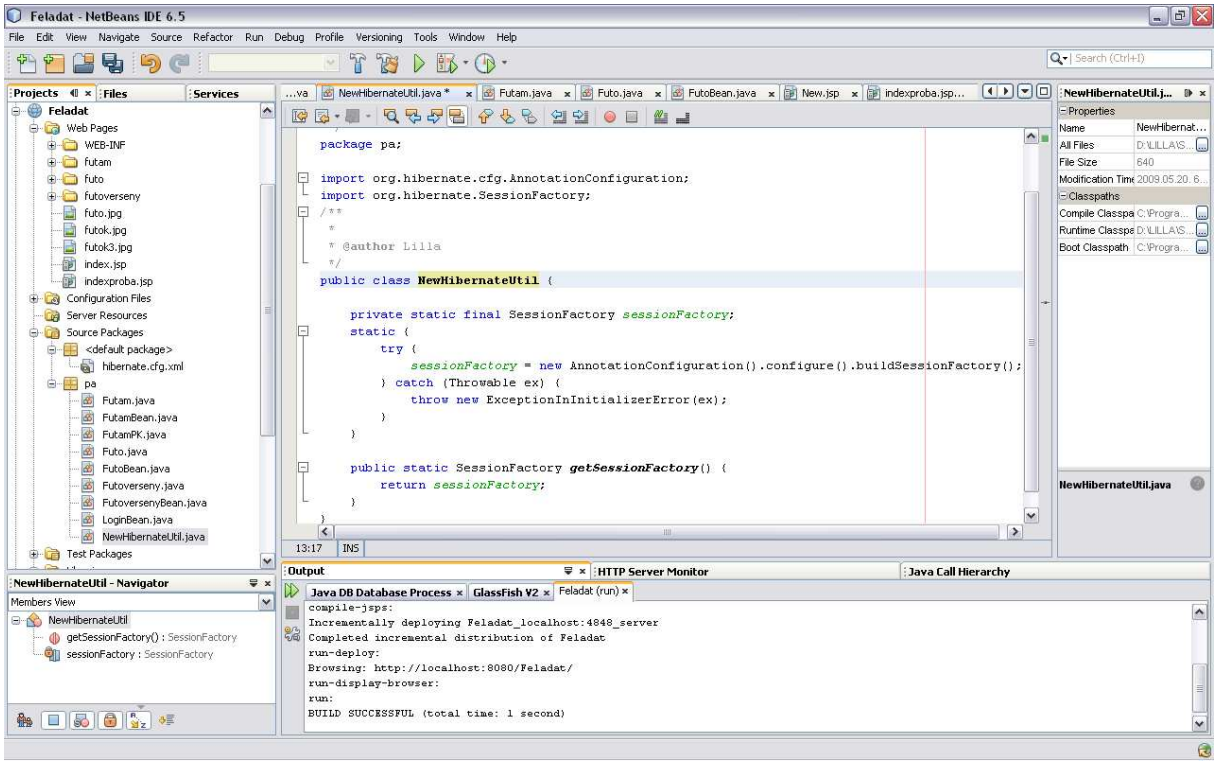
A példaprogramomat Windows operációs rendszerben, a Netbeans 6.5-ös verziója alatt készítettem. Az entitásokra vonatkozó perzisztencia-kezelést a Hibernate-en keresztül értem el, és a felhasználói felület kialakítására JSP-t használtam.
A Hibernate előnye, hogy egyedi HQL lekérdezőnyelvének segítségével adatbáziskezelő-rendszertől független alkalmazásokat fejleszthetünk ki.

A program leírása:

A program futóversenyek kezelését reprezentálja. Három entitás osztályom van:

- `Futo`: bárki jelentkezhet egy futóversenyre, ha már elmúlt 14 éves, ehhez meg kell adnia címét, nevét, és azt, hogy melyik versenyre szeretne jelentkezni. A felhasználó jogosult a futóversenyek idejének, helyszínének megtekintésére, és az adott versenyekre jelentkezők listáját is láthatja.
- `Futoverseny`: a futóverseny tartalmazza a versennyel kapcsolatos információkat (név, dátum, helyszín). A futóversenyek meghirdetését, törlését az `admin` végzi.
- `Futam`: a futam tartalmazza, hogy az egyes versenyekre kik jelentkeztek.

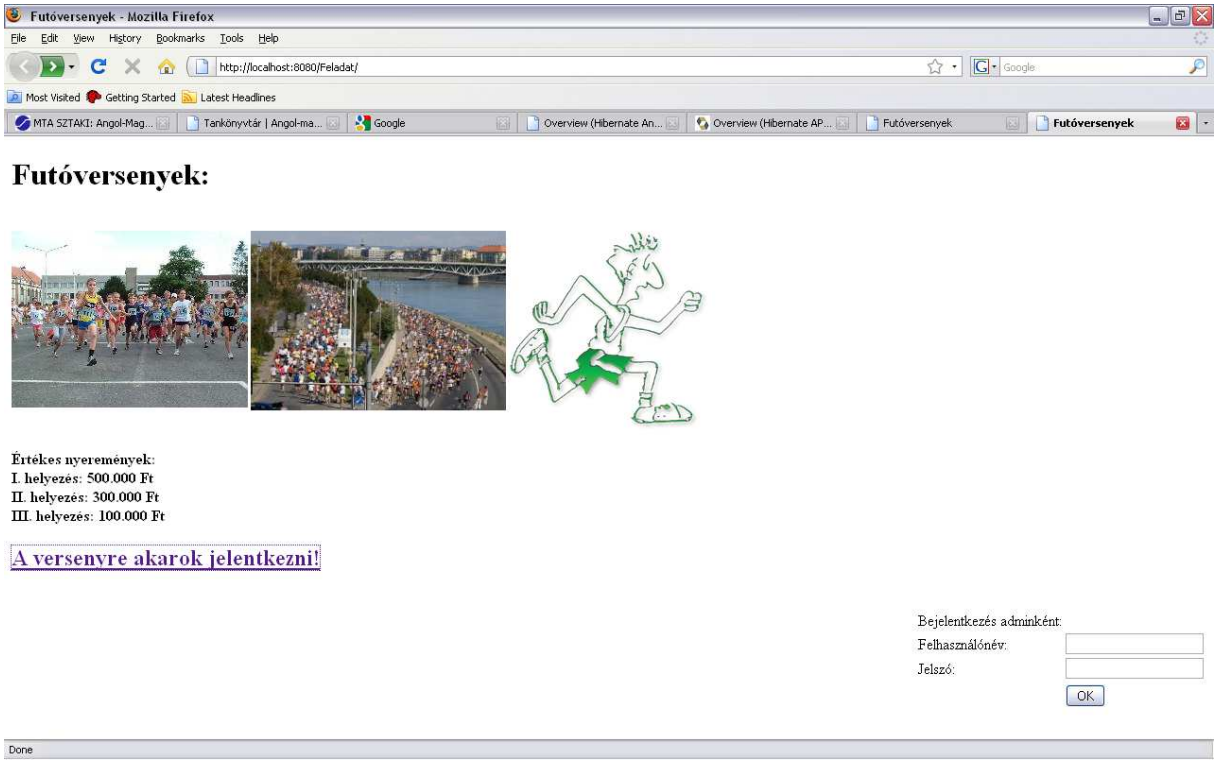
Továbbá rendelkezik három olyan osztállyal, melyek ezen osztályoknak a kezelését végzik, illetve egy `NewHibernateUtil` segédosztállyal, a Hibernate a beállításait ezen keresztül végzi.
A következő ábrán láthatjuk, hogy pontosan milyen osztályokat tartalmaz az alkalmazás, és a `NewHibernateUtil` felépítését. A `SessionFactory`-ről már volt szó. `Configuration()`-t az `AnnotationConfiguration()` váltja fel abban az esetben, ha Hibernate Annotations-t használunk. A `configure()` metódus meghívásakor a Hibernate keresi a `hibernate.cfg.xml`-t, és ha nem találja, akkor kivételt dob.



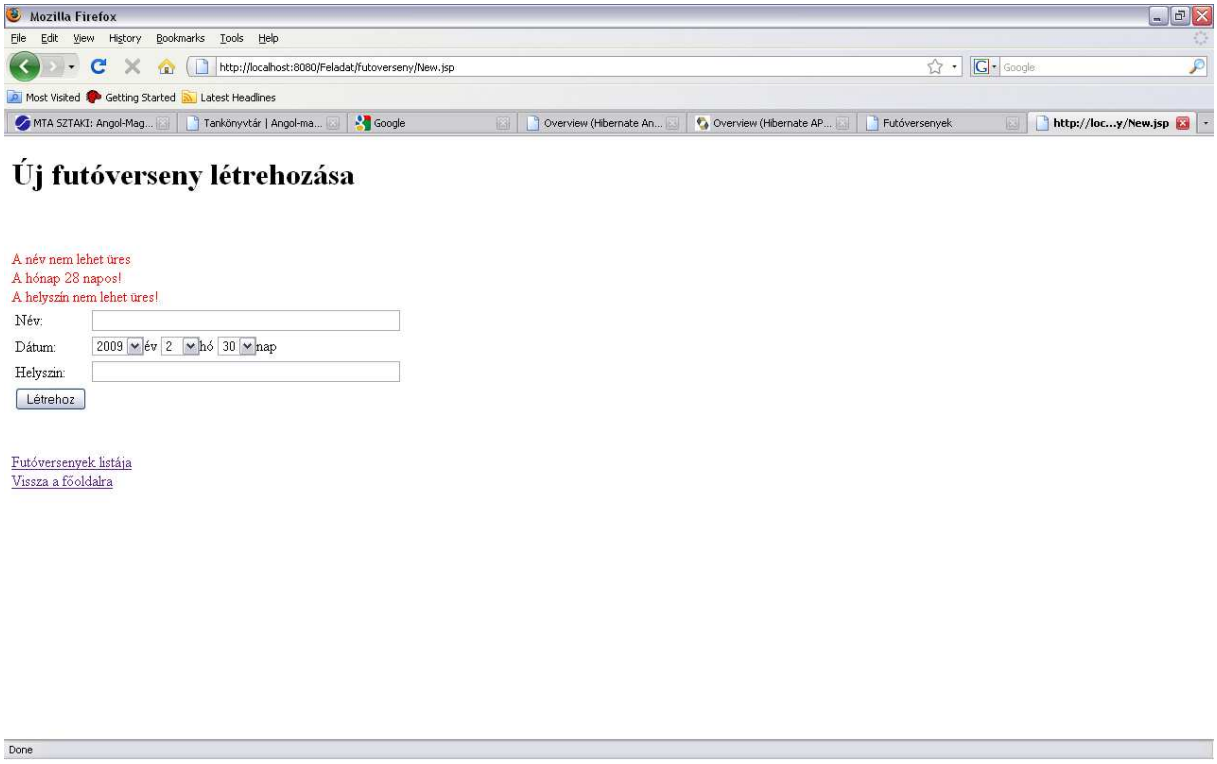
A programhoz tartozó `hibernate.cfg.xml`*:*

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.dialect">org.hibernate.dialect.DerbyDialect</property>
    <property name="hibernate.connection.driver_class"> org.apache.derby.jdbc.ClientDriver
          </property>
    <property name="hibernate.connection.url"> jdbc:derby://localhost:1527/futo_adatb </property>
    <property name="hibernate.connection.username">f</property>
    <property name="hibernate.connection.password">f</property>
    <property name="hibernate.show_sql">true</property>
    <property name="hibernate.current_session_context_class">thread</property>
    <mapping class="pa.Futo" />
    <mapping class="pa.Futam" />
    <mapping class="pa.Futoverseny" />
  </session-factory>
</hibernate-configuration>
```

Az alkalmazás kezdőoldala:



Hibaüzenet megjelenítése:



6. Összefoglalás

A dolgozatomban tárgyalt technológiák összetettek, jól konfigurálhatóak. Mindegyiknek megvannak az előnyei, esetleg hátrányai. Megpróbáltam egy átfogóbb betekintést nyújtani az eszközrendszerek használatába, mikor melyiket érdemes használni, és miért.

Azt mondhatjuk, hogy a JDO az egyik olyan technológia, mely a legtöbb lehetőséget kínálja fel: konkurenciakezelés, relációs és nem relációs tárolási módok, lekérdezések, nagy adathalmaz, Java objektumok, kedvező objektumorientált gondolkodás, tranzakciókezelés, platform függetlenség, egyszerűség.

Függelék

I.2. Mapping fájl létrehozása:

Egy egyszerű Java osztály:

```
package pelda;
public class Example{
        …
        private Long id;
        private String s;
        …
}
```

Az osztályhoz tartozó mapping fájl:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
<hibernate-mapping>
 <class name= "pelda.Example" table= "examples">
         <id name= "id" type="Long" unsavedValue=”null”>
                  <generator class="native"/>
        </id>
        <property name= "s" type= " string" length="40" not-null="true" unique="false" />
</class>
</hibernate-mapping>
```

Köszönetnyilvánítás

Szeretnék köszönetet mondani a témavezetőmnek, Espák Miklósnak, aki észrevételeivel, tanácsaival segítette a szakdolgozat létrejöttét.

Irodalomjegyzék

ORM:

Christian Bauer, Gavin King: Java Persistence with Hibernate

JDO:

http://java.sun.com/developer/technicalArticles/J2SE/jdo/

Hibernate:

http://hibernate.org

https://www.hibernate.org/hib_docs/v3/api/

http://docs.jboss.org/hibernate/stable/core/reference/en/html/

http://docs.jboss.org/hibernate/stable/annotations/reference/en/html/

Christian Bauer, Gavin King: Java Persistence with Hibernate

Java Persistence API:

http://edocs.bea.com/kodo/docs41/full/html/ejb3_overview.html

http://docs.jboss.org/hibernate/stable/annotations/reference/en/html/