A Hibernate egy objektum-relációs leképezést (ORM) megvalósító programkönyvtár Java platformra (a Hibernate-nek létezik egy .NET platformra szánt verziója is NHibernate néven). Segítségével osztályokat és a relációs adatbázisok tábláit tudjuk egymásba leképezni, az adatbázisban lévő rekordokat objektumokként kezelhetjük, és az objektumainkat egyszerűen tárolhatjuk állapotmegőrző módon adattáblákban. A Hibernate egy adatlekérdező nyelvet is biztosít (HQL - Hibernate Query Language), melynek segítségével adatbázis-kezelő rendszerek között hordozható lekérdezéseket írhatunk (ugyanakkor támogatja a natív SQL lekérdezések írását is). A HQL lekérdezésekből a Hibernate generálja az alkalmazás alatt lévő adatbázis kezelő rendszernek megfelelően az SQL lekérdezéseket, és megszabadítja a fejlesztőt az eredményhalmazok objektumokká történő konverziójának nehézségeitől. A Hibernate használható önálló Java alkalmazásokban is, ugyanakkor tipikusan Java EE környezetben alkalmazzák.

**Osztályok leképzése adattáblákra:**

A leképezés az osztályok és az adattáblák között XML fájlok vagy annotációk segítségével történik. Ezen metainformációk segítségével a Hibernate létre is tudja hozni az adatbázisban a megfelelő táblákat. Lehetőség van egy-a-többhöz és több-az-egyhez típusú kapcsolatok leképezésére is. A Hibernate a fejlesztő számára transzparens módon biztosítja a POJO-k (Plain Old Java Object) perzisztenciáját. Az egyetlen fontos követelmény, hogy az osztálynak rendelkeznie kell egy argumentum nélküli konstruktorral, valamint ajánlott, hogy felüldefiniáljuk az equals() és hashCode() metódusokat. A táblák közötti egy-a-többhöz kapcsolat Hibernate entitásban tipikusan valamilyen kollekció (Set vagy List) segítségével képeződik le, megadható, hogy ezek a kollekciók lusta (lazy) vagy mohó (eager) módon töltődjenek be. A lusta betöltés (lazy loading) esetén a kollekció tartalma csak akkor inicializálódik (tehát akkor fut le a benne lévő objektumoknak megfelelő rekordokat beolvasó lekérdezés), amikor először hivatkozunk rá, a mohó betöltésnél pedig azonnal a kollekciót tartalmazó objektum betöltésekor. Szintén megadható, hogy a szülő objektumon végzett műveletek esetén a hozzá kapcsolódó objektumokkal mi történjen. Például, ha van egy szülő objektumunk, a Megrendelés, és ez tartalmaz néhány MegrendelésElem objektumot, akkor lehetőség van arra, hogy a Megrendelés mentése illetve törlése esetén a benne lévő MegrendelésElem objektumok is mentődjenek illetve törlődjenek. Létezik beépített „dirty check”, azaz a Hibernate csak azokat az objektumokat írja vissza az adatbázisba, amelyek valóban változtak, megakadályozva ezzel a felesleges beszúrásokat.

**Hibernate konfiguráció:** Egy xml állományban konfigurálhatjuk a Hibernatet.

Mivel a Hibernate az alatta lévő JDBC-t használja, ezért szükség van információkra, melyekkel használhatja a JDBC-t. Ilyen az url, vagy a username.

<session-factory>

<property name="hibernate.connection.driver\_class">com.mysql.jdbc.Driver</property>

<property name="hibernate.connection.url">jdbc:mysql://localhost:3306/animegame?useUnicode=true&amp;useJDBCCompliantTimezoneShift=true&amp;useLegacyDatetimeCode=false&amp;serverTimezone=UTC</property>

<property name="hibernate.connection.username">root</property>

<property name="hibernate.connection.password"></property>

<property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>

<property name="show\_sql">true</property>

<property name="connection.pool\_size">1</property>

<property name="hibernate.current\_session\_context\_class">thread</property>

</session-factory>

**XML-ben beállított tulajdonságok:**

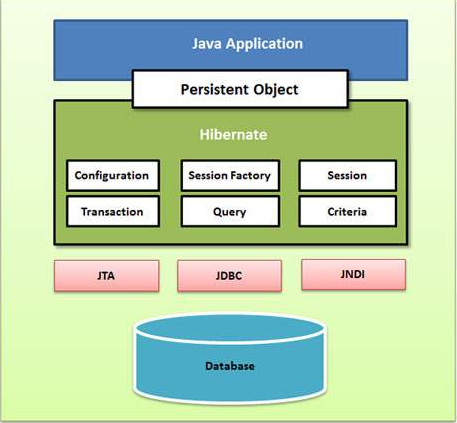
* hibernate.connection.driver\_class:Itt adjuk meg a drivert
* hibernate.connection.url:Az adatbázis címe
* hibernate.connection.username:
* hibernate.connection.password:
* hibernate.connection.pool\_size:Kapcsolatok száma
* hibernate.connection.autocommit:Autocommit módot szabályozhatjuk

**Szükséges Jarok:** Ahhoz, hogy használhassuk a Hibernate álltal nyújtott szolgáltatásokat, nem elég csak a Hibernate maven függőségét megadni, hanem adatbázisspecifikus (oracle,mysql) JDBC jar maven függőségét is meg kell adni, hisz a Hibernate azzal fog dolgozni.

**ORM (Object-Relational Mapping) fogalma:** Ez egy programozási technika, amelynek segítségével objektumokat alakíthatunk át relációs modellben megfelelő adatstruktúrára. Objektum adattábla leképezés történik. Ennek előnyei:

* Üzleti logika objektumokkal dolgozhat, nem pedig táblákkal.
* Elrejti az SQL részleteit.
* JDBC-re épít, de elrejti előlünk
* Gyorsabb fejlesztés

**Hibernate architektúra:**Réteges felépítésű, így az alacsonyabb szintű API-kat elrejti a programozók elől.



**Konfigurációs objektum:** Ez az első Hibernate objektum, amit létre kell hoznunk. Egy alkalmazáson belül elég egyszer létrehozni.

SessionFactory object: Konfigurálja a Hibernatet, a leíró xml állományt felhasználva. Elkészít számunkra egy Session objektumot. Ez egy szál biztos objektum. Ahány különböző adatbázist szeretnénk kezelni, értelem szerűen annyi SessionFactory objektumot kell inicializálnunk.

Session objektum: Ez a fizikai kapcsolat az adatbázissal. Objektumok lekérdezése és kimentése ezen objektumon keresztül történik. A Session objektumok nem szál biztosak, ezért csak arra az időre hagyjuk nyitva, amíg szükséges, akkor hozzuk létre, mikor használni szeretnénk, majd szabadítsuk fel. Használod, majd felszabadítod.

SessionFactory factory = **new** Configuration().

configure("hibernate.cfg.xml").

addAnnotatedClass(Student.**class**).

buildSessionFactory();

Session session = factory.getCurrentSession();

A fent látható kóddal valósítható meg a két objektum. A configure() metódusnak átadjuk értékül az xml konfigurációs fájlt, ha nem adunk meg semmit, akkor az értelmező a CLASSPATH-ból próbálja megkeresni a hibernate.cfg.xml állományt. Az adAnnotationClass() értelem szerűen a felannotált osztály típusát adhatja meg, ez többször is ráhívható egymásra, hisz több annotált osztály is előfordulhat. A buildSessionFactory értelem szerűen felépíti a SessionFactory objetumot.

**Példa egyszerű objektum kimentésére adatbázisba:**

SessionFactory factory = **new** Configuration().

configure("hibernate.cfg.xml").

addAnnotatedClass(Student.**class**).

buildSessionFactory();

Session session = factory.getCurrentSession();

**try**{

Student student1 = **new** Student("Viktor", "Kisz", "viktor123@vipmail.hu");

Student student2 = **new** Student("Viktor", "Kisz", "viktor123@vipmail.hu");

Student student3 = **new** Student("Viktor", "Kisz", "viktor123@vipmail.hu");

session.beginTransaction();

session.save(student1);

session.save(student2);

session.save(student3);

session.getTransaction().commit();

}**catch**(Exception e){

System.***out***.println(e.getMessage());

}**finally**{

factory.close();

}

Látható, hogy try-catch blokkban kezeljük. A beginTransaction metódussal kezdjük el a tranzakciót, majd a save metódusoknak átadjuk a létrehozott objektumokat, amelyeket a háttérben egy insert metódussal fog a Hibernate perzisztálni. Majd commitáljuk a változásokat. A finally blokkban pedig lezárjuk a kapcsolatot.

Látható, hogy az objektumoknak 3 attribúmos konstruktorát hívom meg, pedig van egy id adattag is, amely a primary keykent funkcionál ehhez az objektumhoz. Ezt azért nem mi adjuk meg, mert kihasználjuk a háttérben a MySQL auto\_increment integritási feltételt, amely minden új elem beszúrásánál azt jelenti, hogy a primary key értékét eggyel növeli.

**Session objektum rendes használata:**

session.beginTransaction();

Student student = **new** Student("Kiss", "Viktor", "viktor123", **null**);

session.save(student);

session.getTransaction().commit();

/\*Lezárjuk a session objektumot\*/

session.close();

/\*Újra igénylünk egy session objektumot\*/

session = factory.getCurrentSession();

session.beginTransaction();

Student st = session.get(Student.**class**, 1);

session.close();

**Adatmodell objektum annotálása:**

@Entity

@Table(name="student")

**public** **class** Student {

@Id

@GeneratedValue(strategy=GenerationType.***IDENTITY***)

@Column(name="id")

**private** **int** id;

@Column(name="firstname")

**private** String firstName;

@Column(name="lastname")

**private** String lastName;

@Column(name="email")

**private** String email;

**public** Student() {

// **TODO** Auto-generated constructor stub

}

**public** Student(String firstName, String lastName, String email) {

**super**();

**this**.firstName = firstName;

**this**.lastName = lastName;

**this**.email = email;

}

Mi látható a fenti példán?

A transzormálandó osztálynak kell, hogy legyen paraméter nélküli konstruktora. Az osztály előtt két annotációt adunk meg, melyekkel jelezzük, hogy ez egy Relációs táblával fog megfelelni. A tábla neve a @Table annotációban van megadva. A @Entity pedig kell még plusszba hozzá(ez jelzi hogy entity beanről lesz szó, és van paraméter nélküli konstruktora).

Az adattagok lesznek a tábla oszlopai, azokat a @Column annotációval láthatjuk el, paraméterként megadva a nevüket.

A Primary Key megadása érdekesebb feladatnak minősül: Egyrészről annak az adattagnak a @Id annotációval kell rendelkeznie pluszban a @Column annotáción kívül.

A @GeneratedValue(strategy=GenerationType.***IDENTITY***)

Ez az annotáció állítja be azt a stratégiát, hogy miszerint generáljon értéket a Hibernate a primary Keynek. A fenti megoldás eredménye az, hogy a MySQL auto\_incrementet használja.

A GenerationType-nak nem csak az IDENTITY enum értéke létezik:

* AUTO:
* IDENTITY:MySQL specifikus auto\_increment
* SEQUENCE: Ez az oracles megoldáshoz kell
* TABLE:

**Objektum lekérdezése (retrieve):**

A Session objektumon keresztül történik ez is. Ha nem talál a primary keyyel megfeleltethető mezőt a táblában, akkor null-t ad vissza a get() metódus. Az primary keynek így majd a Hibernate ad egy értéket, és nem mi fogjuk meghívni a setId() vagy bármely setter metódusát, amikor a save() hibernate metódust használjuk, így a lekérdezéskor nyugodtan hívhatjuk a primary key getter metódusát, nem nullal fog visszatérni.

**try**{

Student student1 = **new** Student("Viktor", "Kisz", "viktor123@vipmail.hu");

session.beginTransaction();

session.save(student1);

session.getTransaction().commit();

session = factory.getCurrentSession();

session.beginTransaction();

Student res = session.get(Student.**class**, student1.getId());

session.getTransaction().commit();

System.***out***.println(res);

}**catch**(Exception e){

System.***out***.println(e.getMessage());

}**finally**{

factory.close();

}

Látható, hogy külön beginTransactionben(külön tranzakcióként kell) kezelni az írást és az olvasást is. Ez a megoldás egyetlen elem lekérdezésére jó, primary key alapján.

**HQL (Hibernate Query Language):** Ez az objektum orientált lekérdező nyelve a Hibernatenak. Ahelyett, hogy táblákkal és oszlopokkal dolgozna, itt perzisztes objektumokkal és azok attribútumaival dolgozik. Ezt persze a Hibernate átalakítja natív SQL lekérdezésre. Használhatunk persze natív SQL lekérdezéseket is a Hibernateval párhuzamosan, csak ezt nem javasolja senki, mivel szinkronizációs probléma léphet fel, illetve bízzuk nyugodtan magunkat a Hibernatera, mivel optimalizált lekérdezéseket hat végre, és alkalmaz cache mechanizmust is.

A FROM kulcsszó: Szeretném behúzni a memóriába a tábla összes tartalmát.

Az összes Student lekérdezése:

List<Student> students = session.createQuery("from Student").getResultList();

A session objektum createQuery() metódusának értékül adtuk a from kulcsszót és mögötte az osztály nevét(igen itt java osztály nevet kell megadni, szóval itt akár csomagnévvel együtt is megadható). A getResultList() metódus fogja előállítani belőle a listát. Ezzel lekérdeztük az összes elemet a Student táblából.

Csak azon tanulók lekérdezése, akiknek a vezetéknevük Nagy:

List<Student> students = session.createQuery("from Student s where s.lastName='Nagy'").getResultList();

Látható, hogy a where kifejezés is belemehet. Itt ugyebár már aliasokat is használtam, illetve fontos tudni, hogy az alias attribútumának a neve az az osztály adattag neve kell, hogy legyen. Tehát a Student osztályban nekem van egy lastName attribútumom.

SELECT használata: Lényege, hogy ne az egész objektumot (összes oszlop) kérjük le, hanem csak adott attribútumot.

List<String> students = session.

createQuery("SELECT s.lastName FROM Student s where s.lastName='Nagy'").getResultList();

Természetesen használhatunk orderby és groupby lekérdezéseket is.

**Elnevezett paraméterek használata:** Ezt a módszert használva nem kell védekeznünk az SQLInjectionel szemben.

String hql = "FROM Employee E WHERE E.id = :employee\_id";

Query query = session.createQuery(hql);

query.setParameter("employee\_id",10);

List results = query.list();

Ez a megoldás a háttérben a PreparedStatementre épít.

**Update lekérdezések megvalósítása:**

String hql = "UPDATE Employee set salary = :salary " +

"WHERE id = :employee\_id";

Query query = session.createQuery(hql);

query.setParameter("salary", 1000);

query.setParameter("employee\_id", 10);

int result = query.executeUpdate();

System.out.println("Rows affected: " + result);

**Delete lekérdezés megvalósítása:**

String hql = "DELETE FROM Employee " +

"WHERE id = :employee\_id";

Query query = session.createQuery(hql);

query.setParameter("employee\_id", 10);

int result = query.executeUpdate();

System.out.println("Rows affected: " + result);

**Insert lekérdezés megvalósítása:** Ez a save() már említett metódussal történik.

**Beágyazott objektumok kezelése:** Mivan akkor, ha a az osztály amiből táblát szeretnénk(@Entity), abban nem csak elemi adattípusok vannak, hanem más objektumok is?

Ha létrehozunk egy osztályt az kétféle lehet: Vagy @Entity, ilyenkor külön tábla lesz belőle, vagy lehet @ Embeddable, ilyenkor az ebből készült objektumok már egy tábla részei lesznek.

@Entity

@Table(name="Student")

**public** **class** Student {

@Id

@GeneratedValue(strategy=GenerationType.***IDENTITY***)

@Column(name="id")

**private** **int** id;

@Column(name="firstname")

**private** String firstName;

@Column(name="lastname")

**private** String lastName;

@Column(name="email")

**private** String email;

@Embedded

**private** Address address;

Látható, hogy a Studentnek van egy Address objektuma, aminél jelezzük, hogy ez beágyazott.

Lássuk annak az osztálynak a definícióját:

@Embeddable

**public** **class** Address {

@Column(name="cityty")

**private** String city;

@Column(name="num")

**private** String number;

@Column(name="r")

**private** String road;

Az osztálynál is jelezhetjük, hogy ez beágyazott lesz, és nem pedig külön tábla. Az adattagokat annotáljuk, hogy milyen névvel szerepeljenek (mi legyen az általuk képviselt oszlop neve).



A fenti osztály (Student) hatására ez a tábla jön létre. Látható, hogy a beágyazott osztály adattagjai is oszlopokként szerepelnek.

A hibernate.config.xml-ben

<property name=*"hibernate.hbm2ddl.auto"*>create</property>

Ezzel a sorral azt érjük el, hogy ha még nincs felvéve tábla az adatbázisba ezzel az annotációval, akkor a hibernate autómatikusan létrehoz táblát(Például ha van egy Student annotált soztályom, és hibernate műveletet csinálok(az lehet akár lekérdezés is), akkor a hibernate autómatikusan létrehozza azokat a táblákat, amelyeket még nem hozott létre).Ez csak tábla létrehozásra automatikus.

**Kollekciók kezelése Hibernate segítségével:** Az előző példában láthattuk, hogy Egy tábla annotációs osztály tartalmazhat objektumot, és hogy annak minden attribútumát külön oszlopként kezeljük. De mi van akkor, ha egy objektum listát tárolunk az adott osztályban adattagként?

A feladat a következő: A Student osztályunk a meglévő attribútumai mellé, egy olyan listát is tartalmaz, amelyben Address típusú objektumokat tárol.

@Entity

@Table(name="StudentWithList")

**public** **class** Student {

@Id

@GeneratedValue(strategy=GenerationType.***IDENTITY***)

@Column(name="id")

**private** **int** id;

@Column(name="firstname")

**private** String firstName;

@Column(name="lastname")

**private** String lastName;

@Column(name="email")

**private** String email;

@ElementCollection

@JoinTable(name="StudentAddress",

joinColumns = @JoinColumn(name = "Student\_ID")

)

@GeneratedValue(generator = "increment")

@GenericGenerator(name = "incrementName", strategy = "increment")

@CollectionId(columns = { @Column(name = "ADRESS\_ID") },

generator = "incrementName",

type = @Type(type = "long"))

**private** List<Address> address = **new** ArrayList<Address>();

Látható, hogy a kollekciót kicsit több annotációval kellett ellátni, mint eddigi társait. Lássuk mi mit jelent:

- @ElementCollection: Ez jelzi, hogy itt kollekció elemről lesz szó.

- @JoinTable: Mivel ezt a megoldást úgy oldja meg a Hibernate, hogy külön táblát hoz létre ennek a listában tárolandó objektumnak, majd abban minden elhelyezett elem egy idegenkulcson keresztül mutat a Student primary key-ere. A name attribútum azt adja meg, hogy mi legyen ennek az új táblának a neve(annaka táblának amiben az Addressek vannak). A joinColums sor azt határozza meg, hogy a foreign kulcs oszlopának mi legyen a neve(ha ez nem szerepel, akkor generált nevet ad neki).

- @GeneratedValue: Mivel ezeknek az Adsresseknek is szükségük van valamilyen egyedi azonosítóra(primary key), ezért meg kell határoznunk a Hibernate számára, hogy milyen stratégia alapján generálja le ezt neki. Egyébként a Halmaz kollekciókra ez nem működik, mert ott nincs sorrendiség a tárolásban. Ezzel az annotációval beállítom, hogy növelje az id-k értéket, amolyan auto increment.

- @GenericGenerator: Ez az annotáció lesz egy típusdefiníció, amivel leírjuk magát a generálási stratégiát. A name az a neve lesz amivel hivatkozhatunk rá a későbbiekben, a strategy attribútum pedig az előző generator annotációban megfogalmazott stratégia.

- @CollectionId: Ez az annotáció állítja be, hogy a kollekciónak legyen primary keye, és azt milyen stratégiával generálja a Hibernate.A column name értelem szerűen a primary key oszlopának nevét állítja. A generator attribútumnak értékül adom a @GenericGenerator annotációval definiált generálási stratégiát, ami itt növekvő generálás lesz. A type attribútummal a mező típusát állíthatjuk. Erről az annotációrpl azt kell tudni, hogy ez nincs a JPA interfészben definiálva, ezt a Hibernate implementáció kiegészítésként tette hozzá, így más JPA implementációkban ezt nem biztos, hogy megtaláljuk.

Ezen annotációk használata után lássuk milyen lekérdezéseket adott ki a háttérben a Hibernate:

Hibernate: insert into StudentWithList (email, firstname, lastname) values (?, ?, ?)

Hibernate: select max(ADRESS\_ID) from StudentAddress

Hibernate: insert into StudentAddress (Student\_ID, ADRESS\_ID, cityty, num, r) values (?, ?, ?, ?, ?)

Hibernate: insert into StudentAddress (Student\_ID, ADRESS\_ID, cityty, num, r) values (?, ?, ?, ?, ?)

Hibernate: insert into StudentAddress (Student\_ID, ADRESS\_ID, cityty, num, r) values (?, ?, ?, ?, ?)

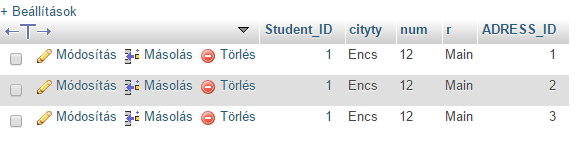
Persze ezek előtt a táblákat is létrehozta, amelyeket az alábbi képen illusztrálok:



A két legenerált osztályt láthatjuk a fenti képen.



Ezen a képen a Student tábla tartalmát láthatjuk.



A fenti képen pedig az Address listának létrehozott táblát láthatjuk, ahol a foreign key neve az amit mi annotációban beállítottunk, a generált primary keyek növelő stratégiával készültek.

**Lazy és Eager inicializáció Hibernateban:** Az előző fejezetben láthattuk a Student osztályunkat, aminek vannak primitív típusú adattagjai, illetve egy Address listája is.

Student st = session.get(Student.**class**, 1);

A fenti lekérdezést a Hibernate, hogy oldja meg a háttérben? Ugyebár a Student osztálynak az alap attribútumai mellett(keresztnév, email), van egy lista adattagja is, mi van akkor ha a listában több ezer elem van? A Hibernate minden ilyen lekérdezéskor automatikusan lekérdezi azokat is? A válasz az, hogy nem. A Hibernate alapértelmezetten Lazy inicializálást használ, ami azt jelenti, hogy a lekérdezésekkor csak a first level (nem külön objektum) attribútumokat inicializálja, a többit csak szükség esetén. Mit jelent az, hogy csak szükség esetén?

st.getAddress().size();

Ha meghívjuk a listára mutató getter metódust, ilyenkor fogja a háttérben lekérdezni a lista elemeit. Tehát azért lusta, mert csak akkor inicializál, amikor arra tényleg szükség van. Az Eager inicializálás ezzel szemben azt jelenti, hogy session.get() hatására mindent lekérdez, legyen az bármilyen erőforrás igényes, és időt pazarló.

De hogyan oldja meg ezt a lusta inicializálást a Hibernate, hisz láthatjuk, hogy

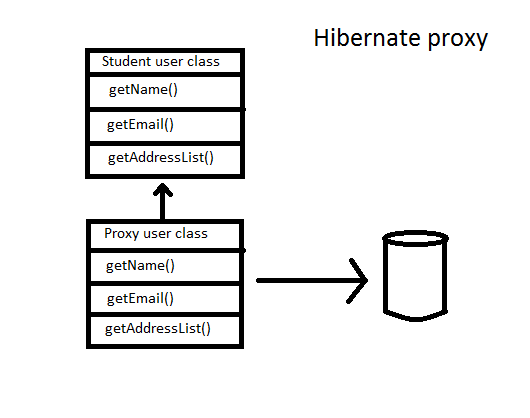
**public** List<Address> getAddress() {

**return** address;

}

ennek a metódusnak a hatására a háttérben ő elindít egy lekérdezést, amiben feltölti a listát, és visszaadja. Hogyan lehetséges ez?

A megoldást a Proxy osztály jelenti. Ahogy az alábbi ábrán is látható, A Hibernate készít egy leszármazást a mi osztályunkból, majd abból példányosít. És amikor a lekérdezés eredményét megkapjuk, akkor a proxy osztályból kapunk egy példányt. Ez azért lehetséges, mert a proxy osztály ugyebár mindent tud, amit a mi osztályunk, csak kiegészíti annyival, hogy a lista getter metódust úgy definiálja felül, hogy először lekérdez az adatbázisból, majd feltölti az adattagot (jelen esetben a listát), majd utána hívja tovább az ős getter metódusát, ami visszaadja a listát. Ezért van az, hogy gettert hívunk, és a háttérben az mégis tud lekérdezést végrehajtani.



Tehát mi nem a saját osztályunk példányát, hanem a proxy osztály egy példányát kapjuk meg, és használjuk. Hogyan lehetne ezt bebizonyítani?

session = factory.getCurrentSession();

session.beginTransaction();

Student st = session.get(Student.**class**, 1);

session.close();

System.***out***.println(st.getAddress().size());

A fenti kódban az történik, hogy a Session objektumtól és a Proxy Student osztály kapom meg, így annak a getAddress() metódusát hívom. Ezért még a lista gettere előtt lezárom a sessiont, így a háttérben a getter metódus nem fog tudni lekérdezést kiadni, így a kiírásnál hiba kell, hogy dobódjon.

„failed to lazily initialize a collection of role: modell.Student.address”

A fent látható hibaüzenetet kapjuk tehát.

**One to One kapcsolat:**

session.beginTransaction();

Student st = **new** Student("kiss", "vik", "123");

NeptunCode code = **new** NeptunCode("Miskolc");

code.setStudent(st);

session.save(code);

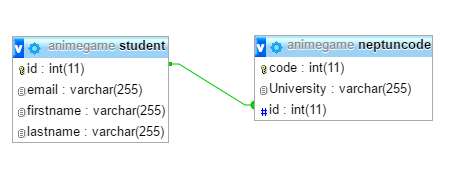
session.getTransaction().commit();

session.close();

A lekérdezése:

NeptunCode code = session.get(NeptunCode.**class**, 3);

System.***out***.println(code.toString());



A fenti képen szereplő relációt szeretnénk megvalósítani(ont to one kapcsolat). Lényege, hogy a studentnek van 4 attribútuma, abból egy a primary key(id), illetve van egy neptuncode is, aminek szintén van egy primary keye(code), illetve egy idegen kulcsa, ami a student id attribútumára mutat. Az alábbiakban ezt az 1-1 kapcsolatot fogom szemlélteni Hibernate keretrendszer segítségével.

A Student osztály:

@Entity

@Table(name="Student")

**public** **class** Student {

@Id

@GeneratedValue(strategy=GenerationType.***IDENTITY***)

@Column(name="id")

**private** **int** id;

@Column(name="firstname")

**private** String firstName;

@Column(name="lastname")

**private** String lastName;

@Column(name="email")

**private** String email;

Látható, hogy itt csak a saját típusai vannak, tehát nem a Student fogja tartalmazni adattagként a neptuncodeot, hanem fordítva. Ez azért logikus, mert a neptuncode tartalmaz lényegében idegenkulccsal hivatkozást a studentre. Ha a student tartalmazná java osztály szinten a neptuncode-ot, akkor az relációs táblában úgy nézne ki, hogy a student foreign keye mutatna a neptuncode primary keyere.

@Entity

@Table(name="NeptunCode")

**public** **class** NeptunCode {

@Id

@GenericGenerator(name="foreignKeyGeneratorOwn", strategy="foreign",

parameters = {@Parameter(value="student", name="property")})

@GeneratedValue(generator="foreignKeyGeneratorOwn")

**private** **int** code;

@Column(name = "University")

**private** String university;

@OneToOne(cascade=CascadeType.***ALL***)

@JoinColumn(name="id")

**private** Student student;

A neptuncoden-nak három attribútuma van. A saját ideje(code), az hogy melyik egyetemhez tartozik, illetve egy referencia a Studentre.

Mivel