İleri Düzey JPA/Hibernate Eğitimi 2

harezmi bilişim çözümleri

Persistent Nesnelere Erişim ve Proxy

JAVA Eğitimleri

- Veritabanına **gereksiz erişimden kaçınmak** içindir
- Proxy, asıl nesneye gerçekten erişim ihtiyacı doğana kadar **veritabanı erişimini öteler**
- Proxy nesne **asıl nesnenin yerine** geçer
- Proxy, asıl nesneye **sadece refer etmek** gerekiyorsa çok faydalıdır, (örneğin M:1 ve 1:1 ilişkileri oluşturmak için)

Persistent Nesnelere Erişim ve Proxy

JAVA Eğitimleri

- Hibernate “**class proxy**” üretmektedir
- Proxy nesnenin yaratılacağı sınıf **SessionFactory build aşaması**nda üretilir
- Proxy nesneler target nesne ile **aynı sınıftan türer**ler ve **target nesneyi wrap eder**ler
- Ancak **proxy nesnedeki attribute'lar NULL** veya sınıf tanımında atanmış **initial değer**leri içerirler

Proxy Nesneler ve Object Equality

- Proxy nesneye bu attribute'ların değerleri için erişildiği vakit **gerçek değerler hedef nesneden getter metotları vasıtası ile** erişilip dönümektedir
- Dolayısı ile **equals** ve **hashCode** metotlarında nesnelerin attribute'larına erişimde **hep getter metotlar üzerinden** olmalıdır

Proxy Nesneler ve Object Equality

```
@Entity
public class User {
    @Id
    @GeneratedValue
    private Long id;

    private String username;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }
    ...
}
```

Proxy Nesneler ve Object Equality

Proxy nesnelerin yer aldığı equality kontrollerinde veya Collection API işlemlerinde sorun olacaktır!

```
@Entity
public class User {
    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null) return false;
        if (getClass() != obj.getClass()) return false;
        User other = (User) obj;
        if (username == null) {
            if (other.username != null)
            return false;
        } else if (!username.equals(other.username)) return false;
        return true;
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((username == null) ? 0 : username.hashCode());
        return result;
    }
}
```

Proxy Nesneler ve Object Equality


```
@Entity
public class User {
    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null) return false;
        if (!getClass().isAssignableFrom(obj.getClass())) return false;
        User other = (User) obj;
        if (getUsername() == null) {
            if (other.getUsername() != null)
            return false;
        } else if (!getUsername().equals(other.getUsername())) return false;
        return true;
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((getUsername() == null) ? 0 :
                                                getUsername().hashCode());
        return result;
    }
}
```

Proxy Kabiliyetinin Özelleştirilmesi

```
@Entity
@org.hibernate.annotations.Proxy(lazy = false)
public class Owner {

}
```

Entity'nin proxy özelliği devre dışı kalacaktır
Bu durumda session.load() metodu proxy dönmeyip get() metodu gibi anında DB'ye bir SELECT atacaktır
M:1 ve 1:1 lazy ilişkiler de EAGER şeklinde davranacaktır

```
@Entity
@org.hibernate.annotations.Proxy(
        proxyClass = OwnerInterface.class)
public class Owner {

}
```

Proxy sınıf üretilirken implement edilecek interface veya extend edilecek üst sınıf da belirtilebilir

Lazy İlişkiler ve Persistent Collection

JAVA Eğitimleri

- Hibernate 1:M ve M:N lazy ilişkileri yönetmek için entity nesneyi yüklerken lazy collection değişkenlerine değer olarak kendi **Persistent collection** sınıflarından uygun bir instance'ı atar
  - Set için PersistentSet
  - List için PersistentList
  - Bag için PersistentBag
  - Map için PersistentMap
- **Lazy collection içeriğine ihtiyaç olduğu vakit** ilgili persistent collection, elemanlarını DB'den **ayrı bir SELECT** ile yükler

Lazy İlişkiler ve Proxy

JAVA Eğitimleri

- Lazy olarak tanımlanmış M:1 ve 1:1 ilişkiler için ise entity yüklenirken M:1 veya 1:1 ilişkisi mevcut ise **hedef entity yerine geçecek bir proxy nesne** yaratılarak bu set edilir
- Entity'nin lazy M:1 veya 1:1 ilişkisine erişildiği vakit **proxy nesne ayrı bir SELECT ile** kendisini initialize edecektir
- Lazy ilişkilerin veya proxy'lerin yüklenebilmeleri için **source entity'nin yüklendiği Session'ın açık olması** gerekir

LazyInitializationException Hatası ve Önleme Yolları

JAVA Eğitimleri

- Aksi takdirde **LazyInitializationException** hatası ortaya çıkacaktır
- Genel olarak **uninitialized ve detached bir proxy** nesneye veya detached entity'nin **uninitialized lazy herhangi bir ilişkisine erişilirse** bu hata ortaya çıkar

LazyInitializationException Hatası ve Önleme Yolları

JAVA Eğitimleri

- Lazy hatası ile karşılaşmamak için değişik çözüm yolları mevcuttur:
- Proxy'nin veya persistent collection'ın herhangi **bir property'sine Session açık iken erişilebilir**
- Hibernate.**initialize**(Object **proxy**) ile proxy nesne veya persistent collection **Session kapanmadan önce** initialize edilebilir
- **Detached entity** veya proxy açık Session'a **reattach** yapılabilir

LazyInitializationException Hatası ve Önleme Yolları

JAVA Eğitimleri

- Spring ile çalışırken **OpenSessionInViewFilter/OpenEntityManagerInViewFilter** kullanılabilir
- Lazy ilişki "**fetch join**" ile eager initialize edilebilir
- Hibernate 4 ile gelen **hibernate.enable_lazy_load_no_trans** property değeri "**true**" yapılabilir

Hibernate Extra Lazy Kabiliyeti

```
@OneToMany
@LazyCollection(LazyCollectionOption.EXTRA)
private Set<Pet> pets = new HashSet<Pet>();
```

- Hibernate'e özel "**extra lazy**" collection eşleme yöntemi kullanılırsa **size()**, **contains()**, **isEmpty()** gibi metotlara erişim de **initialization'ı tetiklemez**
- Sadece **ilgili bilgi** için DB'ye bir sorgu atılır
- Collection tipi Map veya List ise **containsKey()** ve **get()** metodları için de doğrudan DB'ye erişilir

1:M – Bidirectional İlişkiler ve mappedBy

```
public class Owner {

    @OneToMany(mappedBy = "owner")
    private Set<Pet> pets = new HashSet<Pet>();

}
```

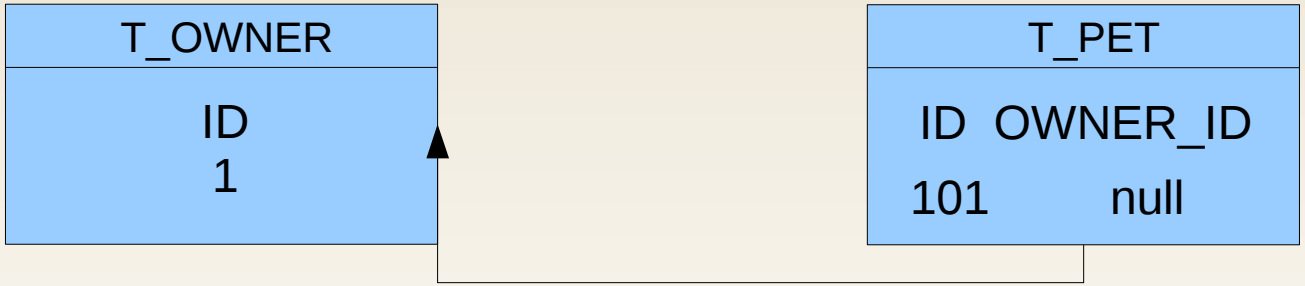
mappedBy attribute çift yönlü ilişkide, ilişkiyi yöneten tarafın kim olduğunu tanımlar

mappedBy attribute olmadığı takdirde aynı FK'yı güncellemeye çalışan **iki farklı SQL ifadesi** söz konusu olurdu

JoinColumn ve **JoinTable** annotasyonları mappedBy ile işaret edilen tarafta tanımlanmalıdır

```
public class Pet {

    @ManyToOne
    @JoinColumn(name = "OWNER_ID")
    private Owner owner;

}
```

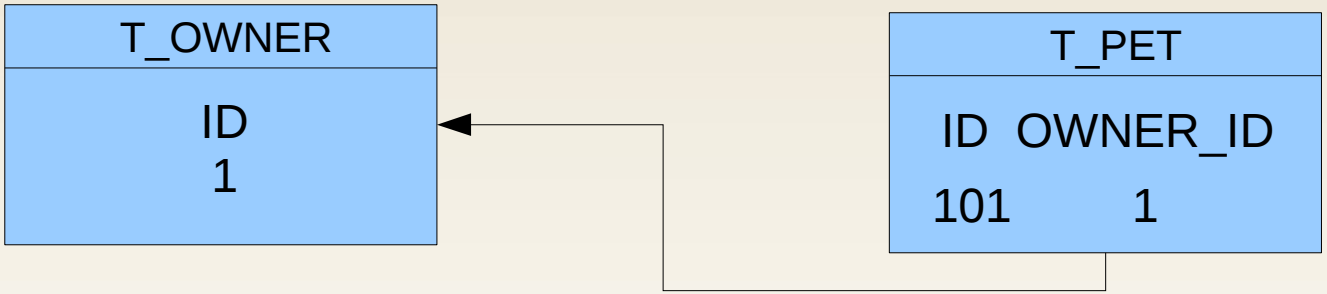
mappedBy'ın İlişki Kurma ve Kaldırmaya Etkisi




```
session.beginTransaction();

Owner owner = session.get(Owner.class, 1L);

Pet pet = session.get(Pet.class, 101L);

owner.getPets().add(pet);

session.getTransaction().commit();
```

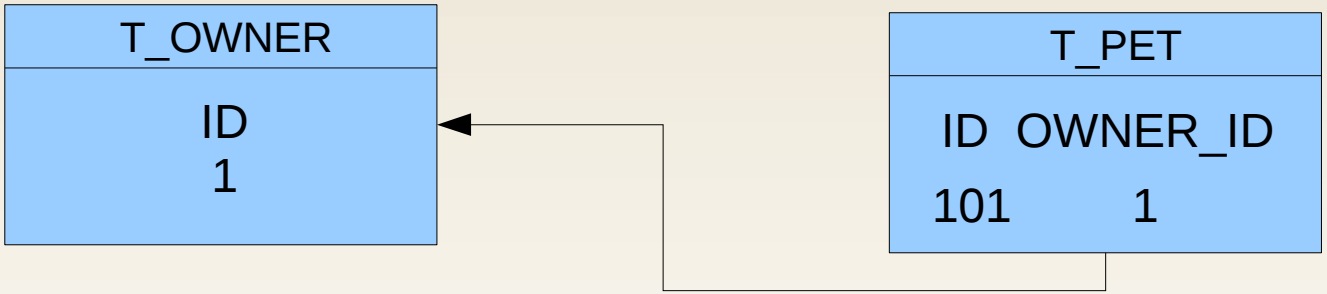
İlişki kurulmaz!

mappedBy'ın İlişki Kurma ve Kaldırmaya Etkisi




```
session.beginTransaction();

Owner owner = session.get(Owner.class, 1L);

Pet pet = session.get(Pet.class, 101L);

pet.setOwner(owner);

session.getTransaction().commit();
```

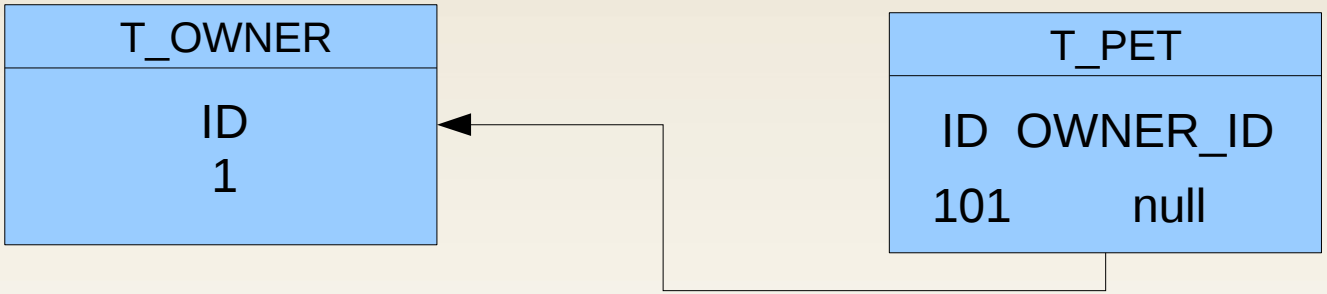
mappedBy'ın İlişki Kurma ve Kaldırmaya Etkisi




```
session.beginTransaction();

Owner owner = session.get(Owner.class, 1L);

Pet pet = session.get(Pet.class, 101L);

owner.getPets().remove(pet);

session.getTransaction().commit();
```

İlişki kaldırılmaz!

mappedBy'ın İlişki Kurma ve Kaldırmaya Etkisi




```
session.beginTransaction();

Owner owner = session.get(Owner.class, 1L);

Pet pet = session.get(Pet.class, 101L);

pet.setOwner(null);

session.getTransaction().commit();
```

İlişki kaldırılır

1:1 İlişkiler ve Performans

- Ayrı bir **foreign key** ile 1:1 ilişki yönetilmesi **veri depolama ve indeksleme maliyetleri** açısından PK yöntemine göre daha kötüdür
- Ayrıca **JPA'da** 1:1 ilişki de **parent tarafı optional=false** olarak belirtilse bile ilişkiyi **LAZY** tanımlamak mümkün değildir
- 1:1 İlişkiler için önerilen mapping yöntemi **primary key üzerinden ilişki kurmak**tır

Primary Key Üzerinden 1:1 İlişki Tanımı

```
@Entity
public class Owner {
    @Id
    @GeneratedValue
    private Long id;
}

@Entity
public class Address {

    @Id
    @GeneratedValue(generator="fkGenerator")
    @GenericGenerator(
            name="fkGenerator",
            strategy="foreign",
            parameters=@Parameter(name="property",value="owner"))
    private Long id;

    @OneToOne
    @PrimaryKeyJoinColumn
    private Owner owner;
}
```

JPA 2.0 öncesi yöntem, Hibernate spesifik foreign key generator stratejisi üzerine kurulu idi

@MapsId ile Primary Key Üzerinden 1:1 İlişki Tanımı

```
@Entity
public class Owner {
    @Id @GeneratedValue
    private Long id;
}

@Entity
public class Address {
    @Id
    private Long id;

    @OneToOne
    @MapsId
    private Owner owner;

}
```

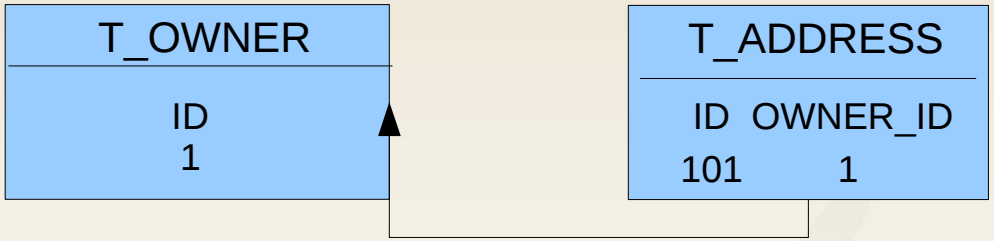
Address entity'sinin PK generation stratejisinin assigned bırakılması gerekir

İlişkili owner entity'sinin PK değerinin Address entity'sinin de PK değeri olmasını sağlar

JPA 2.0 da gelmiştir

1:1 İlişkiler ve Lazy

```
@Entity
public class Owner {
    @Id @GeneratedValue
    private Long id;

    @OneToOne(mappedBy="owner",fetch=FetchType.LAZY)
    private Address address;
}

@Entity
public class Address {
    @Id
    @GeneratedValue
    private Long id;

    @OneToOne
    @JoinColumn(name = "OWNER_ID")
    private Owner owner;
}
```



Owner nesnesi yüklenirken address property'si NULL'mı bırakılacak, yoksa bir proxy address nesnesi mi set edilecek bunun kararına varabilmek için T_ADDRESS tablosuna da bakılması gerekir. Dolayısı ile Address bilgisine bu aşamada erişilmiş olacaktır. **optional=false** set edilir ise bu durumda Hibernate ilişkinin zorunlu olduğunu kabul edip bir proxy nesne set edecektir. **Hibernate 5.x JPA uyumluluğu nedeni ile optional=false yapılsa bile LAZY yapmamaktadır.**

1:1 İlişkiler ve Lazy

```
@Entity
public class Owner {
    @Id @GeneratedValue
    private Long id;

    @OneToOne(mappedBy="owner",fetch=FetchType.LAZY)
    @LazyToOne(LazyToOneOption.NO_PROXY)
    private Address address;
}

@Entity
public class Address {
    @Id
    @GeneratedValue
    private Long id;

    @OneToOne
    @JoinColumn(name = "OWNER_ID")
    private Owner owner;
}
```

Çalışması için ayrıca bytecode enhancement'ın aktive edilmesi gerekir bytecode enhancement maven ile build zamanında veya JPA managed modda runtime'da aktive edilebilir

ByteCode Enhancement

- Hibernate 5 bytecode enhancement yöntemi ile üç farklı işlemi daha verimli biçimde gerçekleştirebilmektedir
  - **Lazy Initialization**: Entity basic attribute'larının lazy biçimde yüklenmesi mümkün olabilmektedir
  - **Dirty Tracking**: Persistence context içerisinde state'i değişmiş entity'leri tespit etmek için kullanılabilmektedir
  - **Association Management**: Çift yönlü ilişkilerin bir tarafında yapılan ekleme/çıkarmanın diğer tarafa otomatik yansıtılmasını sağlamaktadır

ByteCode Enhancement ve Lazy Attribute Initialization

JAVA Eğitimleri

- Entity içerisindeki basic attribute'ların teker teker veya grup şeklinde lazy biçimde yüklenebilmesini sağlar

```
@Entity
public class Image {

    @Column
    @Basic(fetch=FetchType.LAZY)
    private String description;

    @Column
    @Basic(fetch=FetchType.LAZY)
    @LazyGroup("lobs")
    private Blob content;
    ...
}
```

description ve content attribute'ları entity yüklenirken değil, ilk erişildikleri vakit DB'den yükleneceklerdir.

description attribute'unun yüklenmesi content alanının yüklenmesini tetiklemeyecektir.

Her ikisi ayrı lazy gruplarda yer aldıklarından ayrı ayrı erişildiklerinde yükleneceklerdir

ByteCode Enhancement ve Dirty Tracking

- Hibernate default olarak DB'den yüklenen entity'lerin state'lerinin bir kopyasını (**snapshot**) Session'da tutar
- Flush aşamasında da entity'nin **current state'i ile snapshot state'i karşılaştırarak** state değişikliklerini tespit eder
- **Date tipli veya mutable property**'lerin state değişikliğini tespit edebilmek için de en iyi yöntem budur
- Çünkü bu tür property'lerin state'i **entity dışında doğrudan** değiştirilebilmektedir

ByteCode Enhancement ve Dirty Tracking

- Ancak attribute sayısının çok olduğu, Session'a çok fazla sayıda entity yüklendiği durumlarda bu yöntem **performans açısından dezavantaj** yaratmaktadır
- Bytecode enhancement sayesinde **state değişiklikleri entitynin kendi içinde** takip edilir
- Flush aşamasında da Hibernate **sadece entity'ye değişip değişmediğini sorarak** dirty kontrolü yapabilir
- Snapshot yine vardır, ancak dirty kontrolünde kullanılmaz

ByteCode Enhancement ve Assoc Mgmt

JAVA Eğitimleri

- Owner – Pet arasında çift yönlü 1:M bir ilişki olsun
- Uygulama tarafında düzgün çalışabilmek için bu **ilişki iki entity üzerinde de yönetilmeli**dir

```
Owner owner = session.get(Owner.class, 1L);
Pet pet = session.get(Pet.class, 101L);

owner.getPets().add(pet);
pet.setOwner(owner);
```

- Bytecode enhancement sayesinde ilişkinin bir tarafında yapılan değişiklik **otomatik diğer tarafa** da yansıtılır

ByteCode Enhancement Nasıl Gerçekleştirilir?

- **Build-time** ve **Runtime** şeklinde iki farklı biçimde gerçekleştirilebilir
- **Runtime sadece JPA managed ortamda (JavaEE)** gerçekleşebilir
- Ayrıca persistent unit içerisinde **hibernate.ejb.use_class_enhancer=true** property'si tanımlı olmalıdır
- Diğer bir kısıt ise **sadece anotasyonlarla işaretlenmiş entity'ler**de devreye girmesidir

ByteCode Enhancement Nasıl Gerçekleştirilir?

- Build-time için ise Hibernate **gradle** ve **maven plugin'leri** sunmaktadır

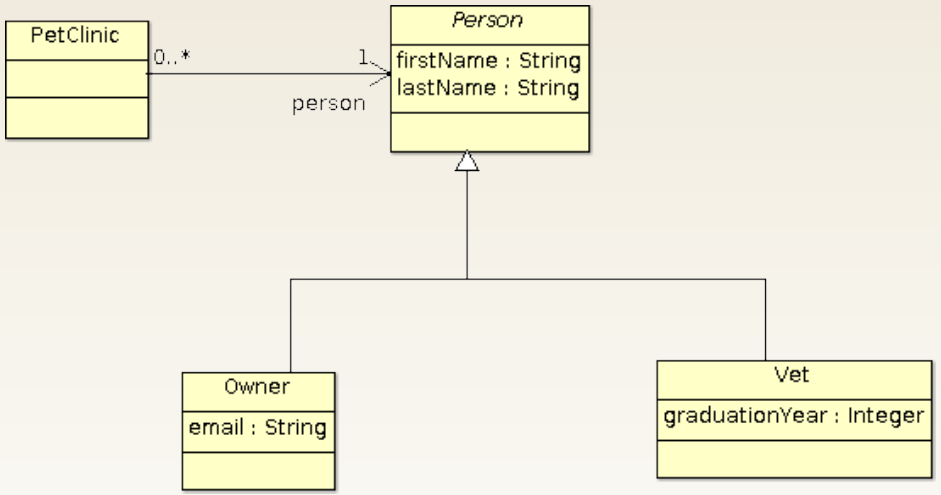
```
<plugin>
    <groupId>org.hibernate.orm.tooling</groupId>
    <artifactId>hibernate-enhance-maven-plugin</artifactId>
    <version>${hibernate.version}</version>
    <executions>
        <execution>
            <configuration>
                <failOnError>true</failOnError>
                <enableLazyInitialization>true</enableLazyInitialization>
                <enableDirtyTracking>true</enableDirtyTracking>
                <enableAssociationManagement>true</enableAssociationManagement>
            </configuration>
            <goals>
                <goal>enhance</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

ByteCode Enhancement ve Proxy

JAVA Eğitimleri

- Build-time **bytecode enhancement** gerçekleştirildiği takdirde Session.**load**(), EntityManager.**getReference**() metotlarından **proxy dönülmesi devre dışı** kalmaktadır
- Bu metotlar Session.**get**() veya EntityManager.**find**() **gibi davranacaklar**dır

Polymorphic M:1/1:1 Lazy İlişkiler




```
@Entity
public class PetClinic extends BaseEntity {

    @ManyToOne(fetch=FetchType.LAZY)
    private Person person;

    public Person getPerson() {
        return person;
    }

    public void setPerson(Person person) {
        this.person = person;
    }
}
```

- **M:1** veya **1:1** ilişkiler **lazy** tanımlanırsa, **proxy dönülebileceği için** polymorphic ilişkinin hedef nesne tipi **instanceof** operatörü ile tespit edilemez

```
Person personProxy = pc.getPerson();

System.out.println(personProxy instanceof Owner);
System.out.println(personProxy instanceof Vet);
```

Her ikiside false dönecektir. instanceof yerine Hibernate'in yardımcı sınıfı kullanılabilir. Bu işlemin yan etkisi olarak proxy initialize edilecektir!

```
System.out.println(Hibernate.getClass(personProxy).isAssignableFrom(Owner.class));
System.out.println(Hibernate.getClass(personProxy).isAssignableFrom(Vet.class));
```

İki Tane Eager Bag Collection Problemi

```
@Entity
public class Foo {

    @OneToMany(fetch = FetchType.EAGER)
    private List<Bar> bars = new ArrayList<Bar>();


    @OneToMany(fetch = FetchType.EAGER)
    private List<Baz> baz = new ArrayList<Baz>();
}
```

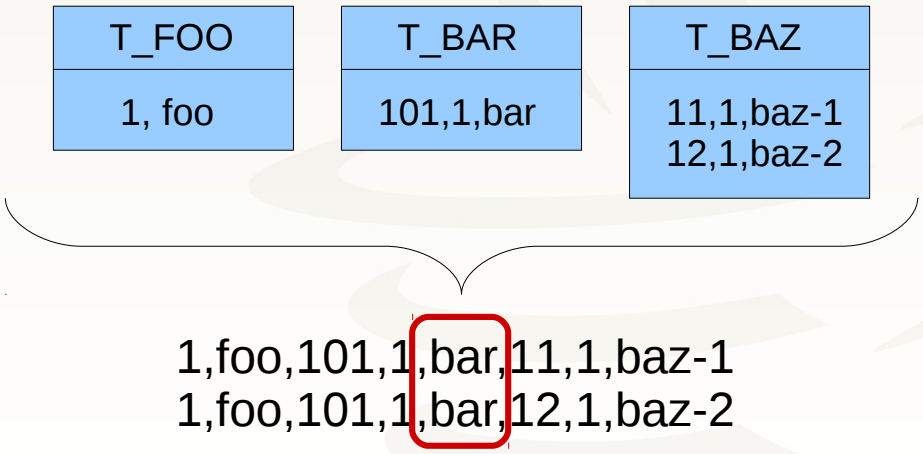
İlişkilerden bir tanesi LAZY tanımlanmak zorundadır!

Hata!

Foo entity'si yüklenirken bar ve baz listeleri eager yüklenmesi için Bar ve Baz tabloları ile LEFT OUTER JOIN yapılmaktadır.

Join sırasında kayıtlarda duplikasyon olabilir. Bag collection ilişkisinde de duplike kayıtlar olabilir.

Dolayısı ile duplikasyonun, join işlemi sonucu mu, yoksa bag collection içerisindeki duplikasyondan mı kaynaklandığı bilinemediği için Hibernate iki eager bag collection mapping'e izin vermemektedir



Parent - Child İlişkiler ve OrphanRemoval

```
@Entity
public class Owner {
    @Id @GeneratedValue
    private Long id;

    @OneToOne(orphanRemoval = true)
    @JoinColumn(name = "ADDRESS_ID")
    private Address address;
}

@Entity
public class Address {
    @Id
    @GeneratedValue
    private Long id;

}
```

FK'nın yer aldığı tabloya karşılık gelen entity child olarak kabul edilir. orphanRemoval ise sadece parent entity tarafında kullanılabilir

Parent - Child İlişkiler ve OrphanRemoval

```
@Entity
public class Owner {
    @Id @GeneratedValue
    private Long id;

    @OneToOne(orphanRemoval = true,mappedBy ="owner")
    private Address address;
}

@Entity
public class Address {
    @Id
    @GeneratedValue
    private Long id;

    @OneToOne
    @JoinColumn(name = "OWNER_ID")
    private Owner owner;
}
```

JPA ile OrderBy

```
@OneToMany
@JoinColumn(name = "PET_ID")
@OrderBy("filePath asc")
private List<Image> images = new ArrayList<Image>();
```

"**Ordered**" collection veritabanı düzeyinde SQL sorgusundaki "**order by**" ifadesi ile sıralanır

@OrderBy attribute değeri bir **JPQL ifadesi**dir.

Hibernate ile OrderBy

```
@OneToMany
@JoinColumn(name = "PET_ID")
@org.hibernate.annotations.OrderBy(clause="FILE_PATH asc")
private List<Image> images = new ArrayList<Image>();
```

"**Ordered**" collection veritabanı düzeyinde SQL sorgusundaki "**order by**" ifadesi ile sıralanır

Hibernate'in @OrderBy attribute değeri bir **SQL ifadesi**dir, Burada bir **SQL function da kullanılabilir**

Hibernate Sort

```
@OneToMany
@JoinColumn(name = "PET_ID")
@org.hibernate.annotations.SortNatural
private SortedSet<Image> images = new TreeSet<Image>();
```

Collection elemanları Comparable arayüzünü implement ediyorsa **SortNatural** kullanılabilir

Sorted* arayüzlerini kullanmak yerine veritabanı düzeyinde “ordering” yapmak daha efektiftir

Hibernate Sort

```
@OneToMany
@JoinColumn(name = "PET_ID")
@org.hibernate.annotations.SortComparator(ImageComparator.class)
private SortedSet<Image> images = new TreeSet<Image>();
```

"**Sorted**" collection Java **Comparator** sınıfı ile hafızada sıralanır

Sorted* arayüzlerini kullanmak yerine veritabanı düzeyinde "ordering" yapmak daha efektiftir

Formula Yöntemi ile Sınıf Hiyerarşisinde Tek Tablo Kullanımı

- Legacy veritabanlarında **discriminator sütun** eklemek mümkün olmayabilir
- Bu durumda "**formula**" yöntemi kullanılabilir
- Discriminator değer üretmek için formula kullanımı JPA da yoktur, **Hibernate spesifik**tir

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@org.hibernate.annotations.DiscriminatorFormula(
"case when GRADUATION_YEAR is not null then 'V' else 'O' end")
public class Person {
   ...
}
```

Hibernate ve Constraintler Arasındaki İlişki

JAVA Eğitimleri

- Constraint tanımları sadece **şema DDL** ifadelerinin üretilmesinde faydalıdır
- Bu sayede Hibernate yardımı ile geliştirme veya test ortamlarında **bütün şema otomatik biçimde** constraintleri ile beraber yaratılabilir
- **Runtime**'da bir fonksiyon icra etmezler

Domain ve Sütun Constraint Eklenmesi: Check

```
@Column(name = "FIRST_NAME", length = 16,
        nullable = false, unique = true)
```

Length, nullable, unique hepsi sütun düzeyinde constraint demektir

```
@org.hibernate.annotations.Check(constraints =
"regexp_like(FIRST_NAME,'^[[:alpha:]]+$')")
private String firstName;
```

```
create table PERSONS (
    …
    FIRST_NAME varchar(16) not null
        check(regexp_like(FIRST_NAME,'^[[:alpha:]]+$')),
    …
);
```

```
<property name="firstName" type="string">
    <column name="FIRST_NAME" check="regexp_like(FIRST_NAME,'^[[:alpha:]]+$')"/>
</property>
```

Satır Düzeyinde Constraint Eklenmesi: Check

```
@Entity
@org.hibernate.annotations.Check(constraints="START_DATE < END_DATE")
public class Meeting {
//...
}
```

Birden fazla sütun'un yer aldığı constraint'lerdir

```
create table MEETING (
    …
    START_DATE timestamp not null,
    END_DATE timestamp not null,
    …
                    check (START_DATE < END_DATE));
```

```
<class name="com.javaegitimleri.petclinic.model.Meeting" table="MEETING"
                                                    check="START_DATE < END_DATE">
</class>
```

Tablo Düzeyinde Constraintler

```
@Entity
@Table(name = "CATEGORY",
   uniqueConstraints = {
        @UniqueConstraint(columnNames ={"CAT_NAME", "PARENT_CATEGORY_ID"} )
   })
public class Category {
//...
}
```

```
create table CATEGORY (
    …
    CAT_NAME varchar(255) not null,
    PARENT_CATEGORY_ID integer,
    …
                unique (CAT_NAME, PARENT_CATEGORY_ID)
);
```

```
<class name="com.javaegitimleri.petclinic.model.Category" table="CATEGORY">
    <id name="id" column="ID"/>
    <properties name="cat_name_parent_cat_id" unique="true">
        <property name="name" column="CAT_NAME"/>
        <many-to-one name="parentCategory" column="PARENT_CATEGORY_ID"/>
    </properties>
    ...
</class>
```

Tablo Düzeyinde Constraintler

```
@Entity
@Table(name="ITEMS", indexes = @Index(
      name = "IDX_INITIAL_PRICE_CURRENCY",
      columnList = { "INITIAL_PRICE", "INITIAL_CURRENCY" }    ))
public class Item {
    @Column(name = "INITIAL_PRICE")
    private BigDecimal price;

    @Column(name = "INITIAL_CURRENCY")
    private Currency currency;
}
```

JPA 2.1'de @**Table** içerisinde kullanılan @**Index** annotasyonu gelmiştir. Bu annotasyon ile bir tablonun bütün index'leri tek bir yerde tanımlanmaktadır.

```
create index IDX_INITIAL_PRICE_CURRENCY on
ITEMS (INITIAL_PRICE,INITIAL_CURRENCY);
```

Veritabanı Düzeyinde Constraintler

```
@Entity
public class Pet {
    @ManyToOne
    @JoinColumn(name = "OWNER_ID",
            foreignKey = @ForeignKey(name = "FK_PET_OWNER_ID"))
    private Owner owner;
}

@Entity
public class Vet {
    @ManyToMany
    @JoinTable(name = "T_VET_SPECIALTY",
            foreignKey = @ForeignKey(name = "FK_VET_ID"),
inverseForeignKey = @ForeignKey(name = "FK_SPECIALTY_ID"))

    private Set<Specialty> specialties =new HashSet<Specialty>();

}
```

JPA 2.1'de @JoinColumn ve @JoinTable içerisinde kullanılan @ForeignKey annotasyonu gelmiştir. Ancak bununla ilgili Hibernate 4'de bazı bug'lar mevcuttur. Örneğin M:N de tanımlanan FK'lar dikkate alınmamaktadır. Hibernate 5'de bu bug fixlenmiştir

Collection İlişkileri ve @OnDelete

Herhangi bir Owner silindiği vakit ona bağlı Pet'lerin de ilgili Tablodan SQL DELETE CASCADE ile silinmesini sağlar

```
@Entity
public class Owner {

    @OneToMany
    @JoinColumn(name="owner_id")
    @OnDelete(action=OnDeleteAction.CASCADE)
    private Set<Pet> pets = new HashSet<>();

}
```

```
alter table T_PET add constraint
FK_h14un5v94coafqonc6medfpv8 foreign key (owner_id)
references T_OWNER on delete cascade
```

Bu DDL ifadesi şema üretimi sırasında oluşturulur

Flush ve Transactional Write Behind

- Hibernate Persistence Context'in flush işlemini mümkün olan **en son aşamaya kadar öteler**
- Bu yaklaşıma "**transactional write-behind**" adı verilmektedir
- Bu aşama genellikle **TX commit** anıdır
- Bir TX içerisinde **birden fazla kez flush** gerçekleşebilir
- FlushMode.AUTO durumunda hem **TX commit aşamasında**, hem de sorgu sonucunun güncelliğini etkileyecek değişiklikler söz konusu ise **sorgu öncesi TX içerisinde** flush çalışabilir

INSERT'in Flush Aşamasına Kadar Ertelenmesi

- **Session.save()** DB id'sini döner, bu nedenle **DB id'si hemen oluşturulmalı**dır
- Pek çok **identifier üretme stratejisi**nde bu bir problem teşkil etmez
- Sequence, uuid vb. sorunsuzdur
- Identity ve auto-increment ise **INSERT ifadesinin hemen çalıştırılması**nı gerekli kılar
- Bu stratejilerle **conversation yönetimi imkansız**dır

INSERT'in Flush Aşamasına Kadar Ertelenmesi

- **persist()** metodu ise bu tür bir probleme neden olmaz
- persist() metodu **identifier'ın hemen oluşturulmasın**ı istemez
- INSERT işlemi **flush aşamasına kadar** ötelenebilir

Collection İlişkileri ve @OptimisticLock

```
@Entity
public class Owner {

  @OneToMany
  @JoinColumn(name = "OWNER_ID")
  @org.hibernate.annotations.OptimisticLock(excluded = true)
  private Set<Pet> pets = new HashSet <Pet>();

}
```

Collection'lara yapılan ekleme ve silmelerin **versiyon artışına neden olması istenmiyor ise** ilişki @OptimisticLock anotasyonu ile tanımlanabilir

```
@Entity
public class Pet {

}
```

@Version Alanı Olmadan Optimistic Lock

JAVA Eğitimleri

- Hibernate ayrı bir **versiyon alanı olmadan da** optimistic lock'a imkan tanır
- Bunun için entity sınıfta **@OptimisticLocking** anotasyonu ile **OptimisticLockType** belirtilmelidir
  - **NONE**: optimistic kilitlemeyi devre dışı bırakır
  - **VERSION**: @Version alanı kullanılır
  - **DIRTY**: WHERE ifadesinde sadece değişen alanlar yer alır
  - **ALL**: WHERE ifadesinde bütün alanlar yer alır

@Version Alanı Olmadan Optimistic Lock

```
@Entity
@org.hibernate.annotations.DynamicUpdate
@org.hibernate.annotations.OptimisticLocking(
OptimisticLockType.DIRTY)
public class Pet {

}
```

- Versiyon alanı olmadan optimistic lock **sadece** "**extended persistence context**" yaklaşımı için uygundur
- **Detached nesneler**le çalışıldığı vakit "**lost update**" problemine neden olabilir

Pessimistic Lock

- Belirtilen entity için DB'de **SELECT...FOR UPDATE** ifadesi çalıştırılabilir
- Bu komut ile kayıt üzerinde **write lock** alınmış olunur
- Bu sayede TX sonuna kadar diğer TX'ler bu entity nesne üzerinde herhangi bir **okuma işlemi dahi yapamazlar**

```
Session session = sf.openSession();
session.beginTransaction();

session.lock(owner, LockMode.PESSIMISTIC_WRITE);

//...
session.getTransaction().commit();
```

Pessimistic Lock

- Session.lock() arka tarafta **LockRequest** üzerinden çalışmaktadır

```
Session session = sf.openSession();
session.beginTransaction();

LockRequest lockRequest = session.buildLockRequest(
                                        LockOptions.UPGRADE);
lockRequest.lock(owner);

//...
session.getTransaction().commit();
```

NaturalId ile Pessimistic Lock

- **Naturel PK** üzerinden de SELECT...FOR UPDATE ile **pessimistic kilit** elde edilebilir

```
User user = session.bySimpleNaturalId(User.class)
                   .with(LockOptions.UPGRADE)
                   .load("ksevindik@gmail.com");
```

Hibernate LockOptions

- **LockOptions.UPGRADE** :
  - Eğer db SELECT...FOR_UPDATE'i destekliyorsa veritabanı düzeyinde **pessimistic write lock** atar
  - Eğer kayıt üzerinde **başka bir lock var ise** bu lock bırakılıncaya veya timeout süresi kadar **bekler**

```
/* UPGRADE lock com.javaegitimleri.petclinic.model.Pet */ select
    ID
from
    T_PET
where
    ID =?
    and VERSION =? for update
```

Hibernate LockOptions

- **LockOptions.UPGRADE** :
  - Versiyon kontrolü yapılır, Eğer write lock alınan nesnenin **versiyon bilgisi güncel değil ise** hata fırlatılır
  - Eğer db SELECT...FOR UPDATE'i desteklemiyor ise kendiliğinden **LockOptions.READ**'a düşer

Hibernate LockOptions

- **LockOptions.READ** :
  - Entity session'da **mevcut ise birşey yapmaz**
  - Detached entity'nin ve DB'deki karşılığının **versiyonlarını karşılaştırır**, eğer nesnenin versiyon bilgisi güncel değil ise hata fırlatır

```
session.buildLockRequest(LockOptions.READ).lock(pet);

session.lock(pet, LockMode.READ);
```

```
/* READ lock com.javaegitimleri.petclinic.model.Pet */ select
    ID
from
    T_PET
where
    ID =?
    and VERSION =?
```

Hibernate LockOptions

- **LockOptions.NO_WAIT**
  - LockOptions.UPGRADE ile benzerdir, fakat destekleniyorsa SELECT ... FOR UPDATE NOWAIT ifadesini kullanır
  - Bu durumda eğer lock elde edilemiyorsa hemen **hata fırlatır**
  - Eğer SQL dialect NOWAIT desteklemiyorsa kendiliğinden **LockOptions.UPGRADE**'e düşer
- **WAIT_FOREVER** ise timeout değerini göz ardı ederek kilidi alana kadar bekler
- **SKIP_LOCKED** ise alınmış kilit varsa bunu göz ardı eder

Hibernate LockOptions

- **LockOptions.NONE**
  - Sadece detached nesnelerin Session'a **reattach** yapılmasını sağlar
  - **Versiyon kontrolü yapmaz**, dolayısı ile versiyon bilgisi güncel olmayan bir entity de Session'a reattach edilebilir

```
session.buildLockRequest(LockOptions.NONE).lock(pet);

session.lock(pet, LockMode.NONE);
```

Hibernate LockModları

- **LockMode.OPTIMISTIC**
  - TX commit aşamasında entity'nin versiyonu DB'deki karşlığı ile **kontrol edilecektir**
  - Fark varsa hata fırlatılır

```
session.beginTransaction();

session.lock(pet, LockMode.OPTIMISTIC);

session.getTransaction().commit();
```

```
/* get version com.javaegitimleri.petclinic.model.Pet */ select
    VERSION
from
    T_PET
where
    ID =?
```

Hibernate LockModları

- **LockMode.OPTIMISTIC_FORCE_INCREMENT**
  - TX commit aşamasında entity'nin versiyonu DB'deki karşlığı ile **kontrol edilir ve bir artırılır**
  - Fark varsa hata fırlatılır

```
session.beginTransaction();

session.lock(pet, LockMode.OPTIMISTIC_FORCE_INCREMENT);

session.getTransaction().commit();
```

```
/* forced version increment */ update
    T_PET
set
    VERSION=?
where
    ID=?
    and VERSION=?
```

Hibernate LockModları

- **LockMode.PESSIMISTIC_FORCE_INCREMENT**
  - LockOptions'da karşılığı yoktur
  - Entity'nin versiyonu mevcut TX içerisinde lock anında hemen bir artırılır

```
session.lock(pet, LockMode.PESSIMISTIC_FORCE_INCREMENT);
```

```
/* forced version increment */ update
    T_PET
  set
    VERSION=?
  where
    ID=?
    and VERSION=?
```

İletişim

- Harezmi Bilişim Çözümleri
- Kurumsal Java Eğitimleri
- http://www.java-egitimleri.com
- info@java-egitimleri.com