İleri Düzey JPA/Hibernate Eğitimi 3

harezmi bilişim çözümleri

IdentifierLoadAcces ile Entity Erişimi

- Hibernate 5 ile gelen bir özelliktir
- **IdentifierLoadAccess** arayüzü ile entity nesneye **PK değeri ile erişilebilir**

```
IdentifierLoadAccess<Pet> byId = session.byId(Pet.class);
```

```
Pet pet = byId.load(1L);
```
→ Session.get() gibidir. Entity döner, bulamaz Ise NULL döner

```
Pet pet = byId.getReference(1L);
```
→ Session.load() gibidir. Uninitialized proxy nesne döner

```
Optional<Pet> optionalPet = byId.loadOptional(1L);
```
→ Load() a benzer, ancak NULL yerine Java 8 Optional nesnesi döner

MultiIdentifierLoadAcces ile Entity Erişimi

- Hibernate 5 ile gelen bir özelliktir
- **MultiIdentifierLoadAccess** arayüzü üzerinden tek sorgu ile birden fazla entity nesne PK değerleri ile yüklenebilir

```
MultiIdentifierLoadAccess<Pet> byId = session.byMultipleIds(Pet.class);

List<Pet> pets = byId.multiLoad(1L,2l,3L);

List<Pet> pets = byId.multiLoad(Arrays.asList(1L,2l,3L));
```

Naturel ID Üzerinden Erişim

- Hibernate 5 ile gelen bir özelliktir
- Entity sınıfta **sentetik PK'nın yanı sıra naturel PK** da tanımlanabilir

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

    @NaturalId(mutable=true)
    private String firstName;

    @NaturalId(mutable=true)
    private String lastName;

    private int age;

    //getter & setters...
}
```

Naturel PK alanları bir veya daha fazla olabilir

Default olarak immutable alanlardır ancak burada olduğu gibi mutable olarak da tanımlanabilirler

Naturel ID Üzerinden Erişim

```
Employee emp = session.byNaturalId(Employee.class)
        .using("firstName", "Kenan")
        .using("lastName", "Sevindik").load();
```

Naturel PK'ya karşılık bulunan Entity dönülür, bulamaz ise NULL döner

```
Employee emp = session.byNaturalId(Employee.class)
        .using("firstName", "Kenan")
        .using("lastName", "Sevindik").getReference();
```

Uninitialized proxy referans döner

```
Optional<Employee> optionalEmp = session.byNaturalId(Employee.class)
        .using("firstName", "Kenan")
        .using("lastName", "Sevindik").loadOptional();
```

Load ile aynı mantığa sahiptir
Ancak NULL değerleri ele almak için
Java 8'deki Optional özelliğini kullanır

Naturel ID Üzerinden Erişim

- Eğer naturel PK alanı tek bir tane ise bu durumda **bySimpleNaturelId** kullanılarak sorgu yazılabilir

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

    @NaturalId
    private String email;

    //getter & setters...
}
```

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

Example ile Sorgulama (QBE)

```
Owner owner = new Owner();
owner.setFirstName("James");

Example exampleOwner =
Example.create(owner)
     .ignoreCase()
     .enableLike(MatchMode.ANYWHERE)
     .excludeProperty("password");

session.createCriteria(Owner.class)
      .add(exampleOwner).list();
```

Sorgulanacak sınıfın bir instance'ı Kullanılır

Instance'ın sorgu kriterinde kullanılacak property'leri doldurulur

Example sorgu instance'ı yaratılır

İstenen property'ler excludeProperty() ile hariç tutulabilir

Sorgulanacak entity için bir criteria nesnesi oluşturulur
Example nesnesi criteria nesnesine eklenerek sorgu gerçekleştirlir

Example ile Sorgulama

```
Owner owner = new Owner();
owner.setFirstName("James");

Example exampleOwner = Example.create(owner)
.ignoreCase().enableLike(MatchMode.ANYWHERE);



session.createCriteria(Owner.class)
.add(exampleOwner).createCriteria("pets")
.add(Restrictions.isNull("birthDate")).list();
```

Çalışma Zamanında İsimlendirilmiş Sorgu Eklemek

- JPA 2.1 ile birlikte **çalışma zamanında yeni bir isimlendirilmiş sorgu** (named query) eklenebilir
- Yeni tanımlanan bu sorgunun kullanım şekli anotasyon veya konfigürasyon dosyası ile tanımlanmış isimlendirilmiş sorgularla aynıdır

```
Query query = em.createQuery("select p from Pet as p");

em.getEntityManagerFactory()
      .addNamedQuery("selectAllPets",query);
```

JPA ile Stored Procedure Çağırmak

JAVA Eğitimleri

- JPA 2.1 ile gelen bir özelliktir
- **İsimlendirilmiş stored procedure sorguları** tanımlamak veya programatik olarak **çalışma zamanında yaratmak** mümkündür
- Her iki yöntem ile de **StoredProcedureQuery** nesneleri elde edilir
- Bu nesneler üzerinden de DB'deki stored procedure **çalıştırılıp dönen sonuç işlenebilir**

JPA ile Stored Procedure Çağırmak

- Aşağıda MySQL ile yazılmış **Pet kayıtlarını isme göre sorgulayan** bir stored procedure örneği vardır

```
DELIMITER $$
CREATE DEFINER=`root`@`localhost`
        PROCEDURE `FIND_PETS_BY_NAME`(in p_name varchar)
begin
SELECT ID, NAME, BIRTH_DATE, TYPE_ID, OWNER_ID
    FROM T_PET
    WHERE NAME = p_name;
end
$$
DELIMITER ;
```

@NamedStored ProcedureQuery Kullanımı

```
@NamedStoredProcedureQueries({
  @NamedStoredProcedureQuery(
    name = "findPetsByName",
    procedureName = "FIND_PETS_BY_NAME",
    resultClasses = { Pet.class },
    parameters = {
        @StoredProcedureParameter(
          name = "p_name",
          type = String.class,
          mode = ParameterMode.IN) })
})
@Entity
@Table(name="T_PET")
public class Pet {
...
}
```

Dört farklı türde parametre modu olabilir

- IN: input parametre
- OUT: output parametre
- INOUT: hem input hem output için kullanılan parametre
- REF_CURSOR: result set dönem için kullanılan ref cursor

@NamedStored ProcedureQuery Kullanımı

```
StoredProcedureQuery storedProcedureQuery =
entityManager.createNamedStoredProcedureQuery(
                                        "findPetsByName",Pet.class);

storedProcedureQuery.setParameter("p_name", "maviş");

List<Pet> pets = storedProcedureQuery.getResultList();

for(Pet pet:pets) {
    System.out.println(pet);
}
```

Çalışma Zamanında Stored ProcedureQuery Oluşturmak

```
StoredProcedureQuery storedProcedureQuery =
entityManager.createStoredProcedureQuery("FIND_PETS_BY_NAME");
storedProcedureQuery.registerStoredProcedureParameter("p_name",
                                    String.class, ParameterMode.IN);

storedProcedureQuery.setParameter("p_name", "maviş");

List<Pet> pets = storedProcedureQuery.getResultList();

for (Pet pet : pets) {
    System.out.println(pet);
}
```

HQL/JQL Fonksiyonları

- Hem HQL hem de JQL bir takım **aggregate ve scalar fonksiyonları** destekler
- HQL **dialect tarafından tanımlanmış** diğer fonksiyonları da destekler
- DB tarafında mevcut olan diğer fonksiyonlar da sonradan programatik olarak **Configuration** nesnesi üzerinden tanımlanabilir
- HQL'de WHERE clause'unda kullanılan bir fonksiyon, eğer **Hibernate için anlamlı değilse** doğrudan **SQL fonksiyonu** olarak kabul edilir

Aggregate Fonksiyonlar

- Aggregate fonksiyonlar çoğunlukla gruplamada kullanılırlar
- İşlevleri SQL karşılıkları ile aynıdır
  - **COUNT**: Long değer döner
  - **AVG**: Double değer döner
  - **MIN**: Argümanın tipinde bir değer döner
  - **MAX**: Argümanın tipinde bir değer döner
  - **SUM**: Dönen değerin tipi toplanan değerlerin tipleri ile uyumludur. Integral değerler toplanıyor ise Long, floating point değerlerde Double gibi

Scalar Fonksiyonlar (HQL ve JQL)

- Int, String, long, date gibi spesifik değer dönen fonksiyonlardır
- CONCAT
- SUBSTRING
- UPPER
- LOWER
- TRIM
- LENGTH
- LOCATE
- ABS
- ABS
- MOD
- SQRT
- CURRENT_DATE
- CURRENT_TIME
- CURRENT_TIMESTAMP

Scalar Fonksiyonlar (HQL spesifik)

- BIT_LENGTH
  - Binary verinin uzunluğunu döner
- CAST
  - Tip dönüşümü yapar
- EXTRACT
  - Tarih değerinin bir bölümünü extract eder
- SECOND
- MINUTE
- HOUR
- DAY
- MONTH
- YEAR
- STR
  - Değeri character dataya cast eder

Collection'lara Özel Fonksiyonlar

- **SIZE**: Collection'ın size'ını hesaplar
- **MAXELEMENT**: Basic tip içeren collectionlarda max elemanı döner
- **MAXINDEX**: Indexed collectionlarda max index'i döner
- **MINELEMENT**:Basic tip içeren collectionlarda min elemanı döner
- **MININDEX**: Indexed collectionlarda min index'i döner

Collection'lara Özel Fonksiyonlar

- **ELEMENTS**: Collection'daki elemanları tümüne refer etmeyi sağlar
  - Genellikle ALL, ANY, SOME ifadeleri ile birlikte kullanılır
- **INDICES**: Indexed collection'lardaki elemanların index/key'lerinin tümüne refer etmeyi sağlar

Custom SQL Fonksiyonu Tanıtmak

```
Configuration cfg = new Configuration();

cfg.addSqlFunction(
     "lpad",
     new StandardSQLFunction("lpad", Hibernate.STRING));

//...

cfg.buildSessionFactory();
```

Hibernate Sorgularında ScrollableResults

- JDBC API'deki “**scrollable resultsets**” gibidir
- Sorgu sonuçlarının **toptan hafızaya yüklenmesinin zor olduğu durumlar** için faydalıdır
- DB sisteminde bir “**cursor**” kullanılarak gerçekleştirilir
- Cursor sorgu sonuç listesindeki herhangi bir satırdaki **kayda işaret** eder
- Uygulama içerisinden cursor **ileri/geri** hareket ettirilebilir

Hibernate Sorgularında ScrollableResults

```
ScrollableResults resultList =
session.createQuery("from Owner").scroll();

resultList.first();
resultList.last();
resultList.get();
resultList.next();
resultList.scroll(3);
resultList.getRowNumber();
resultList.setRowNumber(5);
resultList.previous();
resultList.scroll(-3);
resultList.close();
```

scroll() metodu ScrollableResults nesnesi döner. ScrollableResults vasıtası ile sorgu sonuç listesi üzerinde iterate etmek mümkündür

Hibernate Sorgularında ScrollableResults

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
ScrollableResults resultList =
    session.createQuery("from Owner").scroll();
int count=0;
while ( resultList.next() ) {
    Owner owner = (Owner) resultList.get(0);
    modifyOwner(owner);
    if ( ++count % 100 == 0 ) {
        session.flush();
        session.clear();
    }
}
resultList.close();
tx.commit();
session.close();
```

- ScrollMode.**SCROLL_INSENSITIVE**
  - Cursor açık olduğu müddetçe değişikliğe uğramış veri ile karşılaşılmasını engeller
- ScrollMode.**SCROLL_SENSITIVE**
  - Değişiklikleri ve yeni veriyi resultset anında yansıtır

TX sona ermeden evvel cursor kapatılmalıdır

Query.iterate()

- Query.iterate() ile bir **Iterator** elde edilerek entity'ler üzerinde iterate edilebilir
- Burada ilk SELECT sorgusu ile **sadece PK değerleri** alınır
- Daha sonra Iterator üzerinde dolaşırken nesnelerin state'leri **birincil** (persistence context) ve **ikincil** cache'lerde (second level cache) aranır
- Eğer **bulunamaz ise** her nesne için ayrı bir **SELECT** çalıştırılır

Query.iterate()

- Query.iterate() yöntemi ancak **ikincil cache aktif ise faydalı** olabilir
- Diğer durumda **N+1 SELECT** problemi doğurur
- Hibernate **Criteria** bu yöntemi desteklemez
- Iterator **son kayda** veya **Session kapatılana** değin açık tutulur
- **Hibernate.close(iterator)** ile explicit biçimde kapatılabilir

StatelessSession

- StatelessSession, Hibernate'in bir **SQL mapper** aracı gibi kullanılmasını sağlar
- İşlemler **anında SQL**'e dönüştürülür
- StatelessSession **persistence context içermez**
- Bütün **nesneler detached** durumdadır
- Nesne üzerindeki değişiliklerin DB'ye yansıması için mutlaka **StatelessSession.update()** çalıştırılmalıdır

StatelessSession

- **İkincil önbellek** ile de ilişkide değildir
- İşlemlerin **cascade**'i ilişkili nesnelere aktarılmaz
- Event listener'lar ve interceptor devrede değildir
- **Avantajı** sorgularda Hibernate sınıf eşlemelerinin kullanılabilmesi ve DB platform bağımsızlığı sağlamasıdır

StatelessSession

```
StatelessSession session = sf.openStatelessSession();
Transaction tx = session.beginTransaction();

Owner owner = new Owner("Ali","Güçlü");
Pet pet = new Pet("maviş");
pet.setOwner(owner);

session.save(pet);
session.save(owner);

Pet pet2 = session.createQuery(
        "from Pet p where p.id = 1").uniqueResult();

pet2.setName("karabaş");

session.update(pet);

tx.commit();
session.close();
```

Fetch Profile

- Normalde İlişkiler üzerindeki @Fetch(FetchMode) tanımları sistem genelinde sabittir
- **@FetchProfile** ise FetchMode tanımlarının **Session düzeyinde** duruma göre aktive edilmesini sağlar
- **Global metadata**'dır, herhangi bir sınıf düzeyinde veya **package-info.java** dosyası içerisinde tanımlanabilir

Fetch Profile

```
@FetchProfiles(value = {
    @FetchProfile(
        name = "fetchVisitsWithJoin",
        fetchOverrides =
              @FetchOverride(entity = Pet.class,
                             association = "visits",
                             mode = FetchMode.JOIN)
    )
})
@Entity
public class Pet {
    @OneToMany
    private Set<Visit> visits = new HashSet<Visit>();
}
```

Şu an diğer FetchMode'lar desteklenmemektedir

```
session.enableFetchProfile("fetchVisitsWithJoin");
session.createCriteria(Pet.class).list();
```

enableFetchProfile() ile aktive edilen bir fetch profili, istenildiği vakit disableFetchProfile() ile devre dışı bırakılabilir

JPA Entity Graph

- JPA 2.1 ile gelen bir özelliktir
- Dinamik olarak sorgularda **sadece ihtiyaç duyulan field ve ilişkilerin yüklenmesi**ni sağlar
- Eşleştirme metadata ile **statik olarak** veya çalışma zamanında **dinamik olarak** entity graph tanımlanabilir
- Entity graph tanımları find veya query işlemleri sırasında yüklenecek (eager fetch) **attribute tanımları** içerir

Default Entity Graph

```
@Entity
public class Pet {

    @Id
    private Long id;

    @Basic(fetch = FetchType.EAGER)
    private String name;

    private Date birthDate;

    @OneToMany
    private Set<Visit> visits = new HashSet <Visit>();

    @ManyToOne
    private PetType petType;
}
```

Her entity için default bir entity graph tanımı mevcuttur, bu tanım entity içindeki alan ve ilişkilerden oluşur

Default Entity Graph

- PK ve versiyon alanları **her zaman** yüklenirler, bunlar için attribute tanımlamaya gerek yoktur
- Ancak diğer field'ların hepsi varsayılan durumda eğer `@Basic(fetch = FetchType.EAGER)` şeklinde **tanımlanmamış ise lazy** yüklenirler
- İlişkilerde ise **ilişkilerin varsayılan fetch attribute değerleri** göz önüne alınır
- Normade M:1 ilişki eager fetch edilir,ancak `fetch = FetchType.LAZY` ile lazy tanımlanmış ise entity graph içerisinde de fetch edilmeyecektir

NamedEntityGraph

```
@NamedEntityGraphs({
     @NamedEntityGraph(
      name = "petsWithVisits",
      attributeNodes = {@NamedAttributeNode("visits")})
})
@Entity
public class Pet {
    @Id
    private Long id;

    @Basic(fetch = FetchType.EAGER)
    private String name;

    private Date birthDate;

    @OneToMany
    private List<Visit> visits = new ArrayList <Visit>();

    @ManyToOne
    private PetType petType;
}
```

NamedSubGraph

```
@NamedEntityGraph(
name = "ownersWithPetsAndVisits",
attributeNodes=@NamedAttributeNode(value="pets",subgraph="pets")),
subGraphs = @NamedSubGraph(name = "pets",
                    attributeNodes=@NamedAttributeNode("visits"))
)
@Entity
public class Owner {
    @Id
    private Long id;

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

}

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

    @OneToMany
    private List<Visit> visits = new ArrayList <Visit>();

}
```

Programatik Entity Graph Oluşturma

- EntityManager.**createEntityGraph**() ile çalışma zamanında **dinamik olarak** da entity graph yaratılabilir

```
EntityGraph entityGraph =
            entityManager.createEntityGraph(Pet.class);

entityGraph.addAttributeNodes("name", "visits");
```

Entity Graph Kullanımı

- EntityManager.**getEntityGraph**() veya EntityManager.**createEntityGraph**() metotlarından biri ile oluşturulan EntityGraph persistence işlemlerinde **fetch graph** veya **load graph** şeklinde iki farklı yoldan kullanılabilir
- Fetch Graph
  - Sadece **EntityGraph** içerisinde explicit biçimde tanımlanmış alanları getirir, default entity graph'ı göz ardı eder
- Load Graph
  - Explicit biçimde tanımlananların yanında default entity graph'da yer alan alanları da getirir

Entity Graph Kullanımı

```
EntityGraph entityGraph = entityManager
                    .getEntityGraph("petsWithVisits");

Properties props = new Properties();
props.put("javax.persistence.fetchgraph",entityGraph);

Pet pet = entityManager.find(Pet.class,1L,props);
```

Entity Graph Kullanımı

```
EntityGraph entityGraph = entityManager
                    .getEntityGraph("petsWithVisits");


Query query = entityManager
                   .createQuery("select p from Pet as p");
List<Pet> pets = query
   .setHint("javax.persistence.loadgraph",entityGraph)
   .getResultList();
```

Entity Filtreleme

- Bütün sorgularda where clause'una dahil olacak **ilave kriter** tanımlamayı sağlar
- Session'da enable edildikten sonra **bütün sorgularda** devreye girer
- Identifier kullanılarak gerçekleştirilen nesne erişimleri ise dönen sonuç **filtrelemeden etkilenmez**
- Entity üzerindeki ilişkilerde de tanımlanabilir
- Ancak sadece **1:M** ve **N:M** ilişkilerde filtre **tanımlanabilir**

Entity Filtreleme

Veri filtreleme tanımıdır, global metadatadır

```
@FilterDef(name="limitPetsByOwnerRank", parameters = {
        @ParamDef(name = "currentOwnerRank", type = "int")})
```

```
@Filter(name = "limitPetsByOwnerRank",
    condition=":currentOwnerRank >= " +
              "(select o.RANK from T_OWNERS o" +
              " where o.ID = OWNER_ID)")
@Entity
public class Pet {
}
```

Filter'daki ifade native SQL'dir
Doğrudan DB'ye aktarılır

Filter'ın devreye girebilmesi için Session'da enable edilmesi gerekir

```
Filter filter = session.enableFilter("limitPetsByOwnerRank");

filter.setParameter("currentOwnerRank", owner.getRanking());
```

Collection Filtreleme

- Bütün collection'ı initialize etmeden, **collection içerisinde bir grup elemana erişim** imkanı sunar
- Filtrelenmiş collection, **input collection'dan farklı** bir instance'dır. Asıl collection'a hiç bir biçimde dokunulmaz
- Sadece **persistent collection**'lar üzerinde Uygulanabilir
- Yaygın olarak sorgu sonucunu **daha fazla sınırlandırma sıralama veya pagination için** kullanılır

Collection Filtreleme

```
Query query = s.createFilter(pet.getVisits(),
"where this.visitDate > :oneWeekAgo");

query.setTimestamp("oneWeekAgo", oneWeekAgo);
query.list();
```

```
s.createFilter(pet.getVisits(), "order by
this.visitDate asc").list();
```

```
s.createFilter( pet.getVisits(), "" )
.setFirstResult(50).setMaxResults(100).list();
```

Nesnelerin Replikasyonu

- Nesneleri **bir DB'den diğerine taşımak** için kullanılır
- Hedef db'de kayıt **aynı ID** ile yaratılır
- **ReplicationMode** bu işlemin detaylarını yönetir

```
Session session1 = sessionFactory1.openSession();
Owner owner = (Owner) session1.get(Owner.class, 1L);
session1.close();

Session session2 = sessionFactory2.openSession();
Transaction transaction = session2.beginTransaction();
session2.replicate(owner, ReplicationMode.LATEST_VERSION);
transaction.commit();
session2.close();
```

Nesnelerin Replikasyonu

- **ReplicationMode.IGNORE** : Hedef db'de aynı kayıt söz konusu ise işlemi ignore eder
- **ReplicationMode.OVERWRITE** : Hedef db'deki kaydı, eldeki kayıt ile override eder
- **ReplicationMode.EXCEPTION** : Hedef db'de aynı kayıt varsa exception fırlatır
- **ReplicationMode.LATEST_VERSION** : Hedef db'deki kaydın version değeri eğer eldeki kayıttan eski ise override eder

Hibernate ve Entity Silme İşlemi

- **hibernate.use_identifier_rollback** konfigürasyon değeri **true** ise Hibernate silinen nesnenin id property'sini **NULL**'a set eder

İletişim

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