İleri Düzey JPA/Hibernate Eğitimi 1

harezmi bilişim çözümleri

Konfigürasyon Dosyalarından Bağımsız SessionFactory Oluşturma

JAVA Eğitimleri

```
Properties settings = new Properties();

settings.put("hibernate.connection.driver_class", "org.h2.Driver");
settings.put("hibernate.connection.url", "jdbc:h2:tcp://localhost/~/test");
settings.put("hibernate.connection.username", "sa");
settings.put("hibernate.connection.password", "");
settings.put("hibernate.hbm2ddl.auto", "create");
settings.put("hibernate.show_sql", "true");
settings.put("hibernate.current_session_context_class", "thread");

Configuration cfg = new Configuration();

StandardServiceRegistryBuilder serviceRegistryBuilder =
                            new StandardServiceRegistryBuilder()
                                    .applySettings(settings).build()

SessionFactory sessionFactory = cfg.addProperties(settings)
    .addAnnotatedClass(PetType.class)
    .buildSessionFactory(serviceRegistryBuilder);
```

Konfigürasyon Dosyalarından Bağımsız SessionFactory Oluşturma

```
JdbcDataSource dataSource = new JdbcDataSource();
dataSource.setUrl("jdbc:h2:tcp://localhost/~/test");
dataSource.setUser("sa");
dataSource.setPassword("");

Properties settings = new Properties();

settings.put("hibernate.connection.datasource", dataSource);
settings.put("hibernate.dialect", "org.hibernate.dialect.H2Dialect");
settings.put("hibernate.show_sql", "true");
settings.put("hibernate.current_session_context_class", "thread");

Configuration cfg = new Configuration();

SessionFactory sessionFactory = cfg.addProperties(settings)
    .addAnnotatedClass(PetType.class)
    .buildSessionFactory(new StandardServiceRegistryBuilder()
            .applySettings(settings).build());
```

DataSource nesnesi Properties dosyasından da sağlanabilir. JNDI kullanmak şart değildir.

Hibernate Konfigürasyonu ve Managed DataSource

- **DataSource'u programatik olarak yaratarak** Hibernate konfigürasyonuna tanıtmak da mümkündür

```
JdbcDataSource dataSource = new JdbcDataSource();
dataSource.setUrl("jdbc:h2:tcp://localhost/~/test");
dataSource.setUser("sa");


Configuration cfg = new Configuration();
cfg.configure().getProperties()
                        .put(AvailableSettings.DATASOURCE, dataSource);

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

Proxy DataSource ile SQL İşlemlerini Monitor Etmek

- **DataSource-proxy kütüphanesi** kullanarak Hibernate tarafından üretilen SQL ifadeleri, parametreleri vs loglanabilir

```
JdbcDataSource targetDataSource = new JdbcDataSource();
targetDataSource.setUrl("jdbc:h2:tcp://localhost/~/test");
targetDataSource.setUser("sa");
targetDataSource.setPassword("");

DataSource proxyDataSource = ProxyDataSourceBuilder
            .create(targetDataSource).name("ProxyDS").logQueryToSysOut().build();

Properties settings = new Properties();
settings.put("hibernate.connection.datasource", proxyDataSource);
settings.put("hibernate.dialect", "org.hibernate.dialect.H2Dialect");
settings.put("hibernate.hbm2ddl.auto", "create");

Configuration cfg = new Configuration();
SessionFactory sessionFactory = cfg.addProperties(settings)
    .addAnnotatedClass(PetType.class)
    .buildSessionFactory(new StandardServiceRegistryBuilder()
            .applySettings(settings).build());
```

Proxy DataSource ile SQL İşlemlerini Monitor Etmek (Commons Logging)

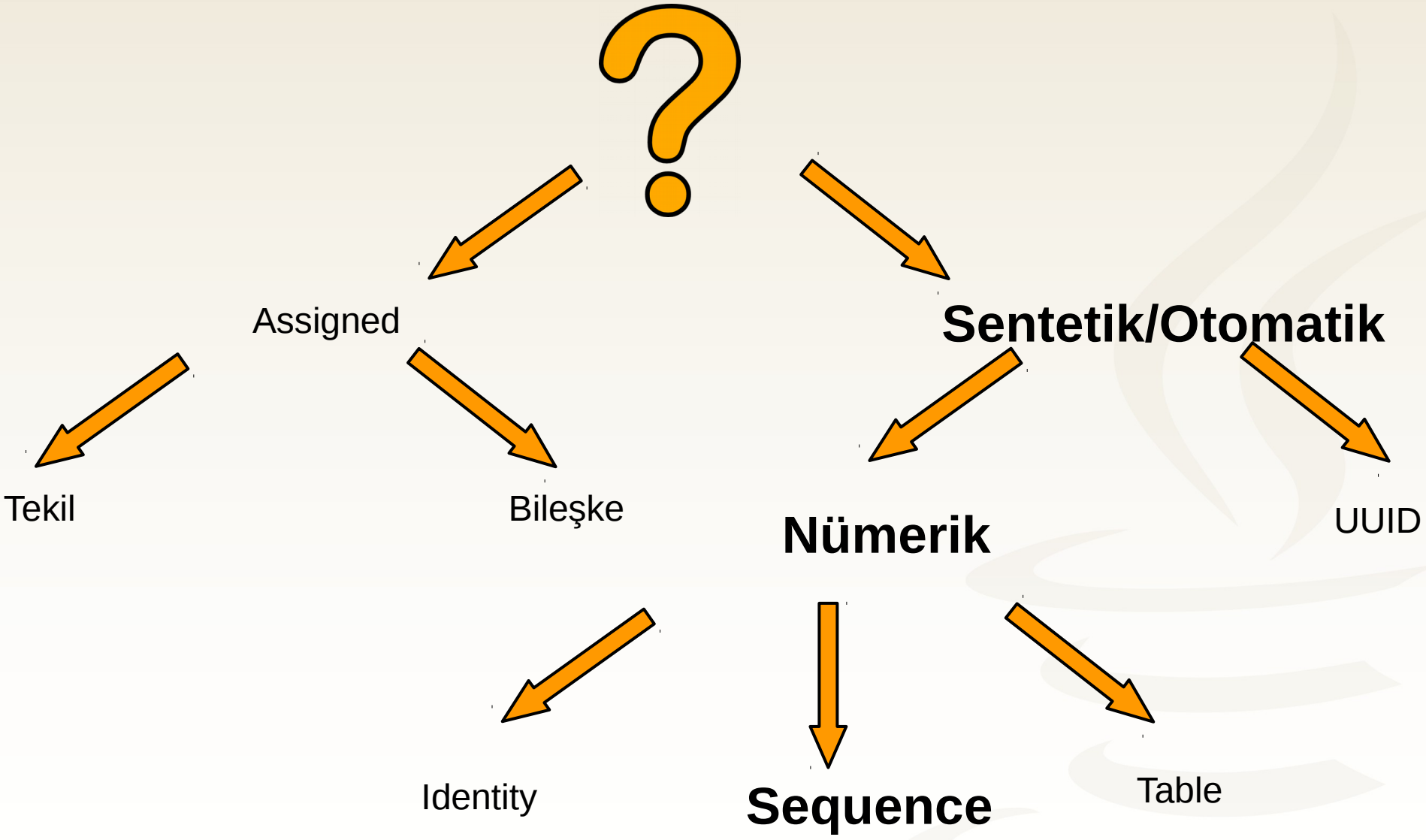
```
JdbcDataSource targetDataSource = new JdbcDataSource();
targetDataSource.setUrl("jdbc:h2:tcp://localhost/~/test");
targetDataSource.setUser("sa");
targetDataSource.setPassword("");

DataSource proxyDataSource = ProxyDataSourceBuilder
                            .create(targetDataSource).name("ProxyDS")
                            .listener(new CommonsQueryLoggingListener())
                            .build();

Properties settings = new Properties();
settings.put("hibernate.connection.datasource", proxyDataSource);
settings.put("hibernate.dialect", "org.hibernate.dialect.H2Dialect");
settings.put("hibernate.hbm2ddl.auto", "create");

Configuration cfg = new Configuration();
SessionFactory sessionFactory = cfg.addProperties(settings)
    .addAnnotatedClass(PetType.class)
    .buildSessionFactory(new StandardServiceRegistryBuilder()
            .applySettings(settings).build());
```

Hangi PK Yöntemi Tercih Edilmeli?



Hangi PK Yöntemi Tercih Edilmeli?

JAVA Eğitimleri

- Eğer **naturel PK tekil ve nümerik bir değer ise** performans açısından sentetik PK yöntemi ile hemen hemen aynıdır
- Ancak iş mantığından bağımsız **sentetik bir PK daha çok esneklik** sağlamaktadır
- Web uygulamalarında veriye erişim, auditing, kayıt düzeyinde yetkilendirme gibi işlemler sentetik id ile **daha kolay/standart biçimde** gerçekleştirilebilir

Hangi PK Yöntemi Tercih Edilmeli?

- **Bileşke PK değerleri** ayrıca veriye erişim açısından join'lerde ve indekslemede performans ve veri depolama alanı açısından daha verimsizdir
- Mecbur kalınmadıkça **tercih edilmeme**lidir
- **UUID** stratejisi daha çok cluster ortamlar için benzersiz PK değeri oluşturmak için tercih edilir
- Ancak **veri depolama alanı** ve **indeksleme** noktalarından dezavantaj yaratabilir

Hangi PK Yöntemi Tercih Edilmeli?

- **Identity** ancak sequence tercih edilemiyorsa kullanılmalıdır, JDBC **batch insert'leri desteklemez**
- **Tablo** üzerinden PK yönetimi **en az verimli** yöntemdir
- PK'nın elde edilmesi için **ayrı bir TX**'e gerek duyar
- Ayrıca **row-lock** yöntemini kullandığı için ölçeklenmede problemler yaratabilmektedir

Hangi PK Yöntemi Tercih Edilmeli?

JAVA Eğitimleri

- **En uygun yöntem sequence** olarak karşımıza çıkmaktadır
- Özellikle **pooled** veya **pooled-lo** yöntemi kullanan bir **sequence stratejisi** oldukça verimlidir
  - Pooled/pooled-lo yöntemlerinin devreye girmesi için ***hibernate.id.new_generator_mappings=true*** tanımlı olması gerekir
  - Bu tanımla default olarak pooled aktive olur
  - ***hibernate.id.optimizer.prefer_lo=true*** tanımı ile pooled-lo'ya geçiş yapılabilir

Her Tablo için Ayrı Bir Sequence Tanımlama Yöntemi

```
@MappedSuperclass
public abstract class BaseEntity {
    @Id

    @GeneratedValue(strategy = GenerationType.SEQUENCE,
                    generator = "sequenceStyleGenerator")

    @org.hibernate.annotations.GenericGenerator(
        name = "sequenceStyleGenerator",
        strategy = "sequence",
        parameters = @org.hibernate.annotations.Parameter(
                    name = "prefer_sequence_per_entity",
                    value = "true"))
    private Long id;
    ...
}
```

Hibernate'in SequenceStyleGenerator isimli sınıfının bir parametresidir. Bu sayede şema genelinde tek bir sequence tanımlamak yerine her bir entity tablosu için ayrı bir sequence üretir. Sequence isimleri default olarak TABLOADI_SEQ şeklinde olacaktır.

Sequence Sentetik ID'lerin Optimize Biçimde Elde Edilmesi

```
@Entity
public class User {

    @Id
    @GenericGenerator(name = "sequenceStyleGenerator",
            strategy = "enhanced-sequence",
            parameters = {
                @Parameter(name = "optimizer", value = "pooled-lo"),
                @Parameter(name = "initial_value", value = "1"),
                @Parameter(name = "increment_size", value = "10") })
    @GeneratedValue(strategy = GenerationType.SEQUENCE,
                        generator = "sequenceStyleGenerator")
    private Long id;

}
```

Hibernate'e özgü pooled veya pooled-lo algoritmalarını kullanarak sequence değerinin blok max veya min değer olarak kullanılması da sağlanabilir

Bileşke Primary Key

- İkincil önbellek kullanımı sırasında PK lookup nedeni ile bileşke primary key sınıfı
  - **Serializable** olmalıdır
  - **equals** ve **hashCode** metotlarını implement etmelidir

Bileşke Primary Key: 1. Yol

```
@Embeddable
public class UserId implements Serializable {
    @Column(name   = "FIRST_NAME")
    private String firstName;

    @Column(name   = "LAST_NAME")
    private String lastName;
    //…
}
```

```
@Entity
public class User {
    @Id
    private UserId id;
    //…
}
```

Bileşke Primary Key: 2. Yol

```
public class UserId implements Serializable {

    private String firstName;

    private String lastName;
    //…
}
```

UserId sınıfının üzerinde hiçbir anotasyon yoktur

User sınıfı içerisinde Id property'si tanımlanırken sütun isimleri belirtilir

```
@Entity
public class User {
   @EmbeddedId
   @AttributeOverrides({
      @AttributeOverride(name   = "firstName",
                column = @Column(name="FIRST_NAME") ),
      @AttributeOverride(name   = "lastName",
                column = @Column(name="LAST_NAME") )})
   private UserId id;
}
```

Bileşke Primary Key: 3. Yol

```
public class UserId implements Serializable {
    private String firstName;
    private String lastName;
    //…
}
```

```
@Entity
@IdClass(UserId.class)
public class User {
    @Id
    @Column(name="FIRST_NAME")
    private String firstName;

    @Id
    @Column(name="LAST_NAME")
    private String lastName;
    //…
}
```

Property Erişiminin Özelleştirilmesi

- @Access anotasyonu ile bir sınıfın **spesifik bir property'sinin erişim yöntemi** değiştirilebilir
- Sınıf düzeyinde kullanılırsa **bütün property'ler**e etki eder
- Bileşenler **ait olduğu parent sınıfın yöntemi**ni kullanır
- @MappedSuperclass property'leri **alt sınıf entity'nin yöntemi** ile erişilir
- İstenirse @**Access** anotasyonu ile bileşenlerin veya mappedsuperclass sınıflarının veya subclass'ların property erişim yöntemi de değiştirilebilir

Property Erişiminin Özelleştirilmesi

```
@Entity
@Table(name="T_PET")
public class Pet {

    @Id
    @GeneratedValue
    private Long id;

    @Column(name="NAME")
    private String name;


    @Temporal(TemporalType.DATE)
    @Column(name="BIRTH_DATE")
    @Access(AccessType.PROPERTY)
    private Date birthDate;
```

```
    public Long getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Date getBirthDate() {
        return birthDate;
    }

    public void setBirthDate(Date birthDate) {
        this.birthDate = birthDate;
    }
}
```

Erişim stratejisi field olarak belirlenmesine rağmen,birthDate Property'sinin erişim stratejisi getter level'a dönüştürülmüştür. Dolayısı ile getter/setter metotlarının her ikisine de ihtiyaç vardır

Field/Property Düzeyinde Erişim ve Proxy Nesneler

```java
@Entity
public class Foo {

    @Id
    @GeneratedValue
    private Long id;

    public Long getId() {
        return id;
    }

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

```java
@Entity
public class Foo {

    private Long id;

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

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

```java
Foo foo = session.load(Foo.class, 1L);

long id = foo.getId();
```

Soldaki field level erişim tanımı söz konusu olduğu vakit getId() metoduna erişim anında bir SELECT çalıştırılır

Sağdaki getter level erişim tanımı söz konusu olduğu vakit ise getId() metoduna Erişim herhangi bir SELECT tetiklemeden id değeri dönülür

Bileşenlerin Eşleştirilmesi

- Bileşenin **bütün property'leri NULL** ise **bileşen değeri NULL** kabul edilir
- Bütün property değerleri NULL olan bir bileşen de persist edilip tekrar yüklenmek istenirse **Hibernate, DB'den NULL dönecektir**
- Bunun önüne geçmek için bileşenin alanlarından birine **default bir değer atayarak persist etmek** çözüm olabilir

Bileşenlerin Ait Oldukları Entity Nesneye Referans Vermesi

```
@Entity
public class User {
    //…
    private Address homeAddress;
    //…
}
```

```
@Embeddable
public class Address {

    @Parent
    private User user;
    //…
}
```

@Parent annotasyonu ile bileşen içerisinden ait olduğu entity nesneye referans verilebilir.

Böylece @ManyToOne mapping'e gerek kalmaz.

Hibernate'e özel annotasyondur

Entity Tanımları ve SQL İfadeleri

JAVA Eğitimleri

- Normalde Hibernate SessionFactory build aşamasında bütün entity'ler için INSERT, DELETE, UPDATE SQL ifadelerini oluşturur
- **UPDATE ifadesi bütün alanları içermektedir**
- **@DynamicInsert** ve **@DynamicUpdate** ile runtime da sadece değişen alanlar için dinamik SQL üretilmesi sağlanabilir

```
@Entity
@org.hibernate.annotations.DynamicInsert(value = true )
@org.hibernate.annotations.DynamicUpdate(value = true )
public class Owner {

}
```

Türetilmiş Property Değerleri

- **@Formula** annotasyonu ile property değeri runtime da çalışacak bir SQL ifadesi ile elde edilebilir
- Property değeri **salt okunur**dur
- Formula attribute'u **SQL** fonksiyonları çağırabilir, subselect içerebilir

```
@Entity
public class Item {

    @org.hibernate.annotations.Formula("PRICE + TAX_RATE * PRICE")
    private BigDecimal priceWithTax;
}
```

Generated Property Değerleri

- Bazı property değerleri DB tarafında,**trigger vs** ile üretilebilir
- **@Generated** ile SQL INSERT ve UPDATE işlemlerinden hemen sonra çalışacak bir SELECT ile property değeri DB'den yüklenebilir

```
@Entity
public class Document {

    @Column(name = "LAST_MODIFIED",
            insertable = false, updateable = false)
    @org.hibernate.annotations.Generated(
       org.hibernate.annotations.GenerationTime.ALWAYS)
    private Date lastModified;

}
```

GenerationTime.INSERT
GenerationTime.ALWAYS
GenerationTime.NEVER
değerlerinden birisini alabilir

Generated Property Değerleri

- Property değerinin zaman zaman **JVM içerisinde dinamik olarak üretilmesi** gerekebilir
- Böyle bir durumda **@GeneratorType** anotasyonu kullanılmalıdır

```
@Entity
public class Item {

    @org.hibernate.annotations.GeneratorType(
        type = PriceWithTaxValueGenerator.class,
        when = org.hibernate.annotations.GenerationTime.ALWAYS)
    private BigDecimal priceWithTax;
}
```

Entity üzerinde yapılacak INSERT ve UPDATE işlemleri sırasında **ValueGenerator** arayüzünü implement eden sınıftan bir nesne ile priceWithTax değeri uygulama içerisinde dinamik olarak üretilecektir

@Formula'ya benzer, ancak @Formula'da değer SQL ifadesi ile oluşturulmaktadır

UPDATE SQL Sırasında Gereksiz Çalışan Trigger'lar

- **Detached** nesne **update()** veya **saveOrUpdate()** yapıldığında snapshot mevcut olmadığından Hibernate **mutlaka update SQL** ifadesini çalıştırır
- Bu da **gereksiz UPDATE**'lere ve **trigger'ların gereksiz yere çalışmasın**a yol açabilir
- Bunun önüne geçmek için **@SelectBeforeUpdate** anotasyonu kullanılabilir

```
@Entity
@SelectBeforeUpdate
public class Pet {
    ...
}
```

Default Property Değerleri

- Yeni bir kayıt INSERT ederken **bazı sütunlara default değer set edilmesi** de generated property'nin özel bir halidir
- Bu sayede property değeri **uygulama tarafında NULL bırakılmış ise** DB tarafında otomatik olarak **default değer** atanacaktır
- INSERT işleminden sonra yapılacak bir SELECT ile **@Generated property değeri** DB'den yüklenir
- Bu özellik için entity'nin **dinamik insert** özelliği aktive edilmelidir

Default Property Değerleri

```
@Entity
@DynamicInsert
public class Item {

    @Column(name = "INITIAL_PRICE",
        columnDefinition = "number(10,2) default 1")
    @org.hibernate.annotations.Generated(
        org.hibernate.annotations.GenerationTime.INSERT)
    private BigDecimal initalPrice;

}
```

Dinamik Temporal Property Değerleri

- @GeneratorType'a benzer ancak **sadece temporal tipteki propertyler** için kullanılabilirler
- **@CreationTimestamp** veya **@UpdateTimestamp** ile işaretlenmiş property'nin değeri yeni kayıt veya güncelleme sırasında **JVM'in güncel zaman değeri** olarak set edilecektir

Dinamik Temporal Property Değerleri

```
@Entity
public class Document {

    @Column(name = "CREATION_TIME")
    @org.hibernate.annotations.CreationTimestamp
    private Date creationTime;

    @Column(name = "LAST_MODIFIED")
    @org.hibernate.annotations.UpdateTimestamp
    private Date lastModified;

}
```

java.util.Date
java.util.Calendar
java.sql.Date
java.sql.Time
java.sql.Timestamp gibi tiplerde kullanılabilir

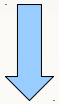
Global Metadata

- **Birden fazla sınıfa etki edecek**, uygulama genelinde kullanılacak **metadata tanımları**na da ihtiyaç duyulmaktadır
- Örneğin,
  - İsimlendirilmiş sorgular (**named query**)
  - Uygulamaya özel eşleştirme tipleri (**UDT**)
  - veri filtre tanımları (**filter**) global metadata'dır
- Global metadata sınıflarda tanımlanabilir
- Ancak tek bir yerde toplanmaları daha iyi bir pratiktir
- Bu toplama işlemi herhangi bir paket altına konan **package-info.java** ile yapılabilir

package-info.java

```
@TypeDefs(value={
@TypeDef(name="money-simple",typeClass=MoneyUserType.class),
@TypeDef(name="money-composite",typeClass=MoneyCompositeUserType.class),
@TypeDef(name="money-
parameterized",typeClass=MoneyParameterizedUserType.class,parameters={@Par
ameter(name="dbCurrencyCode",value="TRL")})})
package com.javaegitimleri.petclinic.model;

import org.hibernate.annotations.Parameter;
import org.hibernate.annotations.TypeDef;
import org.hibernate.annotations.TypeDefs;
```



```
<hibernate-configuration>
    <session-factory>
        ...
        <mapping package="com.javaegitimleri.petclinic.model"/>
    </session-factory>
</hibernate-configuration>
```

package-info.java ve JPA

- **JPA spesifik anotasyonlar** package-info.java içerisinde **kullanılamaz**
- Dolayısı ile isimlendirilmiş sorguları package-info.java içerisinde tanımlamak için **Hibernate anotasyonları**na ihtiyaç duyulur

Şema Export ve Örnek Data Import İşlemi

JAVA Eğitimleri

- Hibernate root classpath'de yer alan **import.sql** isimli bir dosyadaki SQL ifadelerini şema export işlemi sonrası çalıştırabilir
- Bunun için **hibernate.hbm2ddl.auto** değeri "**create**" veya "**create-drop**" olmalıdır

BLOB/CLOB Örneği

- java.lang.String, char[], Character[], veya java.sql.Clob tipi, **CLOB** sütunla eşlenmek istenirse **@Lob** anotasyonu kullanılır
- Benzer biçimde byte[], Byte[], or java.sql.Blob tipleri de **@Lob** anotasyonu kullanılarak **BLOB** sütunla eşlenir

```
@Entity
public class Image {
    ...

    @Lob
    private String description;

    @Lob
    private byte[] content;
}
```

```
@Entity
public class Image {
    ...

    @Lob
    private java.sql.Clob description;

    @Lob
    private java.sql.Blob content;
}
```

BLOB/CLOB Değer Oluşturmak

BLOB/CLOB değer elde etmek için açık bir Session'a ihtiyaç vardır

byte[] veya InputStream tipinde bir nesnedir

```
Blob blobContent = Hibernate.getLobCreator(session).createBlob(content);

Clob clobDesc = Hibernate.getLobCreator(session).createClob(description);
```

String veya Reader tipinde bir nesnedir

```
Image image = new Image();
image.setContent(blobContent);
image.setDescription(clobDesc);
```

BLOB/CLOB Değerlerin DB'den Yüklenmesi

JAVA Eğitimleri

- İlgili entity yüklendiğinde LOB property aslında bir locator nesnedir, yani **pointer**'dır
- LOB değer propety **erişildiğinde yüklenir**
- Yükleme ancak Hibernate **Session açık** olduğu müddetçe yapılabilir
- Ayrıca LOB değerlerin bu lazy davranışı **DB sürücülerine göre** değişiklik gösterebilir
- Eğer DB sürücüsü **desteklemiyorsa** LOB **değer eager yüklenecek**tir

Bazı JDK Tiplerin DB Karşılıkları

Java Tipi	Eşleme Tipi	SQL Tipi	Açıklama
java.lang.Class	class	VARCHAR	Sınıfın FQN ismi saklanır
java.util.Locale	locale	VARCHAR	Locale nesnesinin String gösterimi, örneğin tr_TR, saklanır
java.util.Timezone	timezone	VARCHAR	Timezone nesnesinin String gösterimi, örneğin Europe/Istanbul, saklanır
java.util.Currency	currency	VARCHAR	Currency nesnesinin String gösterimi, örneğin TRY, saklanır

Custom Tiplerle Çalışmak

- Java değeri ile DB değeri arasında **dönüşüm yapılması gerektiğinde** custom tiplere başvurulur
- Basit şekilde custom tip tanımlamak için **UserType** kullanılır
- **CompositeUserType** daha gelişmiş halidir, HQL sorgularında kompleks bir tipin property'lerine de refer etme imkanı sağlar
- **ParameterizedType** arayüzünü implement eden custom tiplere mapping sırasında parametre geçmek mümkündür

Enum Tiplerin Custom Tip ile Ele Alınması

- Eğer Enum tiplerin **ORDINAL veya STRING gösterimleri dışında bir değer**in veritabanında tutulması istenirse bu durumda **custom user type** veya JPA **AttributeConverter** yazmak gerekir

```
public enum Status {
    PENDING(0),SUCCESS(1),FAILED(-1);

    private int code;

    private Status(int code) {
        this.code = code;
    }

    public int getCode() {
        return code;
    }
}
```

Custom UserType Örneği

```
public class StatusEnumType implements UserType {
    @Override
    public void nullSafeSet(PreparedStatement st, Object value, int index,
                                SharedSessionContractImplementor session)
                                throws HibernateException, SQLException {
        if (value == null) {
                st.setNull(index, Types.INTEGER);
        } else {
                st.setInt(index, ((Status)value).getCode());
        }
    }
    @Override
    public Object nullSafeGet(ResultSet rs, String[] names,
                              SharedSessionContractImplementor session,
                              Object owner) throws HibernateException, SQLException {
        int code = rs.getInt(names[0]);
        if(rs.wasNull()) {
                return null;
        }
        for(Object value : returnedClass().getEnumConstants()) {
            if(code == ((Status)value).getCode()) {
                return value;
            }
        }
        throw new IllegalStateException("Unknown code " + code + " for Status");
    }
...
}
```

Custom UserType Kullanımı

```
@TypeDef(name="statusEnumType",typeClass=StatusEnumType.class)
@Entity
public class Message {

    @Id
    private Long id;

    @Type(type="statusEnumType")
    @Column(name="status_code", columnDefinition="integer")
    private Status status;

    ...
}
```

JPA AttributeConverter

- JPA 2.1 ile gelen bir kabiliyettir
- Hibernate'deki **custom type kabiliyeti**ne benzer, **ancak daha kısıtlı**dır

```
public interface AttributeConverter<X,Y> {

    public Y convertToDatabaseColumn (X attribute);


    public X convertToEntityAttribute (Y dbData);
}
```

Java tipleridir

Attribute değerini DB değerine dönüştürür

DB değerini Entity attribute değerine dönüştürür

JPA AttributeConverter Örneği

JAVA Eğitimleri

autoApply=true sayesinde Status enum tipi Entity içerisinde doğrudan kullanılabilir. Aksi durumda attribute üzerinde @Convert anotasyonu ile converter sınıfının explicit biçimde belirtilmesi gerekir

```
@Converter(autoApply=true)
public class StatusConverter
    implements AttributeConverter<Status, Integer> {

    @Override
    public Integer convertToDatabaseColumn(Status attribute) {
        if(attribute == null) return null;
        return attribute.getCode();
    }

    @Override
    public Status convertToEntityAttribute(Integer dbData) {
        if(dbData == null) return null;
        for(Status status:Status.values()) {
            if(status.getCode() == dbData) return status;
        }
        return null;
    }
}
```

İletişim

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