Bölüm-6

ORM Yönetimi ve Hibernate Kütüphanesi

Hibernate, ORM (Object Relational Mapping) kütüphanesidir. Biliyoruz ki Nesneye Dayalı Programlama paradigması da ilişkisel veri tabanları da varlıkları, kavramları ve nesneleri modellemek için uğraşmaktadırlar. Veri tabanında tablolar ile oluşturduğumuz veri modeli aslında uygulamamızdaki veri modelini de belirler. Java tarafında yazılmış olan sınıflar aynı şekilde veri tabanında yer alan tablolara göre şekillenmektedir. Geliştirdiğimiz projelerde Data Layer (DAO Veri-Katmanı) veri tabanıyla etkileşimde olan en alt basamaktır. Direkt olarak veri tabanıyla muhatap olur. Veri tabanından aldığı sonuçları bir üst katman olan Business Layer (İş Katmanı-Servis katmanı) katmanına iletir. Ayrıca, Business Layer'dan aldığı istekleri de veri tabanına iletilir. Çift yönlü ilişki içindedir.

Dolayısıyla DAO katmanında yer alan sınıflar veri tabanındaki yapıyla neredeyse çok benzerlik göstermektedir. Çünkü, bu sınıflar veri tabanındaki veri modelini Java tarafında modeller, yani bir bakıma veri tabanındaki veri modelinin eşleniği gibidir. İşte Java'daki sınıflar ile veri tabanındaki tablolar arasındaki çift taraflı dönüştürülme işine ORM (Object Relational Mapping) denir.

Aşağıdaki Java sınıfı ile ilişkisel veri tabanındaki tabloya ait DDL SQL komutunu incelediğiniz de aslında her ikisinin de aynı varlığı aynı nesneyi modellediğini görebiliriz.

Java Kodu

```java
public class Employee {
   private int id;
   private String first_name;
   private String last_name;
   private int salary;

   public Employee() {}
   public Employee(String fname, String lname, int salary) {
      this.first_name = fname;
      this.last_name = lname;
      this.salary = salary;
   }

   public int getId() {
      return id;
   }

   public String getFirstName() {
      return first_name;
   }

   public String getLastName() {
```

```
      return last_name;
   }

   public int getSalary() {
      return salary;
   }
}
```

SQL Komutu

```
create table EMPLOYEE (
   id INT NOT NULL auto_increment,
   first_name VARCHAR(20) default NULL,
   last_name  VARCHAR(20) default NULL,
   salary     INT  default NULL,
   PRIMARY KEY (id)
);
```

Görüldüğü gibi veri yapısı bütünüyle neredeyse aynıdır. Bu aslında tablodan nesneye, nesneden tabloya otomatik bir dönüşüm yapılabileceğini gösterir.

İşte ORM yukarıda bahsettiğimiz bu dönüşümü yöneten kütüphanedir.

ORM'nin Avantajları

- Java uygulaması içinde veri tabanı tablolarıyla direkt olarak SQL vasıtasıyla etkileşim yerine bu tabloların Java'da karşılığı olarak yaratılmış nesneleriyle çalışmak hem Nesneye Dayalı Programlama pratiğine daha uygundur.
- SQL çalıştırmak için gerekli olan birçok detayından bizi kurtarır. Daha basit şekilde veri tabanı ile etkileşim kurarız. Bu etkileşimi Hibernate gibi ORM araçları bizim sorumluluğumuzdan alır kendileri üstlenirler.
- ORM kütüphaneleri, dolayısıyla Hibernate alt yapı olarak JDBC API'yi kullanır.
- Hibernate ile veri tabanından bağımsız yeniden kullanılabilir kodlar yazılabilir. Örneğin MySQL veri tabanından PostgreSQL veri tabanına geçişte kodlarınızda bir değişiklik gerektirmez.
- Transaction yönetimi Hibernate tarafından otomatik olarak yürütülebilir.
- En önemlisi JDBC kullanılan projelerde projeye büyüdükçe iyice karmaşık bir kod mimarisi oluşabilir. Yazılımcı kodladığı iş mantığı dışında bu durumlar içinde kod düzeltmesi ve zaman harcamak zorunda kalır. Hibernate ile kurumsal projelerde karmaşıklık azaltılır. Yazılımcının yazdığı iş mantığına odaklanması sağlanır. Hızlı bir geliştirme yapmaya imkan verir.

Hibernate

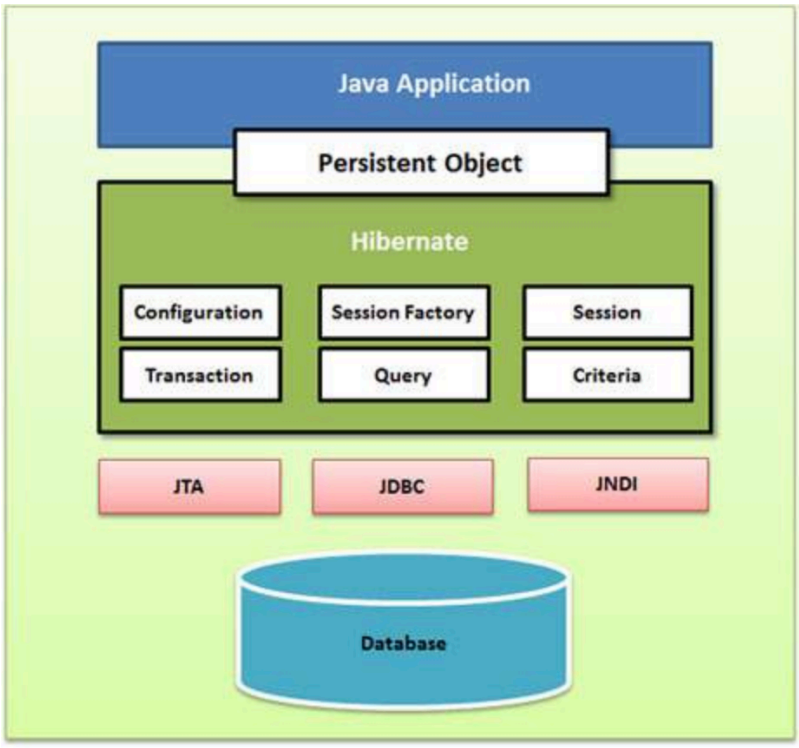
2001 yılında ortaya çıkmıştır. Dediğimiz gibi görevi Java'daki veri tipleriyle, SQL veri tiplerinin birbirine dönüşümünü sağlayarak Java sınıflarını veri tabanı tablolarına eşleştirir.


Hibernate birçok ilişkisel veri tabanını destekler.

HSQL Database Engine
DB2/NT
MySQL
PostgreSQL
FrontBase
Oracle
Microsoft SQL Server Database
Sybase SQL Server
Informix Dynamic Server

JDBC API da bildiğimiz üzere veri tabanı bağımsız bir şekilde çalışmayı sağlıyordu. JDBC ile Java'da yazdığımız kodlar veri tabanı yönetim sisteminin ne olduğundan bağımsız bir şekilde esnek yapıya sahipti. MySQL ile çalışan bir JDBC uygulaması yazdığınızda veri tabanı yönetim sisteminizi PostgreSQL'e veya Oracle'e çevirdiğiniz de çok ta sorun yaşanmıyordu. Önceden yazılmış olan JDBC kodları değişikliğe uğramıyordu. Fakat, JDBC ve veri tabanı arasında Nesneye dayalı Programlama'nın ihtiyacı olan boşluğu dolduracak daha soyut bir ara katman gerekiyordu. Hibernate ile bu katman sağlanmış oldu. Bugün bir çok kurumsal şirkette Hibernate tercih edilmektedir. Hatta, JDBC ile Hibernate'in hibrit olduğu projelerde mevcuttur. Not olarak belirtmekte fayda var, Hibernate üzerinden yalın SQL komutları da çalıştırabiliyoruz.

Hibernate çok katmanlı bir mimariye sahiptir. Böylece, alt yapısında kullandığı kütüphaneleri ve gereksinimleri yazılımı geliştiren kişinin bilmesine gerek bırakmaz. İyi izolasyon ve soyutluluk sağlar.




Configuration Nesnesi

Hibernate uygulaması ayağa kalkarken bir kere oluşturulur. Uygulama çalıştığı süre boyunca kullanılır. Configuration nesnesini doldurmak için bir konfigürasyon dosyası tanımlanır.

Configuration nesnesi iki tane olanak sağlar:

- Kullanılabilir veri tabanı bağlantısı sağlar.
- Java sınıflarının hangi veri tabanı tablolarıyla eşleştirildiği bilgisini tutar.

SessionFactory Nesnesi

Configuration nesnesi kullanılarak bir SessionFactory nesnesi oluşturulur. SessionFactory nesnesi ThreadSafe bir veri yapısıdır. Yani birden fazla Thread'in yer aldığı uygulamalarda sorunsuz çalışabilir.

Her veri tabanı bağlantı konfigürasyonu için aynı bir SessionFactory nesnesi oluşturulur. Yani, Java uygulamamız hem MySQL'e hem de PostgreSQL veri tabanlarına erişmek isterse bunu tek bir SessionFactory nesnesi üzerinden yapamaz. Bu nedenle her iki veri tabanı için ayrı SessionFactory nesnesi oluşturulur.

Session Nesnesi

SessionFactory nesnesi kullanılarak Session nesnesi üretilir. Session nesnesi veri tabanıyla olan kullanılabilir bağlantıyı ifade eder. Session nesnesi ile veri tabanı işlemlerini bitirdiğimizde kapatmak gerekir. Session nesnesi ThreadSafe değildir. Session'la işimiz bittiğinde kapatmakta fayda vardır.

Transaction Nesnesi

Transaction nesnesi JDBC ile JTA (Java Transaction API) alt yapısını kullanarak ilişkisel veri tabanlarında Transaction yönetimini sağlar.

Query Nesnesi

Veri tabanı tablolarında veri sorgulamak, silmek, güncellemek ve veri eklemek gibi işlemleri yapabilmesi sağlayan nesnedir. SQL ve HQL (Hibernate Query Language) dilleriyle veri üzerinde işlem yapabilmeyi sağlar.

Criteria Nesnesi

Nesneye dayalı programlama yöntemiyle sorgu yazmadan, sorgu nesnelerini bir araya getirerek veri tabanı tablolarında işlem yapabilmeyi sağlar. Query nesnesinin alternatifi olarak kullanılabilir.

Hibernate Konfigürasyon Özellikleri

**hibernate.dialect**

Seçilmiş olan veri tabanı yönetim sistemine göre SQL üretmeyi sağlar.

**hibernate.connection.driver_class**

JDBC Driver (Sürücü) sınıfıdır. JDBC örnekleri yaparken bunu kullanmıştık.

**hibernate.connection.url**

Veri tabanına bağlantıyu sağlayan JDBC URL'dir. JDBC uygulamalarında kullanmıştık.

**hibernate.connection.username**

Veri tabanı kullanıcı adı

**hibernate.connection.password**

Veri tabanı kullanıcı şifresi

**hibernate.connection.pool_size**

Veri tabanı bağlantı havuzunun maksimum kaç bağlantı nesnesini tutabileceğini belirtir. Biliyorsunuz Pooling diye bir tasarım deseni vardı. Böylece, kaynakların verimli kullanılmasını sağlamış oluyorduk.

**hibernate.connection.autocommit**

Autocommit özelliği açılıp kapatılabilir. Autocommit açık olursa veri üzerinde yapılan değişiklik otomatik olarak hemen veri tabanına yansır. JDBC'de de bunun örneğini yapmıştık.

Hibernate Annotation'lar ile Entity Tanımlamak

Hibernate Entity sınıfları veri tabanı tarafındaki tabloları modelleyen Java sınıflarıdır. Java uygulamasında kodlama yaparken bu sınıf tablo işlevini görecektir. Bu Entity sınıflarından üretilen her nesne veri tabanı tablosunda bir kaydı ifade edecektir. Tablodaki her satırın Java'da temsili bu sınıflara ait nesnelerdir.

Örnek Entity

```
@Entity
@Table(name = "EMPLOYEE")
public class Employee {
   @Id @GeneratedValue
   @Column(name = "id")
   private int id;

   @Column(name = "first_name")
   private String firstName;

   @Column(name = "last_name")
   private String lastName;

   @Column(name = "salary")
```

```
   private int salary;

   public Employee() {}

   public int getId() {
      return id;
   }

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

   public String getFirstName() {
      return firstName;
   }

   public void setFirstName( String first_name ) {
      this.firstName = first_name;
   }

   public String getLastName() {
      return lastName;
   }

   public void setLastName( String last_name ) {
      this.lastName = last_name;
   }

   public int getSalary() {
      return salary;
   }

   public void setSalary( int salary ) {
      this.salary = salary;
   }
}
```

Yukarıdaki örnekte “Employee” isminde bir sınıf oluşturup, veri tabanı tarafında “EMPLOYEE” isimli tabloyu modelliyoruz. Böylece, Hibernate bu Entity sınıfını gördüğünde bunun veri tabanında bir tablo ile temsil edildiğini anlayacaktır.

@Entity annotation ile bu sınıfı işaretliyoruz. @Table annotation ile bu oluşturduğumuz Entity sınıfının veri tabanı tarafında hangi tabloya karşılık geldiğini belirtiyoruz.

Biliyorsunuz ki tablolarda Primary Key alanlar yer alırdı. Primary Key alanlar tablodaki her satırın eşsiz yani tekil olması sağlardı. Java sınıfı içinde hangi değişkenin Primary Key alan

olduğunu @Id annotation ile belirtiyoruz. Ayrıca, @GeneratedValue etiketiyle bu tekil alanın değerinin ne şekilde artış gösterdiğini belirtiyoruz.

Sonrasında Java sınıfı içindeki değişkenlerimizin tabloda hangi sütunlara denk geldiğini işaretliyoruz. Bunu yaparken @Column annotation'ı kullanıyoruz.

## HQL Sorgulama Dili

Hibernate'in kendine ait bir sorgulama dili vardır. Bu sorgulama diliyle yazılanlar en nihayetinde veri tabanına iletilirken SQL cümlelerine çevrilirler.

```
String hql = "SELECT E.firstName FROM Employee E";
Query query = session.createQuery(hql);
List results = query.list();
```

```
String hql = "FROM Employee E WHERE E.id = :employee_id";
Query query = session.createQuery(hql);
query.setParameter("employee_id",10);
List results = query.list();
```

```
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);
```

```
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);
```

```
String hql = "INSERT INTO Employee(firstName, lastName, salary)"  +
             "SELECT firstName, lastName, salary FROM old_employee";
Query query = session.createQuery(hql);
int result = query.executeUpdate();
System.out.println("Rows affected: " + result);
```

Criteria Query ile Veri Tabanı İşlemleri

HQL veya yalın SQL kullanarak sorgulama yapmak istemeyenler için alternatif olarak Criteria Query API kullanarak tablolarla etkileşeme geçen kodlar yazabililer.

```
Criteria cr = session.createCriteria(Employee.class);
List results = cr.list();
```

```
Criteria cr = session.createCriteria(Employee.class);
cr.add(Restrictions.eq("salary", 2000));
List results = cr.list();
```

```
Criteria cr = session.createCriteria(Employee.class);

// To get records having salary more than 2000
cr.add(Restrictions.gt("salary", 2000));

// To get records having salary less than 2000
cr.add(Restrictions.lt("salary", 2000));

// To get records having fistName starting with zara
cr.add(Restrictions.like("firstName", "zara%"));

// Case sensitive form of the above restriction.
cr.add(Restrictions.ilike("firstName", "zara%"));

// To get records having salary in between 1000 and 2000
cr.add(Restrictions.between("salary", 1000, 2000));

// To check if the given property is null
cr.add(Restrictions.isNull("salary"));

// To check if the given property is not null
cr.add(Restrictions.isNotNull("salary"));

// To check if the given property is empty
cr.add(Restrictions.isEmpty("salary"));

// To check if the given property is not empty
cr.add(Restrictions.isNotEmpty("salary"));
```

AND ve OR yapılarıyla koşulları birleştirmek

```
Criteria cr = session.createCriteria(Employee.class);

Criterion salary = Restrictions.gt("salary", 2000);
```

```
Criterion name = Restrictions.like("firstNname","zara%");

// To get records matching with OR conditions
LogicalExpression orExp = Restrictions.or(salary, name);
cr.add( orExp );

// To get records matching with AND conditions
LogicalExpression andExp = Restrictions.and(salary, name);
cr.add( andExp );

List results = cr.list();
```

Native SQL ile Veri Tabanı İşlemleri

Hibernate kütüphanesinde HQL ve Criteria Query dışında direkt olarak yalın SQL komutları da işletebiliriz.

```
String sql = "SELECT * FROM EMPLOYEE WHERE id = :employee_id";
SQLQuery query = session.createSQLQuery(sql);
query.addEntity(Employee.class);
query.setParameter("employee_id", 10);
List results = query.list();
```