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

```
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

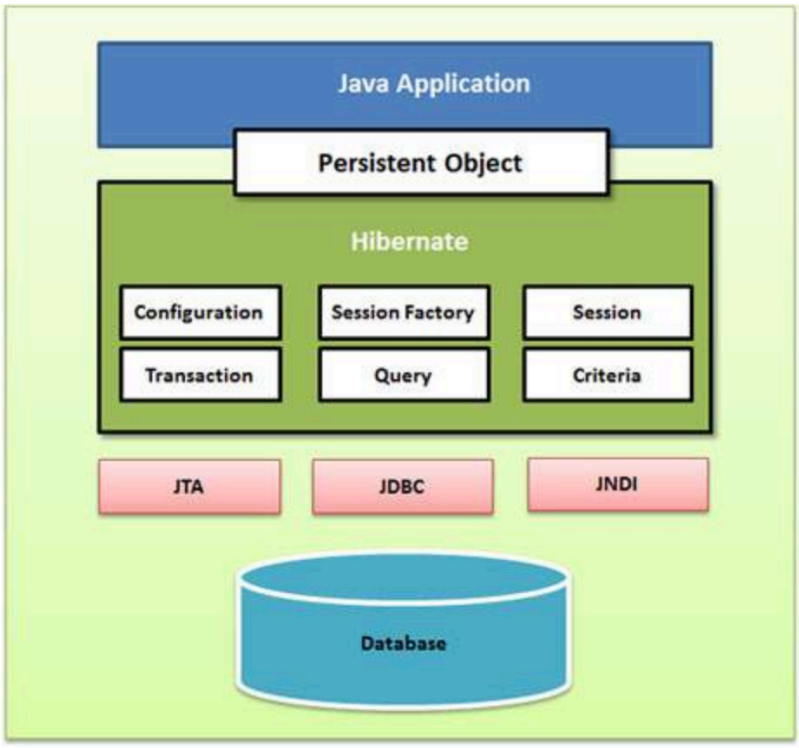
2001 yılında ortaya çıkmıştır. Dediğimiz gibi görevi Java'daki veri tipleriyle, SQL veri tiplerinin birbirine dönüşümünü sağlayarak Java sınıflarını veri tabanı tablolarına eşleştirir.

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 bağlantıyı 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 = "employees")
public class Employee {

   @Id
   @GeneratedValue
   @Column(name = "emp_no")
   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

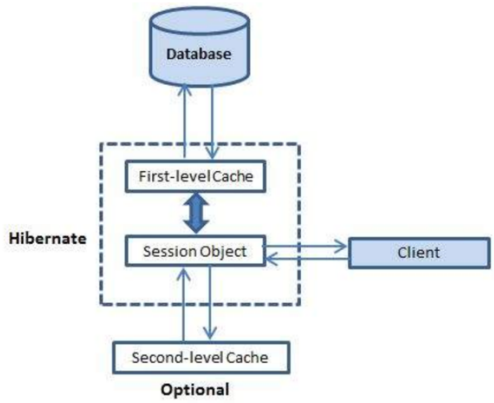
```
Criteria cr = session.createCriteria(Employee.class);

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();
```

Hibernate Cache Mimarisi



Spring Boot ile Hibernate Entegrasyonu

Hibernate kütüphanesinin bir ORM aracı olduğundan bahsetmiştik. ORM araçları ile veri tabanı ile etkileşim sorumluluğunu devretmiş oluyoruz. Aynı zamanda veri tabanımızdaki tabloları Java projesinde modelleyen sınıflar, ki biz bunlara Entity sınıfları diyoruz, elde etmiş olduk. Böylece, veri tabanındaki veriyi Java tarafında modellemiş olduk. Bu sayede veri tabanı sağlayıcımız değişse de ORM aracıyla yazdığımız kodlarda değişiklik yapmaya gerek kalmayacaktı.

Hibernate kütüphanesi de Spring Boot projelerine dahil edilip kullanılabilir. Hatta, projelerde ORM aracı olarak Hibernate tercih edilir. Spring Data kütüphanesi bu entegrasyonu sağlayan kütüphanedir. “spring-boot-starter-data-jpa” kütüphanesi ile Hibernate kütüphanesini projemize dahil etmiş oluruz.

Spring Boot projesinde “application.properties” dosyasına yazdığınız Hibernate ayarları otomatik olarak taranıp konfigürasyon olarak tanımlanır. Fakat, dilersek Hibernate ayarlarını kendimiz de belirleyebiliriz. Bunun için @Configuration etiketi ile bir sınıf tanımlamamız gerekiyor. Aşağıda manuel şekilde tanımlanmış bir Hibernate ayarları bulunmaktadır.

Bilindiği üzere Hibernate konfigürasyonlarında iki tip bilgiler tutulmaktaydı.

- Veri tabanı bağlantı bilgileri
- Entity (Java Sınıfları) ve Veri tabanı tabloları arasındaki eşleştirme

Öncelikle “chapter5-spring-hibernate” isminde bir Maven projesi oluşturuyoruz. Ardından, “pom.xml” dosyasını Spring Boot projesi olacak şekilde ayarlıyoruz.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <groupId>chapter5-spring-hibernate</groupId>
        <artifactId>chapter5-spring-hibernate</artifactId>
        <version>0.0.1-SNAPSHOT</version>

        <name>company-project</name>
        <description>Company Project</description>
        <packaging>jar</packaging>

        <properties>
                <java.version>8</java.version>
                <slogger.version>1.2.7</slogger.version>
                <commons-core.version>1.8.67</commons-core.version>
                <aspectj.version>1.9.4</aspectj.version>
        </properties>
```

```
	<parent>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-parent</artifactId>
		<version>2.1.6.RELEASE</version>
		<relativePath />
	</parent>

	<dependencies>

		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter</artifactId>
		</dependency>

		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-web</artifactId>
		</dependency>

		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-test</artifactId>
			<scope>test</scope>
			<exclusions>
				<exclusion>
					<groupId>org.junit.vintage</groupId>
					<artifactId>junit-vintage-
engine</artifactId>
				</exclusion>
			</exclusions>
		</dependency>

		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-aop</artifactId>
		</dependency>

		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-data-
jpa</artifactId>
		</dependency>

		<dependency>
			<groupId>mysql</groupId>
			<artifactId>mysql-connector-java</artifactId>
		</dependency>
```

```
                <dependency>
                    <groupId>org.apache.commons</groupId>
                    <artifactId>commons-text</artifactId>
                    <version>1.2</version>
                </dependency>

        </dependencies>

        <build>
                <finalName>company-project</finalName>
                <plugins>
                        <plugin>
                                <groupId>org.springframework.boot</groupId>
                                <artifactId>spring-boot-maven-
plugin</artifactId>
                        </plugin>
                </plugins>
        </build>

</project>
```

“pom.xml” içinde “spring-boot-starter-data-jpa” bağımlılığı eklenmiştir. Ayrıca, biliyoruz ki MySQL veri tabanına bağlantı kurabilmek için “mysql-connector-java” isminde bir kütüphane daha ekliyoruz.

Projemiz kullanıma hazır. “com.company.project” isminde bir Java paketi oluşturuyoruz.

Ardından ilk işimiz veri tabanı ile konuşacak olan DAO katmanını şekillendirmek. Ama biz istiyoruz ki Hibernate konfigürasyonumuzu kendimiz yapalım. Bu nedenle “com.company.project.dao” Java paketi oluşturup paket altında “HibernateConfiguration” isminde Java sınıfı oluşturuyoruz.

```
@Configuration
@Order(1)
@PropertySource({ "classpath:application.properties" })
public class HibernateConfig {


}
```

Yukarıdaki sınıfımız @Configuration etiketi ile işaretlendiği için Spring Framework tarafından konfigürasyon sınıfı olarak algılanacaktır. Ayrıca, @Order(1) etiketi ile eğer projede birden fazla konfigürasyon sınıfı varsa ilk önce bu konfigürasyon çalışmasını sağlıyoruz. @Order etiketi ile sıralamayı belirleyebiliyoruz. @PropertySource etiketi ise Java projesinde yer alan Resource dosyalarından belirttiğimiz dosyayı alıp okuyup “HibernateConfig” sınıfında kullanabilmemizi sağlar. “application.properties” dosyasından bahsetmiştik. Şimdi bu özellikler dosyasını dolduralım.

“src/main/resources” klasörü altına “application.properties” isminde bir dosya oluşturuyoruz. Bu dosya anahtar-değer ikilisi şeklinde veriler saklayacaktır. Bu dosyanın içine veri tabanı bağlantı bilgileri ve Hibernate ile ilgili özellikler tanımlanacaktır.

İlk önce veri tabanı bağlantı bilgilerini tanımlıyoruz.

```
spring.datasource.url=jdbc:mysql://remotemysql.com:3306/S9HHYQdP81?useSSL=false
spring.datasource.username=S9HHYQdP81
spring.datasource.password=7mR2jSrEgT
spring.datasource.driver=com.mysql.jdbc.Driver
```

Görüldüğü üzere bağlantı URL’imiz mevcut ve JDBC örneğimizdeki gibi aynı formatta olacaktır. Ardından, username ve password bilgilerini giriyoruz. Biliyoruz ki veri tabanı bağlantısı için bir kullanıcı adı ve şifre gerekiyor. Ve hangi veri tabanı sürücüsünü kullanacağımı belirtiyorum.

Artık HibernateConfig sınıfımızda “DataSource” Spring Bean’i tanımlayabilirim. “DataSource” tipinde Spring Bean, Hibernate konfigürasyonu için kullanılacaktır.

```
@Autowired
private Environment env;

@Bean
public DataSource dataSource() {

        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName(env.getProperty("spring.datasource.driver"));
        dataSource.setUrl(env.getProperty("spring.datasource.url"));
        dataSource.setUsername(env.getProperty("spring.datasource.username"));
        dataSource.setPassword(env.getProperty("spring.datasource.password"));

        return dataSource;
}
```

“DriverManagerDataSource” tipinden bir nesne üretip bunu @Bean etiketiyle Spring Bean’i olarak tanımlıyorum. Böylece, Spring Framework artık bu nesneyi Spring Context’e ekleyip bir bağımlılık nesnesi olarak sunabilecektir. Veri tabanı bağlantısıyla ilgili verileri “application.properties” içinden alıp nesneye tek tek setliyoruz. “application.properties” dosyasındaki verilere erişmemizi sağlayacak “Environment” tipinde bir bağımlılık ekliyoruz. @Autowired etiketi ile bağımlılığı sınıfa dahil ettik. “Environment” sınıfı Spring Framework’ün bize sağladığı bir sınıftır. İçinde “application.properties” dosyasındaki değerleri barındırır. Neden “application.properties” olduğunu nasıl anladı derseniz, hatırlayacaksınız ki @PropertySource isimli etiket ile hangi Resource dosyasını okuyacağımızı belirtmiştik.

Şimdi Hibernate ile ilgili konfigürasyonları yapmaya sıra geldi.

```
@Bean
public EntityManagerFactory entityManagerFactory() throws SQLException {

        HibernateJpaVendorAdapter vendorAdapter = new
HibernateJpaVendorAdapter();
        vendorAdapter.setDatabase(Database.MYSQL);

        LocalContainerEntityManagerFactoryBean factory = new
LocalContainerEntityManagerFactoryBean();
        factory.setJpaVendorAdapter(vendorAdapter);
        factory.setPackagesToScan("com.company.project.dao.entity");
        factory.setDataSource(dataSource());
        factory.afterPropertiesSet();
        factory.setJpaProperties(additionalProperties());
        return factory.getObject();
}

private final Properties additionalProperties() {

        Properties properties = new Properties();
        properties.setProperty("hibernate.hbm2ddl.auto",
env.getProperty("spring.jpa.hibernate.ddl-auto"));
        properties.setProperty("hibernate.dialect",
env.getProperty("spring.jpa.properties.hibernate.dialect"));
        properties.setProperty("hibernate.show_sql",
env.getProperty("spring.jpa.show-sql"));
        properties.setProperty("hibernate.format_sql",
env.getProperty("spring.jpa.properties.hibernate.format_sql"));
        return properties;
}
```

“EntityManagerFactory” tipinde bir Spring Bean tanımlayıp Spring Framework’ün kullanacağı bir EntityManagerFactory oluşturuyoruz.

“EntityManagerFactory” nesnesi veri tabanı için bağlantı bilgileri kapsar. Ayrıca, Entity ve veri tabanı tablolarına ait eşleştirmeyi tutar. Biliyoruz ki @Entity etiketiyle işaretlenmiş bir sınıf veri tabanı tarafında bir tablo ile eşleştirilir. O tablonun Java tarafında bir sınıf modelini temsil eder. Bu nedenle “EntityManagerFactory” aynı zamanda Entity sınıflarının hangi Java paketi altında olduğunu bilmek ister. Bu nedenle Entity sınıflarının bulunduğu Java paketini “EntityManagerFactory” nesnesine “setPackagesToScan” metoduyla setliyoruz.

Ayrıca Hibernate ile ilgili diğer ek özellikleri de “setJpaProperties” fonksiyonu ile setliyoruz.

Yukarıdaki örneğimizde “entityManagerFactory()” isimli fonksiyon ile Bean şeklinde EntityManagerFactory oluşturuyoruz. Bu Bean, Spring Framework tarafından Hibernate ile ilgili işlemler yürütülürken kullanılacaktır.

Fonksiyonun içine baktığımızda “HibernateJpaVendorAdapter” sınıfıntan bir nesne oluştuyoruz. Bu sınıf Spring Framework’ün “EntityManagerFactory” nesnesi kullanırken hangi JPA sağlayıcısını kullandığını ifade edecektir. Spring Boot normalde Hibernate kütüphanesini varsayılan olarak JPA sağlayıcısı olarak kabul eder. Fakat biz manuel konfigürasyon yaptığımız için JPA sağlayıcısının Hibernate olduğunu böylece ifade etmiş olduk.

Ardından, Hibernate Entity sınıflarının hangi paket altında olduğunu belirtmemiz gerekiyor. “com.company.project.dao.entity” paketi altına Hibernate Entity sınıflarını tanımlayacağız. Bu nedenle paket ismini veriyoruz.

“setDataSource” metodu ile bu EntityManagerFactory nesnesinin hangi veri tabanı bağlantısını kullanacağını setliyoruz. Biliyoruz ki üst adımlarda DataSource Bean tanımlamıştık, onu kullanıyoruz.

“setJpaProperties” metoduyla Hibernate için ek özellikleri de EntityManagerFactory’e veriyoruz. Bu özellikler içinde “application.properties” dosyasında tanımladıklarımız var. Örneğin: “hibernate.hbm2ddl.auto” özelliğini “none” işaretlemiştik. Bu veri tabanı tabloları üzerinde herhangi bir yapısal değişiklik yapma olduğu gibi kabul et demektir. Bilindiği gibi Hibernate Entity sınıfları üzerinden veri tabanı tarafında tablo oluşturma işlemleri yapabiliyoruz. Bunu yapabilmek için bu özelliği “create” olarak belirtirsek tabloları Entity sınıflardan yaratmaya çalışacaktır.

“show_sql” ve “format_sql” ile Hibernate kütüphanesi üzerinden geçen ve veri tabanına gidecek olan SQL cümlelerinini konsol ekranda formatlı bir şekilde görmeyi sağlar.

“EntityManagerFactory” Bean nesnemiz artık tanımlandı. Şimdi de veri tabanına fiziksel bağlantı kurmayı sağlayacak Hibernate kütüphanesinin “SessionFactory” nesnesini tanımlayalım. Biliyoruz ki bir nesneyi Spring Context’e dahil etmek istiyorsak onu @Bean etiketiyle Bean olarak tanımlayıp kullanabiliriz.

```
@Bean
@Primary
public SessionFactory getSessionFactory() throws SQLException {

        EntityManagerFactory entityManagerFactory = entityManagerFactory();

    if (entityManagerFactory.unwrap(SessionFactory.class) == null) {
         throw new NullPointerException("factory is not a hibernate
factory");
    }
    return entityManagerFactory.unwrap(SessionFactory.class);
}
```

Yukarıdaki gibi “SessionFactory” nesnesi tanımlıyoruz. Tanımlanan her “SessionFactory” nesnesi veri tabanıyla fiziksel bir bağlantı aracını temsil eder. Yani özetle her veri tabanı için ayrı bir SessionFactory tanımlanmalıdır. SessionFactory nesnesini oluşturana kadar dikkat

ederseniz her adımı zincirleme birbirine bağlayarak tanımladık. Dolayısıyla veri bir veri tabanı bağlantısı kurmak isterseniz yeni bir konfigürasyon sınıfı tanımayıp yukarıdaki adımları tekrarlayabilirsiniz.

“SessionFactory” nesnesini DAO katmanındaki sınıflarımızda bağımlılık olarak kullanacağız.

Böylece Hibernate kütüphanesini Spring Framework ile konfigüre edip kullanıma hazır hale getirdik.

Entity Sınıfların Tanımlanması

“com.company.project.dao.entity” paketi altına Entity sınıfları tanımlıyoruz. @Entity etiketi ile sınıfın bir Hibernate Entity’si olduğunu belirttik. Ardından bu Entity sınıfını veri tabanında temsil eden tablonun ismini yazıyoruz. @Table etiketi ile bu tanımı gerçekleştirdik. @Id etiketi biliyoruz ki Primary Key sütunu ifade ediyordu. @Column etiketi ile Java sınıfındaki değişken ile veri tabanındaki tablonun sütununu eşleştiriyorduk.

```
@Entity
@Table(name = "employees")
public class Employee implements Serializable{

	private static final long serialVersionUID = -8243964832840442 4L;

	@Id
	@Column(name = "emp_no")
	private Long empNo;

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

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

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

	@Column(name = "birth_date")
	@Temporal(TemporalType.DATE)
	private Date birthDate;

	@Column(name = "hire_date")
	@Temporal(TemporalType.DATE)
	private Date hireDate;
```

```
        @OneToMany(mappedBy = "employee", cascade = CascadeType.ALL, fetch
= FetchType.LAZY)
        private List<Salary> salaries;


        @ManyToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
        @JoinTable(
                        name = "dept_emp",
                        joinColumns = { @JoinColumn(name = "emp_no") },
                        inverseJoinColumns =  { @JoinColumn(name =
"dept_no") }
        )
        private List<Department> departments;


        // get and set functions
}
```

DAO Katmanının Tasarlanması

“com.company.project.dao” paketi altında DAO sınıflarını oluşturuyoruz. @Autowired etiketi ile SessionFactory sınıfına ait nesneyi bağımlılık olarak sınıfa dahil ediyoruz. “SessionFactory” veri tabanı bağlantısını sağlayan araçtır.

“openSession” fonksiyonunu kullanarak veri tabanına bir bağlantı açıyoruz. Ardından, Hibernate kütüphanesinin sağladığı “createQuery” fonksiyonu ile HQL sorgu cümlesini yazıp veri tabanından sorgulama yapıyoruz.

```
@Component
public class EmployeeDAO
{

        @Autowired
        private SessionFactory sessionFactory;


        public Long findMaxId() {

                Session session = sessionFactory.openSession();

                Query<Long> query = session.createQuery("select
MAX(e.empNo) from Employee e", Long.class);

                return query.getSingleResult();
        }

        public List<Employee> findAll() {
```

```
                Session session = sessionFactory.openSession();

                Query<Employee> query = session.createQuery("select e from
Employee e", Employee.class);

                return query.getResultList();
        }

        public Employee findById(Long empNo) {

                Session session = sessionFactory.openSession();

                Query<Employee> query = session.createQuery("select e from
Employee e where empNo = :empNo", Employee.class);
                query.setParameter("empNo", empNo);

                return query.getSingleResult();
        }

}
```

Service Katmanının Tasarlanması

İş mantığını kurguladığımız sınıflarımız Service katmanını oluşturuyordu. “com.company.project.service” paketini oluşturup altında Service sınıflarımızı oluşturalım. Biliyoruz ki @Service etiketiyle servis sınıflarımızı işaretliyoruz. Aynı zamanda DAO katmanından ihtiyaç duyduğumuz nesneleri Dependency Injection yöntemiyle sınıfa bağımlılık olarak ekliyoruz. @Autowired etiketi ile DAO bağımlılık nesnesini ekledik.

```
@Service
public class EmployeeService {

        @Autowired
        private EmployeeDAO employeeDAO;


        public Long findMaxId() {

                return this.employeeDAO.findMaxId();
        }

        public List<Employee> findAll() {

                return this.employeeDAO.findAll();
        }

        public Employee findById(Long empNo) {
```

```
                return this.employeeDAO.findById(empNo);
        }

}
```

Spring Framework ile Java Konsol Uygulaması Çalıştırmak

Spring Framework ile Java konsol uygulaması çalıştırmak için sınıfın “CommandLineRunner” interface’den kalıtım alması gerekmektedir. Bunun için “ConsoleApplication” isminde bir sınıf tasarlayıp bu interface’den kalıtım almasını sağlıyoruz.

Ardından konsol uygulamamız Service katmanındaki sınıflara ait nesnelere ihtiyaç duyacaktır. Bu sınıflar ConsoleApplication sınıfı için nesne bağımlılığıdır. Spring Framework zaten bağımlılıkları yönetme görevini üstlenmişti. Bu nedenle @Autowired etiketi ile bağımlılıkları sınıfa ekliyoruz.

```
@Component
public class ConsoleApplication implements CommandLineRunner {

        @Autowired
        private EmployeeService employeeService;

        @Autowired
        private TitleService titleService;


        // Java codes
}
```