Spring ve Hibernate Entegrasyonu

Spring Application Framework ve Hibernate ORM Framework'ün doğuşu hemen hemen aynı dönemlere rastlar. Her iki framework'de EJB spesifikasyonu etrafında şekillenen hantal kurumsal Java programlama modeline bir nevi baş kaldırı olarak ortaya çıkmışlar ve gelişim süreçlerinde kol kola yürümüşlerdir.

İlk Zamanlar

Spring Application Framework kurumsal Java projeleri geliştirmek için ille de monolitik bir uygulama sunucusuna ihtiyaç olmadığını, J2EE ve EJB spesifikasyonlarının vaaz ettiği programlama modelinin tek yol olarak görülemeyeceğini söylüyor, transaction yönetiminin uygulama sunucuları dışında da rahatlıkla gerçekleştirilebileceğini savunuyor ve POJO tabanlı bir programlama modelini tekrar gündemimize sokuyordu.

Hibernate'de hemen aynı dönemde sivrilmeye başlıyor ve nesne dünyası ile ilişkisel dünya arasındaki uyumsuzluğu gidermek için ortaya konan EJB spesifikasyonundaki "entity bean" modelinin aksine POJO tabanlı bir yöntemle, sıradan Java nesneleri ile de "persistence" ihtiyacının karşılanabileceğini söylüyordu.

Spring Application Framework'ün geliştiricileri kurumsal Java teknolojilerinde her şeyi sıfırdan kendilerinin geliştirmesinin yerine, her katmanda öne çıkan ve kendi felsefelerine uygun mimari ortaya koymaya imkan veren çözümleri, framework ve kütüphaneleri bir araya getirmeyi, kurumsal Java spesifikasyonlarının bıraktığı boşlukları dolduran bir "iskelet çözüm" olmayı hedeflemişlerdir. Bu doğrultuda da veri erişim katmanında hantal "entity bean"lere alternatif olarak POJO tabanlı Hibernate'in kullanılmasını kolaylaştıracak çözümleri ilk sürümlerinden itibaren biz geliştiricilere sunmuşlardır.

Hibernate 2 ve Spring

Spring, uygulamaların veri erişim ihtiyaçları için iki temel yöntem sunmaktadır. Bunlardan birisi veri erişim katmanını DAO tasarım örüntüsü ile iş mantığı katmanından soyutlamak ve bunun için sağlanan "**DaoSupport**" sınıflarıdır. Diğeri ise kullanılan veri erişim teknolojisinin "plumbing" olarak tabir edilen kısımlarının uygulama geliştiricilerin omuzlarından alınarak, sadece veri manipülasyon ve sorgulama işlemleri ile dönen sonuçların üzerinde işlem yapan kısımların kodlanmasını sağlayan "**Template**" yapılarıdır.
Hibernate entegrasyonu çerçevesinde de bu sınıfların karşılıkları **HibernateDaoSupport** ve **HibernateTemplate** olarak tanımlanmıştır. Hibernate tarafındaki gelişmeler sonucunda artık bu yapılar "legacy" olarak tabir edilmektedir. Ancak Spring ve Hibernate entegrasyonunun ana kısımlarının anlaşılabilmesi ve mevcut uygulamalarda da halihazırda bu yapıların kullanılmasından ötürü bu bölümde iki yapı üzerinde duracağız.

HibernateDaoSupport

Spring'in veri erişim katmanı oluşturmak için DAO örüntüsü üzerine kurulu DaoSupport sınıflarından birisi de HibernateDaoSupport üst sınıfıdır. Bu sınıfın sunduğu setSessionFactory(...) metodu ile uygulamanın konfigürasyonu sırasında ilgili DAO nesnesine SessionFactory nesnesi enjekte edilir. Daha sonra uygulama içerisinden HibernateDaoSupport sınıfından türeyen bu alt sınıfların nesnelerinde SessionFactory ve HibernateTemplate nesnelerine getSessionFactory() ve getHibernateTemplate() metotları ile erişilebilmektedir.

```
public class ProductDaoImpl extends
HibernateDaoSupport implements
ProductDao {

    public Collection
loadProductsByCategory(String category)
throws DataAccessException {
        return
getHibernateTemplate().find(
            "from Product p where
p.category=?", category);
    }
}
```

HibernateDaoSupport üst sınıfı, veri erişim işlemleri sırasında halihazırdaki Hibernate Session nesnesine erişim sunarak işlem yapılmasını, ayrıca DAO nesnesinin metotlarında meydana gelen exception'ların Spring'in genel DataAccessException hiyerarşisine dönüştürülerek ele alınmasını sağlar. DAO nesnesi içerisinde aktif Session nesnesine erişmek için getSession(...) metodu

kullanılmaktadır. Bu metoda verilen "allowCreate" boolean parametresi ile Session nesnesinin yaratılması kontrol edilir. Bu metoda geçirilen parametrenin değeri genellikle "false" olacaktır. Bu sayede halihazırda Spring'in oluşturduğu transaction ile ilişkili bir Session nesnesinin kullanılması sağlanır. HibernateDaoSupport sınıfı ile sunulan metotların benzerleri SessionFactoryUtils sınıfından da erişilebilir.

HibernateDaoSupport veya SessonFactoryUtils üzerinden yapılan veri erişim işlemlerinin artısı, veri erişim metotları içerisinden checked exception'ların fırlatılabilmesidir. Aşağıda anlatacağımız HibernateTemplate içerisindeki callback sınıflarından ise sadece RuntimeException türevi exception'lar fırlatılabilmektedir. Aslında checked exception fırlatmanın gün geçtikçe demode hale gelmesi ile buna tam olarak bir artı özellik demek de mümkün değildir.

HibernateTemplate

HibernateTemplate sınıfı ise, Hibernate'in Session arayüzünden sunduğu pek çok metodun benzerini ve bu arayüzde olmayıp ORM işlemlerinde geliştiricilerin sıklıkla ihtiyaç duydukları diğer bazı metotları sunmaktadır.

Bu sınıf ile hedeflenen Hibernate Session'a erişmeye ihtiyaç duymadan ORM işlemlerinin yapılmasıdır. Ancak HibernateTemplate tarafından sunulmayan metotlara ihtiyaç duyulması ve doğrudan Session nesnesi üzerinde işlem yapılması gerektiğinde "callback" yöntemi ile Hibernate Session nesnesine erişim sağlanmaktadır.

```
public class ProductDaoImpl implements
ProductDao {
    private HibernateTemplate ht;

    public void
setSessionFactory(SessionFactory
sessionFactory) {
        ht = new
HibernateTemplate(sessionFactory);
    }

    public Collection
loadProductsByCategory(String category)
throws DataAccessException {
      return ht.find("from Product p
where p.category=?", category);
    }
}
```

HibernateTemplate Session nesnelerinin uygun biçimde açılıp, kapatılması, otomatik biçimde Spring'in transaction altyapısına müdahil olmasını sağlamaktadır. HibernateTemplate nesneleri tamamen thread-safe olup, yeniden kullanılabilir nesnelerdir. Bu nedenle Spring ApplicationContext genelinde tek bir HibernateTemplate nesnesi yeterli olmaktadır.

SessionFactory Konfigürasyonu

Spring programlama modelinin temel özelliklerinden birisi uygulama kodu içerisinde ihtiyaç duyulan kaynaklara, JNDI lookup gibi kod içerisinde doğrudan lookup yapılmamasıdır. Veri erişim işlemleri için gerekli olan JDBC DataSource ve Hibernate SessionFactory nesneleri Spring container içerisinde bean olarak tanımlanır ve uygulama koduna doğrudan enjekte edilirler.

```
<bean id="dataSource"
class="com.mchange.v2.c3p0.ComboPooledD
ataSource">
  <property name="driverClass"
value="org.hsqldb.jdbcDriver"/>
  <property name="jdbcUrl"
value="jdbc:hsqldb:hsql://localhost"/>
  <property name="user" value="sa"/>
  <property name="password" value=""/>
</bean>

<bean id="sessionFactory"
class="org.springframework.orm.hibernat
e3.LocalSessionFactoryBean">
  <property name="dataSource"
ref="dataSource"/>
  <property name="mappingResources">
    <list>
      <value>product.hbm.xml</value>
    </list>
  </property>
  <property name="hibernateProperties">
    <value>
      hibernate.dialect=org.hibernate.di
alect.HSQLDialect
    </value>
  </property>
</bean>
```

Hibernate ile çalışırken SessionFactory nesnesinin JNDI context'e bağlanması söz konusu olabilir. Böyle bir durumda JNDI lokasyonlu SessionFactory nesnesi Spring'in JndiObjectFactoryBean factory bean'ı veya Spring 2.0 ile birlikte gelen <jee:jndi-lookup> namespace konfigürasyonu vasıtası ile uygulama koduna sunulabilir. Ancak JNDI lokasyonlu SessionFactory kullanımı EJB'lerle çalışıldığı durumların dışında çok yaygın değildir.

```
<beans>
  <jee:jndi-lookup id="dataSource" jndi-
name="java:comp/env/jdbc/ds"/>
</beans>
```

Spring ile çalışırken container tarafından yönetilen JNDI lokasyonlu SessionFactory ile uygulama içerisinde yerel olarak yönetilen SessionFactory nesneleri arasında geçiş tek satır kod yazmadan konfigüratif olarak gerçekleştirilebilmektedir. Burada SessionFactory'nin nerede tutulacağı tamamen o anda kullanılan transaction stratejisine göre belirlenmelidir.

JNDI lokasyonu SessionFactory ile Spring tarafında tanımlı yerel SessionFactory arasında bir karşılaştırma yapıldığında, JNDI SessionFactory'nin çok büyük bir artısı görülemez. Yalnız Hibernate JCA kullanılırsa, SessionFactory'nin Java EE sunucusunun yönetim altyapısı üzerinden idare edilebilmesi mümkün olmaktadır. Ancak bunun ötesinde bir artı söz konusu değildir.

Spring Transaction Yönetimi ve Hibernate

HibernateTransactionManager, JtaTransactionManager gibi Spring'in hangi transaction stratejisi kullanılsa, ya da doğrudan EJB CMT veya JTA ile çalışılsa bile SessionFactory.getCurrentSession() metodu o anda container (bu container Spring olabilir, JEE container olabilir) tarafından yönetilen transactional Session nesnesini döner. Spring'in LocalSessionFactoryBean'ı da bu çalışma şekli ile tamamen uyumludur.

Spring ile çalışılırken önerilen transaction demarcation yöntemi dekleratiftir. Bu sayede transaction demarcation API çağrıları Java kodu yerine AOP transaction interceptor içerisinde toplanır. Dekleratif transaction kabiliyeti sayesinde iş mantığı katmanındaki nesneler transaction demarcation API çağrılarından arındırılmış olur, geliştiriciler sadece iş mantığı geliştirmeye odaklanabilirler. AOP transaction interceptor'ün konfigürasyonu Java annotasyonları veya XML ile yapılabilmektedir. Transaction yönetimi ile ilgili propagation, izolasyon düzeyi gibi davranışlarda konfigürasyon dosyası içerisinden değiştirilebilir.

Aşağıda attribute tabanlı konfigürasyon örneği görülmektedir.

```
public interface ProductService {
      public void
increasePriceOfAllProductsInCategory(Str
ing category);
      public List<Product>
findAllProducts();
}

public class ProductServiceImpl
implements ProductService {

    private ProductDao productDao;

    public void setProductDao(ProductDao
productDao) {
        this.productDao = productDao;
    }

    @Transactional
    public void
increasePriceOfAllProductsInCategory(fin
al String category) {
        List productsToChange =
      productDao.
loadProductsByCategory(category);
        // ...
    }

    @Transactional(readOnly = true)
    public List<Product>
findAllProducts() {
        return
      productDao.findAllProducts();
    }

}
```

Burada servis katmanındaki metotlar @Transactional annotasyonu ile işaretlenmiştir. Spring container, çalışma zamanında bu annotasyonları tespit eder ve ilgili metotlar için transaction kabiliyetini sağlar. Annotasyon tabanlı konfigürasyon XML tabanlı konfigürasyona göre çok daha basit ve anlaşılırdır. Yapılması gereken Spring container içerisinde bir TransactionManager bean'ı tanımlamak ve <tx:annotation-driven> XML elemanını koymaktan ibarettir.

```
<bean id="transactionManager"
          class="org.springframework.or
m.hibernate3.HibernateTransactionManage
r">
  <property name="sessionFactory"
ref="sessionFactory"/>

</bean>

<tx:annotation-driven/>
```

```
<bean id="productService"
class="com.speedyframework.service.ProductServiceImpl">
  <property name="productDao"
ref="productDao"/>
</bean>
```

Hem programatik transaction demarcation yapılmasını sağlayan TransactionTemplate, hem de dekleratif transaction demarcation'ı mümkün kılan TransactionInterceptor sınıflarının her ikisi de asıl transaction yönetim işini PlatformTransactionManager nesnesine havale ederler. Çalışma zamanında konfigüre edilen PlatformTransactionManager implementasyonu HibernateTransactionManager veya JtaTransactionManager olabilir. Hatta uygulamaya özel bir PlatformTransactionManager implement etmek bile mümkündür. Nihayetinde yapılması gereken, uygun stratejinin Spring container'a belirtilmesinden ibarettir. Bu şekilde konfigüratif yapı sayesinde native Hibernate transaction yönetiminden JTA tabanlı dağıtık transaction yönetimine geçiş sadece konfigürasyon ile halledilebilir bir durumdur. Böyle bir değişiklik sonrasında transaction demarcation işlemleri ve veri erişim kodları hiç problemsiz çalışacaktır.

HibernateTransactionManager, Hibernate JDBC Connection nesnesini doğrudan JDBC işlemlerinde kullanılması için dışarıya erişilebilir kılar. Bunun için ya SessionFactory nesnesinin LocalSessionFactoryBean'ın dataSource property'sine DataSource nesnesi verilerek oluşturulması gerekir, ya da HibernateTransactionManager'ın dataSource property'sine ilgili DataSource bean set edilerek hangi DataSource nesnesi için transaction senkronizasyonu yapacağı söylenir. Bu sayede Hibernate ORM işlemleri ile doğrudan JDBC API ile yapılan veri erişim işlemleri aynı transaction yönetimi içerisinde gerçekleştirilmiş olunur. Böylece tek bir veritabanının kullanıldığı durumlarda Hibernate ve JDBC API erişimlerinin transaction senkronizasyonu için JTA kullanmaya da gerek kalmaz.

Eğer tek bir veritabanına erişim söz konusu ise yerel olarak tanımlanmış bir JDBC DataSource ve Hibernate SessionFactory nesneleri Spring tarafından yönetilen transactionların çalışması için yeterlidir. Spring'in JTA transaction stratejisine ise ancak dağıtık transaction ihtiyaçları söz konusu olduğunda başvurulmalıdır. JCA üzerinden bir konfigürasyon ise container'lara özel deployment adımlarının uygulanmasını gerektirir. Tabiki bu tür bir deployment için öncelikle container'ın JCA desteğinin de olması şarttır. Örneğin WebLogic Express, JCA desteği sağlamaz. Bunun için container'ın kurumsal sürümüne sahip olmanız gerekebilir. Yerel kaynakları kullanan ve yerel transactionlarla tek veritabanına erişerek çalışan Spring uygulamaları ise herhangi bir web container'a deploy edilebilir. Buna ilave olarak bu tür bir uygulamaya ait konfigürasyon rahatlıkla container dışı ortamlarda, örneğin desktop veya birim test ortamlarında da çalıştırılabilir.

Bütün bunları değerlendirdikten sonra söyleyebileceğimiz, eğer EJB kullanmıyor iseniz yerel SessionFactory ve Spring'in transaction stratejilerinden HibernateTransactionManager veya JtaTransactionManager ile çalışmak en kolay ve hızlı yoldur.

Hibernate'in Gelişim Süreci

Hibernate sürüm 2'den sürüm 3'e geçerken önemli mimarisel değişikliklere uğramıştır. Hibernate 2 ile çalışanların temel problemlerinden birisi aynı Hibernate Session nesnesinin bir transaction boyunca nasıl yönetileceği idi. Transaction'lar genel olarak Java thread context'inde yönetildiği için Hibernate Session nesnesinin de benzer bir context'de yönetilmesi en mantıklı olanıydı. Ancak Hibernate 2 içerisinde bununla ilgili bir kabiliyet olmadığı için geliştiriciler kendilerine özel çözümler geliştirmek zorunda kalmışlardır. İşte bu noktada Spring'in transaction boyunca tek bir Hibernate Session nesnesinin kullanılmasını sağlayan altyapısı Spring ve Hibernate uygulamalarının yaygınlaşmasında önemli bir katalizör görevi görmüştür. Hibernate 3 oluşturulurken Session nesnesinin değişik context'lerde yönetilmesi pluggable bir mimari üzerine bina edilmiştir. Bunlardan en yaygın olanı Session nesnesinin web isteği boyunca bir ThreadLocal değişken içerisinde tutularak yönetildiği yapıdır.

Hibernate 2'nin en çok eleştiri aldığı diğer bir nokta ise fırlattığı exception'ların hepsinin "checked exception" olmasıydı. Veri erişim işlemleri sırasında ortaya çıkan exception'ların pek azı uygulama tarafından ele alınabilir, pek çoğu ise "fatal" olarak kabul edilir. Fırlatılan checked exception'ların, metodu çağıran istemci kod tarafında ya ele alınması ya da istemci kodunun bu exception'ları bir üst katman delege ettiğini deklere etmek zorunda kalması bazı kötü kodlama pratiklerine yol açmakadır. Bu nedenle checked exception'lara Java programcıları pek iyi gözle bakmamaktadır. Spring programlama modeli

ise veri erişim hatalarını DataAccessException üst sınıfı ile RuntimeException sınıf hiyerarşisinden türeterek ortak bir exception hiyerarşisine dönüştürmektedir. Bunun yanında da RuntimeException kullanarak ortaya çıkan hataların uygulama tarafında her ne zaman ele alınabilir ise o zaman try/catch blokları ile ele alınmasını sağlamıştır. Hibernate sürüm 3 ile exception hiyerarşisinde de bir değişikliğe gitmiş, HibernateException üst sınıfını RuntimeException'dan türer hale getirerek veri erişim hatalarını istendiği vakit ele alınabilir kılmıştır.

Hibernate3 ve Spring

Yukarıda da belirttiğimiz gibi Hibernate 3 ile birlikte contextual session desteği gelmektedir. Bu sayede Hibernate, bir transaction boyunca tek bir Session nesnesini yönetir ve veri erişim işlemlerinde kullanılmasını sağlar. Bu sayede HibernateTemplate ve HibernateDaoSupport gibi aracı sınıflara ihtiyaç duymadan düz Hibernate 3 API'si kullanarak DAO implementasyonları geliştirmek daha kolay hale gelmiştir.

```
public class ProductDaoImpl implements
ProductDao {

    private SessionFactory sf;

    public void
setSessionFactory(SessionFactory sf) {
        this.sf = sf;
    }

    public Collection
loadProductsByCategory(String category)
{
    return sf.getCurrentSession()
       .createQuery("from Product p
where p.category = ?").setParameter(0,
category).list();

    }
}
```

```
<bean id="productDao"
class="com.speedyframework.dao.ProductD
aoImpl">
  <property name="sessionFactory"
ref="sessionFactory"/>
</bean>
```

Yukarıdaki örnek Hibernate referans dokümantasyonu ve örnekleri ile hemen hemen aynıdır. Buradaki tek fark Spring tarafında yönetilen DAO nesnesinde ki SessionFactory'nin bir instance değişkeninde tutulmasıdır. SessionFactory'nin Hibernate referans dokümantasyonu örneklerindeki gibi HibernateUtil gibi bir yardımcı sınıf üzerinden uygulama genelinde statik olarak paylaşılması çok da tercih edilmemesi gereken bir yaklaşımdır. Genel bir ilke olarak da, gerekmedikçe statik değişkenlerin kullanılması önerilen bir şey değildir.

Bu tür DAO nesnesinin temel avantajı sadece Hibernate 3 API'sine bağımlı olmasıdır. Hibernate 2 ile çalışırken Spring HibernateTemplate'ın kullanılması haklı nedenlerle yaygın bir pratik olmuştu, ancak Hibernate 3 ile birlikte, Spring API'sine bağlı kalmadan DAO katmanını geliştirmek isteyen Hibernate kullanıcıları için bu tür bir kodlama daha uygundur.

Ancak burada da uygulamanın Hibernate ile çok içli dışlı olması durumu ortaya çıkmaktadır. Özellikle fırlatılan exceptionların detayından uygulama davranışının belirlenmesi söz konusu olduğunda, HibernateException'ın spesifik alt sınıflarına kadar bir bağımlılık söz konusudur. Bu noktada da karşımıza veri erişim katmanında Java Persistence API gibi uygulamayı ORM implementasyonlarından soyutlamayı hedefleyen bir yapı çıkmaktadır. Ancak şu an için pratikte bu soyutlama pek gerçekleştirilebilir durumda değildir.

Kısacası Hibernate 3 ile birlikte DAO implementasyonları oldukça kolaylaşmış, Spring'in HibernateTemplate gibi bir yardımcı sınıfına ihtiyaç neredeyse ortadan kalkmıştır. Ancak, veri erişim yöntemlerinin fırlattığı exceptionları Spring'in kendi genel veri erişim exception hiyerarşisine çevrilmesi ve transaction yönetiminde Spring transactionların kullanılmasının artıları hala sürmektedir.

Sonuç

Her ne kadar iki framework'ün geliştirici grupları arasında zaman zaman söz düelloları yaşansa da bizim kanaatimiz, her iki framework'ün de daha uzun bir süre etle tırnak gibi bir arada kullanılmaya devam edeceği yönündedir. Kış uykusundan çıkıp bahara gireli çok oldu, hatta bu yazı yayımlandığında yazın ortalarına gelmiş olacağız. Herkese Spring ve Hibernate ile bol Java'lı günler dileriz.

Yazar: ODTÜ Bilgisayar Mühendisliği'nden 1999 yılında mezun olan Kenan Sevindik o dönemden bu yana pek çok büyük ölçekli kurumsal yazılım projesinde görev yapmıştır. Halihazırda kurumsal Java teknolojileri ile yazılım geliştirme, eğitim ve danışmanlık hizmeti sunan yazarımız, değişik ortamlarda teknoloji içerikli konuşma ve sunumlar da yapmaktadır. Edindiği bilgi birikimi ve deneyimleri http://www.harezmi.com.tr/blog adresinden sanal alemde sizlerle paylaşmaktadır. Kendisi ile iletişime geçmek için ksevindik@gmail.com adresine mesaj gönderebilirsiniz.