Data Access with the Spring Framework

file:///F:\Belgelerim\Yayınlar\Java\Makaleler\Araçlar\Spring\Data%20Access%20with%20the%20Spring%20Framework.doc

Faydaları:

- Hibernate oturumlarıyla ve transactionlarıyla uğraşmadan veri erişimi
- try-catch yok
- SQLExceptionın anlamlı hatalara dönüştürülmesi. SQL hata kodlarının dahi.

Kaynak tanımlanması. Bir jdbc driver (Data source nesnesini JNDI ile bulunuyor) ve session factory:

```
<beans>

  <bean id="myDataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName">
      <value>jdbc/myds</value>              jdbc sürücüsü için
    </property>
  </bean>

  <bean id="mySessionFactory" class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
    <property name="mappingResources">
      <list>
        <value>product.hbm.xml</value>
      </list>
    </property>
    <property name="hibernateProperties">
      <props>
        <prop key="hibernate.dialect">net.sf.hibernate.dialect.MySQLDialect</prop>
      </props>
    </property>
    <property name="dataSource">
      <ref bean="myDataSource"/>
    </property>
  </bean>

  ...

</beans>
```

jdbc driverını, hibernate/jdo/jdbc daolarını, transaction managerı kolayca değiştirmek mümkün.

DAO: data access object. Veritabanı erişim nesneleri. Yani kalıcı nesneler

Data source nesnesini JNDI ile bulmak yerine, lokal olarak tanımlamak için:

```
<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
  <property name="driverClassName">
    <value>org.hsqldb.jdbcDriver</value>
  </property>
  <property name="url">
    <value>jdbc:hsqldb:hsql://localhost:9001</value>
   </property>
  <property name="username">
    <value>sa</value>
  </property>
  <property name="password">
    <value></value>
  </property>
</bean>
```

Bir DAO´nun konfigürasyonu:

Bir session factory sunmalı, arayüzünde.

```
<beans>

  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="sessionFactory">
      <ref bean="mySessionFactory"/>
    </property>
```

```
  </bean>

  ...

</beans>
public class ProductDaoImpl implements ProductDao {

  private SessionFactory sessionFactory;

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

  public List loadProductsByCategory(final String category) {
    HibernateTemplate hibernateTemplate =
      new HibernateTemplate(this.sessionFactory);

    return (List) hibernateTemplate.execute(
      new HibernateCallback() {
        public Object doInHibernate(Session session) throws HibernateException {
          List result = session.find(
           "from test.Product product where product.category=?",
           category, Hibernate.STRING);
          // do some further stuff with the result list
          return result;
        }
      }
    );
  }
}
```

Veri erişimi bir callback ve template kalıbıyla oluyor. hibernateTemplate nesnesi, işlemi, belli bir oturum açarak bir transaction içinde gerçekleştiriyor.

template nesneleri thread güvenlidirler bu yüzden kapsayan sınıfta örnek değişkenleri olarak tutulabilir.

Basit find, load, saveOrUpdate ve delete metotları için kolaylaştırıcı metotlar sunuyor, hibernate template:

```
public class ProductDaoImpl extends HibernateDaoSupport implements ProductDao {

  public List loadProductsByCategory(String category) {
    return getHibernateTemplate().find(
     "from test.Product product where product.category=?", category,
     Hibernate.STRING);
  }
}
```

extends HibernateDaoSupport yerine kompozisyon ve delegasyon uygulamalı. Hatta bunu da bir aspekt olarak tanımlamalı.

Transaction:

```
<beans>

  ...

  <bean id="myTransactionManager"
class="org.springframework.orm.hibernate.HibernateTransactionManager">
    <property name="sessionFactory">
      <ref bean="mySessionFactory"/>
    </property>
  </bean>

  <bean id="myProductService" class="product.ProductServiceImpl">
    <property name="transactionManager">
      <ref bean="myTransactionManager"/>
    </property>
    <property name="productDao">
      <ref bean="myProductDao"/>
    </property>
  </bean>
```

```
</beans>

public class ProductServiceImpl implements ProductService {

  private PlatformTransactionManager transactionManager;
  private ProductDao productDao;

  public void setTransactionManager(PlatformTransactionManager transactionManager) {
    this.transactionManager = transactionManager;
  }

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

  public void increasePriceOfAllProductsInCategory(final String category) {
    TransactionTemplate transactionTemplate = new TransactionTemplate(this.transactionManager);
    transactionTemplate.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRED);
    transactionTemplate.execute(
      new TransactionCallbackWithoutResult() {
        public void doInTransactionWithoutResult(TransactionStatus status) {
          List productsToChange = productDAO.loadProductsByCategory(category);
          ...
        }
      }
    );
  }
}
```

Hibernate template yerine transaction template kullanıyoruz. Bunun içinde istenildiği kadar veri erişim hizmeti uygulanabilir. Bir problem çıkarsa, hepsi bir transaction olarak kabul edilir.

Deklaratif Transaction Sınırları (Demarcation):

Aspektler yoluyla iş nesnelerin transaction sınırlandırma kodundan ayırabiliriz. Daha temiz kod tutabiliriz.

Ayrıca yayılma (propagation) davranışını ve izolasyon seviyesini bir konfigürasyon dosyasında tanımlamamız mümkün olur.

```
<beans>

  ...

        transaction yöneticisi:
  <bean id="myTransactionManager"
class="org.springframework.orm.hibernate.HibernateTransactionManager">
    <property name="sessionFactory">
      <ref bean="mySessionFactory"/>
    </property>
  </bean>

        Aspekt:
  <bean id="myTransactionInterceptor"
class="org.springframework.transaction.interceptor.TransactionInterceptor">
    <property name="transactionManager">
      <ref bean="myTransactionManager"/>
    </property>
    <property name="transactionAttributeSource">
      <value>
        product.ProductService.increasePrice*=PROPAGATION_REQUIRED
        product.ProductService.someOtherBusinessMethod=PROPAGATION_MANDATORY
      </value>
    </property>
  </bean>

  <bean id="myProductServiceTarget" class="product.ProductServiceImpl">
    <property name="productDao">
      <ref bean="myProductDao"/>
    </property>
  </bean>
```

```
  <bean id="myProductService" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces">
      <value>product.ProductService</value>
    </property>
    <property name="interceptorNames">      Proxynin arabiriminde bulunuyor olmalı...
      <list>
        <value>myTransactionInterceptor</value>
        <value>myProductServiceTarget</value>
      </list>
    </property>
  </bean>

</beans>
public class ProductServiceImpl implements ProductService {

  private ProductDao productDao;

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

  public void increasePriceOfAllProductsInCategory(final String category) {
    List productsToChange = this.productDAO.loadProductsByCategory(category);
    ...
  }
}
```

Kod tertemiz. Ne transaction, ne veri erişimi...

Şimdi hangi exceptionda nasıl davranılacak?

Unchecked istisnalarda, (yani runtime hataları, yani programlama hataları) bütün transaction yöneticileri rollback yapıyor.

HibernateInterceptor ve HibernateInterceptor, ayrıca checked hatalarda da rollback yapıyor, sanırım...

Transaction Yönetim Stratejileri:

TransactionTemplate ve TransactionInterceptor transaction işlemlerini, PlatformTransactionManager nesnesine devreder. Yani yukarıdaki tanımda:

```
<bean id="myTransactionManager" class="org.springframework.orm.hibernate.HibernateTransactionManager">
    <property name="sessionFactory">
      <ref bean="mySessionFactory"/>
    </property>
  </bean>
```

and

Transaction yöneticisi, JtaTransactionManager veya özel bir implementasyon da olabilir. Sanırım, JTA yöneticisi, çok session factory kullanılması gereken durumlarda kullanılıyor. Niye böyle bir şey gereksin?

Dolayısıyla hibernate transaction yönetiminden, JTA´ye geçmek sadece bir satırlık bir konfigürasyon meselesi. Bütün transaction sınırlandırmaları ve veri erişim kodu etkilenmeden çalışacaktır.

Dağınık transactionlarda, JtaTransactionManager nesnesini birçok LocalSessionFactoryBean tanımıyla bağla. Her bir veri erişim nesnesi, özel bir SessionFactory referansı alır.

```
<beans>

  <bean id="myDataSource1" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName">
      <value>jdbc/myds1</value>
    </property>
  </bean>

  <bean id="myDataSource2" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName">
      <value>jdbc/myds2</value>
    </property>
  </bean>

  <bean id="mySessionFactory1" class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
```

```
    <property name="mappingResources">
      <list>
        <value>product.hbm.xml</value>
      </list>
    </property>
    <property name="hibernateProperties">
      <props>
        <prop key="hibernate.dialect">net.sf.hibernate.dialect.MySQLDialect</prop>
      </props>
    </property>
    <property name="dataSource">
      <ref bean="myDataSource1"/>
    </property>
  </bean>

  <bean id="mySessionFactory2" class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
    <property name="mappingResources">
      <list>
        <value>inventory.hbm.xml</value>
      </list>
    </property>
    <property name="hibernateProperties">
      <props>
        <prop key="hibernate.dialect">net.sf.hibernate.dialect.OracleDialect</prop>
      </props>
    </property>
    <property name="dataSource">
      <ref bean="myDataSource2"/>
    </property>
  </bean>

  <bean id="myTransactionManager" class="org.springframework.transaction.jta.JtaTransactionManager"/>

  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="sessionFactory">
      <ref bean="mySessionFactory1"/>
    </property>
  </bean>

  <bean id="myInventoryDao" class="product.InventoryDaoImpl">
    <property name="sessionFactory">
      <ref bean="mySessionFactory2"/>
    </property>
  </bean>

  <bean id="myProductServiceTarget" class="product.ProductServiceImpl">
    <property name="productDao">
      <ref bean="myProductDao"/>
    </property>
    <property name="inventoryDao">
      <ref bean="myInventoryDao"/>
    </property>
  </bean>

  <bean id="myProductService"
class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
    <property name="transactionManager">
      <ref bean="myTransactionManager"/>
    </property>
    <property name="target">
      <ref bean="myProductServiceTarget"/>
    </property>
    <property name="transactionAttributes">
      <props>
        <prop key="increasePrice*">PROPAGATION_REQUIRED</prop>
        <prop key="someOtherBusinessMethod">PROPAGATION_MANDATORY</prop>
      </props>
    </property>
  </bean>

</beans>
```

Uygulama bağlamı (Application Context):

Konfigürasyon, application context dosyalarında yapılıyor. Bunun da farklı türleri var. Classpathde yapılabilir. XML ile yapılabilir. Web uygulamaları için yapılabilir. Mesela, bir web uygulamasının bağlamı, WEB-INF/applicationContext.xml dosyasında tanımlıdır.

```
ApplicationContext context = WebApplicationContextUtils.getWebApplicationContext(servletContext);
ProductService productService = (ProductService) context.getBean("myProductService");
ApplicationContext context =
  new FileSystemXmlApplicationContext("C:/myContext.xml");
ProductService productService =
  (ProductService) context.getBean("myProductService");
ApplicationContext context =
  new ClassPathXmlApplicationContext("myContext.xml");
ProductService productService =
  (ProductService) context.getBean("myProductService");
```

Uygulama bağlamı ne işe yarar?

Bir sınıf modelindeki ilişkileri uygulama bağlamında konfigüre edebiliriz. Böylece statik ilişkilerin kurulmasıyla ilgili kod yazmamız gerekmez. Bunlar farklı yerlere dağılmaz...