Spring ile Veri Erişimi

Veri Erişim Exception Dönüşümü

- Her bir veri erişim teknolojisinin **kendine özel bir exception hiyerarşisi** vardır
- Spring bu exception'ları yakalayarak **ortak bir veri erişim exception hiyerarşisi**ne dönüştürür
- Böylece servis katmanı ve üstü sadece Spring'in **DataAccessException** hiyerarşisi ile muhatap olarak farklı teknolojileri aynı anda rahatlıkla kullanabilir

Veri Erişim Exception Dönüşümü:JDBC

- JDBC ile veri erişiminde JdbcTemplate bu dönüşümü **SQLExceptionTranslator** arayüzüne sahip bir nesne ile gerçekleştirir
- Default durumda **SQLErrorCodeSQLExceptionTranslator** kullanılır
- Vendor spesifik **SQL hata kodları** resolve edilir
- İlave kodlar classpath'de **sql-error-codes.xml** dosyasında tanımlanabilir

Veri Erişim Exception Dönüşümü:ORM

- ORM'de ise **PersistenceExceptionTranslator** arayüzü kullanılır
- **LocalSessionFactoryBean** ve **LocalContainerEntityManagerFactoryBean** sınıfları ayrıca bu arayüzü implement ederler
- Dolayısı ile ORM exception dönüşümleri bu bean'lar ile gerçekleştirilir

Veri Erişim Exception Dönüşümü:ORM

- DAO bean'inde herhangi bir ORM exception'ı (**HibernateException** veya **PersistenceException**) meydana geldiğinde bunun yakalanıp translator ile uygun Spring exception'ına **transparan biçimde dönüştürülme**si gerekir
- Bunun için Spring Container içerisinde **PersistenceExceptionTranslationPostProcessor** bean tanımı yapılmalıdır

Veri Erişim Exception Dönüşümü:ORM

```
<bean class="org.springframework.dao.annotation
.PersistenceExceptionTranslationPostProcessor"/>
```

- Bu bean ile @**Repository** anotasyonuna sahip bean'ler bir proxy ile wrap edilir
- DAO bean'lerinde meydana gelecek veri erişim exception'ları handle edilerek **PersistenceExceptionTranslator** ile exception dönüşümü gerçekleştirilir

Hibernate Java Tabanlı Konfigürasyon

```
@Configuration
public class AppConfig {

    @Bean
    public SessionFactory sessionFactory(DataSource dataSource) {
        Properties properties = new Properties();
        properties.put("hibernate.dialect",
                       "org.hibernate.dialect.H2Dialect");
        properties.put("hibernate.show_sql", "true");
        properties.put("hibernate.hbm2ddl.auto", "create");

        LocalSessionFactoryBuilder builder =
                new LocalSessionFactoryBuilder(dataSource);

        SessionFactory bean = builder
                    .scanPackages("com.javaegitimleri.petclinic.model")
                    .setProperties(properties)
                    .buildSessionFactory();
        return bean;
    }
}
```

JPA Java Tabanlı Konfigürasyon

```
@Configuration
public class AppConfig {

    @Bean
    public LocalContainerEntityManagerFactoryBean
                        entityManagerFactory(DataSource dataSource) {
        Map<String, String> jpaProperties = new HashMap<>();
        jpaProperties.put("hibernate.dialect",
                          "org.hibernate.dialect.H2Dialect");
        jpaProperties.put("hibernate.show_sql", "true");
        jpaProperties.put("hibernate.hbm2ddl.auto", "create");

        LocalContainerEntityManagerFactoryBean bean =
                          new LocalContainerEntityManagerFactoryBean();
        bean.setDataSource(dataSource);
        bean.setPackagesToScan("com.javaegitimleri.petclinic.model");
        bean.setJpaPropertyMap(jpaProperties);
        bean.setJpaVendorAdapter(new HibernateJpaVendorAdapter());
        return bean;
    }
}
```

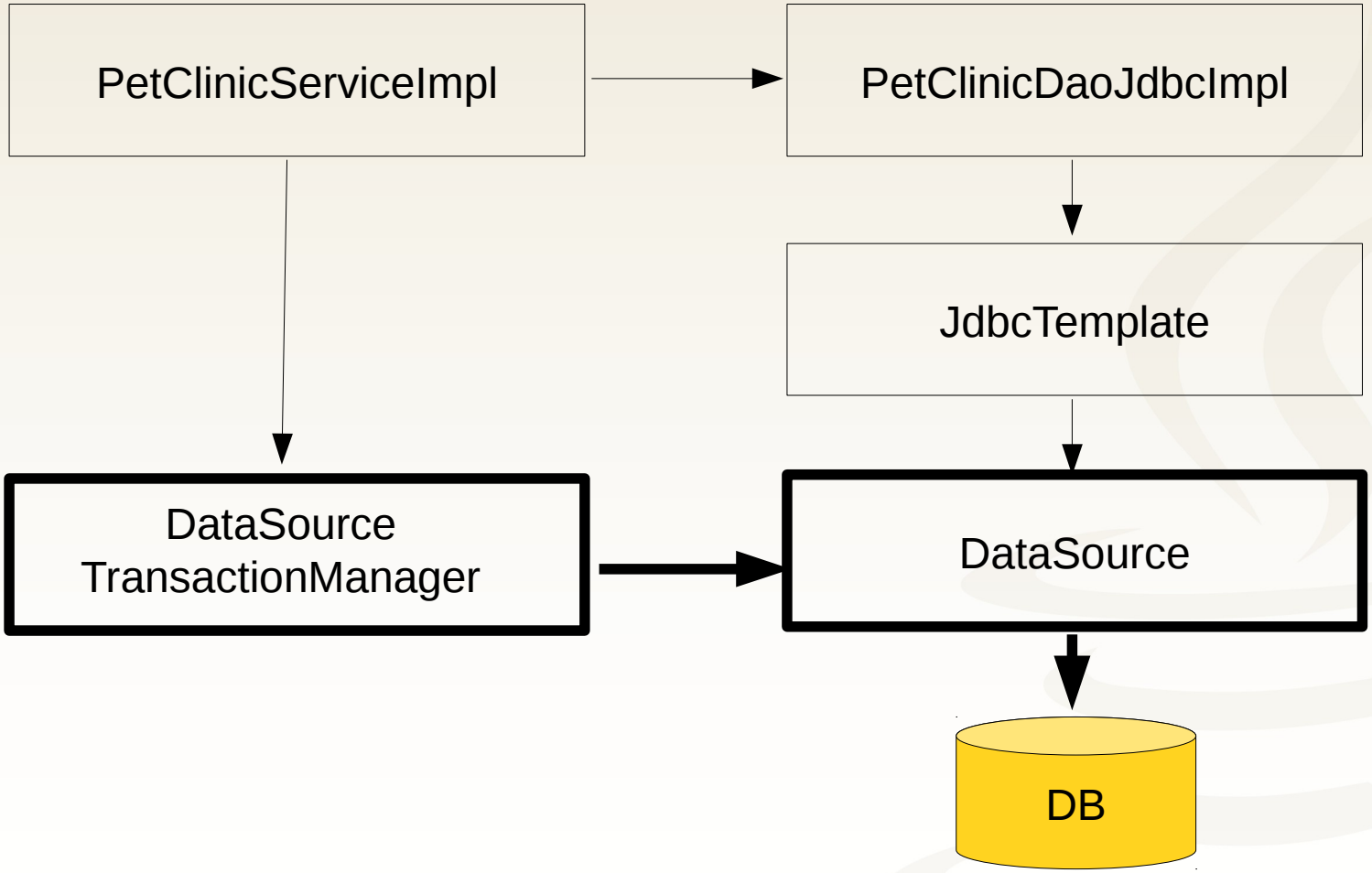
<tx:advice> ile XML Tabanlı Dekleratif TX Yönetimi

```
<bean id="fooService" class="x.y.service.DefaultFooService"/>

<tx:advice id="txAdvice" transaction-manager="txManager">
    <tx:attributes>
        <tx:method name="get*" read-only="true"/>
        <tx:method name="*" rollback-for="Throwable"/>
    </tx:attributes>
</tx:advice>

<aop:config>
  <aop:pointcut id="fooServiceMethods"
                expression="bean(fooService)"/>

  <aop:advisor advice-ref="txAdvice"
                pointcut-ref="fooServiceMethods"/>
</aop:config>
```

Programatik Transaction Yönetimi

- Asenkron biçimde **arka planda** ve **uzun süre çalışacak** kod bloklarını transactional yapmak için kullanılabilir
- Ya da bir **metodun sadece belirli bölümleri**ni transactional yapmak için kullanılabilir
- Spring programatik TX için iki yol sunar
  - **TransactionTemplate** üzerinden
  - Doğrudan **PlatformTransactionManager** kullanarak

Programatik Transaction Yönetimi



TransactionTemplate Kullanımı

- Servis metotlarındaki transactional işlemler **TransactionTemplate isimli bir nesne üzerinden** yürütülür
- Spring'in genel **template mantığı**na sahiptir
- Transactional kod bloğunu TransactionTemplate'a iletmek için **Callback** yöntemi kullanılır

TransactionTemplate Kullanımı

TransactionTemplate'ı oluşturmak için **PlatformTransactionManager**'a ihtiyaç duyulur

```
TransactionTemplate transactionTemplate = new
TransactionTemplate(transactionManager);

transactionTemplate.execute(new TransactionCallbackWithoutResult() {

    protected void doInTransactionWithoutResult(TransactionStatus status){

    //transactional is mantigi...

    }
});
```

Metot başlangıcında Spring bir TX başlatır. Metot başarılı sonlandığında da TX'i commit eder. İş bloğu içerisinde Meydana gelecek herhangi bir **exception durumunda ise TX rollback** edilir

Eğer transactional iş bloğu sonucunda bir değer dönülmesi gerekiyorsa Bu durumda **TransactionCallback** arayüzü kullanılabilir

TransactionTemplate Kullanımı

- TransactionTemplate'in **default ayarları @Transactional ile aynı**dır
- Ancak **herhangi bir exception'da rollback** gerçekleşir
- Rollback kuralını değiştirmek mümkün değildir
- Diğer ayarlar değiştirilebilir

PlatformTransactionManager Kullanımı

- **PlatformTransactionManager** bean'ı servis bean'ine enjekte edilir
- Servis metodu içerisinde **TransactionDefinition** ile bir TX tanımı oluşturulur
- Bu tanım ile TX manager üzerinden bir **TransactionStatus** elde edilir
- İş mantığı yürütülür
- İşlem sonucunda TransactionStatus ile **commit** veya **rollback** gerçekleştirilir

PlatformTransactionManager Kullanımı

TransactionDefinition TX tanımını ifade eder
TX'in Isolation, Propagation, Timeout, read-only özellikleri tanımlanabilir

```
DefaultTransactionDefinition txDef = new
DefaultTransactionDefinition();
txDef.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRED);

TransactionStatus status =
transactionManager.getTransaction(txDef);
```

TransactionStatus transaction execution'ı kontrol eder

```
try {
  // is mantigi calistirilir...
  transactionManager.commit(status);
} catch (MyException ex) {
  transactionManager.rollback(status);
  throw ex;
}
```

Birden Fazla TransactionManager ile Çalışmak

- Bazı durumlarda uygulama içerisinde **bazı metotların farklı transactionManager bean'leri** veya bean konfigürasyonları ile çalışmaları gerekebilir
- Bu durumda @**Transactional** anotasyonuna kullanılması istenen **transactionManager bean'inin qualifier değeri** yazılmalıdır

Birden Fazla TransactionManager ile Çalışmak

```
@Service
public class TransactionalService {

    @Transactional(transactionManager="txManager1")
    public void foo() {
        //transactional logic...
    }

    @Transactional(transactionManager="txManager2")
    public void bar() {
        //transactional logic...
    }

    @Transactional
    public void baz() {
        //transactional logic...
    }

}
```

Sistemdeki **default transactionManager bean'i** ile çalıştırılır

Birden Fazla TransactionManager ile Çalışmak

```
<beans...>
    <bean id="txManager1"
class="org.springframework.orm.hibernate4.HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>

    <bean id="txManager2"
class="org.springframework.transaction.jta.JtaTransactionManager"/>

    <alias name="txManager1" alias="transactionManager"/>

    <tx:annotation-driven transaction-manager="transactionManager" />
</beans>
```

Transaction Sonrasında İşlem Yapmak

- Transaction sonrası için ise **TransactionSynchronization** arayüzü implement edilmelidir
- Ardından bu sınıftan bir nesne aktif bir TX içerisinde **TransactionSynchronizationManager** ile register edilmelidir
- Eğer aktif TX olmadan bu işlem yapılırsa **IllegalStateException** fırlatılır

Transaction Sonrasında İşlem Yapmak

- **Birden fazla** tx synchronization nesnesi kayıt ettirilebilir
- Aralarındaki çalışma sırası **Ordered arayüzü** implement edilerek belirtilebilir
- **TransactionSynchronizationAdapter** isimli bir sınıf ile tx synchronization sınıfı sadece ilgili metotlar override edilerek yazılabilir

Transaction Sonrasında İşlem Yapmak

```
public class MyTransactionSynchronization extends
                    TransactionSynchronizationAdapter {

    @Override
    public void beforeCommit(boolean readOnly) {
        System.out.println("tx is about to be committed");
    }

    @Override
    public void afterCommit() {
        System.out.println("tx commited");
    }

    @Override
    public void afterCompletion(int status) {
        System.out.println("tx result :" + status);
    }
}
```

Transaction Sonrasında İşlem Yapmak

```
public interface TransactionSynchronization extends Flushable {

    int STATUS_COMMITTED = 0;
    int STATUS_ROLLED_BACK = 1;
    int STATUS_UNKNOWN = 2;

    void suspend();
    void resume();
    void flush();

    void beforeCommit(boolean readOnly);
    void beforeCompletion();
    void afterCommit();
    void afterCompletion(int status);

}
```

Transactional resource'u, örneğin Hibernate Session, TransactionSynchronizationManager'a bind/unbind etmek ve resource içindeki işlemleri DB'ye yansıtmak için kullanılırlar

Transaction Sonrasında İşlem Yapmak

```
@Service
public class TransactionalService {

    @Transactional(propagation=Propagation.REQUIRED)
    public void transactionalMethod() {

        TransactionSynchronizationManager
                .registerSynchronization(
                    new MyTransactionSynchronization());

        //perform transaction logic...
    }
}
```

Register işlemi mutlaka aktif bir transaction içerisinde gerçekleşmelidir
PlatformTransactionManager konfigürasyonunda synchronization
Kabiliyetinin ne zaman devrede olacağı kontrol edilebilir

Transaction Sonrasında İşlem Yapmak

```
<bean id="transactionManager"
class="org.springframework.orm.hibernate4.HibernateTransactionManager">

    <property name="sessionFactory" ref="sessionFactory" />

    <property name="transactionSynchronization" value="0"/>

    <property name="transactionSynchronizationName"
                                value="SYNCHRONIZATION_ALWAYS" />
</bean>
```

```
SYNCHRONIZATION_ALWAYS=0
SYNCHRONIZATION_ON_ACTUAL_TRANSACTION=1
SYNCHRONIZATION_NEVER=2
```

ALWAYS (default) değerinde transaction propagation kuralı ne olursa olsun synchronization yapılabilir. Eğer değer ACTUAL_TRANSACTION ise sadece REQUIRED, REQUIRES_NEW ve MANDATORY'de synchronization kod register edilebilir.

Transaction Öncesinde İşlem Yapmak

- Transaction öncesinde veya sonrasında **uygulamaya özel kod çalıştırmak** mümkündür
- **Transaction öncesi** kod çalıştırmak için **Spring AOP**'tan yararlanılır
- Bir **advice** yazılıp, transaction davranışı öncesi devreye girecek biçimde ayarlanması gerekir

Servis Bean'larını Transactional Yapmak

- **@Service** anotasyonu ile oluşturulan bean'ların public metotları normalde transactional değildir
- Bu bean'ların bütün metotlarını transactional yapmak için ayrıca **@Transactional** anotasyonunu kullanmak gerekir
- Bu **iki anotasyonu bir araya getiren** bir **anotasyon** yazarak servis bean'larını default durumda transactional'a yapmak mümkün olabilir

Servis Bean'larını Transactional Yapmak

```
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

@Service
@Transactional
@Target(value=ElementType.TYPE)
public @interface TransactionalService {
    String value() default "";
}
```

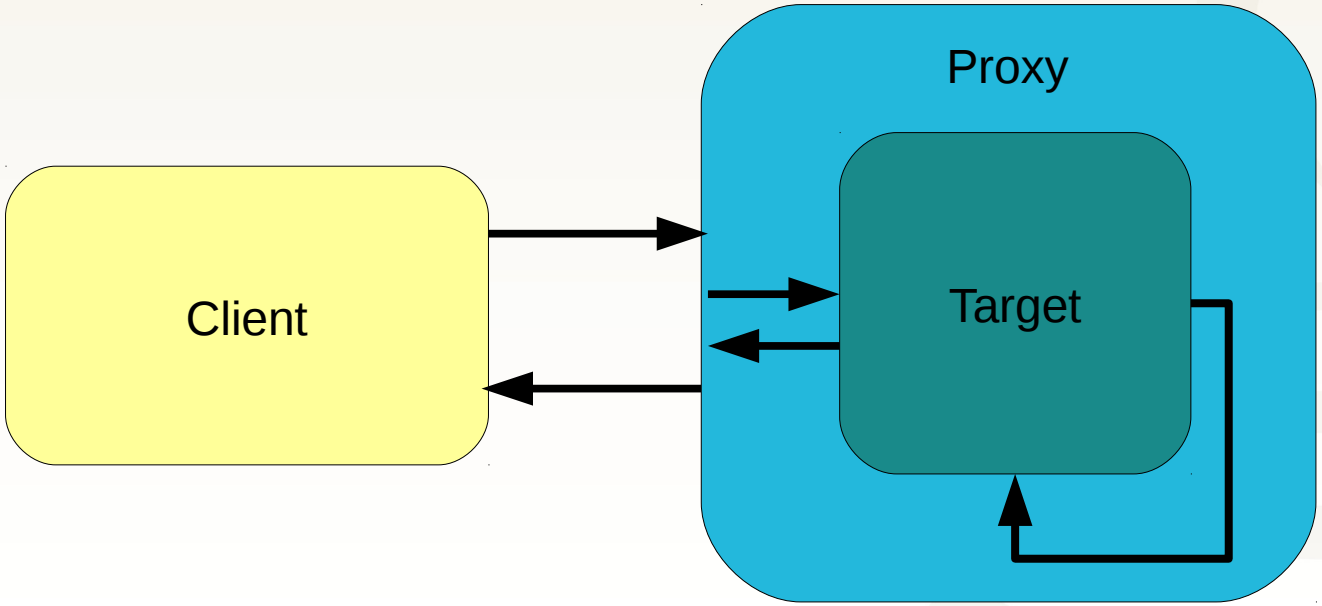
```
@TransactionalService("petClinicService")
public class PetClinicServiceImpl implements PetClinicService {
...
}
```

Aynı Bean İçerisinde Başka Bir Transactional Metot Çağırmak

- Spring transaction yönetimi **proxy örüntüsü** üzerine kuruludur
- Transaction davranışı **sadece public metot invokasyonları**nda devreye girebilir
- Bu metot invokasyonları ayrıca **proxy dışından gerçekleşmeli**dir
- Bir metot içerisinde **aynı bean'in başka bir metodu**na erişildiğinde bu proxy üzerinden gerçekleşmez

Aynı Bean İçerisinde Başka Bir Transactional Metot Çağırmak

- Dolayısı ile bu çağrı proxy üzerinden gerçekleşmediği için **transaction davranışı da devreye girmeyecek**tir



Aynı Bean İçerisinde Başka Bir Transactional Metot Çağırmak

```
@Service
public class FooService {

   @Transactional(propagation=Propagation.REQUIRED)
   public void foo() {
      this.bar();
   }

   @Transactional(propagation=Propagation.REQUIRES_NEW)
   public void bar() {

   }
}
```

Aynı Bean İçerisinde Başka Bir Transactional Metot Çağırmak

- Bu probleme **üç farklı çözüm yolu** vardır
  - İkinci transactional metodu **ayrı bir bean**'e taşımak
  - Metot içerisinde kendine **ApplicationContext üzerinden lookup** yapmak
  - Metot içerisinde o anki **proxy nesneye erişim** sağlamak

İkinci Metodu Ayrı Bir Bean'e Taşımak

```
@Service
public class FooService {

    @Autowired
    private BarService barService;

    @Transactional(propagation = Propagation.REQUIRED)
    public void foo() {
        barService.bar();
    }
}


@Service
public class BarService {
    @Transactional(propagation = Propagation.REQUIRES_NEW)
    public void bar() {

    }
}
```

Metot İçerisinde Kendine Lookup Yapmak

JAVA Eğitimleri

```
@Service
public class FooService
                implements ApplicationContextAware, BeanNameAware {

    private ApplicationContext applicationContext;
    private String beanName;

    @Override
    public void setApplicationContext(
                ApplicationContext applicationContext) {
        this.applicationContext = applicationContext;
    }

    @Override
    public void setBeanName(String beanName) {
        this.beanName = beanName;
    }

...
}
```

Metot İçerisinde Kendine Lookup Yapmak

JAVA Eğitimleri

```
@Service
public class FooService
                implements ApplicationContextAware, BeanNameAware {

...

    @Transactional(propagation = Propagation.REQUIRED)
    public void foo() {
        FooService fooService = applicationContext
                        .getBean(beanName,FooService.class);
        fooService.bar();
    }

    @Transactional(propagation = Propagation.REQUIRES_NEW)
    public void bar() {
    }
}
```

Metot İçerisinde Proxy Nesneye Erişim Sağlamak

```
@Service
public class FooService {

    @Transactional(propagation=Propagation.REQUIRED)
    public void foo() {
        FooService proxy =
            (FooService)AopContext.currentProxy();
        proxy.bar();
    }

    @Transactional(propagation=Propagation.REQUIRES_NEW)
    public void bar() {

    }
}
```

Metot İçerisinde Proxy Nesneye Erişim Sağlamak

- AopContext'in o anki proxy nesneyi dönebilmesi için AOP altyapısında **expose proxy özelliğ**inin aktive edilmiş olması gerekir

```
<beans...>
    <aop:config expose-proxy="true">
    </aop:config>

    <tx:annotation-driven/>
</beans>
```

Hangi @Transactional'ı Kullanmalı ?

- **JTA 1.2** API'si ile gelen bir **@Transactional** anotasyonu da mevcuttur
- Spring bu anotasyonu da **desteklemektedir**
- JTA'nın anotasyonunda **sadece propagation ve rollback kuralları** değiştirilebilmektedir

Hibernate API ile DAO Geliştirilmesi

```
@Repository
public class ProductDaoImpl implements ProductDao {

    private SessionFactory sessionFactory;

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

    public Collection loadProductsByCategory(String category) {
        return sessionFactory.getCurrentSession()
                .createQuery("from test.Product product where
product.category=?")
                .setParameter(0, category)
                .list();
    }
}
```

JPA API ile DAO Geliştirilmesi


```
@Repository
public class ProductDaoImpl implements ProductDao {

    private EntityManager entityManager;

    @PersistenceContext
public void setEntityManager(EntityManager entityManager) {
        this.entityManager = entityManager;
    }

    public Collection loadProductsByCategory(String category) {
        return entityManager.createQuery(
                "select p from Product p where p.category=?1")
                .setParameter(1, category)
                .getResultList();
    }
}
```

İletişim

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