

# Tasarım Örüntüleri ile Spring Eğitimi 4

# Spring JDBC ile Veri Erişimi & Transaction Yönetimi

# Spring ile Veri Erişimi'nin Özellikleri

- Spring değişik teknolojiler için kullanımı **kolay ve standart bir veri erişim** desteği sağlar
  - JDBC, JPA, Hibernate vb desteklenir
- Farklı veri erişim teknolojileri aynı anda kullanılabilir
- **Kapsamlı ve transparan** bir transaction yönetim altyapısına sahiptir
- Veri erişim teknolojilerinin exception hiyerarşilerini **standart bir exception hiyerarşisine** çevirir

# JDBC API ile Veri Erişimi

- JDBC API kullanarak gerçekleştirilen veri tabanı işlemlerinde yazılan kod blokları genel olarak birbirlerine **benzer bir akışa** sahiptir
- Hepsinde veritabanı bağlantısı oluşturma, SQL'i çalıştırma, dönen sonuçlar üzerinde işlem yapma, TX varsa commit/rollback yapma, hataları ele alma ve bağlantıyı kapatma gibi **işlemler standarttır**
- **Değişen kısımlar** SQL, parametreler ve dönen sonucu işleyen kod bloğu olur

# JDBC API ile Veri Erişimi

- Veritabanı bağlantı parametrelerinin belirtilmesi ve bağlantının kurulması
- *SQL sorgusunun oluşturulması*
- Oluşturulan Statement'ın derlenip çalıştırılması
- Dönen ResultSet üzerinde işlem yapan bir döngünün kurulması
- *Bu döngü içerisinde her bir kaydın işlenmesi*
- Meydana gelebilecek hataların ele alınması
- Transaction'ın sonlandırılması ve veritabanı bağlantısının kapatılması

Sadece kırmızı font ile işaretlenen kısımlar değişkenlik gösterir, diğer adımlar bütün persistence işlemlerinde standarttır

# JDBC API ile Veri Erişimine Örnek

```
public Collection<Document> findDocuments() {  
    Connection c = null;  
    Statement stmt = null;  
    try {  
        c = DriverManager.getConnection(  
            "jdbc:h2:tcp://localhost/~test", "sa", "");  
        c.setAutoCommit(false);  
        stmt = c.createStatement();  
        ResultSet rs = stmt  
            .executeQuery("select * from T_DOCUMENT");  
        Collection<Document> result = new ArrayList<Document>();  
        while (rs.next()) {  
            Document doc = new Document();  
            doc.setName(rs.getString("doc_name"));  
            doc.setType(rs.getInt("doc_type"));  
            result.add(doc);  
        }  
    }  
}
```

# JDBC API ile Veri Erişimine Örnek


```
        c.commit();
        return result;
    } catch (SQLException ex) {
        try {
            c.rollback();
        } catch (Exception e) {}
        throw new DataRetrievalFailureException(
            "Cannot execute query", ex);
    } finally {
        try {
            stmt.close();
        } catch (Exception e) {}
        try {
            c.close();
        } catch (Exception e) {}
    }
}
```


# Spring Üzerinden JDBC ile Veri Erişimi

- Spring veri erişiminde bu tekrarlayan kısımları ortadan kaldırmak için **Template Method örüntüsü tabanlı** bir kabiliyet sunar
- **JdbcTemplate** merkez sınıftır
- Utility veya helper sınıflarına benzetilebilir
- JdbcTemplate sayesinde Template Method tarafından dikte edilen **standart bir kullanım şekli** kod geneline hakim olur



# JdbcTemplate ile Veri Erişimi

```
public Collection<Document> findDocuments() {  
    Collection<Document> result = jdbcTemplate.query(  
        "select * from T_DOCUMENT",  sorgu  
        new RowMapper<Document>() {  
  
            @Override  
            public Document mapRow(ResultSet rs, int rowNum)  
                throws SQLException {  
                Document doc = new Document();  
                doc.setName(rs.getString("doc_name"));  
                doc.setType(rs.getInt("doc_type"));  
                return doc;  
            }  
        }  
    );  
    return result;  
}
```



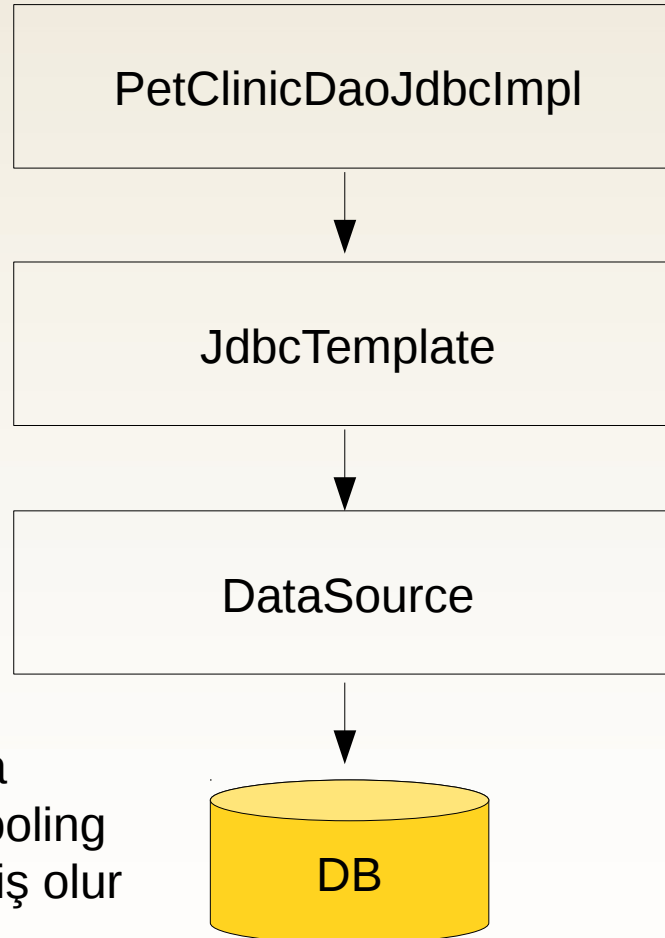
# JdbcTemplate Konfigürasyonu

Uygulama tarafında  
JdbcTemplate ile  
çalışırken sadece  
**Callback** yazmak yeterlidir

Spring veritabanı  
bağlantılarını **DataSource**  
nesnesinden alır

DataSource  
**connection factory**'dir

DataSource'un kendi başına  
yönetilmesi ile connection pooling  
vs. uygulamadan izole edilmiş olur



JDBC ile veri erişimi  
**JdbcTemplate** üzerine  
kurulmuştur

Çalışması için  
**DataSource** nesnesine  
ihtiyaç vardır

Thread safe'dir, birden  
fazla bean tarafından  
erişilebilir

# Veritabanı Bağlantılarının Yönetilmesi

- Spring ile veri erişiminde veritabanı bağlantıları **javax.sql.DataSource** arayüzü üzerinden elde edilir
- **DriverManagerDataSource** yaygın kullanılan Spring DataSource sınıfıdır
- Ancak Connection **pooling kabiliyeti yoktur**
- Dolayısı ile **prod ortamı için uygun değildir**

# Veritabanı Bağlantılarının Yönetilmesi

- Hikari, Apache **DBCP** veya **C3P0** gibi ile **connection pool** özelliğine sahip kütüphaneler ile DataSource oluşturulabilir
- Ya da **application server**'ın **dataSource**'una JNDI ile erişilebilir
- Uygulama sunucuları üzerinde konfigüre edilmiş DataSource instance'ları **genellikle pooling kabiliyetine sahiptir**

# DataSource Tanım Örneği

```
<beans...>
```

```
    <bean id="dataSource"  
class=  
"org.springframework.jdbc.datasource.DriverManagerDataSource">  
  
    <property name="driverClassName"  
value="org.h2.Driver"/>  
    <property name="url"  
value="jdbc:h2:tcp://localhost/~test"/>  
    <property name="username"  
value="sa"/>  
    <property name="password"  
value=""/>  
  
    </bean>  
  
</beans>
```

# DataSource Tanım Örneği

```
<beans...>
```

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

```
</beans>
```

# JdbcTemplate Kullanım Örnekleri

```
Collection<Document> result = jdbcTemplate.query(  
    "select * from T_DOCUMENT",  
    new RowMapper<Document>() {  
  
        @Override  
        public Document mapRow(ResultSet rs, int  
rowNum)  
            throws SQLException {  
            Document doc = new Document();  
            doc.setName(rs.getString("doc_name"));  
            doc.setType(rs.getInt("doc_type"));  
            return doc;  
        }  
    }  
);
```

# JdbcTemplate Kullanım Örnekleri

```
String result = jdbcTemplate.queryForObject("select  
last_name from persons where id = ?", new Object[]{1212L},  
String.class);
```

```
List<String> result = jdbcTemplate.queryForList("select  
last_name from persons", String.class);
```

```
Map<String, Object> result =  
jdbcTemplate.queryForMap("select last_name, first_name from  
persons where id = ?", 1212L);
```

```
List<Map> result = jdbcTemplate.queryForList("select * from  
persons");
```



# JdbcTemplate Kullanım Örnekleri

```
int insertCount = jdbcTemplate.update(  
    "insert into persons (first_name, last_name) values (?, ?)",  
    "Ali", "Yücel");
```

```
int updateCount = jdbcTemplate.update(  
    "update persons set last_name = ? where id = ?", "Güçlü", 1L);
```

```
int deleteCount = jdbcTemplate.update(  
    "delete from persons where id = ?", 1L);
```

```
int result = jdbcTemplate.update(  
    "call SUPPORT.REFRESH_PERSON_SUMMARY(?)", 1L);
```

# JdbcTemplate Kullanım Örnekleri

```
KeyHolder keyHolder = new GeneratedKeyHolder();

PreparedStatementCreator psc = new PreparedStatementCreator() {
    @Override
    public PreparedStatement createPreparedStatement(
        Connection con) throws SQLException {

        PreparedStatement stmt = con.prepareStatement(
            "insert into persons(first_name,last_name) values(?,?)");
        stmt.setString(1, person.getFirstName());
        stmt.setString(2, person.getLastName());
        return stmt;
    }
};

int insertCount = jdbcTemplate.update(psc, keyHolder);

Long id = (Long) keyHolder.getKey();
```

# JdbcTemplate Kullanım Örnekleri

```
int[] batchUpdateCount = jdbcTemplate.batchUpdate(  
    "update t_item set active = true",  
    "delete from persons");
```

```
List<Object[]> args = new ArrayList<Object[]>();  
args.add(new Object[]{"Kenan", "Sevindik", 1L});  
args.add(new Object[]{"Muammer", "Yücel", 2L});
```

```
int[] batchUpdateCount = jdbcTemplate.batchUpdate(  
    "update persons set first_name = ?, last_name = ? where id = ?",  
    args);
```

# JdbcTemplate Kullanım Örnekleri

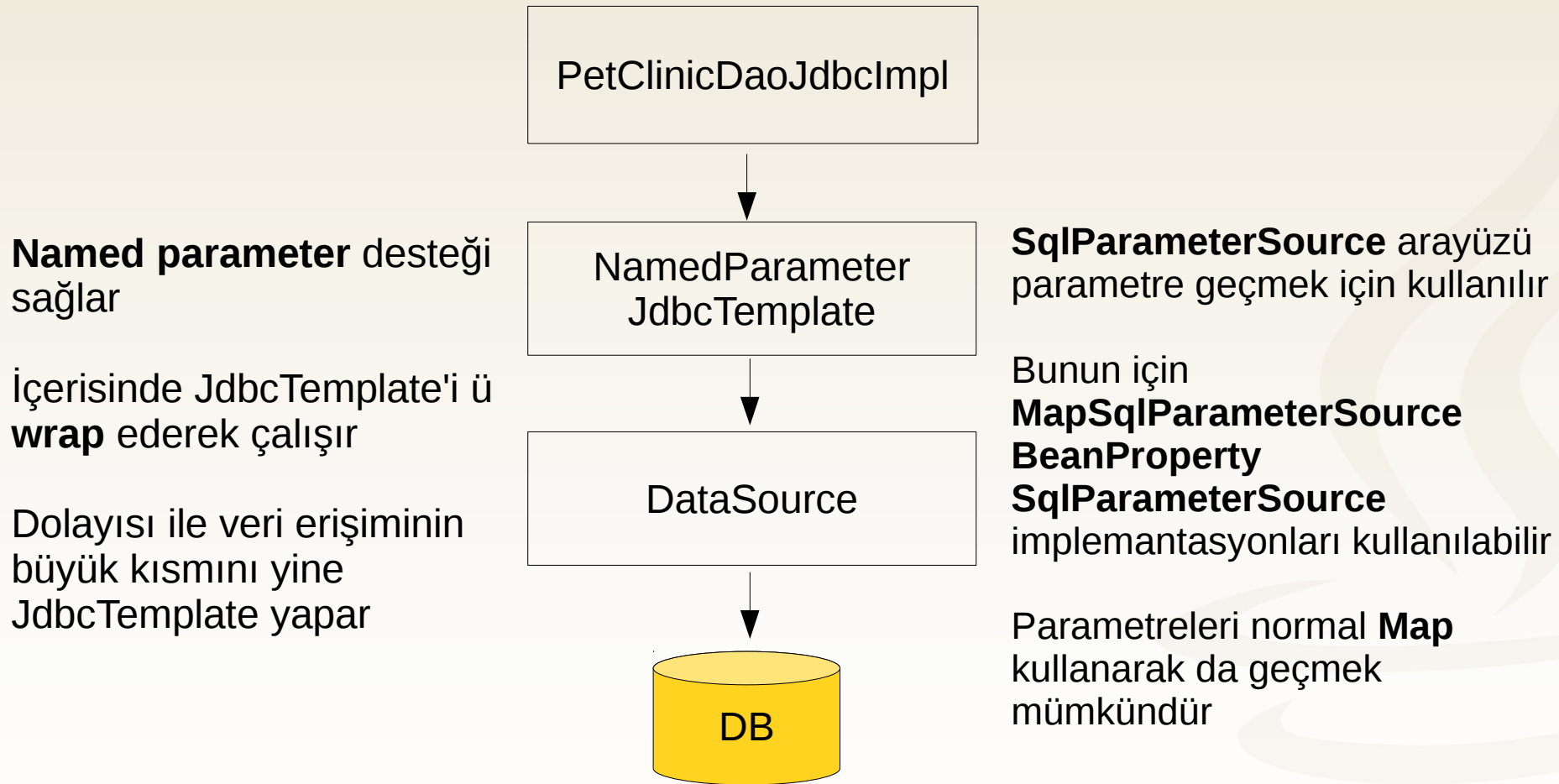
```
jdbcTemplate.execute("create table mytable (id integer, name  
varchar(100))");
```

```
Collection<PetType> result = jdbcTemplate.execute(new  
StatementCallback<Collection<PetType>>(){  
    @Override  
    public Collection<PetType> doInStatement(Statement stmt)  
        throws SQLException, DataAccessException {  
        ResultSet resultSet = stmt.executeQuery("select * from  
t_pet_type");  
        List<PetType> types = new ArrayList<>();  
        while(resultSet.next()) {  
            PetType type = new PetType();  
            type.setId(resultSet.getLong("ID"));  
            type.setName(resultSet.getString("NAME"));  
            types.add(type);  
        }  
        return types;  
    }  
});
```

# Named Parameter Desteği

- JdbcTemplate default olarak **pozisyonel parametreleri** destekler
- Parametreler SQL ifadesi içerisinde ? **işareti** ile belirtilir
- Değerler de bir **Object array** ile verilir
- Spring veri erişiminin **isimlendirilmiş parametre desteği** de mevcuttur
- İsimlendirilmiş parametreler SQL ifade içerisinde **:paramName** şeklinde gösterilir

# NamedParameterJdbc Template Konfigürasyonu



# NamedParameterJdbc Template Kullanım Örneği

```
String sql = "select count(*) from persons where first_name =  
:firstName";
```

```
Map namedParameters = new HashMap();  
namedParameters.put("firstName", "Kenan");
```

```
int count = namedParameterJdbcTemplate  
            .queryForObject(  
                sql,  
                namedParameters,  
                Integer.class);
```

Sorgu ve isimleri ile eşleştirilmiş parametreler

Sorgu return değeri tipi

# IN Clause'una Değişken Sayıda Parametre Geçilmesi

```
String sql = "select * from users where id in (:idList)";
```

```
Map namedParams = Collections.singletonMap(  
    "idList"  
    Arrays.asList(new Integer[]{1,2,3}));
```

```
RowMapper<User> rowMapper = new RowMapper<User>() {  
    public User mapRow(ResultSet rs, int rowNum) throws  
    SQLException {  
        User user = new User();  
        user.setFirstName(rs.getString("first_name"));  
        user.setLastName(rs.getString("last_name"));  
        return user;  
    }  
};
```

```
List<User> result=namedParameterJdbcTemplate.query(sql,  
namedParams,rowMapper);
```



# Spring ile Transaction Yönetiminin Özellikleri

- Spring, değişik veri erişim yöntemlerini aynı anda kullanmayı sağlayan **ortak bir transaction yönetim API'sine** sahiptir
- **Dekleratif** ve **programatik** TX yönetimi mümkündür
- **Global** (dağıtık) ve **lokal** (tek DB) transactionları destekler
- Spring'in temel transaction soyutlaması **PlatformTransactionManager**'dir

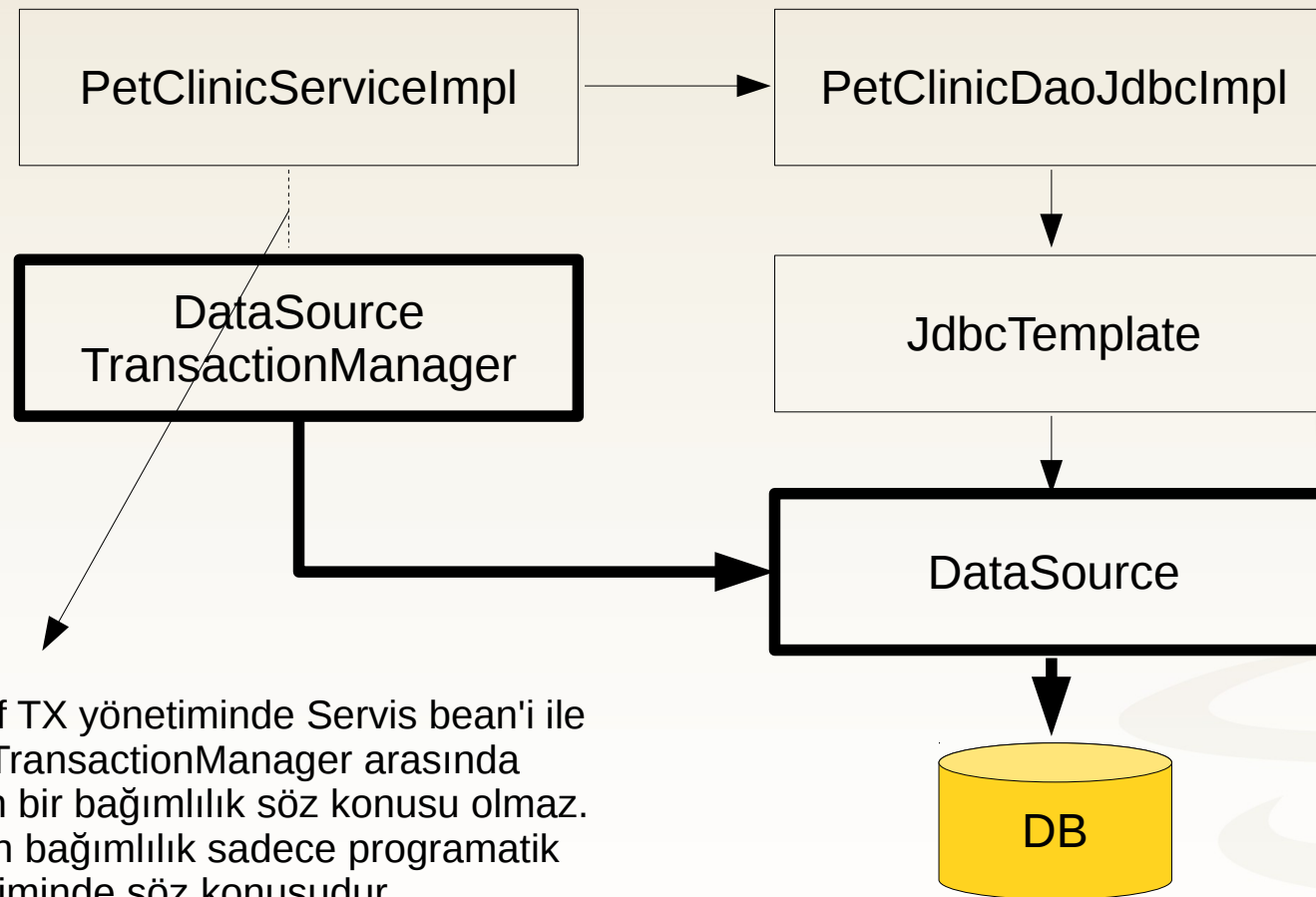
# Spring ile Transaction Yönetiminin Özellikleri

- **PlatformTransactionManager** sayesinde veri erişimi, TX altyapısından **tamamen bağımsız** hale gelir
- Bu sayede uygulamalar **lokal TX'den global TX'e transparan biçimde geçiş** yapabilir
- Ya da **veri erişim teknolojisi** değiştiği vakit uygulama tarafında bir değişikliğe gerek kalmadan **transaction yönetim altyapısı** rahatlıkla değiştirilebilir
- Bunun için **sadece bean tanımlarında değişiklik** yeterlidir

# PlatformTransactionManager Konfigürasyonu

- Spring ile transaction yönetiminde **en önemli nokta** veri erişim teknolojisine uygun **PlatformTransactionManager** tanımlanmasıdır
- Farklı veri erişim teknolojileri için **farklı implemantasyonlar** mevcuttur
  - **JDBC**: DataSourceTransactionManager
  - **Hibernate**: HibernateTransactionManager
  - **JPA**: JpaTransactionManager
  - **JTA**: JtaTransactionManager

# PlatformTransactionManager Konfigürasyonu - JDBC



Dekleratif TX yönetimde Servis bean'i ile PlatformTransactionManager arasında doğrudan bir bağımlılık söz konusu olmaz. Doğrudan bağımlılık sadece programatik TX yönetimde söz konusudur.

# PlatformTransactionManager Konfigürasyonu - JDBC

```
<bean id="transactionManager"  
class="org.springframework.jdbc.datasource.DataSourceTr  
ansactionManager">  
    <property name="dataSource" ref="dataSource" />  
</bean>
```

```
<bean id="dataSource"  
class="org.springframework.jdbc.datasource.DriverManage  
rDataSource">  
    <property name="driverClassName" value="$  
{jdbc.driverClassName}" />  
    <property name="url" value="${jdbc.url}" />  
    <property name="username" value="${jdbc.username}" />  
    <property name="password" value="${jdbc.password}" />  
</bean>
```

# Dekleratif Transaction Yönetimi

- Uygulama içinde TX yönetimi ile ilgili **kod yazılmaz**
- Servis metot çağrısı geldiği vakit Spring Container tarafından **yeni bir TX başlatılır**
- Metot başarılı sonlandığı vakit **TX commit** edilir
- **Sınıf veya metot düzeyinde** TX yönetimi yapılabilir
- En sık **tercih edilen yöntemdir**

# Dekleratif Transaction Yönetimi ve Rollback

- Transactional bir metot içerisinde bir exception meydana geldiğinde **exception türüne** bakılır
- Default olarak runtime exception'larda **TX rollback** edilir
- Checked exception'larda ise **TX commit** edilir
- Ancak çoğunlukla **bu davranış değiştirilir**

# Dekleratif Transaction Yönetimi

- Dekleratif TX yönetimi **@Transactional** anotasyonu ile yapılır
- **Sınıf veya metot düzeyinde** kullanılabilir
- Sadece **public metotlarda** kullanılmalıdır
- **<tx:annotation-driven/>** elemanı **@Transactional** anotasyonlarını devreye sokar



# @Transactional ile Dekleratif TX Yönetimi

```
@Transactional
public class DefaultFooService implements FooService {

    public Foo getFoo(String fooName) {
        // ...
    }

    public void updateFoo(Foo foo) {
        // ...
    }
}
```

# @Transactional ile Dekleratif TX Yönetimi

```
public class DefaultFooService implements FooService {  
  
    public Foo getFoo(String fooName) {  
        // ...  
    }  
  
    @Transactional  
    public void updateFoo(Foo foo) {  
        // ...  
    }  
}
```

# @Transactional ile Dekleratif TX Yönetimi

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

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

```
<bean id="transactionManager"  
class="org.springframework.jdbc.datasource.DataSource  
ceTransactionManager">
```

```
    <property name="dataSource" ref="dataSource"/>
```

```
</bean>
```

# @Transactional Default Değerleri

- **@Transactional** anotasyonundaki attribute'ların default değerleri:
  - **Propagation** REQUIRED
  - **Isolation** DEFAULT
  - **Transaction** read/write
  - **Timeout** sistem default
  - **Rollback** Herhangi bir RuntimeException

# @Transactional Default Değerleri

```
@Transactional(rollbackFor = Exception.class)
public class DefaultFooService implements FooService {

    @Transactional(readOnly = true)
    public Foo getFoo(String fooName) {
        // ...
    }

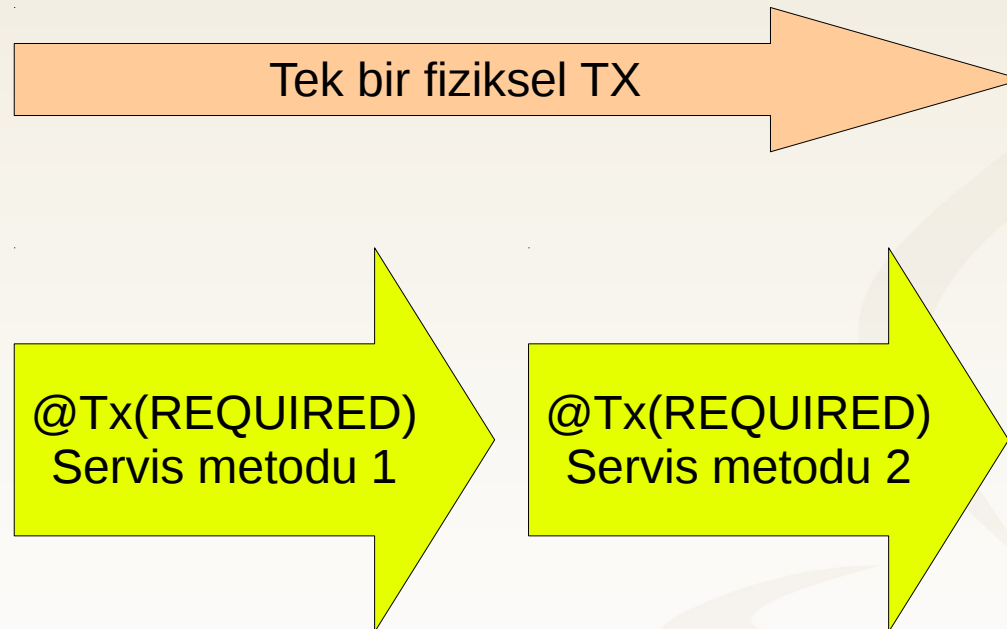
    @Transactional(propagation = Propagation.REQUIRES_NEW)
    public void updateFoo(Foo foo) {
        // ...
    }
}
```

# Transaction Propagation: PROPAGATION\_REQUIRED

Her bir servis metodu için  
**ayrı mantıksal TX'ler** yaratılır

İçteki TX **setRollbackOnly**  
yaparsa bütün **diğer TX'ler**  
**etkilenir**

Dıştaki commit yapsa bile  
**UnexpectedRollback**  
**Exception** fırlatılır

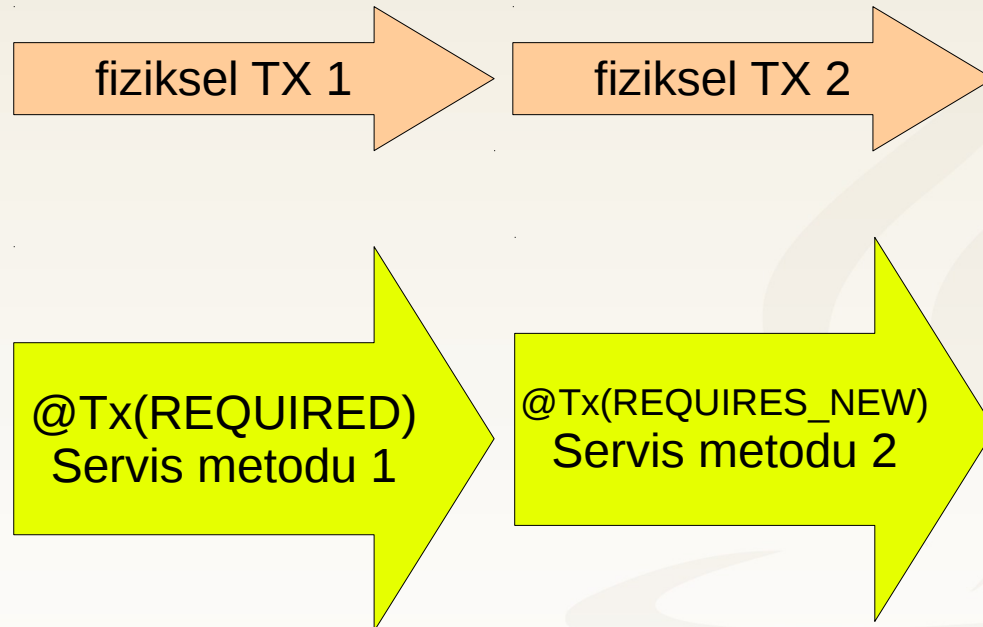


# Transaction Propagation: PROPAGATION\_REQUIRES\_NEW

Birbirinden **bağımsız fiziksel TX'ler** vardır

İkinci servis metodu çalışırken  
İlk servis metodunun TX'i  
Suspend edilir

Herbirisi kendi başına  
commit/rollback yapılabilir



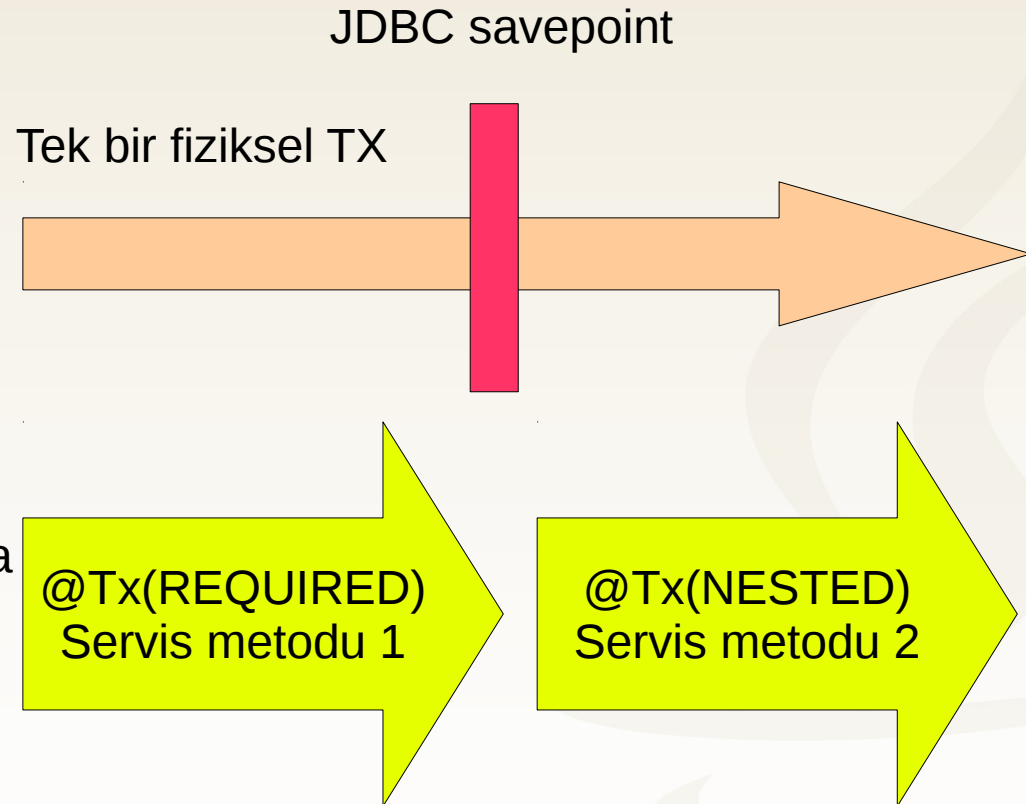
# Transaction Propagation: PROPAGATION\_NESTED

Birden fazla savepoint vardır

**inner TX** kendi içinde **rollback** Yapabilir

Sadece Savepoint'e kadar yapılan işlemler rollback olur

Sadece **JDBC**'de yani **DataSourceTransactionManager**'da anlamlıdır, Hibernate desteklemez





# Diğer TX Propagation Değerleri

- **SUPPORTS:**
  - Servis metodu TX varsa TX içinde çalıştırılır
  - TX yoksa, TX'siz çalıştırılır
- **NOT\_SUPPORTED:**
  - TX varsa suspend edilir, servis metodu TX dışında çalıştırılır
- **NEVER:**
  - Servis metodu çağrıldığında TX varsa exception fırlatılır
- **MANDATORY:**
  - Servis metodu çağrıldığında TX yoksa exception fırlatılır

# Spring ve Entegrasyon Birim Testleri

## Framework'ün Özellikleri

- Entegrasyon testlerinin **standalone ortamda yazımı ve çalıştırılmasını** mümkün kılar
- ApplicationContext'i oluşturan **bean tanımlarının doğru yapılp yapılmadığının** kontrolü sağlanır
- JDBC ve ORM araçları ile **veri erişiminin testi** yapılır
  - SQL, HQL sorgularının kontrolü yapılır
  - ORM mapping'lerin düzgün yapılp yapılmadığı test edilmiş olur

# Spring TestContext Framework'ün Özellikleri

- Test sınıflarına **dependency injection** yapılabilir
- TestCase içerisinde **ApplicationContext'e erişmek** de mümkündür
- Test metodlarının **transactional context** içerisinde çalıştırılması mümkündür
- Otomatik **ApplicationContext yönetimi** yapar
- Spring container'ı test sınıfları ve test metotları arasında yeniden oluşturmamak için **cache desteği** sağlar

# Spring TestContext Framework Konfigürasyonu

ApplicationContext yüklenmesi  
Dependency injection  
Transactional test desteği aktive olur

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
public class FooTests {
    // ...
}
```


Default: classpath:/com/example/FooTests-context.xml  
**locations** attribute ile farklı dosyalar belirtilebilir

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("/appContext.xml")
public class BarTests {
    // ...
}
```

# Entegrasyon Testleri ve ApplicationContext Yönetimi

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("/base-context.xml")
public class BaseTest {
    // ...
}
```

```
@ContextConfiguration("/extended-context.xml")
public class ExtendedTest extends BaseTest {
    // ...
}
```



ExtendedTest sınıfı BaseTest sınıfından türediği için bu sınıftaki test metotları için yaratılacak olan ApplicationContext base-context.xml ve extended-context.xml dosyaları yüklenerek oluşturulacaktır

# Entegrasyon Testleri ve Dependency Injection

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
public class SpringTests {

    @Autowired
    private FooService fooService;

    ...
}
```

# ApplicationContext Yönetimi ve ÖnBellekleme

- ApplicationContext test sınıfı bazında sadece **bir kereliğine** yüklenir
- ApplicationContextAware arayüzünü implement ederek veya **@Autowired** ile enjekte ederek **test sınıfı içerisinde ApplicationContext'e erişilebilir**
- Herhangi bir test sonrası ApplicationContext'in yeniden yüklenmesi istenirse, **test sınıfı veya test metodu düzeyinde @DirtiesContext** kullanılabilir



# ApplicationContext Yönetimi ve ÖnBellekleme

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("/appcontext/beans-*.xml")
public class SpringTests {
```

```
    @Autowired
    private ApplicationContext applicationContext;
```

```
    @Test
    @DirtiesContext
    public void testMethod1() {
    }
```

Bu test metodu çalıştıktan sonra ApplicationContext bir sonraki test metodu çalıştırılmadan önce yeniden yaratılacaktır

```
    @Test
    public void testMethod2() {
    }
}
```

# Testler ve Java Tabanlı Container Konfigürasyonu

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes={AConfig.class,BConfig.class})
public class SpringTests {
    @Autowired
    private Foo foo;

    @Test
    public void testFoo() {
        Assert.assertNotNull(foo.getBar());
    }
}
```

XML konfigürasyon dosyaları  
yerine konfigürasyon  
sınıfları listelenir

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
public class SpringTests {
    @Configuration
    static class TestContextConfiguration {
        @Bean
        public Foo foo() {
            return new Foo();
        }
    }
}
```

Test'e özel bean konfigürasyon  
sınıfı inner class olarak  
tanımlanabilir

# Testleri Ortama Göre Çalıştırmak

- **@IfProfileValue** anotasyonu ile belirli bir ortam veya sistem değişkeninin değerine göre testler enable/disable edilebilir
- Sınıf veya metot düzeyinde kullanılabilir

```
@IfProfileValue(name="targetPlatform", value="test")  
@Test  
public void testMethod() {  
    // ...  
}
```

Test metodu eğer ifade true olarak evaluate ediyor ise çalıştırılır. Sınıf düzeyinde kullanılırsa o sınıftaki hiçbir test metodu çalıştırılmayacaktır

# Testler ve Spring Bean Profile Kabiliyeti

- **@ActiveProfiles** anotasyonu ile test çalışırken aktif olması istenen Spring bean profilleri belirtilebilir

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("/appcontext/beans.xml")
@ActiveProfiles({"test", "oracle"})
public class FooTests {

    @Test
    public void testFoo() {
        ...
    }
}
```

# Testler ve PropertySource Kullanımı

- Property tanımları testlere özgü de belirlenebilir
- Bunun için **@TestPropertySource** anotasyonu kullanılabilir

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
@TestPropertySource({"fooName=myFoo", "barName=myBar"})
public class Springtests {

    @Autowired
    private FooService fooService;

    ...
}
```

# Entegrasyon Testleri ve Transaction Yönetimi

- **@Transactional** anotasyonu test sınıfı veya metot düzeyinde tanımlanabilir
- Container'da tanımlı bir **PlatformTransactionManager** bean mevcut olmalıdır
- Test metodu başladığında bir **TX başlatılır**
- Test metodu sonlandığında da bu TX **rollback edilir**
- **@Rollback(false)** ile TX **commit** ettirilebilir

# Entegrasyon Testleri ve Transaction Yönetimi

```
@ContextConfiguration
@Transactional
public class TransactionalTests {
    @Test
    public void testWithRollback() {
        // ...
    }

    @Rollback(false)
    @Test
    public void testWithoutRollback() {
        // ...
    }
}
```

# Entegrasyon Testleri ve Transaction Yönetimi

- **@TransactionConfiguration** ile sınıf düzeyinde TX konfigürasyonu özelleştirilebilir
- **@Before** ve **@After** metotları TX içinde çalıştırılır
- **@BeforeTransaction**, **@AfterTransaction** ile de TX dışında setup ve destroy işlemleri yapılabilir



# Entegrasyon Testleri ve Transaction Yönetimi

```
@ContextConfiguration  
@Transactional  
@TransactionConfiguration(transactionManager="txMgr",  
defaultRollback=false)  
public class CustomConfiguredTransactionalTests {
```

```
    @Before  
    public void setUp() {  
    }
```

} **@Before** anotasyonu ile işaretlenmiş setUp metodu TX içerisinde çalıştırılır. TX dışında çalışması için **@BeforeTransaction** kullanılmalıdır

```
    @Rollback(true)  
    @Test  
    public void testProcessWithoutRollback() {  
    }
```

```
    @After  
    public void tearDown() {  
    }
```

} **@After** anotasyonu ile işaretlenmiş tearDown metodu da TX içerisinde çalıştırılır. TX dışında çalışması için **@AfterTransaction** kullanılmalıdır

```
}
```

# İletişim

- **Harezmi** Bilişim Çözümleri
- Kurumsal Java Eğitimleri
- <http://www.java-egitimleri.com>
- [info@java-egitimleri.com](mailto:info@java-egitimleri.com)



**harezmi**  
bilişim çözümleri

JAVA  
Eğitimleri 