LUXOFT TRAINING

# JVA-000
# Java Persistence with Hibernate

## Module 8
## Spring Framework Integration

## Objectives

Observe facilities the Spring framework provides for JPA enabled applications

- Configuration
- Dependency injection
- Transaction management

## JPA support in Spring

Spring offers comprehensive support for JPA:

- Environment setup
- `EntityManagerFactory` reference injection
- `EntityManager` reference injection
- Integration into Spring transaction management support

## Environment Setup

Three ways to setup `EntityManagerFactory`:

- Obtaining `EntityManagerFactory` from JNDI
- Using `LocalEntityManagerFactoryBean`
- Using `LocalContainerEntityManagerFactoryBean`

## Obtaining `EntityManagerFactory` from JNDI

Use this option when deploying application to the Java EE environment.

Obtaining an `EntityManagerFactory` from JNDI is simply a matter of changing the configuration:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:jee="http://www.springframework.org/schema/jee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
            http://www.springframework.org/schema/jee
            http://www.springframework.org/schema/jee/spring-jee.xsd">

    <jee:jndi-lookup id="entityManagerFactory" jndi-name="persistence/samplePersistenceUnit"/>

</beans>
```

## `LocalEntityManagerFactoryBean`

Use this option in simple deployment environments such as stand-alone applications and integration tests

The `LocalEntityManagerFactoryBean` creates an `EntityManagerFactory` suitable for the simple deployment environments.

The factory bean uses the JPA `PersistenceProvider` auto-detection mechanism (according to JPA's Java SE bootstrapping)

## `LocalEntityManagerFactoryBean`

This form of JPA deployment is the simplest and the most limited:

- No way to refer existing `DataSource` bean
- No support for global transactions
- No declarative transaction management
- Transaction management via JPA `Transaction` API only

## LocalEntityManagerFactoryBean

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context">

    <context:annotation-config/>

    <bean class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
        <property name="persistenceUnitName" value="samples.spring" />
    </bean>
    <bean class="samples.spring.service.CompanyServicePUnitImpl"/>
</beans>
```

```
@Repository
public class CompanyServicePUnitImpl implements CompanyService {

    @PersistenceUnit
    private EntityManagerFactory emf;

    @Override
    public void removeCompany(int id) {
        EntityManager em = emf.createEntityManager();

        em.getTransaction().begin();

        Company company = em.find(Company.class, id);
        em.remove(company);

        em.getTransaction().commit();
    }
}
```

Reference to EntityManagerFactory is injected by Spring container for @PersistenceUnit fields.

Injected EntityManagerFactory used to create EntityManager

Transaction started manually via JPA API

Transaction commited manually via JPA API

LocalEntityManagerFactoryBean provides limited facilities

## `LocalContainerEntityManagerFactoryBean`

- The most powerful option to setup JPA
- Allows flexible configuration
- Supports links to existing JDBC `DataSource`
- Supports both local and global transactions
- Supports declarative transaction management (via Spring `PlatformManager` implementation)

# LocalContainerEntityManagerFactoryBean

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:jee="http://www.springframework.org/schema/jee">

    <context:annotation-config/>

    <jee:jndi-lookup id="dataSource" jndi-name="jdbc/DataSource"/>

    <bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
        <property name="persistenceXmlLocation" value="classpath:META-INF/persistence.xml"/>
        <property name="persistenceUnitName" value="samples.spring" />
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <bean class="samples.spring.service.CompanyServicePContextImpl"/>

</beans>
```

More powerful way to create EntityManagerFactory

```
@Repository
@Transactional(Transactional.TxType.REQUIRED)
public class CompanyServicePContextImpl implements CompanyService {

    @PersistenceContext
    private EntityManager em;

    @Override
    public void removeCompany(int id) {
        Company company = em.find(Company.class, id);
        em.remove(company);
    }

}
```

Reference to EntityManager is injected by Spring Container for @PersistenceContext field

Contains the only business logic.
No need of transaction management code.

LocalContainerEntityManagerFactoryBean provides more facilities

## Using `LocalContainerEntityManagerFactoryBean`

`PersistenceUnitManager` defines abstraction for finding/mapping JPA configuration

In `LocalContainerEntityManagerFactoryBean` used for loading JPA configuration

Default implementation is provided by Spring is `DefaultPersistenceUnitManager`:

- Reads configuration from `META-INF/persistence.xml`
- Supports spring-based scanning for entity classes
- Supports JDBC `DataSources` specifying that JPA persistence provider is supposed to use

## Implementing DAOs

Spring supports injection of:

- `EntityManagerFactory` (`@PersistenceUnit`)
- `EntityManager` (`@PersistenceContext`)

Annotations are supported both on field and method levels

`PersistenceAnnotationBeanPostProcessor` needs to be enabled:

- Via bean definition
- Via `<context:annotation-config/>`

## Implementing DAOs

```
public interface JpaDao <T, K extends Serializable> {
    List<T> findAll();
    T findByKey(K key);
    void persist(T entity);
    void remove(T entity);
}
```

DAO interface common to all JPA DAOs:
- entity type - parameterized
- entity key - parameterized

```
public class CompanyDao implements JpaDao<Company, Integer> {
    @PersistenceContext
    private EntityManager em;

    public List<Company> findAll() {
        TypedQuery<Company> query =
                em.createQuery("SELECT c FROM Company c ORDER BY c.name", Company.class);
        return query.getResultList();
    }

    public Company findByKey(Integer key) {
        return em.find(Company.class, key);
    }

    public void persist(Company entity) {
        em.persist(entity);
    }

    public void remove(Company entity) {
        em.remove(entity);
    }
}
```

DAO implementation
for Company entity

Sample of DAO implementation for Spring or EJB3 container

## Implementing DAOs

**Question**: Do we really need DAO when using JPA?

DAO advantages:

- Single point of JPA access
- Ability to limit operations for entity (ex. no removal)
- With DAO it’s easy change data access technique
- You can centralize all queries on certain entity instead of scattering them through your code

**Answer**: It depends how complex your application really is.

## Transactions :: ACID

**Atomicity** -  if one part of the transaction fails, the entire transaction fails, and state is left unchanged

**Consistency** - any data written to database must be valid according to all defined rules

**Isolation** – changes being made in concurrent transaction not visible until it’s allowed

**Durability** - if transaction is committed, it will remain so  (even in the event of power loss, crashes, or errors)

## Transactions :: Types

### Local

- Local to the transactional resource (ex. database)
- Used in JSE
- Used in non-managed J2EE

### Global

- Used in managed J2EE
- Managed by Application Server
- Controlled via Java Transaction API

## Transactions :: Spring facilities

`PlatformTransactionManager` the main abstraction to handle transaction in Spring

Existing implementation:

- `JpaTransactionManager`
- `JtaTransactionManager`
- `HibernateTransactionManager`
- `DataSourceTransactionManager`

## Transactions :: Spring facilities

Spring support transaction management:

**Declarative**

- Based on aspects
- Applicable for public methods only
- Rules can be described via annotations

**API based**

- `PlatformTransactionManager`
- `TransactionTemplate`

## Transactions :: Spring facilities

To enable Spring transaction management:

1. Configure needed `PlatformTransactionManager`
2. Register configure transaction manager
3. Apply rules on public methods of Spring beans

**Note**: Spring can manage the only objects which are created by Spring (means defined as beans)

*Objects created using new operator aren't managed by Spring*

## Transactions :: Spring facilities

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:jee="http://www.springframework.org/schema/jee">

    <context:annotation-config/>

    <jee:jndi-lookup id="dataSource" jndi-name="jdbc/DataSource"/>

    <bean id="emf" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
        <property name="persistenceUnitName" value="samples.spring" />
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <bean id="txManager" class="org.springframework.orm.jpa.JpaTransactionManager">
        <property name="entityManagerFactory" ref="emf" />
        <property name="dataSource" ref="dataSource" />
    </bean>

    <tx:annotation-driven transaction-manager="txManager"/>

    <bean class="samples.spring.service.CompanyServicePContextImpl"/>

</beans>
```

Configure Tx manager

Enabe declarative Tx management
and register Tx manager

Service object that needs
Tx management to be applied

Spring configuration with Tx management configured/enabled

# Transactions :: Spring facilities

```
@Transactional(propagation = Propagation.REQUIRED)
@Repository
public class CompanyServicePContextImpl implements CompanyService {

    @PersistenceContext
    private EntityManager em;

    @Transactional(propagation = Propagation.REQUIRES_NEW, readOnly = true)
    public Company getCompany(int id) {
        return em.find(Company.class, id);
    }

    public void removeCompany(int id) {
        Company company = em.find(Company.class, id);
        em.remove(company);
    }

}
```

Tx attributes default to all methods in class

Individual Tx attributes for the method

Declarative Tx management using annotations

## Transactions :: Spring facilities

Transaction parameters:

- **Isolation** – level of Tx isolation
- **Propagation** – defines how Tx passed between methods
- **Timeout** – Tx timeout
- **Read-only** - prohibits any changes in Tx

## Transactions :: Spring facilities

Tx isolation levels (class `Isolation`):

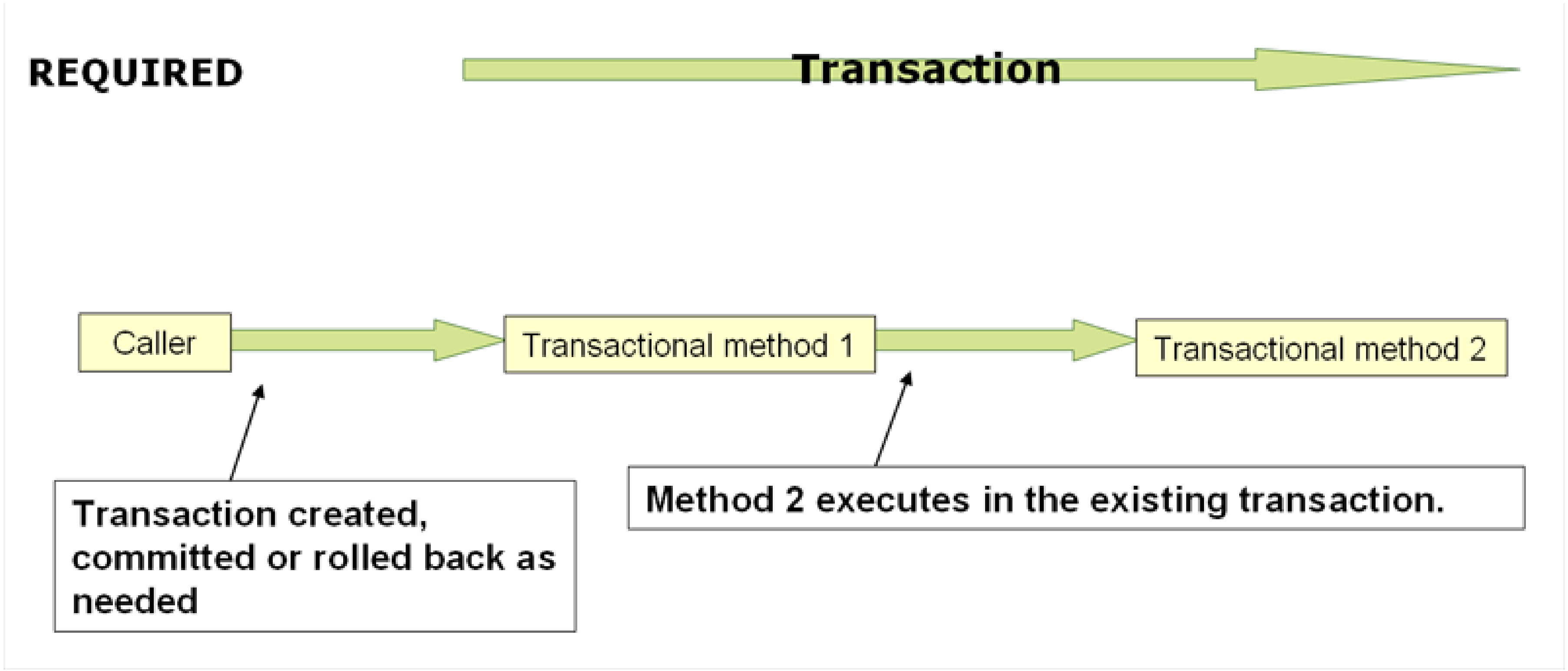
- **DEFAULT** – use the default to DB level of isolation
- **READ_UNCOMMITTED** – allows reading not committed changes. Dirty reads, non-repeatable reads and phantoms are possible.
- **READ_COMMITTED** – Non-repeatable reads and phantoms are possible.
- **REPEATABLE_READ** – any changes in parallel Tx are not visible. Phantoms are still possible.
- **SERIALIZABLE** – max restricted level. No deviations are possbible.

## Transactions :: Spring facilities

Tx propagation (class `Propagation`):

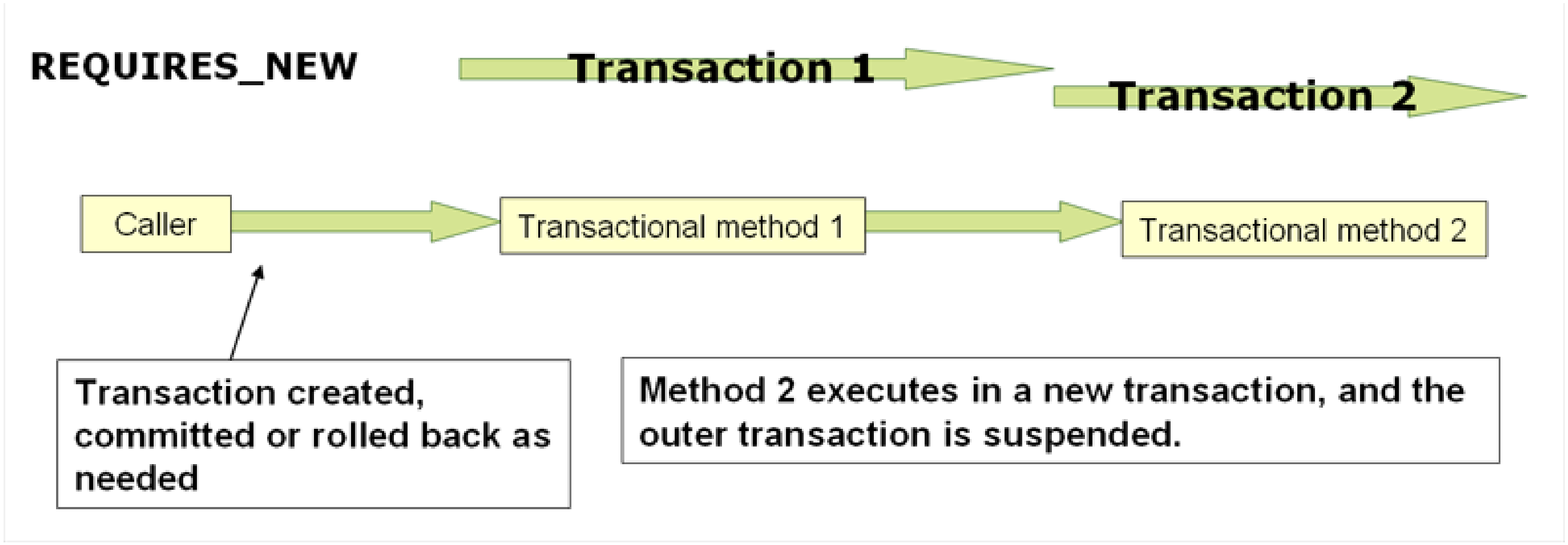
- **REQUIRED** – support current transaction, create a new one if none exists
- **SUPPORTS** – support a current transaction, execute non-transactionally if none exists.
- **MANDATORY** – support a current transaction, throw an exception if none exists.
- **REQUIRES_NEW** – create a new transaction, suspend the current transaction if one exists.
- **NOT_SUPPORTED** – execute non-transactionally, suspend the current transaction if one exists.
- **NEVER** - execute non-transactionally, throw an exception if a transaction exists.

## Transactions :: Spring facilities


REQUIRED
Transaction
Caller
Transactional method 1
Transactional method 2
Transaction created, committed or rolled back as needed
Method 2 executes in the existing transaction.

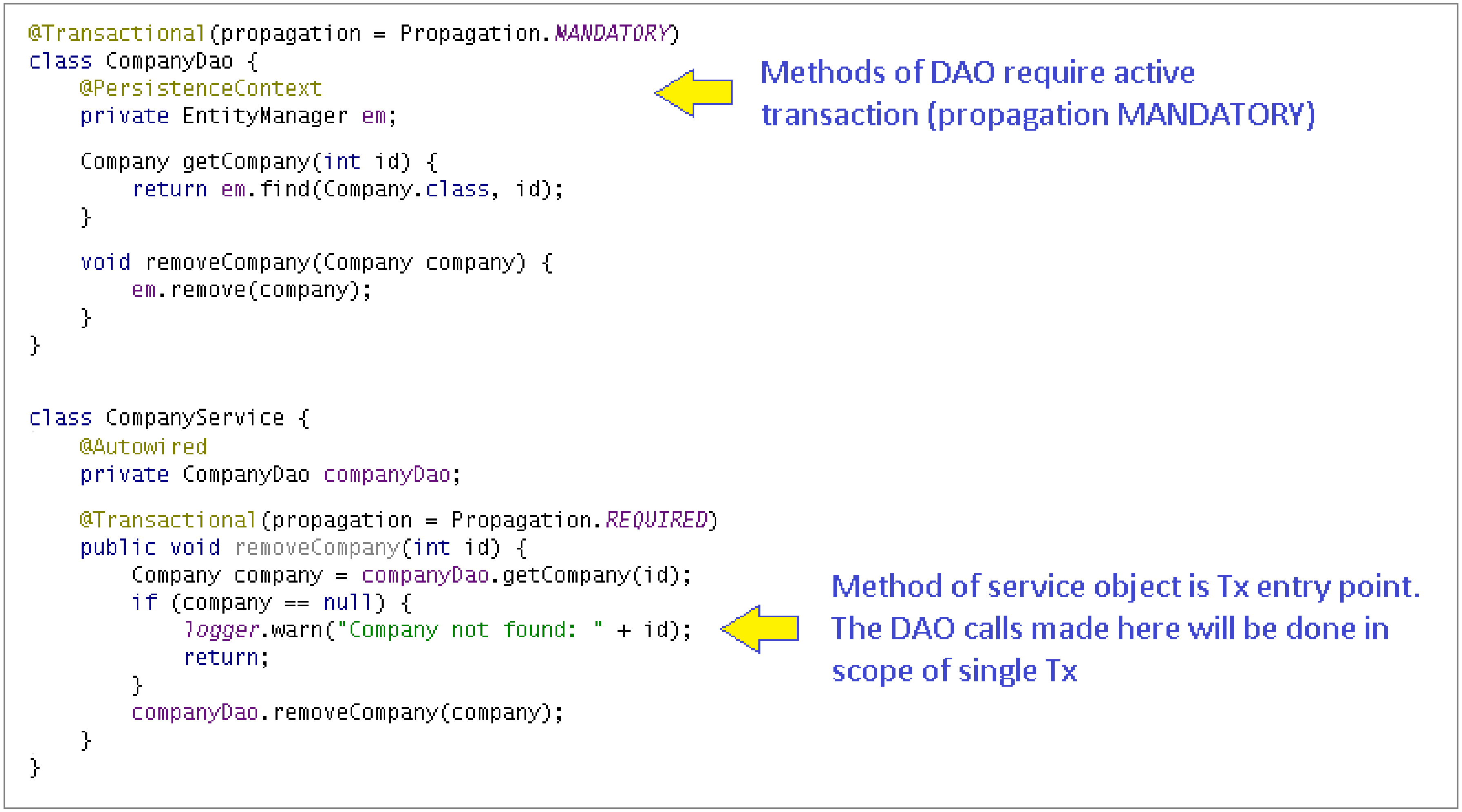
Propagation **REQUIRED**: support current transaction, create a new one if none exists

# Transactions :: Spring facilities


REQUIRES_NEW
Transaction 1
Transaction 2
Caller
Transactional method 1
Transactional method 2
Transaction created, committed or rolled back as needed
Method 2 executes in a new transaction, and the outer transaction is suspended.

Propagation **REQUIRES_NEW**: create a new transaction, suspend the current transaction if one exists.

# Transactions :: Spring facilities



```
@Transactional(propagation = Propagation.MANDATORY)
class CompanyDao {
    @PersistenceContext
    private EntityManager em;

    Company getCompany(int id) {
        return em.find(Company.class, id);
    }

    void removeCompany(Company company) {
        em.remove(company);
    }
}


class CompanyService {
    @Autowired
    private CompanyDao companyDao;

    @Transactional(propagation = Propagation.REQUIRED)
    public void removeCompany(int id) {
        Company company = companyDao.getCompany(id);
        if (company == null) {
            logger.warn("Company not found: " + id);
            return;
        }
        companyDao.removeCompany(company);
    }
}
```

Methods of DAO require active transaction (propagation MANDATORY)

Method of service object is Tx entry point. The DAO calls made here will be done in scope of single Tx

Tx organization when using DOA with service object

## Practical work

**Exercise 7:**

Integration with Spring Framework

LUXOFT TRAINING

Thank you for your attention!

Questions?