



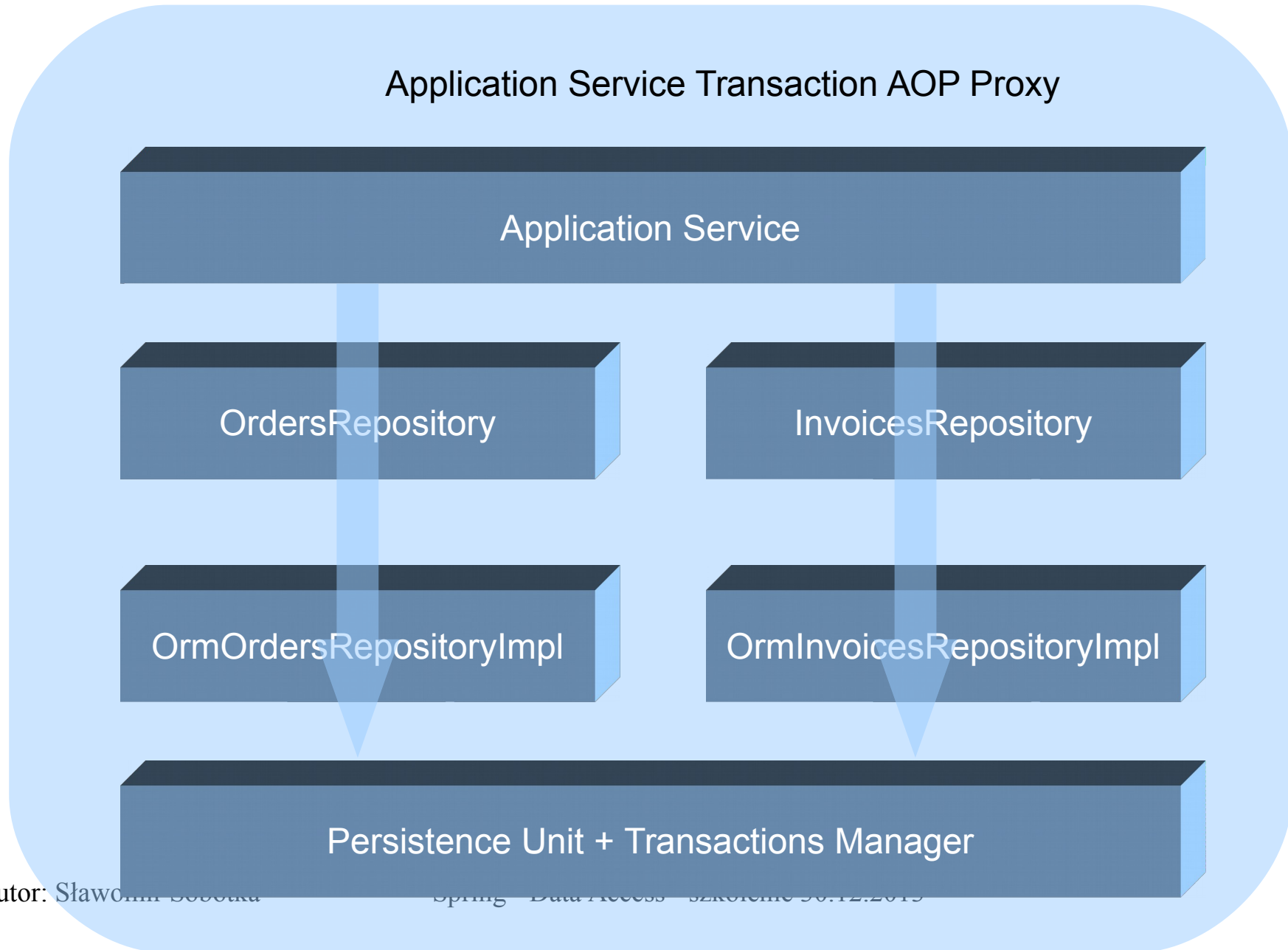
Praktycy dla Praktyków
Szkolenia i doradztwo

Spring Framework

Data Access

Dowiesz się:

- ACID, Anomalie, Izolacja
- Zarządzanie transakcjami
- Propagacja transakcji



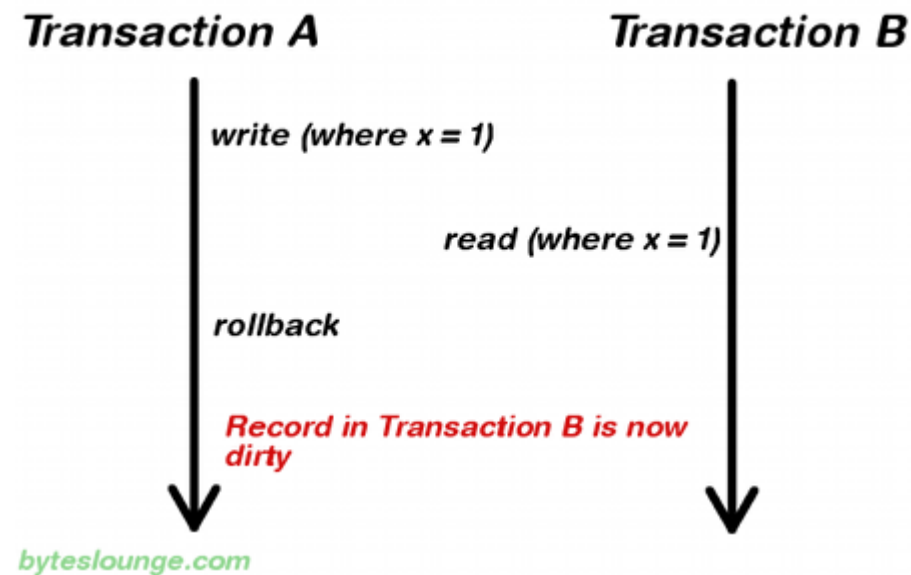
- **Atomic** – all or nothing
- **Consistent** – transaction will bring the database from one valid state to another; consistent, no integrity violation
- **Isolated** – transaction should be able to interfere with another transaction (depends on isolation level)
- **Durable** - means that once a transaction has been committed, it will remain so, even after crash

- **dirty read** – TX1 read data changed by TX2 even if later TX2 is rolled back. TX1 read non-existing data
- **unrepeatable read** – TX1 performs the same query but receives different data (row attributes). TX2 modifies attributes
- **phantoms** - TX1 performs the same query but receives different data sets (amount of rows). TX2 modifies rows.

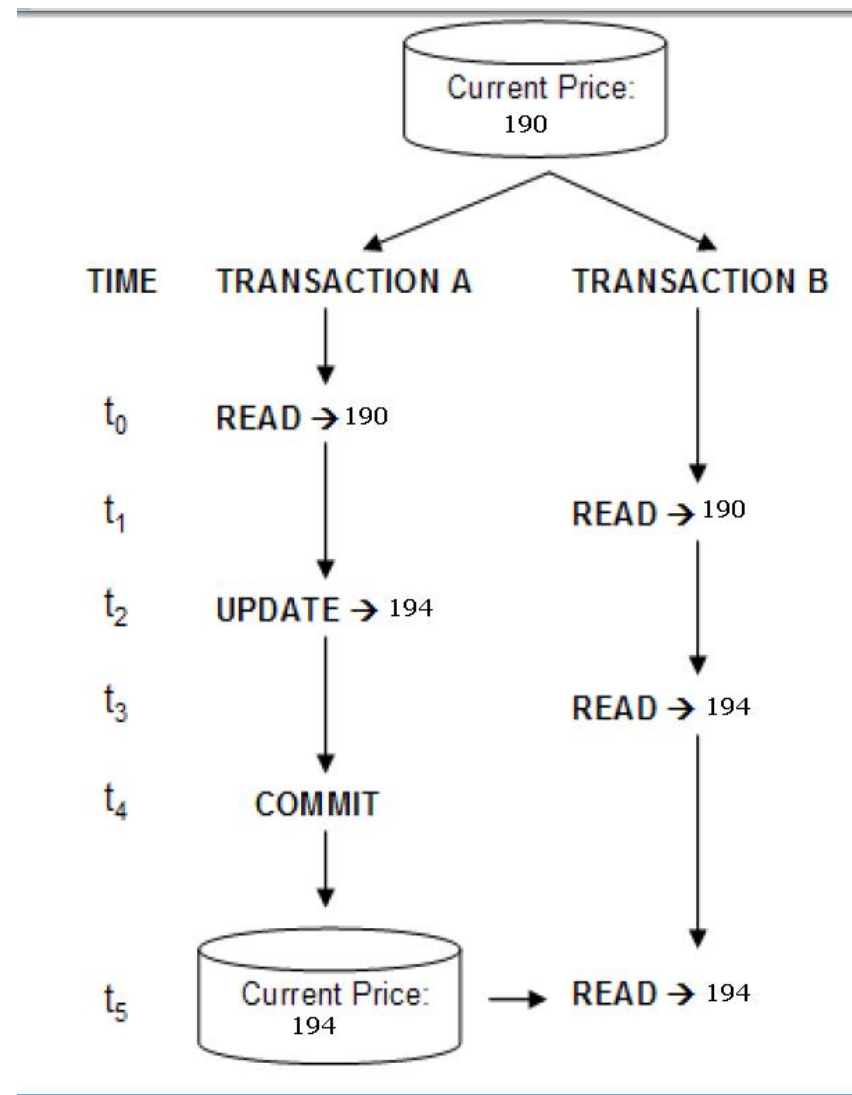
Dirty Read is crucial problem for data integrity.
Unrepeatable reads and Phantoms can be tolerated.

- **read_uncommitted** – uncommitted changes made by TX1 are visible for TX2 (dirty-reads, non-repetable-reads, phantoms)
- **read_committed** – changes made by TX1 are visible only after commit (non-repetable-reads, phantoms)
- **repetable_read** – loaded data are blocked, so repeated reads returns the same data (phantoms)
- **serializable** – transactions are serialized (no anomalies)

DIRTY READ

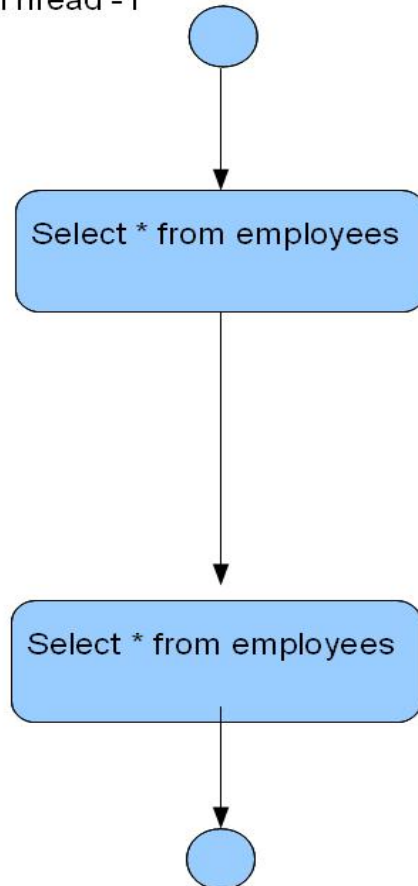


UNREPEATABLE READ

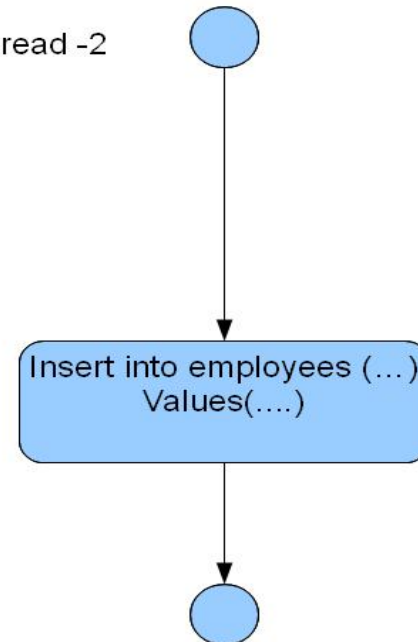


PHANTOM READ

Thread -1



Thread -2



```
<beans>
```

```
<!-- transakcje sterowane adnotacjami -->
```

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

```
<!-- Odpowiednia implementacja TxManagera (JDBC, Hibernate,...) -->
```

```
<bean id="txManager"
```

```
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
```

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

```
</bean>
```

```
</beans>
```

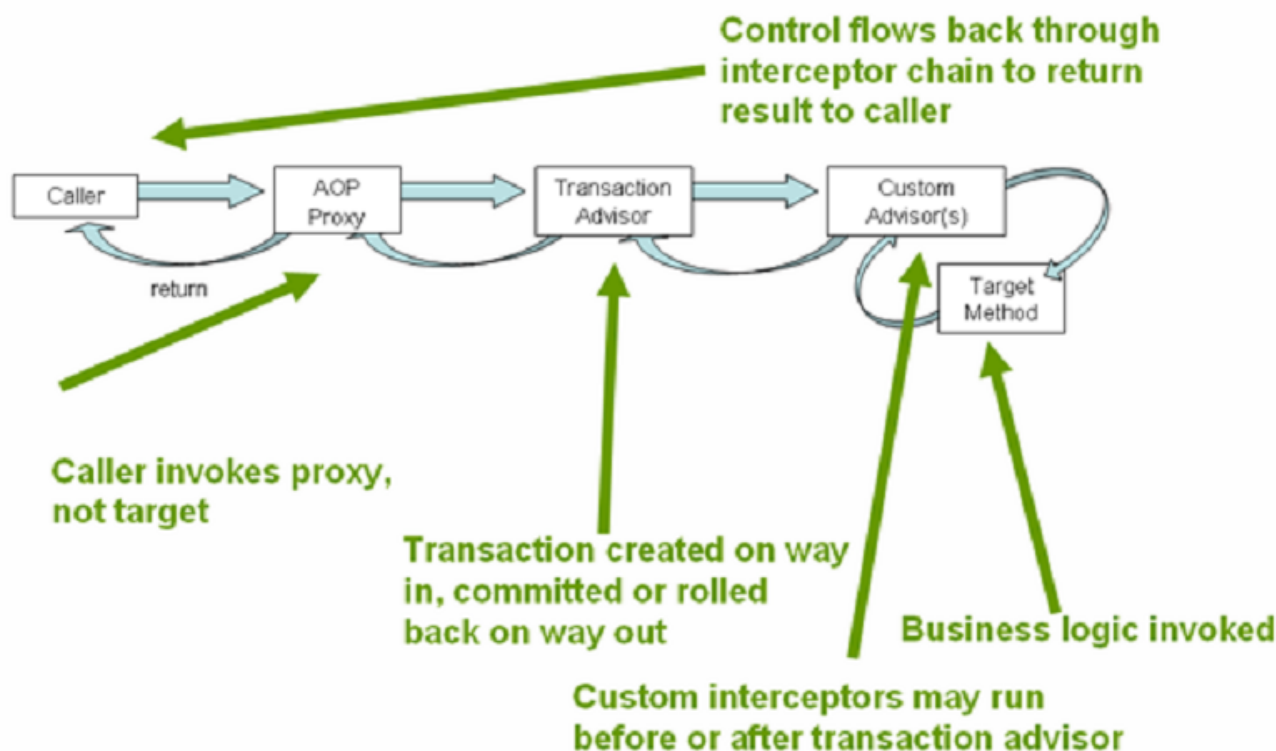
Spring dostarcza abstrakcji ponad dostępnymi podejściami do zarządzania transakcjami

- lokalne
- JEE
- rozproszone (JTA)

Manager transakcji działa na abstrakcyjnym źródle danych

Zarządzanie transakcjami

- deklaratywne: Adnotacje, XML AOP Advices
- imperatywne: TxManager API, TransactionTemplate callbacks



źródło: Spring Reference

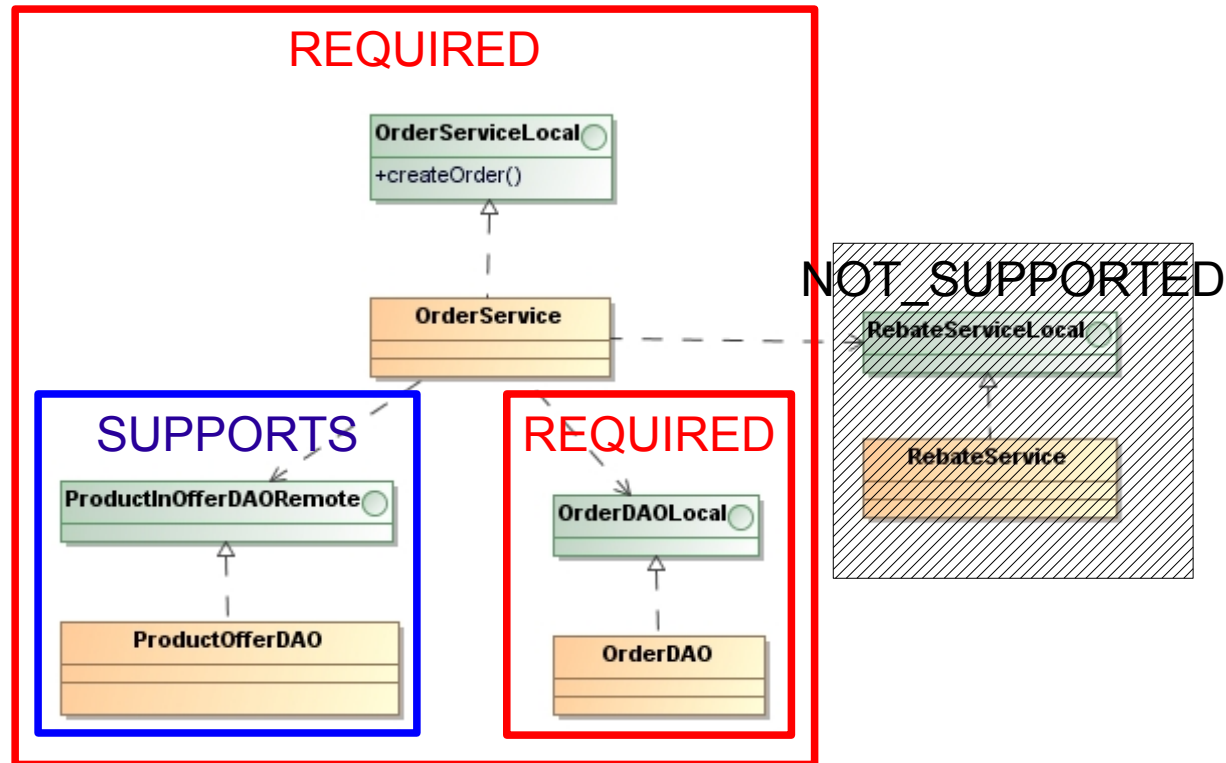
Sterowanie wyjątkami

- weryfikowalne (Exception) – nie wycofują tx
- nieweryfikowalne (RuntimeException, Error) - rollback

- Adnotacja klasy (obejmuje wszystkie metody)
- **UWAGA: Wołając metodę z tej samej klasy obowiązują reguły metody nadrzędnej** – ograniczenie implementacji AOP, obejście wymaga AspectJ
- Adnotacja metody (ew. nadpisuje zasady klasy)
 - tylko dla metod publicznych
 - gdy metoda jest wołana przez inną z danej klasy to pośrednik AOP nie jest nakładany – nie można sterować transakcją z osobna

```
@Transactional(readOnly = true)
public class DefaultFooService implements FooService {
    public Foo getFoo(String fooName) {
    }

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



- **REQUIRES - domyślnie**
 - Jeżeli transakcja nie istnieje, to wówczas jest tworzona
 - Jeżeli metoda rozpoczyna transakcję to musi również ją zakończyć (commit/rollback)
 - Jedna transakcja fizyczna
 - Podczas propagacji są tworzone osobne logiczne transakcje
 - Marker rollbackOnly per metoda
 - Zewnętrzna metoda może obsłużyć wyjątek wewnętrznej
- **REQUIRES_NEW**
 - Jeżeli transakcja istnieje to jest zawieszana
 - Osobne transakcje fizyczne
 - Zewnętrzna metoda może kontynuować pomimo rollback wewnętrznej – jeżeli przechwyci wyjątek
- **NESTED:** <http://forum.springsource.org/archive/index.php/t-16594.html>
 - Jedna transakcja fizyczna + Savepoints
 - Commit na końcu całości
 - Zewnętrzna metoda może kontynuować pomimo rollback wewnętrznej

```
@Autowired
private MyDAO myDAO;

@Autowired
private InnerBean innerBean;

@Override
@Transactional(propagation=Propagation.REQUIRED)
public void testRequired(User user) {
    myDAO.insertUser(user);
    try{
        innerBean.doRequired();
    } catch (RuntimeException e) {
        // handle exception
    }
}
```

RuntimeException oznacza transakcję logiczną do wycofania (mimo catch) – żadne dane nie będą zapisane

```
@Override
@Transactional(propagation=Propagation.REQUIRED)
public void doRequired() {
    throw new RuntimeException("Rollback this transaction!");
}
```

```
@Autowired  
private MyDAO myDAO;
```

```
@Autowired  
private InnerBean innerBean;
```

```
@Override  
@Transactional(propagation=Propagation.REQUIRED)  
public void testRequiresNew(User user) {  
    myDAO.insertUser(user);  
    try{  
        innerBean.doRequiresNew();  
    } catch (RuntimeException e){  
        // handle exception  
    }  
}
```

**RuntimeException wycofuje
transakcję fizyczną, ale nie
wpływa na transakcję metody
zewnętrznej**

```
@Override  
@Transactional(propagation=Propagation.REQUIRES_NEW)  
public void doRequiresNew() {  
    throw new RuntimeException("Rollback this transaction!");  
}
```



```
@Autowired
private MyDAO myDAO;

@Autowired
private InnerBean innerBean;

@Override
@Transactional(propagation=Propagation.REQUIRED)
public void testNested(User user) {
    myDAO.insertUser(user);
    try{
        innerBean.doNested();
    } catch (RuntimeException e){
        // handle exception
    }
}
```

RuntimeException wycofuje do
savePoint.
**Gdyby nie został przechwycony,
wówczas wycofa całość.**

```
@Override
@Transactional(propagation=Propagation.NESTED)
public void donested() {
    throw new RuntimeException("Rollback this transaction!");
}
```

- SUPPORTS
 - Może działać bez kontekstu transakcji
 - Jeżeli kontekst istnieje to jest do niego dołączana (widoczność Session/EntityManager)
 - Odpowiednia propagacja do odczytów
 - Metoda odczytująca „widzi” dane zapisane (jeszcze bez commit) przez metodę nadrzędną – dane z transaction log
 - Przy braku transakcji odczyt nastąpi z danych z tabeli
- MANDATORY
 - Dołącza się do istniejącego kontekstu transakcji
 - Jeżeli takowy nie istnieje to wyjątek
 - Odpowiednia propagacja dla transakcji zarządzanych przez kod kliencki

- NOT_SUPPORTED
 - Zawiesza istniejącą transakcję (brak widoczności zasobów taki jak Session)
 - Odpowiednie dla wywołania procedur, które same zarządzają transakcjami i łącznie (chaining) nie jest wspierane
- NEVER
 - Wyjątek gdy transakcja istnieje
 - Zastosowanie: ???
 - Testowanie: jeżeli metoda zwraca wyjątek, oznacza to, że kontekst transakcji istniał

- **REQUIRES_NEW**

- Zawsze jest tworzona nowa transakcja
 - Jeżeli używamy JPA to pracujemy z nową instancją EntityManager/Session (osobny L1 cache)
- Uwaga na rekursje!
- Możemy niezależnie sterować izolacją takiej transakcji
 - Pozwala nakładać silną izolację na krótsze jednostki czasu (np. generator kluczy)

- **NESTED**

- Przetwarzanie dużych ilości „pod-ścieżek”
- Warto używać NOT_SUPPORTED i NEVER
- XA tylko gdy naprawdę ich potrzebujemy
 - Warto używać obu managerów transakcji: XA i local

- Transakcje na warstwie API (serwisów)
 - Serwisy REQUIRED
 - Odczyt SUPPORTS
- Transakcje zorientowane na współbieżność
 - Skracanie czasu w transakcji
 - Odczyty najpierw (SUPPORTS)
 - Zapis później (REQUIRED)
- Transakcje zorientowane na wydajność
 - Zarządzane przez bazę (local) – każda operacja jest zapisywane
 - Mechanizm kompensacyjny
- Transakcje sterowane przez kod kliencki
 -

- **value** – (opcjonalnie) wsazanie Tx Managera (jeżeli zdefiniowano kilka w systemie)
- **propagation** – polityka propagacji (domyślnie REQUIRED)
 - **required, requires_new** (niezależne transakcje, zewnętrzna jest zawieszana), **nested** (jdbc savepoints)
- **isolation** – poziom izolacji (domyślny ba źródła)
- **readOnly** – true w celu optymalizacji zasobów DB
- **timeout**
- **rollbackFor/noRollbackFor** – klasy wyjątków (nie)wycofujących transakcję

```
public class SimpleService {  
    private final TransactionTemplate transactionTemplate;  
  
    public SimpleService(PlatformTransactionManager transactionManager) {  
        this.transactionTemplate = new TransactionTemplate(  
            transactionManager);  
    }  
  
    public Object someServiceMethod() {  
        transactionTemplate.execute(new TransactionCallbackWithoutResult()  
  
            protected void doInTransactionWithoutResult(  
                TransactionStatus status) {  
  
                try {  
                    //...  
                } catch (SomeBusinessException ex) {  
                    status.setRollbackOnly();  
                }  
            }  
        });  
    }  
}
```

org.springframework.transaction.PlatformTransactionManager

```
DefaultTransactionDefinition def = new DefaultTransactionDefinition();  
def.setName("SomeTxName");  
def.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRED);
```

```
TransactionStatus status = txManager.getTransaction(def);  
try {  
    // ...  
}  
catch (MyException ex) {  
    txManager.rollback(status);  
    throw ex;  
}  
txManager.commit(status);
```


<http://www.ibm.com/developerworks/java/library/j-ts1/index.html?ca=drs->

- ReadOnly dla JDBC
 - Nie ma sensu z SUPPORTS
 - supports podłącza się do istniejącej transakcji, która domyślnie istnieje w tym kontekście
 - Działa tylko dla REQUIRED
 - Zatem readOnly wymusza stosowanie transakcji w ogóle
 - Co powoduje zbędny narzut
- RadOnly dla JPA
 - Ma sens jedynie z SUPPORTS – domyślnie mamy REQUIRED
 - Generalnie: zależy od implementacji JPA