Praktycy dla Praktyków
Szkolenia i doradztwo

Spring Framework

Data Access

Dowiesz się:

- Jak konfigurować dostętp do danych
- Jak używać JDBC
  - Dlaczego warto używać iBatis
- Jak zarządzać transakcjami
- Jak używać JPA

Konfiguracja źródła danych

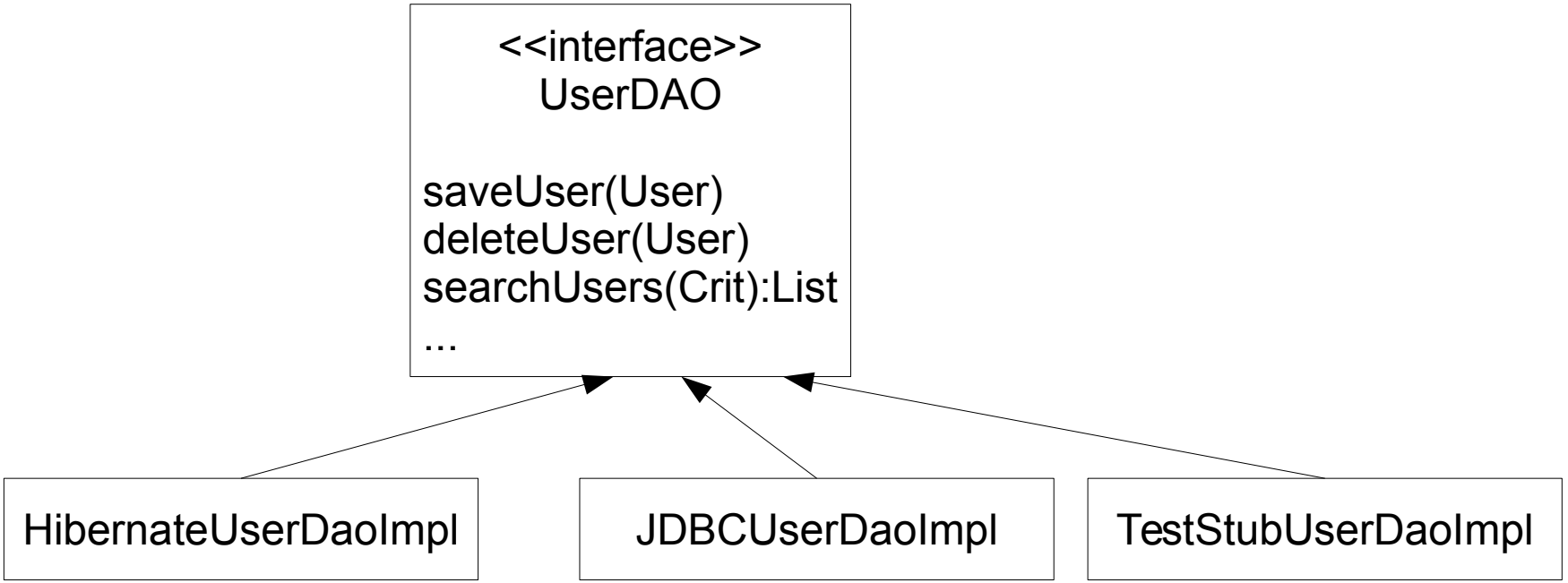
- Konfiguracja źródła danych

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

- Wartości zmiennych `${...}` zapisane są w pliku `jdbc.properites`

```
<context:property-placeholder location="classpath:jdbc.properties" />
```

Data Access Objects



- Hermetyzacja dostępu do danych
  - Ukrycie skomplikowanej logiki zapytań
  - Reużywalność logiki zapytań
  - Możliwość zmiany źródła danych
    - Rzadko występuje w projekcie
      - Przydatne podczas refaktoryzacji z powodu wydajności ORM->JDBC, Baza->RAM
    - Przydatna w testach jednostkowych (redukcja czasu)

Tworzenie szablonu

```
public class MyJdbcDao implements MyDao {
    private JdbcTemplate jdbcTemplate;

    @Autowired
    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }
    //...
}
```

Wsparcie dla JDBC
Szablony

Action	Spring	You
Define connection parameters.		X
Open the connection.	X	
Specify the SQL statement.		X
Declare parameters and provide parameter values		X
Prepare and execute the statement.	X	
Set up the loop to iterate through the results (if any).	X	
Do the work for each iteration.		X
Process any exception.	X	
Handle transactions.	X	
Close the connection, statement and resultset.	X	

Źródło: Spring Reference

Szablony Zapytanie

- `jdbcTemplate.query(sql, params, new RowCallbackHandler() {...})`
- `jdbcTemplate.query(sql, params, types, new RowCallbackHandler() {...});`
  - `sql`
    - instrukcja sql
  - `params`
    - parametry do obiektu `PreparedStatement`
  - `Types`
    - typy parametrów w obiekcie `PreparedStatement`. Parametr opcjonalny dodawany celem prawidłowej obsługi wartości `null`, jeżeli taka będzie zwracana w obiekcie `ResultSet`
  - `RowCallbackHandler()` posiada metodę do implementacji:
    - `public void processRow(ResultSet rs) throws SQLException {...}`
    - w ciele tej metody należy ustawić parametry zwracanego obiektu

Zapytanie Przykład

```
public Phone getPhone(final int phoneId) {
    String sql = "select * from PHONE where PHONE_ID = ?";
    final Phone phone = new Phone();

    Object params[] = new Object[] { new Integer(producerId)} ;
    jdbcTemplate.query(sql, params, new RowCallbackHandler() {
        public void processRow(ResultSet rs) throws SQLException {
            phone.setId(rs.getString("PHONEID"));
            phone.setNumber(rs.getString("NUMBER");
            phone.setDirection(rs.getString("DIRECTION"));
        }
    });
return phone;
}
```

Szablony CRUD

- Operacje kasowania i modyfikacji rekordów wykonywane są też w oparciu o metodę
  - `jdbcTemplate.update(sql, params, types);`
- Wymaga zmiany tylko parametru `sql`. Pozostała część kodu jest stała.
- Interpretacja zwracanej wartości pozostaje taka sama: liczba rekordów, które uległy zmianie
  - kasowanie – liczba skasowanych rekordów
  - modyfikacja – liczba zaktualizowanych rekordów

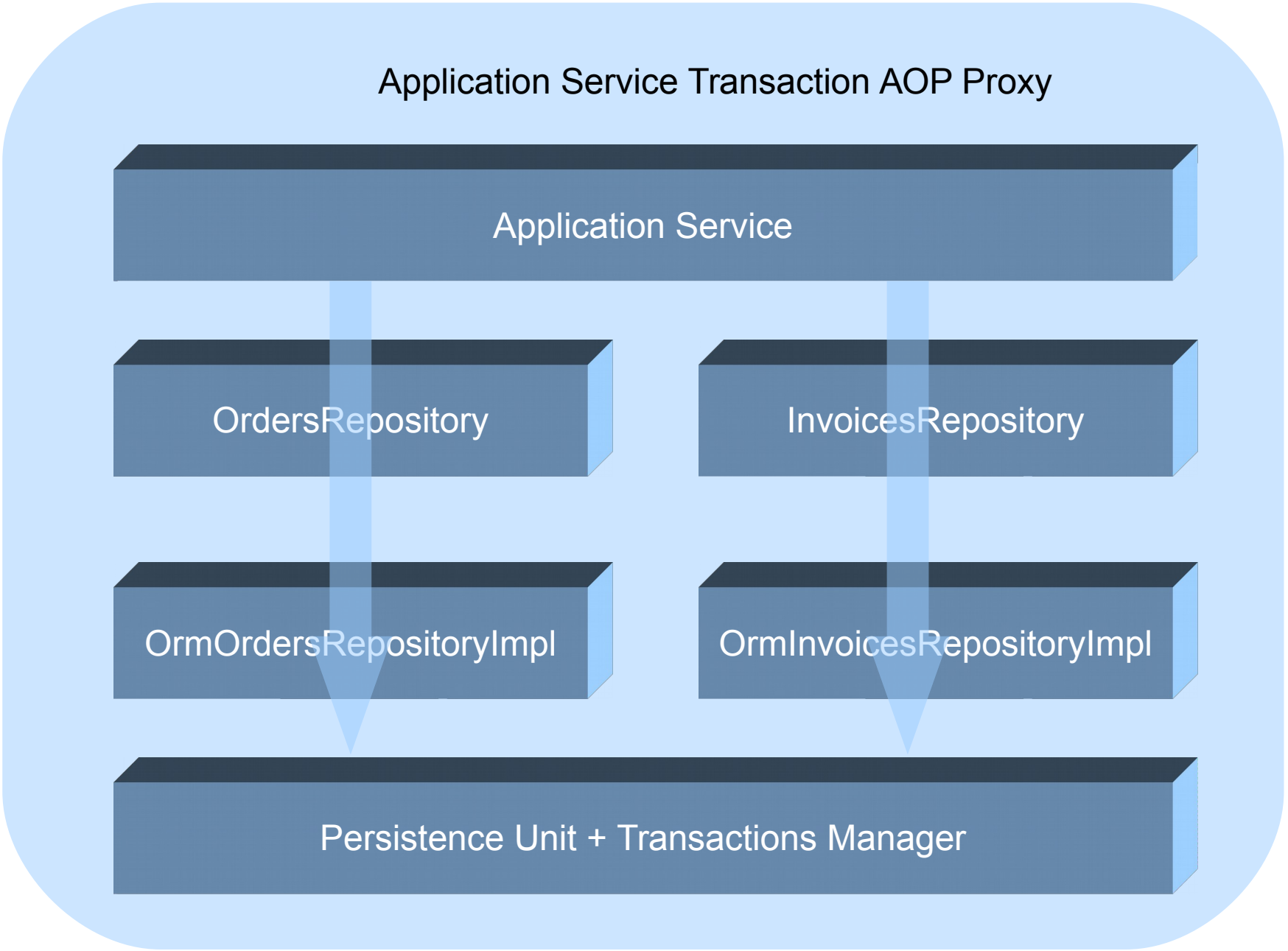
CRUD Przykład

```
String sql = "insert into PHONE(PHONE_ID, NUMBER, DIRECTION) VALUES (?, ?, ?)";
Object [] params = new Object[] {
    phone.getId(),
    phone.getNumber(),
    phone.getDirection()
}

int [] types = new int[] {
    Types.INTEGER,
    Types.INTEGER,
    Types.INTEGER
}

jdbcTemplate.update(sql, params, types);
```

Dowiesz się:

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

Wiele repozytoriów Transakcje



Transakcje ACID

- **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

Anomalie odczytu

- **dirty read** – TX1 read data changed by TX2 even if later TX2 is rolled back. TX1 red 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.

Poziomy izolacji
Wydajność / spójność

- **read_uncommited** – uncommitted changes made by TX1 are visible for TX2 (dirty-reads, non-repetable-reads, phantoms)

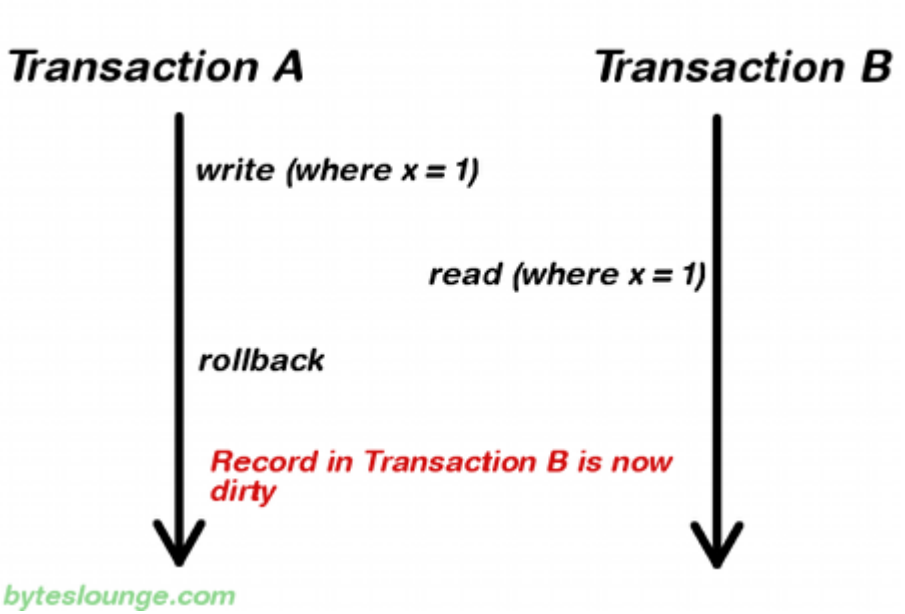
- **read_commited** – changes made by TX1 are visible only after commit (non-repetable-reads, phantoms)

- **repetable_read** – loaded data are blocked, so repeated reads returns the same data (phantoms)

- **serializable** – transactions are serialized (no anomalies)

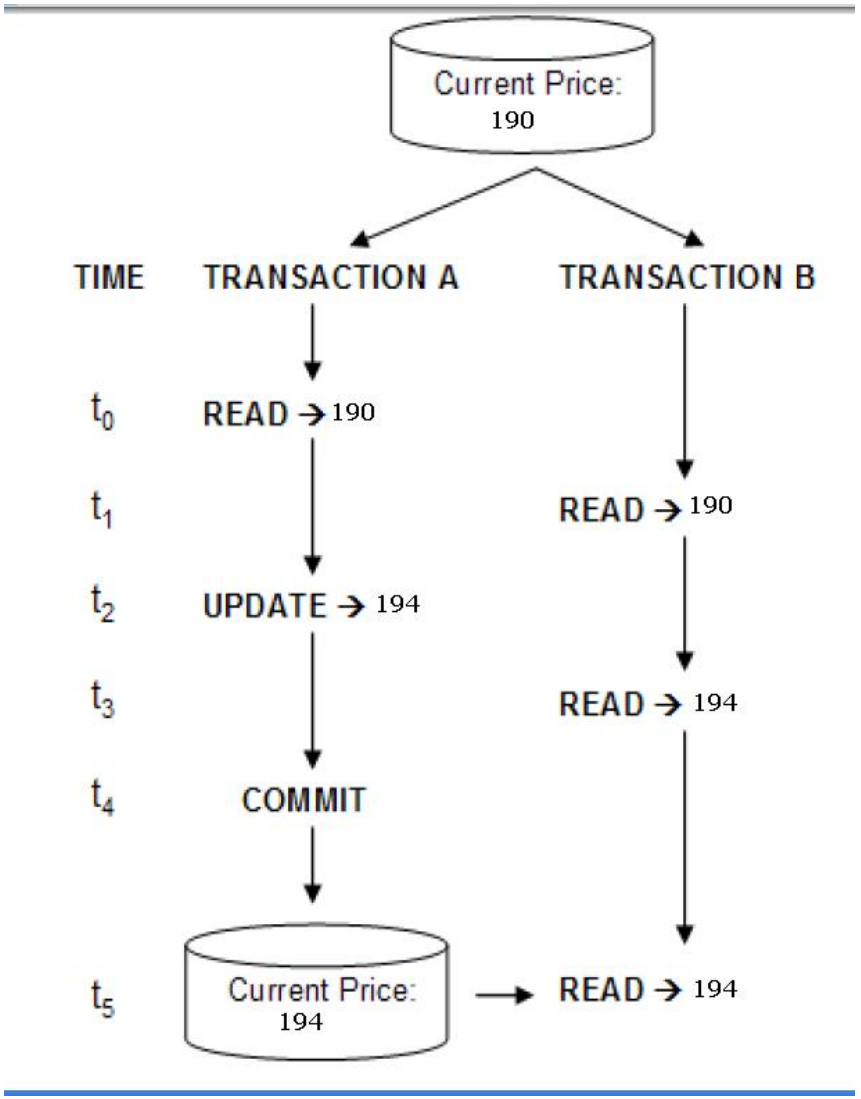
Poziomy izolacji
Wydajność / spójność

DIRTY READ



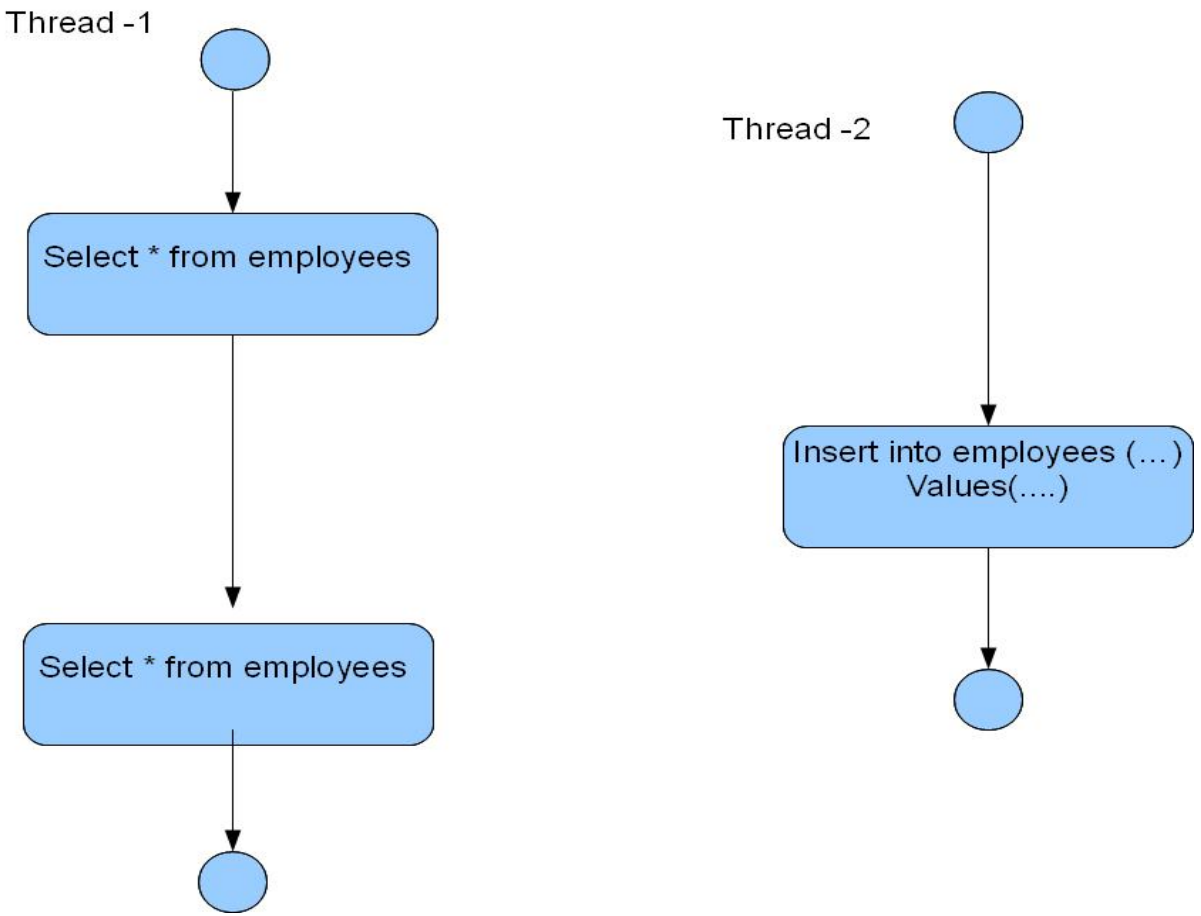
Poziomy izolacji
Wydajność / spójność

UNREPEATABLE READ



Poziomy izolacji
Wydajność / spójność

PHANTOM READ



Transakcje Konfiguracja

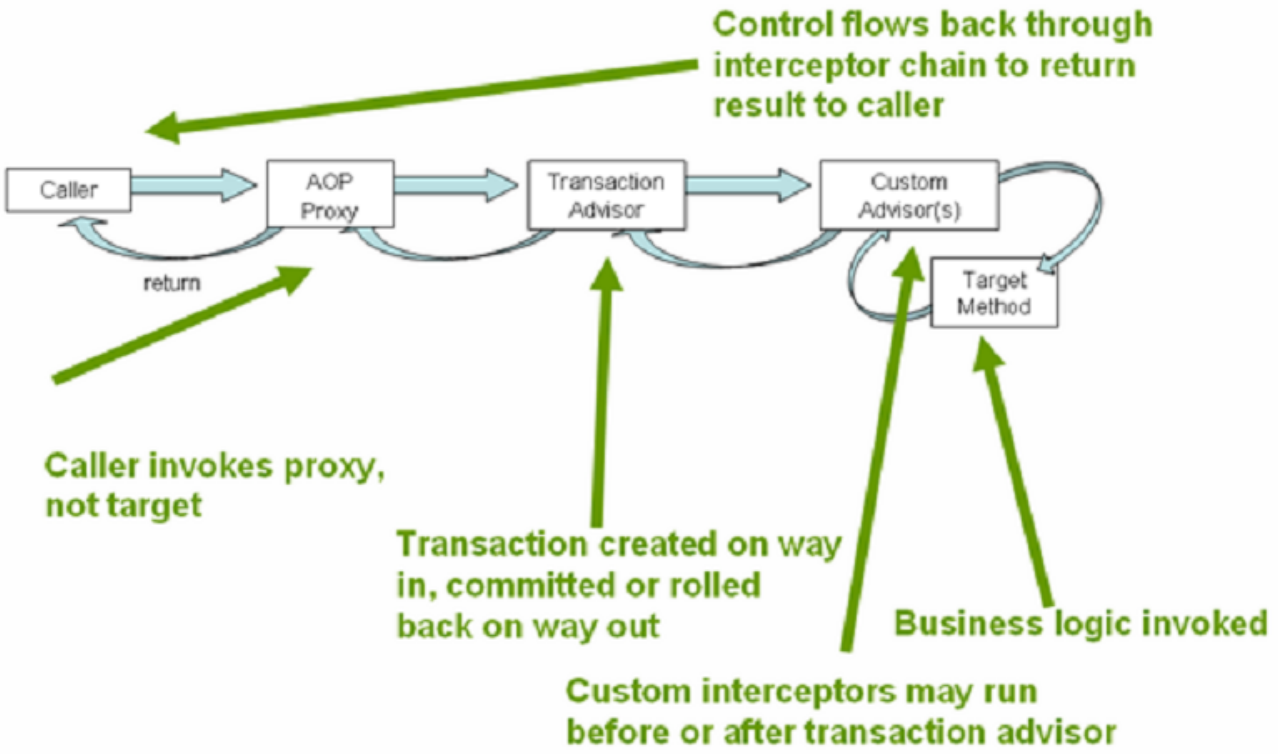
```
<beans>

    <!-- 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

Deklaratywne zarządzanie transakcjami Wykorzystanie AOP



źródło: Spring Reference

Sterowanie wyjątkami

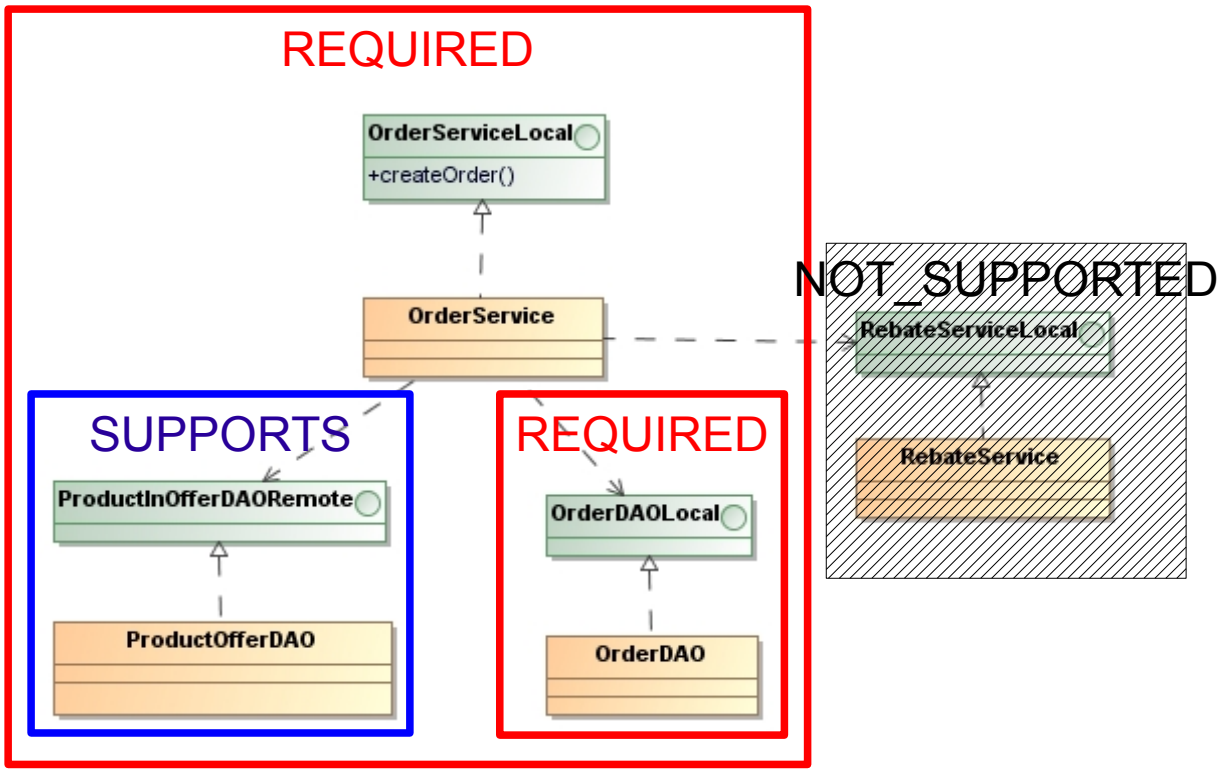
- weryfikowalne (Exception) – nie wycofują tx
- niewryfikowalne(RuntimeException, Error) - rollback

Transakcje deklaratywne adnotacja @Transactional

- Adnotacja klasy (obejmuje wszystkie metody)
- **UWAGA: Wołając metododę 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) {

    }
}
```

Propagacja transakcji



Propagacja transakcji

- 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 rolbackOnly per metoda
    - Zewnętrzna metoda może obsłużyć wyjątek wewnętrznej
- REQUIRES_NEW
  - Jeżeli transakcje 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

Propagacja REQUIRED

```
@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
  }
}

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

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

Propagacja REQUIRES_NEW

```
@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
  }
}

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

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

Propagacja NESTED

```
@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
  }
}

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

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

Propagacja transakcji

- 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

Propagacja transakcji

- 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

Wydajność transakcji

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

Wydajność transakcji
Strategie

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

Parametry transakcji (adnotacji @Transactional)

- **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

Samodzielne zarządzanie Tx Template

```
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 (SomeBusinessExeption ex) {
                        status.setRollbackOnly();
                }
          }
       });
}
```

Samodzielne zarządzanie TX Manager

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);
```

Pułapki

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

Dowiesz się:

- Jaka jest idea frameworka
- Poznasz przykładowe zastosowanie jako alternatywnego sposobu implementacji wyszukiwania

- Projekt Apache Foundation
  - Java i .NET
- Warstwa abstrakcji nad SQL
- Z wykorzystaniem XML
  - Mapowanie wyników kwerend na DTO
  - Mapowanie DTO na INSERT/UPDATE/DELETE

iBatis
Przykład mapowania

```
<sqlMap namespace="Person">

<select id="getPerson" resultClass="domain.Person">
SELECT
    ID as id,
    FIRST_NAME as firstName,
    LAST_NAME as lastName,
FROM PERSON
WHERE ID = #value#
</select>

<insert id="insertPerson" parameterClass="domain.Person">
INSERT INTO PERSON (ID, FIRST_NAME, LAST_NAME)
VALUES (#id#, #firstName#, #lastName#)
</INSERT>

</sqlMap>
```

iBatis
Przykład użycia

```
SqlMapClient sqlMap = MyAppSqlMapConfig.getSqlMapInstance();

Person person = (Person) sqlMap.queryForObject("getPerson", 1);
```

```
Person person = new Person();
sqlMap.insert("insertPerson", person);
```

iBatis
Mapowanie przez adnotacji i dynamiczne kwerendy

```
public interface BlogMapper {
  @Select("SELECT * FROM blog WHERE id = #{id}")
  Blog selectBlog(int id);
}

BlogMapper mapper = session.getMapper(BlogMapper.class);
Blog blog = mapper.selectBlog(101);
```

```
private String selectPersonLike(Person p){
  BEGIN(); // Clears ThreadLocal variable
  SELECT("P.ID, P.USERNAME, P.PASSWORD, P.FIRST_NAME, P.LAST_NAME");
  FROM("PERSON P");
  if (p.id != null) {
  WHERE("P.ID like #{id}");
  }
  if (p.firstName != null) {
  WHERE("P.FIRST_NAME like #{firstName}");
  }
  if (p.lastName != null) {
  WHERE("P.LAST_NAME like #{lastName}");
  }
  ORDER_BY("P.LAST_NAME");
  return SQL();
}
```

```
<bean id="sqlMapClient"
    class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
    <property name="configLocation" value="WEB-INF/sqlmap-config.xml"/>
    <property name="dataSource" ref="dataSource"/>
</bean>
```

```
public class SqlMapAccountDao extends SqlMapClientDaoSupport
     implements AccountDao {

    public Account getAccount(String email) throws DataAccessException {
        return (Account) getSqlMapClientTemplate()
        .queryForObject("getAccountByEmail", email);
    }

    public void insertAccount(Account account) throws DataAccessException {
        getSqlMapClientTemplate().update("insertAccount", account);
    }
}
```

Spring i ORM
Wsparcie dla JPA (Hibernate)

- Zarządzanie kontekstem persystencji
  - wstrzykujemy EntityManager/Session
  - ten sam kontekst w każdej metodzie w obrębie transakcji
- Zarządzanie transakcjami
  - deklaratywne - @Transactional
  - imperatywne – Transaction Template

Spring i ORM
Konfiguracja JPA

```
ontext:property-placeholder location="classpath:jdbc.properties" />
cor
<!--transakcje sterowane adnotacjami-->
<tx:annotation-driven transaction-manager="transactionManager" />
<!--umożliwia używanie @PersistenceContext zamiast @Autowired-->
<bean class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor" />
<!--źródło danych (dodać pulę!)-->
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource"
    p:driverClassName="${jdbc.driverClassName}" p:url="${jdbc.url}" />
<!--transakcje na źródle-->
<bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager"
    p:entityManagerFactory-ref="entityManagerFactory" />
<!--zarządzanie fabryką EM na źródle-->
<bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"
    p:dataSource-ref="dataSource" p:jpaVendorAdapter-ref="jpaAdapter">
    <property name="loadTimeWeaver">
        <bean class="org.springframework.instrument.classloading.InstrumentationLoadTimeWeaver" />
    </property>
    <property name="persistenceUnitName" value="defaultPU" />
</bean>
<!--reużywalny szablon-->
<bean id="jdbcTemplate" class="org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate">
    <constructor-arg ref="dataSource" />
</bean>
<!--adapter pozwala ustawić parametry unikając hibernate-cfg.xml;
użycie wspólnego DS dla Hibernate i JDBC template-->
<bean id="jpaAdapter" class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter"
    p:database="${jpa.database}" p:showSql="${jpa.showSql}" />
```

Generic JPA DAO

```
public abstract class GenericJpaDao<T> {
  private Class<T> entityBeanType;
  @Autowired
  protected EntityManager entityManager;

  public GenericJpaDao() {
    this.entityBeanType = ((Class<T>) ((ParameterizedType) getClass()
        .getGenericSuperclass()).getActualTypeArguments()[0]);
  }

  public T findById(Serializable id) {
     return entityManager.find(getEntityBeanType(), id);
  }

  public List<T> findAll() {
      return entityManager
           .createQuery("FROM " + getEntityBeanType().getName() )
          .getResultList();
  }

  protected Class<T> getEntityBeanType() {
      return entityBeanType;
  }

  //...
}
```

Generic JPA DAO Implementacja

```
@Repository
public JpaUserDao extends GenericJpaDao<User> implements UserDao {
 //implementacja metod niegenerycznych,
 //niezaimplementowanych przez GenericJpaDao
}
```

Kontener może wstrzyknąć odpowiednie zasobu (EntityManager, Session, DataSource)
•podłączone do transakcji okalającej metodę @Transactional
•współdzielone w sesji perystencji (wiele wywołań metod w jednej transakcji działa na tym samym EM)

Spring dostarcza abstrakcji nad dostępem do danych – archetyp @Repository
•Przepakowanie specyficznych dla rozwiązania wyjątków na ogólne wyjątki dostępu do dancyh
•W przyszłości być może dodatkowe funkcjonalności