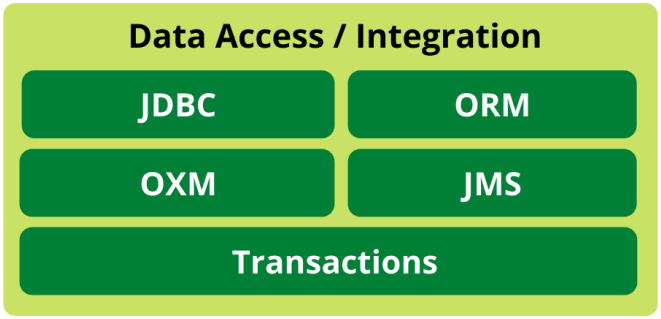
Spring Data access - Transakcje

Spis treści

Spring Transaction

Patrząc na ogólny zarys modułów Spring Framework, można zauważyć wydzielony moduł *Data Access / Integration*, który dotyczy interakcji pomiędzy warstwą danych a warstwą logiki biznesowej.



Obraz 1. Moduły Data Access / Integration

W tej części przyjrzymy się bliżej modułom *JDBC* i *Transactions*.

Wszystko, co jest potrzebne do zarządzania transakcjami w Springu, znajduje się w *Spring Transaction*.

Zależność Spring Transaction:

```
// https://mvnrepository.com/artifact/org.springframework/spring-tx
```

```
implementation "org.springframework:spring-tx:$springVersion"
```

Klasyczne zarządzanie transakcjami z JDBC

Pracę z transakcjami zaczniemy od podstaw, czyli od JDBC. Wynika to z tego, że wszystko, co Spring robi pod spodem z bazami danych, opiera się na podstawach JDBC. Także zaczynamy od podstaw, krok po kroku.

Normalnie, korzystając z samego JDBC, chcąc wywołać jakąś metodę w transakcji, trzeba by było użyć bloku `try-catch`, wyłączyć autocommit, wywołać metodę, która może składać się z wielu operacji na bazie i ostatecznie zacommitować transakcję po jej wykonaniu. Dodatkowo cała obsługa wyjątków spada na dewelopera i wymaga dodatkowego kodu jak poniżej:

```
public class ClassicJdbcExample {
    private final String url = "jdbc:postgresql://localhost:5432/zajavka";
    private final String user = "postgres";
    private final String password = "password";

    public void classicJdbcUseExample() throws SQLException {
        Connection connection = DriverManager.getConnection(url, user, password); ①
        try {
            connection.setAutoCommit(false); ②
            someMethod(); ③
            connection.commit(); ④
        } catch (SQLException e) {
            connection.rollback(); ⑤
        }
    }

    private void someMethod() {
        // ...
    }
}
```

① Utworzenie połączenia do bazy.

② Wyłączenie trybu *autocommit*, bo inaczej każde zapytanie będzie wywoływane w oddzielnej transakcji. Nawet jeżeli nazwa jest myląca to to jest właśnie sposób na rozpoczęcie transakcji w Javie.

③ Wywołanie metody, która wymaga oddzielnej transakcji.

④ *commit* po zmianach wykonanych przez metodę `someMethod()`.

⑤ Obsłużenie błędu, poprzez *rollback* zmian wprowadzonych przez metodę `someMethod()`.

My jednak nie chcemy pisać tyle powtarzalnego kodu i dobrze, bo Spring może zająć się tym w naszym imieniu ☺ i to na kilka różnych sposobów.

Ważne jest również to, że w kolejnych materiałach będziemy dodawali kolejne poziomy abstrakcji przy pracy z bazami danych, żeby finalnie upraszczać sobie kod, który będzie pisany. Jednakże kolejne biblioteki i technologie finalnie i tak zarządzają transakcjami w sposób analogiczny do pokazanego powyżej. Można powiedzieć, że nawet jeżeli w kolejnych przykładach będziemy używali Springa do zarządzania transakcjami, to pod spodem będzie on w uproszczeniu robił to, co zostało pokazane w

> powyższym przykładzie.

Programowalne zarządzanie transakcją

Przejdźmy do transakcji w Springu. Mniej popularną opcją jest programowalne zarządzanie transakcją za pomocą *TransactionTemplate*. Wywołując metodę TransactionTemplate#`execute()` trzeba przygotować implementację interfejsu `TransactionCallback` z kodem wymagającym transakcji. Jest to mało wygodne, bo przecież zależy nam na uproszczeniu, a jak sam/sama zaraz zobaczysz, kodu nadal jest całkiem sporo.

Konfiguracja DataSourceConfiguration

```
@Configuration
@ComponentScan(basePackages = "pl.zajavka")
public class DataSourceConfiguration {

    @Bean
    public SimpleDriverDataSource databaseDataSource() {
        SimpleDriverDataSource dataSource = new SimpleDriverDataSource();
        dataSource.setDriver(new Driver());
        dataSource.setUrl("jdbc:postgresql://localhost:5432/zajavka");
        dataSource.setUsername("postgres");
        dataSource.setPassword("postgres");
        return dataSource;
    }

    @Bean
    public PlatformTransactionManager transactionManager() {
        return new DataSourceTransactionManager(databaseDataSource());
    }

    @Bean
    public TransactionTemplate transactionTemplate(
        PlatformTransactionManager transactionManager
    ) {
        return new TransactionTemplate(transactionManager);
    }

}
```

Przykład z użyciem **TransactionCallback***:*

```
@Service
@AllArgsConstructor
public class TransactionTemplateExample {

    private TransactionTemplate transactionTemplate; ①

    public void transactionTemplateUseExample() {

        transactionTemplate.execute(new TransactionCallback<Object>() { ②
            @Override
            public Object doInTransaction(TransactionStatus status) { ③
                return someMethod();
            }
        });
```

```
    }

    private Object someMethod() { ④
        return null;
    }
}
```

① Wstrzyknięcie obiektu `TransactionTemplate`.

② Wywołanie metody `someMethod()` w transakcji z użyciem TransactionTemplate#`execute()`.

③ Implementacja metody `doInTransaction()` z interfejsu funkcyjnego `TransactionCallback` jako klasa anonimowa.

④ Przykładowa metoda, która wymaga oddzielnej transakcji.

Całe szczęście przedstawiony przykład można trochę uprościć, stosując wyrażenie lambda tak, jak poniżej:

```
transactionTemplate.execute(status -> someMethod());
```

Zwróć uwagę, że w przypadku gdy korzystamy ze Springa, nie musimy pisać bloku `try-catch`, nie musimy łapać wyjątków i jednocześnie możemy pracować na Spring Beanach - `TransactionTemplate`. Jest to usprawnienie w stosunku do tego gdybyśmy chcieli zarządzać transakcjami z poziomu JDBC, natomiast nie jest to jeszcze ostateczny poziom "uproszczenia", do którego przejdziemy później.

Przypadek z metodą void

W przypadku metod, które nie zwracają żadnej wartości, można zastosować klasę `TransactionCallbackWithoutResult`, która implementuje interfejs `TransactionCallback`.

Przykład z użyciem **TransactionCallbackWithoutResult***:*

```
@Service
@AllArgsConstructor
public class TransactionTemplateExample {

    private TransactionTemplate transactionTemplate; ①

    public void transactionTemplateUseExample() {
        transactionTemplate.execute(new TransactionCallbackWithoutResult() { ②
            @Override
            protected void doInTransactionWithoutResult(TransactionStatus status) { ③
                someVoidMethod();
            }
        });
    }

    private void someVoidMethod() {} ④
}
```

① Wstrzyknięcie obiektu `TransactionTemplate`.

② Wywołanie metody `someMethod()` w transakcji z użyciem TransactionTemplate#`execute()`.

③ Implementacja metody TransactionCallbackWithoutResult#`doInTransactionWithoutResult()` jako klasa anonimowa.

④ Przykładowa metoda, która wymaga oddzielnej transakcji.

Manualne wycofanie transakcji

Jeżeli chcielibyśmy dodatkowo zaprogramować wycofanie transakcji, czyli jej *rollback*, w jakimś specyficznym przypadku, trzeba by było skorzystać z metody TransactionStatus#`setRollbackOnly()`, tak jak poniżej:

Przykład rollback z użyciem **TransactionCallback***:*

```
@Slf4j
@Service
@AllArgsConstructor
public class TransactionTemplateExample {

    private TransactionTemplate transactionTemplate;

    public void transactionTemplateUseExample() {
        transactionTemplate.execute(new TransactionCallback<Object>() {
            @Override
            public Object doInTransaction(TransactionStatus status) {
                Object result = someMethod();
                if (true) { ①
                    status.setRollbackOnly();
                    log.error("Transaction rollback");
                }
                return result;
            }
        });
    }

    private Object someMethod() {
        return null;
    }
}
```

① Celowo wycofujemy transakcję, pomimo że zapytania na bazie danych zostały wykonane pomyślnie.

Przedstawiony przykład można uprościć, stosując wyrażenie lambda tak, jak poniżej:

```
transactionTemplate.execute(status -> {
    Object result = someMethod();
    if (true) {
        status.setRollbackOnly();
        log.error("Transaction rollback");
    }
    return result;
});
```

Przykład rollback z użyciem **TransactionCallbackWithoutResult***:*

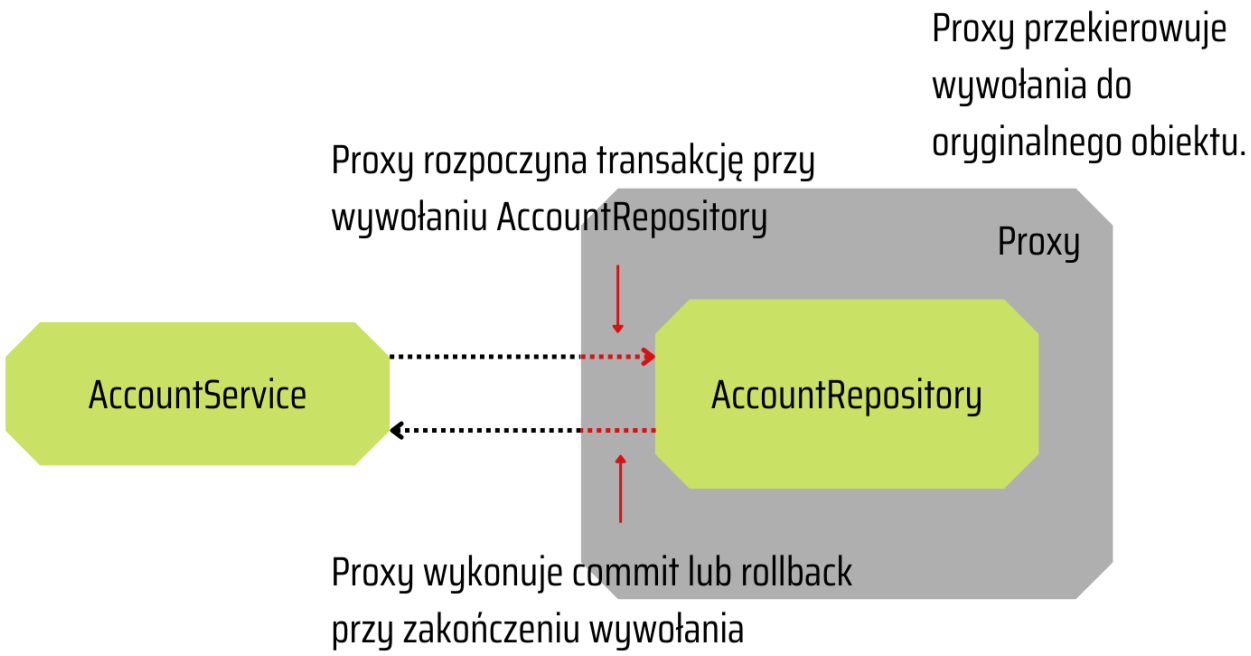
```
@Service
@AllArgsConstructor
```

```
public class TransactionTemplateExample {

    private TransactionTemplate transactionTemplate;

    public void transactionTemplateUseExample() {
        transactionTemplate.execute(new TransactionCallbackWithoutResult() {
            @Override
            protected void doInTransactionWithoutResult(TransactionStatus status) {
                someVoidMethod();
                if (true) {
                    status.setRollbackOnly();
                    log.error("Transaction rollback");
                }
            }
        });
    }

    private void someVoidMethod() {}
}
```

Deklaratywne zarządzanie transakcją

Zdecydowanie bardziej popularnym sposobem jest stosowanie adnotacji *@Transactional*. Można ją stosować nad definicją klasy, wtedy adnotacja dotyczy wszystkich metod publicznych w klasie, lub nad konkretnymi metodami publicznymi. Jej działanie opiera się o wzorzec **proxy**. Tłumacząc to w uproszczeniu, Spring tworzy obiekt **proxy** dla wszystkich klas oznaczonych adnotacją `@Transactional`, zarówno w klasie, jak i w dowolnej metodzie. Stosowanie wzorca **proxy** umożliwia Springowi wstrzyknięcie logiki transakcyjnej przed i po uruchomionej metodzie, głównie w celu rozpoczęcia i zatwierdzenia transakcji.

Czyli Spring stworzy za nas obiekt **proxy**, który dołoży zachowania transakcyjne "wokół" wywołań naszych metod. Inaczej mówiąc, stworzony przez Spring obiekt proxy przed wywołaniem naszej metody rozpocznie transakcję i zakończy ją po zakończeniu działania naszego kodu. Ważne dla nas są ułatwienia, jakie niesie ze sobą stosowanie tego podejścia. Poniższy schemat przedstawia opisane zachowanie:



Obraz 2. Deklaratywne zarządzanie transakcją

Czyli Spring otworzy za nas transakcję, oddeleguje dalsze wywołanie do faktycznej metody, która ma wykonać jakąś pracę i zamknie transakcję. Z perspektywy `AccountService` nadal rozmawiamy z `AccountRepository`, robimy to jedynie przez obiekt **proxy**.

Przypomnij sobie, że gdy po raz pierwszy rozmawialiśmy o proxy, wzorzec ten mógł być zaimplementowany w ten sposób, że jednocześnie implementowaliśmy / dziedziczyliśmy konkretny interfejs/klasę i jednocześnie odwoływaliśmy się z poziomu **proxy** do tego interfejsu / klasy w formie pola w obiekcie **proxy**.

Dlaczego takie zarządzanie nazywane jest deklaratywnym? Jest tak, ponieważ stosując adnotację `@Transactional`, określamy **co** program powinien zrobić, ale nie definiujemy **jak**. Za to, **jak** osiągnąć zadeklarowany cel odpowiadają mechanizmy Springowe.

Adnotacja *@Transactional*

Przykład konfiguracji z użyciem `@Transactional`:

```
@Service
@AllArgsConstructor
public class PersonService {
    private PersonRepository personRepository;

    @Transactional
    public Long createPerson(Person person) {
        // ... kod odpowiedzialny za przygotowanie person
        return personRepository.create(person);
    }
}
```

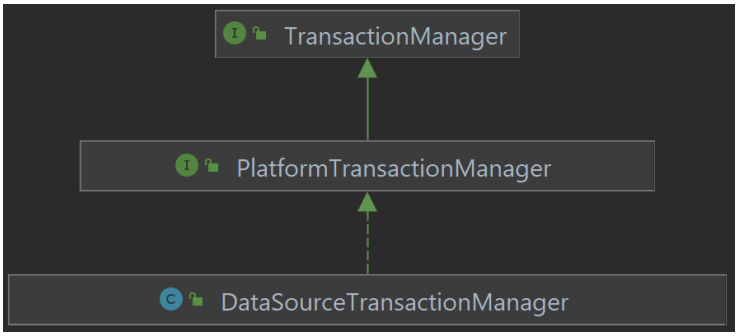
Gdy korzystamy z deklaratywnego zarządzania transakcjami, to w bardzo podstawowym przypadku jedyne co musimy zrobić to napisać adnotację `@Transactional`. Przy tym podejściu, Spring stworzy obiekt **proxy**, który będzie "opakowywał" nasze wywołanie i w tym **proxy** Spring doda kod rozpoczynający i

commitujący lub rollbakujący transakcję. Jedyne, o co programista musi się wtedy martwić, to gdzie taka transakcja powinna być rozpoczęta.

Powyżej adnotacja `@Transactional` jest zdefiniowana na poziomie metody z serwisu. Oznacza to, że cała logika, która jest wykonywana w obrębie tego serwisu w metodzie `createPerson()`, będzie traktowana jako jedna transakcja bazodanowa. Możemy również dodawać adnotację `@Transactional` nad metodami z repozytorium `PersonRepository`.

Jak włączyć adnotacje *@Transactional* w projekcie

Żeby włączyć możliwość korzystania z adnotacji `@Transactional`, trzeba rozszerzyć klasę konfiguracyjną o adnotację *@EnableTransactionManagement* i uzupełnić klasę o bean menadżera transakcji *PlatformTransactionManager*, który jest głównym interfejsem Springa do pracy z transakcjami. Jego implementacją stworzoną do użytku z pojedynczym JDBC `DataSource` jest *DataSourceTransactionManager*.



Obraz 3. Hierarchia TransactionManager

Klasa konfiguracyjna przygotowana do używania `@Transactional`:

```
@Configuration ①
@EnableTransactionManagement  ②
public class AppConfig {

    @Bean
    public SimpleDriverDataSource dataSource() { ③
        SimpleDriverDataSource dataSource = new SimpleDriverDataSource();
        dataSource.setDriver(new Driver());
        dataSource.setUrl("jdbc:postgresql://localhost:5432/zajavka");
        dataSource.setUsername("postgres");
        dataSource.setPassword("password");
        return dataSource;
    }

    @Bean
    public PlatformTransactionManager txManager() { ④
        return new DataSourceTransactionManager(dataSource());
    }

}
```

① Adnotacja klasy konfiguracyjnej.

② Adnotacja włączająca możliwość zarządzania transakcjami oparte o adnotację `@Transactional`.

③ Konfiguracja `DataSource`.

④ Konfiguracja `PlatformTransactionManager`.

A po co musimy definiować `PlatformTransactionManager`? Jeżeli zajrzymy do kodu źródłowego klasy `DataSourceTransactionManager`, znajdziemy tam kod analogiczny do tego poniżej. Poniższy przykład jest kopiuj-wklej z klasy `DataSourceTransactionManager` i spora część kodu skopiowanego źródłowego została uproszczona, żeby pokazać istotę istnienia `PlatformTransactionManager`:

```
public class DataSourceTransactionManager implements PlatformTransactionManager {

    // ... metody

    @Override
    protected void doBegin(Object transaction, TransactionDefinition definition) {
        Connection newCon = obtainDataSource().getConnection();
        // ...
        con.setAutoCommit(false);
        // widzieliśmy już to wcześniej przy JDBC
    }

    // ... metody

    @Override
    protected void doCommit(DefaultTransactionStatus status) {
        // ...
        Connection connection = status.getTransaction().getConnectionHolder().getConnection();
        try {
            con.commit();
        } catch (SQLException ex) {
            throw new TransactionSystemException("Could not commit JDBC transaction", ex);
        }
    }

    // ... metody
}
```

`TransactionManager` jest definiowany po to, żeby "ktoś" faktycznie zarządzał naszymi transakcjami. Zatem podsumowując, Spring wykryje, że dana metoda jest oznaczona adnotacją `@Transactional` i stworzy obiekt **proxy**. Obiekt **proxy** poprosi zdefiniowanego `TransactionManagera` o otwarcie i zamknięcie transakcji. Natomiast `TransactionManager` zrobi za nas to co napisaliśmy podczas pokazywania prostego przykładu zarządzania transakcjami z poziomu JDBC.

Tematy które są teraz poruszane powoli zahaczają o sens istnienia frameworków. Zauważ, że to co robimy, to zaczynamy od "niskopoziomowego" podejścia do jakiegoś problemu i pokazujemy później jak można zrobić to prościej przy wykorzystaniu Springa. I właśnie o to chodziło gdy mówiliśmy sobie o upraszczaniu konfiguracji i niepisaniu schematycznego kodu. Tutaj mamy pokazany przypadek, gdzie Spring będzie za nas zarządzał transakcjami, wystarczy jedna adnotacja. Inaczej musielibyśmy robić to wszystko ręcznie.

Transakcje fizyczne vs transakcje logiczne

Przejdźmy teraz do wyjaśnienia jaka jest różnica pomiędzy transakcjami fizycznymi a logicznymi. Spójrz na poniższy przykład:

Klasa AccountService

```
@Service
@AllArgsConstructor
public class AccountService {

    private InvoiceService invoiceService;

    @Transactional
    public void saveBill() {
        invoiceService.saveBill();
        // ... pozostały kod
    }
}
```

Klasa InvoiceService

```
@Service
public class InvoiceService {

    @Transactional
    public void saveBill() {
        // ... logika
    }
}
```

Klasa `AccountService` ma zdefiniowaną metodę `saveBill()`, która jest oznaczona adnotacją `@Transactional`. Metoda ta wywołuje inną metodę `saveBill()` z klasy `InvoiceService`, która również jest oznaczona adnotacją `@Transactional`.

W rozumieniu transakcji bazodanowej, powinna to być cały czas jedna i ta sama transakcja. Inaczej mówiąc powinna to być jedna **fizyczna transakcja**.

Z perspektywy Springa natomiast mamy dwie **logiczne transakcje**, jedną w `AccountService` i drugą w `InvoiceService`. Spring musi sam wywnioskować, że obie metody oznaczone adnotacją `@Transactional` powinny finalnie być jedną **fizyczną transakcją** w bazie danych. Podsumujmy zatem terminy:

- **fizyczna transakcja** - transakcja, która będzie miała miejsce w bazie danych,
- **logiczna transakcja** - transakcja w Spring, która może być potencjalnie zagnieżdżona, tak jak w przypadku powyżej.

Prowadzi nas to do pojęcia **propagacji transakcji**.

Propagacja transakcji

Istnieje kilka sposobów, w jaki Spring może obsługiwać transakcje. Nazywa się to **propagacją**, bo określa, co ma się zadziać z transakcją podczas wywoływania kolejnej metody. Opcji jest kilka, bo przykładowo można wymusić, że dana metoda będzie zawsze wywoływana w oddzielnej / nowej / własnej transakcji. Można też zadecydować odwrotnie, że metoda ma zostać wykonana poza jakąkolwiek transakcją. Wszystkie dostępne opcje propagacji zabrane są w specjalnym enumie *Propagation*.

Poniżej pokrótce omówimy wybrane rodzaje propagacji transakcji. Znajdziesz je w enumie org.springframework.transaction.annotation.`Propagation`:

- **REQUIRED** (default) - Spring używa istniejącej transakcji, a jeżeli jej nie ma, tworzy nową. Jest to domyślny rodzaj propagacji `@Transactional`.
- **SUPPORTS** - Spring używa istniejącej transakcji, a jeżeli jej nie ma, działa bez.
- **MANDATORY** - Spring używa istniejącej transakcji, a jeżeli jej nie ma, wyrzuca wyjątek.
- **REQUIRES_NEW** - Spring zawsze tworzy nową transakcję.
- **NOT_SUPPORTED** - Spring działa bez transakcji, a jeżeli transakcja już istnieje, to ją wstrzymuje.

Jeżeli wrócimy teraz do poprzedniego przykładu, co by się stało gdybyśmy zmienili kod na taki?

Klasa AccountService

```
@Service
@AllArgsConstructor
public class AccountService {

    private InvoiceService invoiceService;

    @Transactional
    public void saveBill() {
        invoiceService.saveBill();
        // ... pozostały kod
    }
}
```

Klasa InvoiceService

```
@Service
public class InvoiceService {

    @Transactional(propagation = Propagation.REQUIRES_NEW)
    public void saveBill() {
        // ... logika
    }
}
```

Powyższa zamiana mówi Springowi, że metoda InvoiceService#`createBill()` musi być wykonana w swojej własnej transakcji, niezależnie od jakiejkolwiek już istniejącej. Parametr ten mówi Springowi, aby zawsze tworzył nową transakcję fizyczną. Taka transakcja będzie wtedy rozumiana jako nowa transakcja, która będzie miała swoje własne ustawienia. Tutaj należy pamiętać, że nowa transakcja wymaga nowego połączenia z bazą danych. W takim przypadku "zewnętrzna" transakcja będzie miała swoje połączenie do bazy danych, podczas gdy `REQUIRES_NEW` utworzy nową "wewnętrzną" transakcję fizyczną i powiąże z nią nowe połączenie z bazą danych. Przebieg transakcji będzie zatem wyglądał w ten sposób:

- Uruchamiamy zewnętrzną transakcję,
- Uruchamiamy wewnętrzną transakcję i zawieszamy zewnętrzną transakcję, ale jej połączenie z bazą danych pozostaje otwarte,

- Po zacommitowaniu wewnętrznej transakcji, zewnętrzna transakcja jest wznawiana i albo zakończy się przez *commit* albo *rollback*.

Podejście to jest stosowane wtedy gdy potrzebujemy, żeby zewnętrzna transakcja fizyczna została zacommitowana, nawet jeżeli wewnętrzna transakcja skończy się rollbackiem. Jeżeli zewnętrzna transakcja skończy się rollbackiem, to nie ma to wpływu na wewnętrzną transakcję, która mogła zostać zacommitowana.

Gdybyśmy się nad tym głębiej zastanowili to można dojść do wniosku, że większość trybów propagacji tak naprawdę nie ma nic wspólnego z bazą danych ani JDBC. Bardziej odnosi się to do tego jak ułożymy swój program i jak / kiedy / gdzie Spring oczekuje, że będą tam transakcje. Jeżeli spojrzymy na poniższy przykład:

```
public class AccountService {

     @Transactional(propagation = Propagation.MANDATORY)
     public void someMethod() {
        // call do bazy danych
     }

}
```

W powyższym przypadku Spring będzie oczekiwał, że transakcja będzie otwarta za każdym razem, gdy wywołamy `someMethod()`. Inaczej mówiąc, wywołanie `someMethod()` musi się odbywać zawsze po rozpoczęciu już wcześniej transakcji. Spring nie otworzy jej na etapie `someMethod()` samodzielnie, jeżeli transakcja nie będzie już otwarta wcześniej, to na tym etapie Spring wyrzuci wyjątek.

Poziomy izolacji

Transakcje w Spring pozwalają także na ustawienie wymaganego poziomu izolacji. Rodzaje izolacji transakcji znajdziesz w enumie org.springframework.transaction.annotation.`Isolation`:

- **DEFAULT** - Rodzaj transakcji jest brany z konfiguracji używanej bazy,
- **READ_UNCOMMITTED** - cytując dokumentację: "A constant indicating that dirty reads, non-repeatable reads and phantom reads can occur".
- **READ_COMMITTED** - cytując dokumentację: "A constant indicating that dirty reads are prevented; non-repeatable reads and phantom reads can occur."
- **REPEATABLE_READ** - cytując dokumentację: "A constant indicating that dirty reads and non-repeatable reads are prevented; phantom reads can occur."
- **SERIALIZABLE** - cytując dokumentację: "A constant indicating that dirty reads, non-repeatable reads and phantom reads are prevented."

Wszystkie wartości liczbowe użyte w enumach `Propagation` i `Isolation` pochodzą z interfejsu *org.springframework.transaction.TransactionDefinition*

Należy jednak pamiętać, że jeśli chodzi o przełączanie poziomów izolacji transakcji z poziomu Springa, należy sprawdzić jak w takim przypadku zachowa się sterownik JDBC oraz baza danych. Może wystąpić taka sytuacja, że dany poziom nie będzie

> wspierany przez sterownik albo przez bazę danych.

Wiedząc już jakie parametry możemy ustawić podczas tworzenia transakcji, przejdźmy do tego jak wspomniane wartości można ustawić w konkretnej transakcji:

Przykład konfiguracji z użyciem `TransactionTemplate`*:*

```
@Service
@AllArgsConstructor
public class TransactionTemplateExample {

    private TransactionTemplate transactionTemplate;

    public void transactionTemplateUseExample() {
        transactionTemplate.setIsolationLevel(Isolation.READ_UNCOMMITTED.value());
        transactionTemplate.setPropagationBehavior(Propagation.REQUIRED.value());

        // ... pozostałe operacje
    }
}
```

Przykład konfiguracji z użyciem `@Transactional`*:*

```
@Transactional(isolation = Isolation.READ_UNCOMMITTED, propagation = Propagation.REQUIRED)
```

Najbezpieczniej jest stosować ustawienia domyślne. Czyli w przypadku propagacji jest to `Propagation.REQUIRED`, a w przypadku poziomu izolacji `Isolation.DEFAULT`. Każde własnoręczne ustawianie tych parametrów powinno być przetestowane w zestawieniu z używaną bazą danych i zastosowanym sterownikiem. W takim przypadku dobrze jest też przeczytać dokumentację, bo ona zwraca nam uwagę w których miejscach powinniśmy zachować "szczególną ostrożność".

Błędy popełniane przy wykorzystaniu @Transactional

Wywołanie z tej samej klasy

Z racji natury działania adnotacji `@Transactional` trzeba pamiętać o jednym dosyć ważnym szczególe. Spring utworzy transakcję tylko gdy wywołamy metodę oznaczoną adnotacją `@Transactional` **z zewnątrz**. Inaczej mówiąc, ten przykład nie zadziała:

```
@Service
public class AccountService {

    @Transactional
    public void saveBill() {
        createInvoice(); ①
    }

    @Transactional(propagation = Propagation.REQUIRES_NEW) ②
    public void createInvoice() {
        // ...
    }
```

```
}
```

① Wywołujemy metodę oznaczoną adnotacją `@Transactional`, która znajduje się w tej samej klasie.

② Chcemy, żeby została utworzona nowa transakcja.

Wynika to z tego, że do obsługi transakcji Spring tworzy obiekt **proxy**. Spring jest w stanie "interweniować" w nasze wywołanie i "wstawić" tam obiekt proxy tylko i wyłącznie wtedy gdy wywołanie takie następuje pomiędzy metodami w różnych klasach. Jeżeli, tak jak w powyższym przypadku, wywołując metodę `createInvoice()` chcemy stworzyć nową transakcję fizyczną, to nie zostanie dodany obiekt **proxy** i nie zostanie utworzona nowa transakcja fizyczna. Cały czas będziemy działać w obrębie jednej transakcji. Rozwiązaniem jest przeniesienie metody `createInvoice()` do innej klasy.

O tym samym zachowaniu należy pamiętać, jeżeli kiedykolwiek będziemy korzystali z funkcjonalności Springa, które opierają się o obiekt **proxy**.

@Transactional vs checked exception

Chciałem to specjalnie zapisać w oddzielnym podpunkcie. Domyślnie, *rollback* transakcji wystąpi tylko jeżeli w obrębie działania transakcji wystąpi **RuntimeException**. Wyrzucenie wyjątku **checked** nie spowoduje wycofania transakcji.

Gdzie najlepiej używać @Transactional

Pytanie z nagłówka jest w sumie ciekawe. Stanie się to o wiele jaśniejsze gdy przejdziemy już do projektu, natomiast spróbujmy zastanowić się nad tym na razie teoretycznie. Transakcja ma zapewnić, że albo poprawnie wykonają się wszystkie zapytania, albo żadne. Nie ma zatem większego sensu umieszczać tej adnotacji nad "metodą realizującą zapytanie". Jeżeli nie będziemy wykorzystywali tej adnotacji, to każde zapytanie będzie realizowane w oddzielnej transakcji.

Wydaje się zatem sensowne stosowanie tej adnotacji "nad jakąś logiką", która w swoim obrębie skupia wykonanie kilku zapytań do bazy danych, które albo mają udać się wszystkie albo żadne. Wniosek z tego płynie taki, że często najwięcej sensu będzie miało stosowanie tej adnotacji w klasach, które są serwisami, czyli tych oznaczonych przez `@Service`. Zobaczysz to w praktyce gdy przejdziemy już do projektu.

A co z testami? Czy ma sens stosowanie adnotacji `@Transactional` nad metodami będącymi np. testami integracyjnymi, które wykonywane są na bazie danych?

Mówimy tutaj o testach opartych o Spring context.

To jest ciężkie pytanie, a odpowiedź na nie to: *To zależy*. W przypadku gdy, oznaczymy test adnotacją `@Transactional`, podczas uruchamiania kontekstu aplikacji Spring, automatycznie konfiguruje się `TransactionalTestExecutionListener`. Klasa ta zapewnia obsługę testów, które są oznaczone przez `@Transactional`, czy to na poziomie klasy czy metody. Ten "listener" zauważa adnotacje `@Transactional` i tworzy nowe transakcje, które domyślnie są wycofywane po zakończeniu testu. Domyślne zachowanie `@Transactional` na teście to **rollback**. Oczywiście są sposoby na zmianę tej konfiguracji, ale nie będziemy się w to zagłębiać.

Należy jednak pamiętać, że mówimy tutaj o transakcji rozpoczętej na poziomie testu. Jeżeli w kodzie pojawi się kolejna adnotacja z ustawioną propagacją `REQUIRES_NEW`, to ta "zagnieżdżona" transakcja zostanie zacommitowana, a "testowa" transakcja wycofana.

No dobrze, ale jakie to ma znaczenie? Commitowanie transakcji na koniec wykonania testu zmniejszy ryzyko powstania nieprzewidywalnych / dziwnych i trudnych do znalezienia błędów. Przypomnij sobie chociażby, że jeżeli transakcja nie będzie zacommitowana to zmiany przez nią wprowadzane mogą nie być widoczne dla innych transakcji (kwestia ustawionego poziomu izolacji). Dlatego najlepiej jest commitować transakcje na koniec wykonania testu - zbliżamy się wtedy maksymalnie do "rzeczywistości" w testowanym przez nas przypadku.

Określenie konfiguracji rollback

Dodatkowo adnotacja `@Transactional` pozwala na zdefiniowanie warunków, kiedy transakcja ma zostać wycofana.

- **rollbackFor** - tablica klas wyjątków, których wystąpienie ma spowodować *rollback* transakcji.

```
@Transactional(rollbackFor = { RuntimeException.class })
```

- **rollbackForClassname** - tablica nazw klas wyjątków, których wystąpienie ma spowodować *rollback* transakcji.

```
@Transactional(rollbackForClassName = {"NullPointerException"})
```

- **noRollbackFor** - tablica klas wyjątków, których wystąpienie ma `nie` spowodować *rollback* transakcji.

```
@Transactional(noRollbackFor = { RuntimeException.class })
```

- **noRollbackForClassname** - tablica nazw klas wyjątków, których wystąpienie ma `nie` spowodować *rollback* transakcji.

```
@Transactional(noRollbackForClassname = {"NullPointerException"})
```