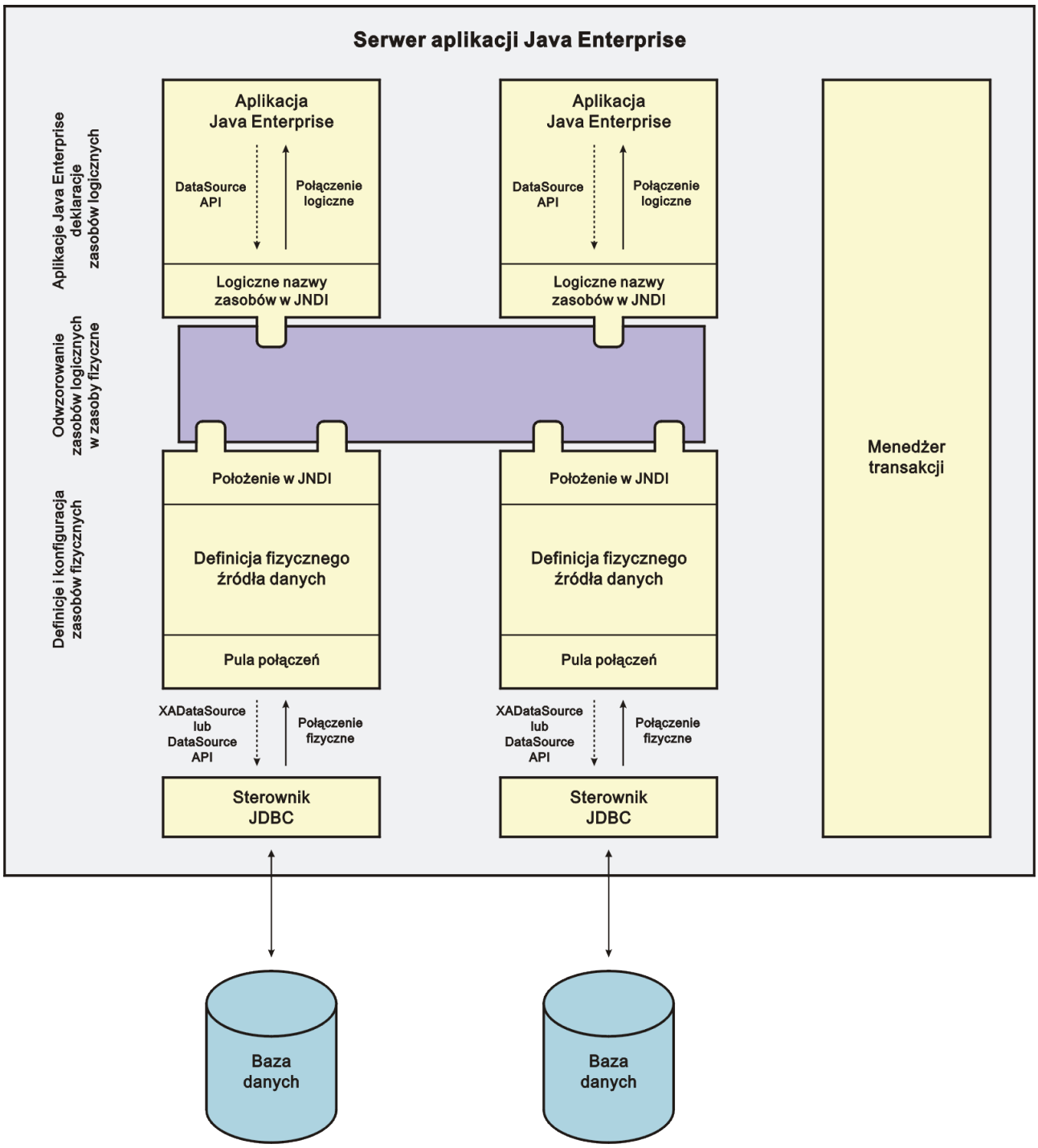
Transakcje w systemach Java Enterprise

Korzystanie z baz danych w serwerach aplikacji

Dostęp do baz danych w środowisku serwera aplikacji

Korzystanie z baz danych w aplikacjach Java Enterprise różni się nieco od korzystanie z nich w aplikacjach konsolowych, czy desktopowych. Różnice te polegają przede wszystkim na odmiennym sposobie uzyskiwania połączenia do baz danych. W standardowych aplikacjach pisanych w środowisku Java Standard Edition korzystamy bezpośrednio ze sterownika JDBC, z którego uzyskujemy fizyczne połączenie do bazy danych. W aplikacjach Java Enterprise dostęp do bazy danych jest o wiele bardziej złożony. Przyjrzyjmy mu się dokładniej.

Na rysunku poniżej zostało przedstawione środowisko jakie serwer aplikacji tworzy dla aplikacji korzystających z baz danych.



Rysunek 1 Dostęp do baz danych w środowisku tworzonym przez serwer aplikacji

Środowisko to składa się z następujących elementów:

- Deskryptora aplikacji, w którym twórca aplikacji definiuje logiczne zasoby, z których będzie korzystał (np. bazy danych). Dla aplikacji internetowych zasoby te są definiowane za pomocą elementów `<resource-ref>` w pliku `web.xml`. Aplikacje mogą się odwoływać jedynie do zasobów, które zostały zdefiniowane w deskryptorze. Z punktu widzenia aplikacji zasoby te są widoczne w drzewie JNDI pod zdefiniowanymi w deskryptorze nazwami.
- Definicji fizycznych zasobów (np. baz danych) wraz z ich parametrami konfiguracyjnymi takimi jak: adres serwera bazy danych, nazwa bazy danych, użytkownik, hasła. Zasoby te mogą być dostępne dla wszystkich aplikacji zainstalowanych na serwerze. Każdy zasób jest widoczny pod odpowiednią nazwą w drzewie JNDI. Ze względów wydajnościowych obiekty reprezentujące zasoby fizyczne utrzymują pulę otwartych połączeń do bazy danych. Sposób konfiguracji zasobu fizycznego jest specyficzny dla danego serwera aplikacji.
- Odwzorowania zdefiniowanych w aplikacjach zasobów logicznych na zasoby fizyczne zdefiniowane w konfiguracji serwera. Odwzorowanie to jest realizowane za pomocą deskryptorów specyficznych dla danego serwera aplikacji i jest ustanawiane w momencie instalacji aplikacji na serwerze.
- Menedżera transakcji, który w imieniu aplikacji potrafi zarządzać transakcjami rozproszonymi, w których może uczestniczyć wiele zasobów transakcyjnych (np. baz danych).

Taki sposób organizacji dostępu do bazy danych ma następujące konsekwencje:

- Aplikacja nie zawiera konfiguracji dostępu do bazy danych i nie zarządza połączeniem do bazy danych.
- Aplikacja operuje na połączeniu logicznym udostępnianym przez serwer aplikacji.
- Konfiguracja parametrów fizycznego połączenia do bazy danych znajduje się na poziomie serwera aplikacyjnego.
- Wiele aplikacji może korzystać z tego samego fizycznego zasobu. Wiąże się to również ze współdzieleniem puli połączeń.

Wszystko to powoduje, że w serwerze aplikacyjnym mamy bardzo elastyczną strukturę dostępu do bazy danych, niestety okupioną dużą złożonością jej konfiguracji.

Przyjrzyjmy się jak taka konfiguracja może wyglądać w praktyce na przykładzie prostej aplikacji internetowej, która chce skorzystać z bazy danych.

Tworząc naszą aplikację musimy zadeklarować w jej deskryptorze (plik `web.xml`), że będziemy korzystać z bazy danych i że obiekt umożliwiający dostęp do bazy danych (`javax.sql.DataSource`) powinien się znajdować w drzewie JNDI pod określoną nazwą (tutaj `jdbc/SampleDS`):

```
<resource-ref>
    <description>Sample DS</description>
    <res-ref-name>jdbc/SampleDS</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
</resource-ref>
```

W serwerze aplikacji musimy zdefiniować zasób fizyczny wraz konfiguracją parametrów połączenia do bazy danych oraz konfiguracją parametrów puli połączeń (np. wielkość puli). W każdym serwerze konfigurowanie zasobu fizycznego wygląda nieco inaczej. Dla serwera JBoss w naszym przykładzie będzie to plik sample-ds.xml umieszczony w katalogu `$JBOSS_HOME/server/$PROFILE_NAME/deploy`:

```
<?xml version="1.0" encoding="UTF-8"?>
<datasources>
    <local-tx-datasource>
        <jndi-name>jdbc/PhysicalSampleDS</jndi-name>
        <connection-url>jdbc:postgresql://localhost:5432/sampledb</connection-url>
        <driver-class>org.postgresql.Driver</driver-class>
        <user-name>testuser</user-name>
        <password>123456</password>
        <new-connection-sql>select 1</new-connection-sql>
        <use-java-context>false</use-java-context>
        <metadata>
            <type-mapping>PostgreSQL 8.3</type-mapping>
        </metadata>
    </local-tx-datasource>
</datasources>
```

Ostatnią rzeczy, którą musimy przygotować jest deskryptor specyficzny dla serwera, który zawiera odwzorowanie logicznej nazwy zasobu zdefiniowanej w aplikacji na jego fizyczny odpowiednik zdefiniowany w serwerze. W przypadku JBoss'a takie mapowanie odbywa się za pomocą pliku `jboss-web.xml` i może wyglądać następująco

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss-web>
    <resource-ref>
        <res-ref-name>jdbc/SampleDS</res-ref-name>
        <jndi-name>jdbc/PhysicalSampleDS</jndi-name>
    </resource-ref>
</jboss-web>
```

Warto również pamiętać, że sterowniki JDBC powinny być umieszczone w odpowiednich katalogach serwera aplikacyjnego, a nie w archiwach aplikacji (war, ear). Przykładowo w JBoss'ie powinniśmy je umieszczać w katalogu: `$JBOSS_HOME/server/$PROFILE_NAME/lib` gdzie `JBOSS_HOME` to ścieżka, w której zainstalowaliśmy serwer a `PROFILE_NAME` to nazwa profilu, w ramach którego ma pracować nasza aplikacja. Umieszczenie sterownika JDBC w archiwum aplikacji to proszenie się o trudne do wychwycenia problemy związane z hierarchią ładowania klas w serwerze aplikacji.

Tworząc aplikacje Java Enterprise trzeba pamiętać, żeby nigdy korzystać bezpośrednio ze sterownika JDBC do uzyskiwania połączeń do bazy danych (`DriverManager.getConnection(...)`). Zawsze należy korzystać z mechanizmów dostarczanych przez serwer aplikacji.

Transakcje lokalne

Rozważania związana z transakcjami w dostępie do baz danych zacznijmy od tzw. transakcji lokalnych. Są to transakcje realizowane na poziomie menedżerów poszczególnych zasobów (np. baz danych czy kolejek JMS). Zarządzając takimi transakcjami korzystamy wyłącznie z właściwości udostępnianych przez interfejsy specyficzne dla danego zasobu. Dla baz danych będzie to interfejs JDBC. Korzystamy tutaj z tego, że menedżer zasobów (np. silnik bazy danych) jest jednocześnie menedżerem transakcji. W transakcjach lokalnych nie jest wykorzystywany menedżer transakcji serwera aplikacji. Rola serwera aplikacji w przypadku transakcji lokalnych sprowadza się jedynie na umożliwieniu dostępu do zasobu.

Transakcje lokalne możemy wykorzystać jeżeli w naszym systemie mamy jedną bazę danych, co jest dość typowym przypadkiem.

Zarządzanie transakcjami bazadanowymi odbywa się na poziomie połączenia do bazy danych, reprezentowanego w interfejsie JDBC przez obiekt `java.sql.Connection`. W serwerze aplikacji obiekty te uzyskujemy z obiektu `javax.sql.DataSource` pobranego przez naszą aplikację z JNDI. Obiekt `java.sql.Connection` udostępnia następujące metody służące do zarządzania transakcjami:

- `void setAutoCommit(boolean autoCommit)` – umożliwia wyłączenie trybu autocommit, co jest równoznaczne z rozpoczęciem transakcji. Domyślnie wszystkie zwracane przez serwer połączenia są ustawiane w tryb autocommit, co oznacza, że każde zapytanie do bazy danych wykonywane jest w oddzielnej transakcji.
- `void commit()` – powoduje zatwierdzenie bieżącej transakcji. Jednocześnie powoduje rozpoczęcie nowej transakcji.
- `void rollback()` – powoduje wycofanie bieżącej transakcji. Jednocześnie powoduje rozpoczęcie nowej transakcji.
- `void setTransactionIsolation(int level)` – ustawia wskazany poziom izolacji transakcji. Więcej na ten temat w dalszej części artykułu.
- `Savepoint setSavepoint(String name)` – umożliwia ustawienie punktu kontrolnego o danej nazwie, do którego będzie się można wycofać bez wycofywania całej transakcji.
- `void releaseSavepoint(Savepoint savepoint)` – powoduje usunięcie danego punktu kontrolnego z bieżącej transkacji.
- `void rollback(Savepoint savepoint)` – powoduje wycofanie zmian do podanego punktu kontrolnego.

W praktyce tworząc aplikacje Java Enterprise, które zazwyczaj mają charakter aplikacji OLTP (Online Transaction Processing) bardzo rzadko korzysta się z punktów kontrolnych transakcji. Mają one głównie zastosowanie w aplikacjach, w których występują długotrwające transakcje. W takich aplikacjach występują zazwyczaj złożone i kosztowne operacje, których wycofywanie w całości byłoby nieopłacalne. W aplikacjach internetowych ten przypadek

jest bardzo rzadko spotykany. W praktyce mamy więc do czynienie tylko z trzema metodami – `setAutocommit(...)`, `commit ()` oraz `rollback()`.

Przyjrzyjmy się jak za pomocą tych metod zrealizować transakcję bazodanową wpisującą dane do dwóch różnych tabel z poziomu serwletu (pominąłem w tym przykładzie właściwą obsługę wyjątków):

```
public class JDBCTransactionDemoServlet extends HttpServlet {
    protected void doGet(HttpServletRequest req, HttpServletResponse res)
                            throws ServletException, IOException {
        ...
        try {
            Context ctx = new InitialContext();

            // uzyskanie połączenia
            DataSource ds1 =
                (DataSource)ctx.lookup("java:comp/env/jdbc/TransactionDemoDS1");
            Connection conn = ds1.getConnection();

             // rozpoczęcie transakcji
             conn.setAutoCommit(false);

            // wykonanie operacji
            PreparedStatement stmt1 = conn.prepareStatement(
                 "insert into a values (1, 'dane_x'");
            PreparedStatement stmt2 = conn.prepareStatement(
                 "insert into b values (2, 'dane_y'");
            stmt1.executeUpdate();
            stmt2.executeUpdate();
            stmt1.close();
            stmt2.close();

            // zatwierdzenie transakcji i zamknięcie połączenia
            conn.commit();
            conn.close();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
        ...
    }
}
```

Jak można zauważyć koncepcyjnie zarządzanie transakcjami z poziomu interfejsu JDBC nie wydaje się szczególnie trudne i składa się z następujących kroków:

- Uzyskania połączenia z serwera aplikacji
- Rozpoczęcia transakcji.
- Wykonania operacji na bazie danych w ramach otwartej transakcji.
- Zatwierdzenia transakcji.
- Zamknięcia połączenia. W serwerze aplikacji połączenie nie jest fizycznie zamykane, a jedynie wraca do puli połączeń i może być ponownie wykorzystane – niekoniecznie przez naszą aplikację.

Jednak jak to zwykle bywa, diabeł tkwi w szczegółach. W naszym przypadku jest to właściwa obsługa sytuacji wyjątkowych. W trakcie działania przedstawionego wyżej kodu może dojść do wielu sytuacji, które wymagają specjalnego obsłużenia, w szczególności:

- Może nie udać się pobranie połączenia (np. wyczerpana pula połączeń).
- Mogą nie udać się operacje wykonywane na danych w bazie (np. błąd w aplikacji albo naruszenie więzów integralności danych na poziomie bazy danych). Powinno to skutkować wycofaniem transakcji.
- Może nie udać się zatwierdzenie transakcji (np. wskutek wykrytego konfliktu podczas modyfikacji danych w bazie – przypadek typowy dla baz z optymistycznym blokowaniem).
- Problemy komunikacyjne pomiędzy serwerem aplikacji a bazą danych na każdym z wymienionych etapów. Mój ulubiony przykład z życia to zrywanie połączeń w związku z przekroczeniem zdefiniowanej liczby stanów w zaporze ogniowej (*statefull firewall*) pracującej pomiędzy serwerem aplikacji a serwerem bazy danych.

Zobaczmy jak można sobie z tymi sytuacjami poradzić:

```
Context ctx = new InitialContext();
DataSource ds1 = (DataSource)ctx.lookup("java:comp/env/jdbc/TransactionDemoDS1");

Connection conn = null;
boolean ok = false;

try {
    conn = ds1.getConnection();
} catch (SQLException e) {
    throw new RuntimeException(e);
}

RuntimeException exception = null;
try {
    conn.setAutoCommit(false);

    ...
    // tutaj wykonujemy operacje na bazie
    // w rozpoczętej transakcji
    ...

    ok = true;
} catch (SQLException e) {
    exception = new RuntimeException(e);
} catch (RuntimeException e) {
    exception = e;
} finally {
    if (ok) {
        try {
            conn.commit();
        } catch (SQLException e) {
            if (exception != null) {
                exception = new RuntimeDoubleException(exception, e);
            } else {
                exception = new RuntimeException(e);
            }
        }
    } else {
        try {
            conn.rollback();
        } catch (SQLException e) {
            if (exception != null) {
                exception = new RuntimeDoubleException(exception, e);
            } else {
                exception = new RuntimeException(e);
            }
        }
    }

    try {
        conn.close();
    } catch (SQLException e) {
        if (exception != null) {
            exception = new RuntimeDoubleException(exception, e);
        } else {
            exception = new RuntimeException(e);
        }
    }
}

if (exception != null) {
    throw exception;
}
```

Jak widać pełna obsługa tego typu transakcji wymaga sporej ilości kodu, którego działanie nie jest już takie łatwe do prześledzenia.

Teraz nasuwa się pytanie gdzie taką obsługę wyjątków umieścić w kodzie naszej aplikacji.

W pierwszym podejściu wydawałoby się, że w każdym miejscu gdzie realizujemy nasze transakcje. Jeżeli jednak zadanie prawidłowego zarządzania transakcjami i połączeniami oddamy w ręce programistów poszczególnych modułów czy ekranów aplikacji to nie możemy oczekiwać, że nasz system będzie odpowiednio wysokiej jakości. Nawet dobremu programiście zdarzy się jakiś element tej obsługi przeoczyć lub zrealizować nieprawidłowo. W najlepszym przypadku będzie to prowadziło do wycieków z puli połączeń, a w najgorszym do ulotnych, ciężkich do zlokalizowania błędów w logice naszej aplikacji. Poza tym takie podejście powoduje gigantyczną duplikację kodu. Nawet w niewielkich systemach miejsc, w których należałoby umieścić przedstawiony wyżej kod byłoby pewnie z kilkadziesiąt. Prowadzi to w oczywisty sposób do pogorszenie czytelności kodu (obsługa wyjątków zajmuje więcej niż logika operacji na bazie) i kłopotów w jego utrzymaniu.

Podejście drugie polega na zamknięciu powyższego kodu w jedną klasę usługową i wywoływaniu go z tych miejsc naszej aplikacji, gdzie występuje zarządzanie transakcjami.

Sytuacja komplikuje się wtedy, gdy budujemy złożony system, w którym liczba możliwych interakcji pomiędzy fragmentami kodu realizującym logikę transakcyjną jest trudna do oszacowania, a co za tym idzie do oprogramowania. Wtedy zostaje nam podejście trzecie, czyli scentralizowane zarządzania transakcjami.

Podejście to polega na umieszczeniu przetwarzania całego żądania do naszej aplikacji w jednej transakcji JDBC. Można to zrealizować na poziomie metod `doGet(...) czy doPost(...)` serwleta lub kodu kontrolera z modelu MVC a połączenie do bazy danych umieścić w parametrach żądania HTTP, przez co zawsze mamy do niego dostęp z dowolnego miejsca w naszej aplikacji. Takie podejście bardzo upraszcza zarządzanie transakcjami i połączeniami do bazy danych w naszej aplikacji. Oczywiście jak wszystkie „złote środki" takie podejście ma swoje ograniczenia. Jeśli zdecydujemy się na takie rozwiązanie należy przede wszystkim pamiętać, że w przypadku wykonywania przez nas długotrwałych operacji (np. skomplikowany rendering treści) cały czas blokujemy jedno połączenie na poziomie serwera aplikacji oraz zasoby (wiersze/tabele) na poziomie bazy danych.

W praktyce najlepiej sprawdza się podejście trzecie uzupełniane podejściem drugim. To znaczy standardowo zawsze wszystko obejmujemy transakcją, a tylko w pewnych krańcowych sytuacjach (np. ze względów wydajnościowych) odstępujemy od tej reguły i obsługujemy zachowanie transakcyjne naszego systemu na poziomie poszczególnych komponentów.

Transakcje zarządzane przez menedżer transakcji serwera aplikacji (transakcje rozproszone)

Jeśli w systemie operujemy na więcej niż jednym zasobie transakcje lokalne przestają wystarczać. Musimy wtedy skorzystać z dobrodziejstwa menedżera transakcji rozproszonej, który jest składową serwera aplikacji. Przyjrzyjmy się jak w takim przypadku wygląda korzystanie z baz danych. Poniższy przykład pokazuje fragment aplikacji, która operuje na dwóch bazach danych w ramach pojedynczej transakcji.

```
...
Context ctx = new InitialContext();
UserTransaction ut = (UserTransaction)ctx.lookup("java:comp/UserTransaction");;

ut.begin();

DataSource ds1 = (DataSource)ctx.lookup("java:comp/env/jdbc/TransactionDemoDS1");
DataSource ds2 = (DataSource)ctx.lookup("java:comp/env/jdbc/TransactionDemoDS2");

Connection conn1 = ds1.getConnection();
Connection conn2 = ds2.getConnection();

doSomethingInFirstDatabase(conn1);
doSomethingInSecondDatabase(conn2);

conn1.close();
conn2.close();

ut.commit();
...
```

Jak widać z poziomu kodu uczestnictwo naszych operacji na bazach danych w transakcji rozproszonej nie wymaga żadnych specjalnych zabiegów. Widzimy również, że nie korzystamy tutaj z żadnych metod interfejsu JDBC, które służyły nam do zarządzania transakcjami lokalnymi. Wszystkie te szczegóły ukrywa przed nami menedżer transakcji i odpowiednie implementacje sterowników do baz danych.

Dla czytelności pominąłem obsługę wyjątków. Oczywiście operacje pobierania i zamykania połączeń powinny być nią objęte. Również kod całej transakcji powinien być obsłużony w taki sposób, w jaki przedstawiłem to w pierwszej części artykułu (rozdział „Obsługa wyjątków i sytuacji brzegowych przy samodzielnym zarządzaniu transakcjami").

Aby móc operować na bazach danych w ramach transakcji rozproszonej musi zostać spełnionych kilka warunków, których nie widać z przedstawionego wyżej przykładu:

- Przede wszystkim silnik bazy danych, którego używamy musi wspierać dwufazowy protokół zatwierdzania transakcji (na przykład w tak popularnej bazie danych jak PostgreSQL wsparcie dla tego protokołu pojawiło się dopiero od wersji 8.1).
- Po drugie sterownik JDBC do bazy danych musi implementować odpowiednie interfejsy pozwalające na współpracę z menedżerem transakcji (XADataSource, XAConnection, XAResource). W przypadku wspomnianej wyżej bazy PostgresSQL funkcjonalność ta pojawiła się od wersji `8.1dev-403` sterownika.
- Wreszcie musimy we właściwy sposób skonfigurować źródło danych w serwerze aplikacji (czyli jako źródło XA). Ponieważ sterowniki JDBC do danej bazy danych przychodzą zazwyczaj w jednej paczce zawierającej zarówno sterowniki XA jak i nie XA to dość łatwo o pomyłkę. Sprzyja temu większość przykładów użycia sterownika, które pokazują konfigurację źródła danych nie wspierającego uczestnictwa bazy danych w transakcji rozproszonej. Skonfigurowanie źródła danych, które nie wspiera transakcji rozproszonych to jeden z najczęstszych błędów popełnianych podczas tworzenia systemów transakcyjnych.

Korzystając z baz danych w ramach transakcji rozproszonej (JTA) nie wolno nam używać następujących metod zdefiniowanych w obiekcie `java.sql.Connection`:

- `setAutoCommit(...)`
- `commit()`
- `rollback()`
- `setSavepoint()`

Zarządzanie transakcjami przez menedżer transakcji serwera aplikacji równie dobrze sprawdza się w przypadku, gdy korzystamy z jednej bazy danych. Tak więc możemy zarządzać transakcjami w całym systemie w jednolity sposób, korzystając interfejsu `UserTranscation`, bez potrzeby wnikania w aspekt zarządzania transakcjami z poziomu JDBC. Zachęcam do takiego podejścia nawet w systemach z jednym źródłem danych.

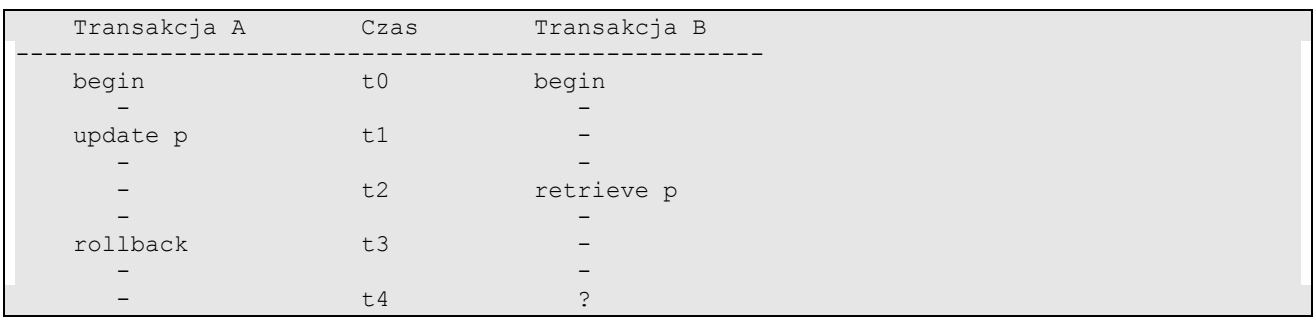
Poziomy izolacji transakcji

Omówienie zastosowania baz danych w systemach transakcyjnych nie może się obyć bez poruszenia tematyki poziomów izolacji transakcji w bazach danych. Być może jest to nawet najważniejszy aspekt budowy systemów transakcyjnych korzystając z baz danych. W pierwszej części artykułu opisując właściwości izolacji transakcji wspominałem, że w odniesieniu do baz danych własność ta nastręcza wielu kłopotów. Przyjrzyjmy się dlaczego tak jest.

W idealnym świecie transakcje nie widzą żadnych efektów działań wykonywanych przez inne transakcje dopóki tamte nie zostaną zatwierdzone. Obserwując taki świat z zewnątrz mielibyśmy wrażenie, że wszystkie transakcje wykonują się po kolei (są uszeregowane). Ponieważ taki idealny świat działał by zbyt wolno, to zaczęto poszukiwać sposobów na złagodzenie tego wymagania i zwiększenie wydajności. W ten sposób narodziły się w bazach danych różne poziomy izolacji transakcji. W zależności od poziomu dopuszczają one istnienie określonych anomalii przy współbieżnym wykonywaniu transakcji. Aby dobrze zrozumieć poziomy izolacji transakcji musimy najpierw przyjrzeć się tym anomaliom.

Brudne odczyty (*ang. dirty reads*)

Brudny odczyt oznacza, że transakcje mogą widzieć zmiany wykonywane przez inne transakcje zanim zmiany te zostaną zatwierdzone. W takim przypadku istnieje możliwość, że w przypadku wycofania zmian inne transakcje będą dalej pracowały na niewłaściwych danych. Sytuację brudnego odczytu ilustruje poniższy diagram:



Transakcje A i B rozpoczynają się w tej samej chwili t0. Następnie transakcja A aktualizuje wiersz p, który transakcja B odczytuje w chwili t2. Niestety w chwili t3 transakcja A wycofuje wykonane wcześniej zmiany, co oznacza, że transakcja B używa wartości p, która tak naprawdę nigdy nie znalazła się w bazie. Najłatwiej sobie wyobrazić jakie „zniszczenia" może powodować takie zachowanie systemu jeśli p oznacza wysokość oprocentowania naszego rachunku bankowego.

Niepowtarzalne odczyty (*ang. nonrepeatable reads*)

Niepowtarzalny odczyt oznacza, że transakcja, która wielokrotnie odczytuje ten sam wiersz, może otrzymać różne wyniki chociaż z punktu widzenia spójności danych oba poprawne, czyli zatwierdzone przez inne transakcje. Sytuację niepowtarzalnego odczytu ilustruje diagram poniżej:

```
    Transakcja A        Czas        Transakcja B
-------------------------------------------------
    begin               t0          begin
       -                               -
    retrieve p          t1              -
       -                               -
       -                t2          update p
       -                               -
       -                t3          commit
       -                               -
    retrieve p?         t4              -
```

Transakcja A pobiera dwa razy wiersz p (t1 i t4), za każdym razem otrzymując inne wyniki. Jeśli inne dane są modyfikowane w oparciu o wartość p, może to prowadzić to niespójności, które jest trudno wykryć.

Fantomy (ang. phantom reads)

Anomalia ta polega na tym, że jeżeli transakcja dwa razy odczytuje zbiór danych według tych samych warunków to może ona otrzymać dwa różne wyniki. Sytuację taką ilustruje diagram poniżej:

```
    Transakcja A          Czas        Transakcja B
-------------------------------------------------
    begin                 t0          begin
       -                                  -
    retrieve q -> a,b     t1              -
       -                                  -
       -                  t2          insert c into q
       -                                  -
       -                  t3          commit
       -                                  -
    retrieve q -> a,b,c!  t4              -
       -                                  -
```

Jak można zauważyć transakcja A dwa razy odczytuje tabelę q, za każdym razem otrzymując inne dane, co jest związane z zatwierdzeniem w międzyczasie transakcji B.

W oparciu o eliminację powyższych anomalii stworzono model poziomów izolacji transakcji. Model zdefiniowano w standardzie języka SQL. Specyfikacja JDBC również opiera się na tym modelu wprowadzając jeden dodatkowy poziom. Przyjrzyjmy się teraz poziomom izolacji transakcji zdefiniowanych w specyfikacji JDBC (stałe w `java.sql.Connection`), w kolejności od najmniej do najbardziej restrykcyjnych:

- `TRANSACTION_NONE` - wskazuje że sterownik nie wspiera transakcji. Ten poziom nie ma nic wspólnego z omawianymi wyżej anomaliami dostępu do danych. Może on być używany w sytuacji, kiedy za pomocą interfejsu JDBC udostępniane są dane lub systemy, w których nie da się zdefiniować żadnego sensownego zachowania transakcyjnego. Takim przykładem jest biblioteka `csvjdbc` (http://csvjdbc.sourceforge.net/), która za pomocą interfejsu JDBC umożliwia dostęp do plików CSV.
- `TRANSACTION_READ_UNCOMMITED` - poziom, na którym mogą występować wszystkie wymienione anomalie.
- `TRANSACTION_READ_COMMITED` - poziom oznaczający, że zmiany wykonywane przez transakcje nie są widoczne dla innych transakcji do momentu jej zatwierdzenia, czyli mamy tutaj ochronę przed brudnymi odczytami. Na tym poziomie dalej mogą występować niepowtarzalne odczyty i fantomy.
- `TRANSACTION_REPEATABLE_READ` - ten poziom chroni zarówno przed brudnymi odczytami jak i przed niepowtarzalnymi odczytami. Fantomy dalej mogą na nim występować.

- `TRANSACTION_SERIALIZABLE` - na tym poziomie nie mogą występować żadne z opisywanych anomalii.

W kontekście tak przyjętej definicji poziomów izolacji chciałbym zwrócić uwagę na jeden znaczący fakt. Ustawienie poziomu izolacji SERIALIZABLE wcale nie oznacza, że mamy zagwarantowane wykonywanie transakcji w sposób uszeregowany (jedna po drugiej). Niestety przez nieszczęśliwy, moim zdaniem, dobór nazwy najwyższego poziomu izolacji wiele osób jest przekonanych, że tak się właśnie dzieje.

Piękny świat poziomów izolacji zdefiniowanych w JDBC API i standardzie SQL psują dostępne implementacje baz danych, które czasami niektórych poziomów nie wspierają albo stosują inną nomenklaturę. Przykładowo w bazie PostgreSQL są tylko dwa poziomy: READ_COMMITED i SERIALIZABLE. W DB2 są cztery, ale inaczej się nazywają, a dodatkowo te same nazwy używane są do określenia różnych poziomów. Tą złożoną sytuację ilustruje tabela poniżej.

DB2 Transaction Isolation Level	JDBC Transaction Isolation Level
Uncommitted Read	READ_UNCOMMITED
Cursor stability	READ_COMMITED
Read stability	REPEATABLE_READ(*)
Repeatable read (*)	SERIALIZABLE

Z tego względu zawsze przy nowej bazie danych czeka nas lektura dokumentacji sterownika JDBC jak i samej bazy danych.

Niestety na tym nie kończą się komplikacje związane z poziomami izolacji transakcji. Dostawcy baz danych implementują poziomy izolacji korzystając z różnych mechanizmów i algorytmów. Zasadniczo podział przebiega między algorytmami wykorzystującymi pesymistyczne i optymistyczne blokowanie zasobów. Przyjrzymy się jaki może mieć to wpływ na transakcje wykonywane na tym samym poziomie izolacji w bazach danych, które stosują odmienne podejścia implementacyjne (PostgreSQL i IBM DB2).

Dla obu baz danych będziemy chcieli wykonać jednocześnie dwie transakcje (symulowane za pomocą dwóch oddzielnych konsol) na tym samym poziomie izolacji (`READ COMMITTED`):

- jedna odczytująca zawartość tabeli `foo` (transakcja A),
- druga modyfikująca wybrany wiersz tabeli `foo` (transakcja B).

W obu przypadkach sprawdzamy jakie jest zachowanie odczytu w transakcji A w przypadku, w którym doszło już do aktualizacji danych przez transakcję B.

Przed wykonaniem tych transakcji zawartość tabeli `foo` była następująca:

```
id       data
----------------
  1      dane 1
  2      dane 2
  3      dane 3
```

PostgreSQL (baza danych z optymistycznym blokowaniem z wykorzystaniem mechanizmu Multi-Version Concurrency Control)

Transakcja A	Transakcja B

test=> BEGIN;
test=> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
	test=> BEGIN;
	test=> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
	test=> UPDATE foo SET data='dane 2 zmienione' WHERE id=2;

test=> SELECT * FROM foo;
id | data
----+--------
1 | dane 1
2 | dane 2
3 | dane 3
(3 rows)
Odczyt daje się wykonać chociaż transakcja B zmieniała dane w tabeli.
test=> COMMIT;
test=>
	test=> COMMIT;
	test=>

IBM DB2 (baza danych z pesymistycznym blokowaniem)

Transakcja A	Transakcja B
db2 => SET ISOLATION cs db2 => SELECT * FROM foo **W tym momencie konsola zawisa czekając na zwolnienie blokady nałożonej przez transakcję B. Wynik pojawi się dopiero po wykonaniu operacji COMMIT na transakcji B.** ID DATA ---------- ---------- 1 dane 1 2 dane 2 - zmienione 3 dane 3 3 record(s) selected. db2 => COMMIT db2 =>	db2 => SET ISOLATION cs db2 => UPDATE foo SET data='dane 2 - zmienione' WHERE id = 2 db2 => COMMIT db2 =>

Jak widać mimo, że wykonujemy identyczne transakcje na takim samym poziomie izolacji, to bazy zachowują się odmiennie - chociaż w obu przypadkach założenie co braku odpowiednich anomalii jest spełnione. Niestety ma to bezpośredni wpływ na działanie naszego systemu, szczególnie w zakresie jego wydajności. Musimy taki aspekt uwzględnić przy projektowaniu i implementacji naszego systemu.

Kolejnym dość złożonym zagadnieniem jest ustawienie żądanego poziomu izolacji z poziomu aplikacji Java Enterprise. Niestety tutaj każdy serwer aplikacji zachowuje się inaczej:

- Pierwsze rozczarowanie przeżywamy nie mogąc na poziomie deskryptora aplikacji ustawić poziomu izolacji źródła danych. Takie rozwiązanie wydaje się najbardziej naturalne, gdyż poziom izolacji transakcji wynika z charakteru aplikacji. Niestety specyfikacja Java Enterprise pomija całkowicie ten istotny aspekt aplikacji. Konfigurowanie tego na poziomie konfiguracji fizycznego zasobu w serwerze wydaje się pomysłem karkołomnym, stawiającym pod dużym znakiem zapytania całą koncepcję rozdzielenia zasobów logicznych od fizycznych.
- Zazwyczaj mamy możliwość ustawienia domyślnego poziomu izolacji na poziomie konfiguracji zasobu fizycznego w serwerze aplikacji. Ale jeśli mamy sytuację, w której chcemy skorzystać z zasobu na różnych poziomach izolacji mamy problem, bo musimy konfigurować więcej niż jedno fizyczne źródło danych do tej samej bazy danych. Niestety często jest to jedyne rozwiązanie.
- W niektórych serwerach aplikacji daje się ustawić poziom izolacji na poziomie logicznego źródła danych za pomocą specyficznego deskryptora deploymentu. Np. w serwerze aplikacji IBM Websphere można to zrobić tak (fragment pliku `ibm-web-ext.xmi`):

```
<resourceRefExtensions xmi:id="ResourceRef_ext_1"
                       isolationLevel="TRANSACTION_READ_COMMITTED">
   <resourceRef href="WEB-INF/web.xml#ResRef_1"/>
</resourceRefExtensions>
```

  gdzie do zasobów wymienionych w deskryptorze `web.xml` dodajemy dodatkowe własności, które będą uwzględniane w momencie, gdy zażądamy dostępu do danego zasobu.
- Ostatnią deską ratunku wydaje się metoda `setTransactionIsolation` wywołana bezpośrednio na uzyskanym z serwera aplikacji połączeniu (`java.sql.Connection`). Niestety nie mamy gwarancji, że metoda ta zadziała w przypadku gdy transakcja jest już rozpoczęta, a z taką sytuacją mamy do czynienia gdy korzystamy z transakcji JTA. Niektóre bazy danych i sterowniki do nich pozwalają zmienić poziom izolacji w trakcie trwania transakcji (np. baza IBM DB2 pozwala zmienić poziom izolacji transakcji na poziomie pojedynczego zapytania SQL).

Jak wynika z powyższych rozważań JDBC API w zakresie poziomów izolacji w praktyce nie jest w stanie ukryć przed programistą rzeczywistych implementacji silników baz danych i sterowników do nich. Zawsze musimy wniknąć w szczegóły działania bazy danych, konfiguracji sterownika do niej, konfiguracji serwera aplikacji i procesu instalacji aplikacji.

Można się jeszcze zastanawiać w jakich przypadkach stosować konkretne poziomy izolacji. Ciężko na to pytanie jednoznacznie odpowiedzieć nie znając wymagań dla konkretnego systemu. Można się jedynie kierować pewnymi heurystykami wynikającymi z dotychczasowego doświadczania. I tak z mojej praktyki wynika, że:

- Najlepiej zacząć od ustawienia domyślnego poziomu izolacji transakcji na `READ_COMMITTED`. Zapewnia on rozsądny kompromis między wydajnością systemu, a zapewnieniem spójności danych.
- Poziomu `READ_UNCOMMITTED` możemy użyć przy wyciąganiu mniej istotnych danych na potrzeby prezentacji. Ma to szczególne znaczenie w przypadku korzystania z bazy danych z pesymistycznym blokowaniem.
- Poziomu `SERIALIZABLE` można użyć do realizacji semafora na bazie danych oraz oczywiście do implementacji operacji transferu środków pomiędzy kontami bankowymi.

Pozostałe aspekty korzystania z baz danych z poziomu serwera aplikacji

Na zakończenie chciałbym przedstawić kilka problemów/aspektów, na które można się natknąć w codziennej praktyce zawodowej, a co których dość ciężko znaleźć wyjaśnienie czy rozwiązanie w dokumentacji produktów czy w Internecie.

Timeout transakcji JTA a timeout'y na poziomie bazy danych

Pomiędzy timeout'em transakcji JTA a działaniami wykonywanymi przez nas bazie danych nie ma żadnego związku, poza tym, że po przekroczeniu timeout'u JTA wszystkie wykonane przez nas operacje na bazie danych powinny zostać wycofone. Wynika to z przyczyn, o których pisałem w pierwszej części artykułu.

Co więcej, znane mi bazy danych nie oferują funkcjonalności, która by umożliwiała ustawienie maksymalnego czasu trwania transakcji w bazie danych.

Wiem, że wiele osób poszukuje takiego rozwiązania. Pewne przybliżone rozwiązanie można uzyskać korzystając ze specyficznych własności konkretnych silników baz danych. Możemy tu mówić w zasadzie o dwóch typach rozwiązań:

- Część baz danych (np. PostgreSQL) umożliwiają ustawienie maksymalnego czasu trwania pojedynczego zapytania. Odbywa się to na poziomie konfiguracji serwera bazy danych (`statement_timeout` w PostgreSQL) lub za pomocą JDBC API - `java.sql.Statement.setQueryTimeout` (niestety w większości sterowników niezaimplementowane, gdyż silnik bazy tego nie wspiera).
- Część baz danych (np. Oracle, IBM DB2, MS SQL Server) umożliwia ustawienie maksymalnego czasu utrzymywania blokady na zasobie (wiersz, tabela). Zazwyczaj znane jest to pod pojęciem `lock-timeout`. Rozwiązanie to jednak nie daje żadnej gwarancji i ma duże ograniczenia. Na przykład w przypadku pełnego przeglądania dużej tabeli (*ang. full scan*) blokada może się zmieniać przy przechodzeniu z wiersza na wiersz, przez co nigdy nie uzyskamy wyjątku oznaczającego `lock-timeout` a jednocześnie dostęp do takiej tabeli będzie mocno utrudniony.

Korzystanie z bazy danych poza otwartą transakcją JTA

Czasami chcemy zrealizować jakieś operacje na bazie danych poza aktualnie trwającą transakcją. W takim przypadku możemy skorzystać bezpośrednio z menedżera transakcji. Przykład takiego rozwiązania przedstawiłem w pierwszej części artykułu w sekcji „Bezpośrednie korzystanie z menedżera transakcji".

Korzystanie z połączenia przed rozpoczęciem transakcji JTA

Spotykam się czasami z pytaniem, jak zachowuje się połączenie do bazy wzięte przed rozpoczęciem transakcji JTA i co się dzieje jeśli po rozpoczęciu transakcji JTA będę chciał go dalej używać? Co się stanie jeśli po rozpoczęciu transakcji JTA wezmę połączenie z tego samego źródła danych? Mamy więc do czynienia z przykładem kodu jak poniżej:

```
Context ctx = new InitialContext();
DataSource ds1 = (DataSource)ctx.lookup("java:comp/env/jdbc/TransactionDemoDS1");
Connection conn1 = ds1.getConnection();

// zrób coś na połączeniu conn1

UserTransaction ut = (UserTransaction)ctx.lookup("java:comp/UserTransaction");

ut.begin();

ds1 = (DataSource)ctx.lookup("java:comp/env/jdbc/TransactionDemoDS1");

Connection conn2 = ds1.getConnection();

// zrób coś na połączeniu conn2
```

```
// zrób coś na połączeniu conn1

ut.commit();
```

Zachowanie systemy w takim przypadku jest następujące:

- Z połączeniem wziętym przed transakcją nic się nie dzieje. Można go dalej używać, tak jak byśmy go używali w transakcji lokalnej. Transakcja JTA nie na to połączenia żadnego wpływu.
- Pobrania jeszcze raz połączenia z tego samego źródła danych powoduje przydzielenie nowego połączenia i przypisanie go do transakcji JTA.
- Serwer aplikacyjny zapewnia więc prawidłową pracę na obu połączeniach, które są wzajemnie odseparowane.

Domyślne poziomy izolacji

Każdy serwer aplikacji stosuje własne domyślne poziomy izolacji transakcji, co więcej mogą się one różnić w zależności od bazy danych lub użytego sterownika JDBC. Na przykład:

- IBM Websphere w wersji 5.1 dla większości baz danych stosuje domyślny poziom `REPEATABLE_READ`, ale dla bazy Oracle poziome `READ_COMMITED`.
- Weblogic korzysta z ustawień domyślnych bazy danych.

Tak więc zawsze trzeba sprawdzić jaki jest domyślny poziom izolacji w naszym specyficznym przypadku, a najlepiej nadpisać konfigurację własnymi ustawieniami, co chroni nas przed sytuacją, w której dostawca serwera w drobnej poprawce aktualizującej zmienia domyślny poziom, a nasza produkcyjna aplikacja dotychczas świetnie działająca załamuje się nawet pod niewielkim obciążeniem.

Ze względu na obszerność zagadnienia związanego z wykorzystaniem baz danych przy budowie systemów transakcyjnych obiecany w poprzednim artykule temat wykorzystania systemów kolejkowania postanowiłem przenieść do kolejnego artykułu.

Literatura

[1] Java Transaction API Specification, http://java.sun.com/javaee/technologies/jta/index.jsp
[2] Mark Little, Jon Maron, Greg Pavlik, Java Transaction Processing, Prentice Hall, 2004
[3] Specyfikacja oraz API JDBC, http://java.sun.com/javase/technologies/database/
[4] Dokumentacja do bazy danych PostgreSQL
[5] Dokumentacja do bazy danych IBM DB2
[6] Dokumentacja do serwera aplikacji JBoss
[7] Dokumentacja do serwera aplikacji JOnAS
[8] Dokumentacja do serwera aplikacji IBM Websphere
[9] Dokumentacja do serwera aplikacji BEA Weblogic (obecnie Oracle)