Transakcje w systemach Java Enterprise

Wprowadzenie do systemów transakcyjnych oraz mechanizmów obsługi transakcji w serwerach aplikacji

Artykułem tym rozpoczynam cykl związany z szeroko pojętą tematyką budowy systemów transakcyjnych w środowisku Java Enterprise. Temu podstawowemu zagadnieniu związanemu bezpośrednio z tworzeniem solidnych systemów informatycznych poświęcono do tej pory niewiele książek i artykułów, a zdobycie praktycznej wiedzy w tym zakresie jest stosunkowo trudne. Z tego powodu w ramach cyklu w kolejnych artykułach postaram się w sposób systematyczny przedstawić następujące zagadnienia:

- Podstawowe pojęcia i mechanizmy związane z budową systemów transakcyjnych.
- Sposób obsługi transakcji w serwerze aplikacji.
- Korzystanie z baz danych i systemów kolejkowania a transakcje w serwerze aplikacji.
- Transakcje w komponentach EJB.
- Strategie obsługi transakcji w aplikacjach JEE.
- Problemy i ograniczenia związane z budową transakcyjnych aplikacji w technologii JEE.

Szczególną uwagę będę starał się poświęcić rzeczywistym problemom na jakie możemy się natknąć tworząc systemy transakcyjne i praktycznym rozwiązaniom, które możemy zastosować w codziennej pracy.

Wprowadzenie

Zastosowanie systemów transakcyjnych w aplikacjach bankowych, e-commerce, czy innych, w których w grę wchodzą pieniądze w zasadzie nie podlegają dyskusji. Ale można się zastanawiać, czy warto do prostszych systemów internetowych, systemów zarządzania treścią czy np. aplikacji forum internetowego dokładać jeszcze dodatkowy aspekt w postaci transakcji. Według mnie warto, co najmniej z jednego powodu. Dzięki transakcjom możemy zachować spójność danych w systemie. Nie ma nic gorszego niż próba naprawy niespójności danych. Dodajmy, że próba bez gwarancji sukcesu. Jednym słowem używając transakcji gwarantujemy sobie pewien poziom spokoju.

Rozważania na temat transakcji zacznę od przypomnienia kilku podstawowych definicji.

Transakcja - zestaw operacji na danych, który traktujemy jaką jedną całość i który cechuje się następującymi właściwościami:

- Jest niepodzielny (*atomicity*), czyli albo wszystkie operacje w ramach transakcji zostaną wykonane albo żadna.
- Zachowuje spójność danych (*consistency*). To znaczy, że jeśli na dane obrabiane przez system nałożone są pewne warunki logiczne, to warunki te muszą być spełnione zarówno przed jak i po wykonaniu transakcji. Należy zwrócić uwagę, że nie ma wymagania, aby w trakcie trwania transakcji dane były spójne!
- Jest izolowany (*isolation*). Jeśli w jednym systemie wiele transakcji wykonywane jest współbieżnie to z punktu widzenia pojedynczej transakcji powinno to wyglądać tak, jakby wszystkie transakcje były wykonywane po kolei. Mówimy wtedy, że transakcje wykonywane są we wzajemnej izolacji. Ta właściwość, szczególnie w odniesieniu do baz danych przysparza sporo problemów. Jest to spowodowane potrzebą zwiększenia wydajności kosztem niepełnej izolacji transakcji. Temat ten postaram się szczegółowo omówić w części związanej z bazami danych.
- Jest trwały (*durability*). Zmiana danych, których dokonała transakcja muszą być trwałe, nawet w przypadku awarii systemu tuż po zakończeniu transakcji.

Wymienione wyżej właściwości określane są skrótem *ACID* pochodzącym od pierwszych liter ich angielskich nazw. A samą transakcję zazwyczaj określa się jako *transaction*, ale często również używa terminu *unit of work* (UOW).

System transakcyjny (*transactional system*) – system, w którym wszystkie operacje na danych grupowane są w transakcje.

Zasób transakcyjny (*transactional resource*) – system (podsystem), który umożliwia operowanie na danych w sposób transakcyjny (np. baza danych).

Zakres transakcji (*transaction scope*) – obszar działania programu od momentu rozpoczęcia transakcji do jej zatwierdzenia lub wycofania.

Zatwierdzanie transakcji (*transaction commit*) – operacja trwałego wprowadzenia zmian w danych, które zaszły od momentu rozpoczęcia transakcji.

Wycofanie transakcji (*transaction rollback*) – operacje wycofania wszystkich zmian w danych, które zaszły od momentu rozpoczęcia transakcji.

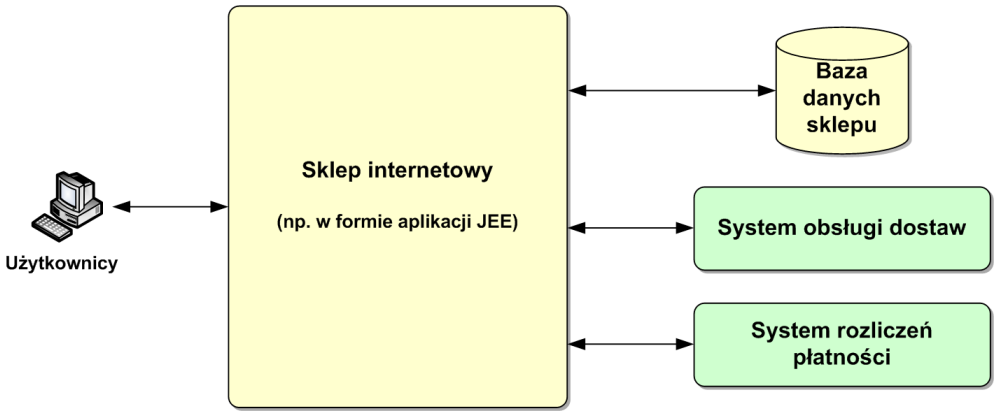
Typowym przykładem systemu transakcyjnego (a jednocześnie zasobu transakcyjnego) są systemy relacyjnych baz danych. Przez wiele lat systemy baz danych były w praktyce synonimem systemów transakcyjnych. Wymagania związane z budową złożonych systemów, w szczególności systemów, które muszą się integrować z innymi systemami zmieniły świat systemów transakcyjnych. Nie da się jednak zaprzeczyć, że bazy danych grają w nim jedną z głównych ról.

Transakcje rozproszone

Aby zagadnienie integracji systemów dokładniej zilustrować przyjrzyjmy się dość typowej konstrukcji współczesnego sklepu internetowego poglądowo przedstawionej na rysunku 1. System sklepu jest wyposażony we własną bazę danych, w której gromadzone są zarówno informacje o ofercie sklepu jak i o zamówieniach składanych przez klientów.

Ale to nie wszystko. Zamówione towary trzeba jakoś dostarczać. Obsługą tego procesu zajmuje się zazwyczaj odrębny system wspierających logistykę dostaw. Zamówienia trzeba również rozliczać, fakturować itd. Dochodzi więc system rozliczeń płatności, czy system finansowo księgowy. Oczywiście można to dalej komplikować przez dodawanie kolejnych systemów, takich jak obsługa płatności elektronicznych, system gospodarki magazynowej, systemy powiadomień o statusie zamówień (mail, sms) itd.

Oczywiście końcowego użytkownika nie interesuje to, ile systemów w rzeczywistości stoi za aplikacją sklepu internetowego. Składając zamówienie spodziewa się on, że dostanie towar i fakturę, czyli traktuje swoje działanie jako pojedynczą transakcję. To na aplikacji sklepu ciąży zadanie doprowadzenia do sytuacji, w której wszystkie systemy zagrają razem.



Rysunek 1 Przykład sklepu internetowego w architekturze rozproszonej

Taki mniej więcej jest obraz współczesnych systemów informatycznych, które nam przychodzi w ostatnim czasie tworzyć, a w których występuje nieustająca potrzeba integracji różnych systemów.

Rodzi to z kolei innego rodzaju problem, mianowicie problem transakcji rozproszonej, czyli takiej w której pojedyncza aplikacja wykonuje operacje na różnych systemach, które nic o sobie nie wiedzą, jednocześnie zachowując wszystkie właściwości transakcji, o których wspominałem na początku (*ACID*).

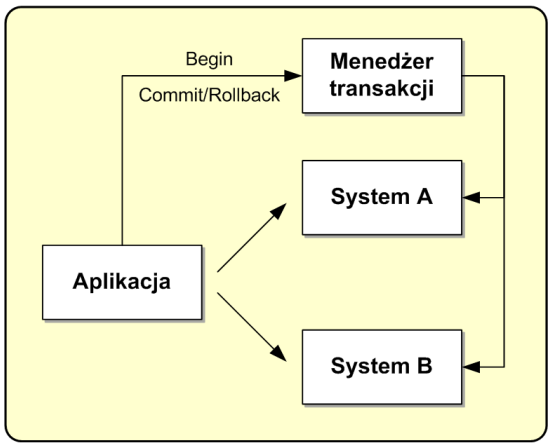
Oczekiwany przebieg transakcji rozproszonej wygląda następująco:

- Rozpoczęcie transakcji.
- Użycie kilku zasobów transakcyjnych.
- Zatwierdzenie lub wycofanie transakcji, co powinno spowodować określone skutki we wszystkich zasobach uczestniczących w transakcji.

Implementacja pojedynczego systemu transakcyjnego, takiego jak na przykład relacyjna baza danych nie jest zadaniem trywialnym. W przypadku systemu, który musi wspierać transakcje rozproszone skala trudności znacząco rośnie. Podstawową trudnością jest rozwiązanie problemu komunikacji między systemami, które często były tworzone jako całkowicie niezależne produkty. Poradzono sobie z tym konstruując oprogramowanie menedżera transakcji i wymyślając dwufazowy protokół zatwierdzania transakcji, o którym teraz kilka słów.

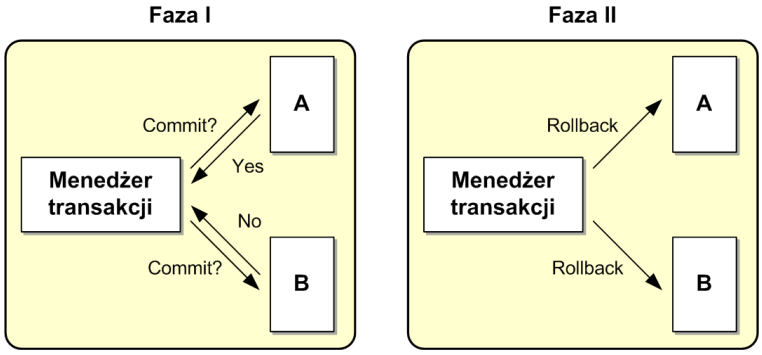
Dwufazowy protokół zatwierdzania transakcji

Przyjrzyjmy się jak działa dwufazowy protokół zatwierdzania transakcji. Przykład na rysunku 2 pokazuje dwa niezależne systemy A i B, które muszą uczestniczyć w transakcji rozproszonej. Wprowadzony jest również menedżer transakcji, który w imieniu aplikacji korzystającej z systemów A i B zarządza transakcją. Po wykonaniu operacji na systemach A i B, aplikacja żąda od menedżera zatwierdzenia transakcji.

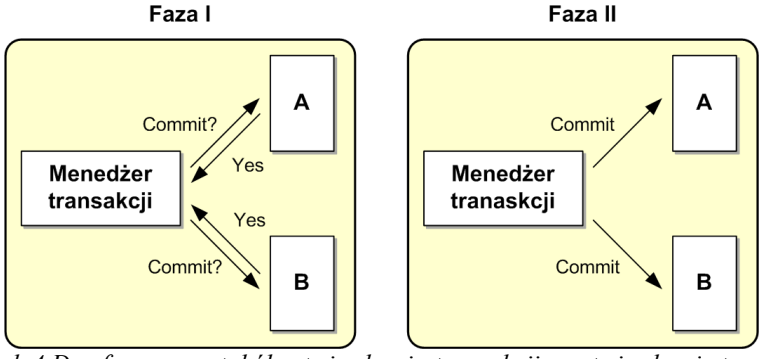


Rysunek 2 Infrastruktura systemu zarządzania transakcjami rozproszonymi

Menedżer transakcji w pierwszej fazie, którą określa się fazą przygotowania (*prepare*), pyta wszystkie systemy, czy są gotowe do zatwierdzenia swoich lokalnych zmian. Każdy system może udzielić jednej z dwóch odpowiedzi: tak lub nie. Jeśli choć jeden z systemów nie zgodzi się na zatwierdzenie transakcji menedżer transakcji w fazie drugiej do wszystkich systemów wysyła rozkaz wycofania transakcji (tą sytuację ilustruje rysunek 3). Jeśli wszystkie systemy odpowiedzą, że są gotowe do zatwierdzenia transakcji menedżer transakcji w fazie drugiej do wszystkich systemów wysyła rozkaz zatwierdzenia transakcji (tą sytuację ilustruje rysunek 4). Po zatwierdzeniu lub wycofaniu transakcji sterowanie wraca do aplikacji.



Rysunek 3 Dwufazowy protokół zatwierdzenia transakcji – wycofanie transakcji



Rysunek 4 Dwufazowy protokół zatwierdzenia transakcji – zatwierdzenie transakcji

W rzeczywistych implementacjach, gdzie mamy do czynienia z systemami rozproszonymi w sensie logicznym lub fizycznym zachodzi szereg warunków brzegowych, z którymi zarówno systemy i menedżer transakcji musi sobie radzić. Przede wszystkim występuje szereg możliwości wystąpienia awarii:

- jednego z systemów,
- komunikacji między systemem a menedżerem transakcji,
- wreszcie samego menedżera transakcji.

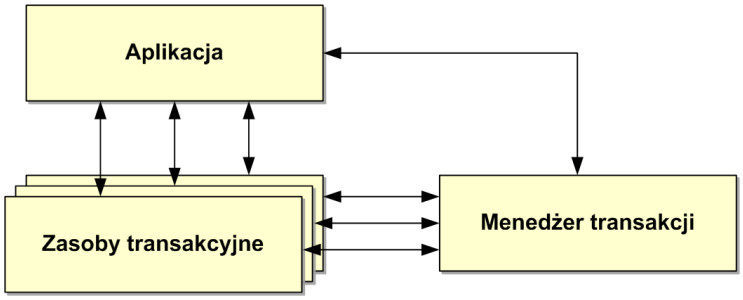
System transakcyjny musi sobie z tymi problemami umieć radzić. Temat związany z odtwarzaniem systemów po awarii jest bardzo obszerny i nie mamy tutaj miejsca na jego choćby skromne omówienie. Warto według mnie zwrócić uwagę jedynie na to, że w większości przypadków przywrócenie systemu do działania po awarii może następować automatycznie. Jedynie część specyficznych awarii związanych z awarią samego menedżera transakcji uniemożliwia automatyczne odtworzenie i wymaga interwencji operatora.

Istotą dwufazowego protokołu zatwierdzania transakcji jest to, że jeżeli w pierwszej fazie system zgłosił gotowość do zatwierdzenia transakcji, to w drugiej fazie nie może się już z tej decyzji wycofać. Takie zachowanie musi gwarantować implementacja systemu uczestniczącego w transakcji rozproszonej. Jest to więc swoistego rodzaju kontrakt między menedżerem transakcji rozproszonej a systemem w niej uczestniczącym.

Uczestnicy transakcji rozproszonej

Architektura systemu, w którym realizowane są transakcje rozproszone składa się z trzech głównych elementów (patrz rysunek 5):

- Aplikacji, czyli tego co zazwyczaj piszemy jako deweloperzy systemów.
- Zasobów transakcyjnych, z których nasza aplikacja korzysta (np. baz danych).
- Menedżera transakcji, który zarządza transakcją rozproszoną w imieniu aplikacji (zwykle dostarczany przez serwer aplikacji).



Rysunek 5 Uczestnicy transakcji rozproszonej

Każdy z tych elementów ma swoje specyficzne zadania do zrealizowania. Zacznijmy od zadań, które ciążą na aplikacji:

- Aplikacja (a więc my) przede wszystkim zarządza zakresem transakcji, czyli decyduje gdzie transakcja ma się zacząć, a gdzie zakończyć.
- Przeprowadza operacje na zasobach transakcyjnych, takich jak bazy danych, systemy kolejkowania czy transakcyjne systemy plików.

Przyjrzyjmy się teraz zadaniom menedżera transakcji:

- Menedżer transakcji przede wszystkim tworzy transakcję i zarządza kontekstem transakcji.
- Kojarzy również transakcję z zasobami w niej uczestniczącymi.
- Wreszcie prowadzi operacje zatwierdzania lub wycofywania transakcji bazując na dwufazowym protokole zatwierdzania.

Na koniec zadania zasobów transakcyjnych:

- Podstawową ich rolą jest umożliwienie aplikacji operowanie na danych, które przechowują.
- Dodatkowo zasoby transakcyjne muszą umieć współpracować z menedżerem transakcji, w szczególności w ramach dwufazowego protokołu zatwierdzania.

Kilka zdań wyjaśnienia należy się pojęciu kontekstu transakcji (*transaction context*). Kontekst transakcji to nic innego jak bieżący stan transakcji, w szczególności identyfikator transakcji oraz informacje o zasobach uczestniczących w transakcji. W czasie trwania transakcji jej kontekst przekazywany jest poszczególnym uczestnikom transakcji, co jest znane pod pojęciem propagacji kontekstu transakcji.

Na tym chciałbym zakończyć ogólne wprowadzenie do systemów transakcyjnych. Więcej szczegółów można znaleźć w materiałach pomocniczych [3,5]. Teraz przejdźmy do omówienia jak to jest zrealizowane w serwerze JEE.

Transakcje w środowisku serwera aplikacyjnego JEE

Zacznijmy od omówienia modeli transakcji jakie serwer aplikacji udostępnia deweloperem piszącym aplikacje. W praktyce mamy możliwość pisania systemów transakcyjnych na trzy różne sposoby.

Transakcje lokalne. Serwer aplikacji umożliwia wykonywania operacji w sposób transakcyjny na pojedynczym zasobie, np. na bazie danych. Przy czym zarządzanie transakcjami realizuje programista korzystając z właściwości konkretnego zasobu, np. transakcje w obrębie pojedynczej bazy danych realizujemy z poziomu API zdefiniowanego przez specyfikację JDBC. Rola serwera aplikacji ogranicza się tutaj jedynie do udostępnienia zasobów, na których operuje aplikacja, np. źródła danych do bazy danych (`javax.sql.DataSource`).

Transakcje zarządzane przez programistę z wykorzystaniem interfejsów zdefiniowanych przez specyfikację JTA. Są to takie transakcje, w których programista pisząc kod, jawnie określa początek i koniec transakcji korzystając z JTA. Musi również umieć obsłużyć szereg sytuacji brzegowych, czy wyjątkowych związanych z korzystaniem z tych interfejsów.

Transakcje zarządzane przez kontener, tzw. deklaratywne. W tym przypadku programista tworzy komponenty EJB i w deskryptorze komponentów określa jakie ma być zachowanie transakcyjne poszczególnych metod. Całością obsługi transakcji zajmuje się serwer aplikacji, a ściślej kontener komponentów EJB we współpracy z menedżerem transakcji.

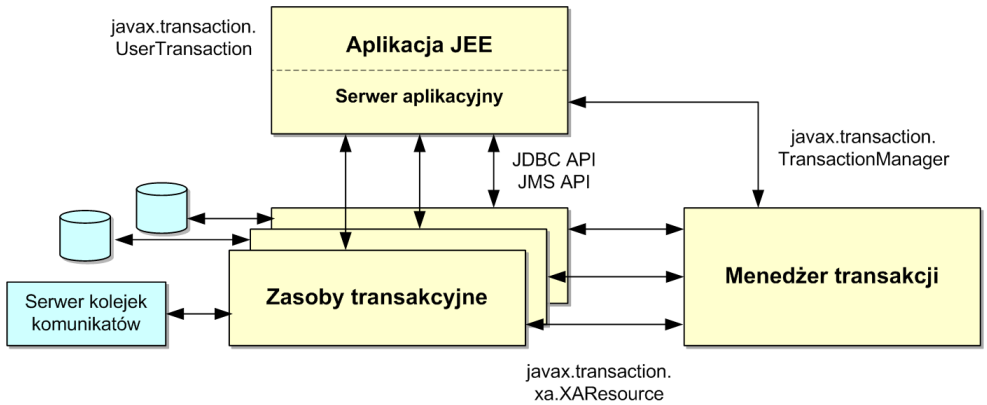
Transakcje lokalne oraz transakcję zarządzane przez kontener omówię w oddzielnych artykułach. Natomiast w tym artykule przedstawię dokładniej transakcje zarządzane przez programistę, które moim zdaniem najlepiej ilustrują wsparcie serwera aplikacji w tworzeniu systemów transakcyjnych.

Środowisko transakcyjne dla aplikacji JEE jest określone przez dwie specyfikacje:

- **Java Transaction API (JTA)** - definiuje ona sposób zarządzania transakcjami z punktu widzenia programisty. Określa również sposób współpracy z zasobami uczestniczącymi w transakcji rozproszonej (jest to odwzorowanie fragmentu standardu X/Open DTP - XA Interface).
- **Java Transaction Service (JTS)** - określa sposób implementacji menedżera transakcji (w szczególności wspierającego JTA), aczkolwiek specyfikacja JEE nie wymaga, żeby JTA było koniecznie implementowane w postaci JTS. Na rynku istnieje sporo implementacji JTA nie korzystających z JTS. JTS jest wymagane, jeśli myślimy o współpracy menedżerów transakcji w środowisku rozproszonym (transakcja rozproszona pomiędzy kilkoma serwerami aplikacji różnych dostawców). JTS jest tak naprawdę mapowaniem CORBA Object Transaction Service na język Java.

Najłatwiej zrozumieć związek pomiędzy tymi specyfikacjami przez analogię. JTA ma się mniej więcej tak do JTS jak specyfikacja JDBC do sterownika do bazy danych. Z punktu widzenia programisty praktyczne znaczenie ma JTA i tym będę się wyłącznie zajmował.

Przyjrzyjmy się teraz interfejsom jakie w serwerze aplikacji używane są do współpracy poszczególnych elementów uczestniczących w przetwarzaniu transakcji. Szczegóły przedstawiłem na rysunku 6.



Rysunek 6 Uczestnicy transakcji rozproszonych w środowisku serwera aplikacji JEE

Dla aplikacji pracującej w serwerze aplikacji, która chce zarządzać transakcją został przygotowany interfejs `javax.transaction.UserTransaction`. Wywołania tego interfejsu serwer aplikacji deleguje do interfejsu `javax.transaction.TransacationManager`, który stanowi reprezentację menedżera transakcji w serwerze aplikacji (implementacja tego interfejsu stanowi serce menedżera transakcji).

Aplikacja z poszczególnymi zasobami transakcyjnymi komunikuje się przez interfejsy specyficzne dla danych zasobów, np. przez interfejs JDBC albo JMS. Zachowanie transakcyjne zasobów jest przezroczyste z punktu widzenia aplikacji. Oczywiście sterowniki do zasobów muszą być odpowiednio zaimplementowane, ale zadanie to spoczywa na dostawcy sterownika, a nie na programiście aplikacji.

Pozostaje jeszcze interfejs `XAResource` za pomocą którego odbywa się komunikacja między menedżerem transakcji a zasobami transakcyjnymi. Aby móc uczestniczyć w transakcji rozproszonej zasoby te muszą implementować `javax.transaction.xa.XAResource`. Interfejs ten stanowi kontrakt określający sposób współpracy menedżera transakcji z zasobami transakcyjnymi.

Jeżeli używamy transakcji JTA **ważne jest upewnienie się, czy zasoby na których operujemy implementują `XAResource`** oraz czy są odpowiednio skonfigurowane w serwerze. W praktyce dość często można zaobserwować przypadki korzystania z baz danych w transakcjach rozproszonych z nieprawidłowo skonfigurowanym zasobem. Wynika to z tego, że zazwyczaj sterowniki JDBC w jednym fizycznym archiwum `jar` zawierają zarówno klasy implementujące zwykły dostęp jak i przystosowany do pracy w transakcji rozproszonej. Łatwo zatem o pomyłkę, zwłaszcza, że serwery nie ostrzegają przed użyciem zasobu nie będącego XAResource'em w transakcji JTA. Jest to zresztą zrozumiałe, gdyż nie wszystkie zasoby, na których operujemy są w stanie uczestniczyć w takiej transakcji - np. wysyłanie poczty elektronicznej.

Zarządzanie transakcjami za pomocą interfejsu UserTransaction

Przejdźmy do omówienia najważniejszego interfejsu z punktu widzenia programisty zainteresowanego zarządzaniem transakcjami. W serwerze aplikacji jest to interfejs `UserTransaction`. Jest to skromny interfejs o następujących metodach (pominąłem specyfikację wyjątków, które mogą być rzucone przez te metody):

```
void begin();
void commit();
void rollback();

void setRollbackOnly();
int getStatus();
void setTransactionTimeout(int seconds);
```

Jak widać interfejs składa się z dwóch grup metod. Pierwszej pozwalającej zarządzać zakresem transakcji, czyli rozpocząć transakcję (`begin`) oraz ją zakończyć zatwierdzając (`commit`) lub wycofując (`rollback`). Drugiej niejako pomocniczej pozwalającej sterować pewnymi elementami transakcji a także uzyskać informacje o aktualnym stanie transakcji.

Pierwsza grupa metod moim zdaniem nie wymaga szerszego komentarza. Natomiast druga grupa metod jest bardziej intrygująca i wymagająca dodatkowych objaśnień ponad to, co można wyczytać z suchego Java Transaction API.

Zacznijmy od metody `setRollbackOnly`. Pozwala ona na oznaczenie transakcji do wycofania. Oznacza to, że jedynym dopuszczalnym działaniem, które kończy transakcję jest jej wycofanie. Próba zatwierdzenia takiej transakcji zakończy się wyjątkiem. Zwróćmy uwagę na to, że metoda ta nie przerywa działania naszego programu - ustawia jedynie odpowiedni status transakcji. Zazwyczaj używamy tej metody w przypadku, gdy z pewnych warunków biznesowych lub współpracy z innymi systemami w sposób nietransakcyjny wynika, że musimy transakcję wycofać, ale powinniśmy wykonać jeszcze szereg innych działań (czyli nie przerywać wykonania naszego programu).

Z kolei za pomocą metody `getStatus` możemy zbadać w jakim obecnie stanie znajduje się nasza transakcja. Specyfikacja JTA za pomocą stałych z interfejsu `javax.transaction.Status` definiuje szereg stanów, w których może znaleźć się transakcja:

```
NO_TRANSACTION
ACTIVE
PREPARING
PREPARED
COMMITTED
COMMITTING
ROLLING_BACK
ROLLEDBACK
MARKED_ROLLBACK
UNKNOWN
```

Większość z tych stanów związana jest z realizacją dwufazowego protokołu zatwierdzania transakcji i jest wykorzystywana wewnętrznie przez menedżer transakcji. Z praktycznego punktu widzenia dla programisty istotne są poniższe stany:

- `ACTIVE` - oznaczający, że transakcja trwa (została rozpoczęta).
- `NO_TRANSACTION` - oznaczający, że z aktualnym wątkiem nie jest związana żadna transakcja. Status ten można wykorzystać do zweryfikowania, czy w metodach, które wymagają do prawidłowego działania rozpoczętej wcześniej transakcji, została ona faktycznie rozpoczęta.
- `MARKED_ROLLBACK` - oznaczająca, że transakcja została oznaczona do wycofania. Status ten sprawdzamy przy samodzielnym zarządzaniu transakcjami, aby podjąć decyzję o zatwierdzeniu czy wycofaniu transakcji. Często również na jego podstawie możemy zaprezentować użytkownikowi odpowiednią informację o rezultacie transakcji, którą próbował wykonać.

Na koniec omawiania interfejsu `UserTransaction` kilka słów o timeout'ach transakcji. Metoda `setTransactionTimeout(...)` pozwala na ustawienie maksymalnego dopuszczalnego czasu trwania transakcji. Jednak jej działanie jest chyba najmniej oczywiste ze wszystkich metod związanych z zarządzaniem transakcjami (nie tylko w obrębie `UserTransaction`). Przyjrzyjmy się jej dokładnie:

- Po pierwsze z ustawienia maksymalnego czasu trwania transakcji wcale nie wynika, że transakcja zakończy się po tym czasie. Co więcej, nie możemy oczekiwać, że nawet w dowolnym momencie po przekroczeniu tego czasu działanie naszej aplikacji zostanie przerwane. W rzeczywistości po przekroczeniu tego czasu serwer aplikacji jedynie oznacza tą transakcję do wycofania (`MARKED_ROLLBACK`). Tak więc transakcja wykonuje się cała (nawet jeśli trwałaby godzinami), a dopiero na końcu jest wycofywana z powodu przekroczenia tego czasu. Jest to zachowanie sprzeczne z zazwyczaj wyrobioną intuicją programisty, która podpowiada, że działanie aplikacji powinno zostać przerwane (tak jak jest to na przykład przy obsłudze socket'ów). Takie zachowanie transakcji rodzi wiele problemów, które szerzej będę omawiał w oddzielnym artykule.
- Po drugie należy pamiętać, że jeśli samodzielnie ustawiamy maksymalny czas trwania transakcji musimy to zrobić przed wywołaniem metody `begin`.
- Po trzecie należy pamiętać, że w serwerze aplikacji zawsze jest zdefiniowany jakiś domyślny czas trwania transakcji. Jest on specyficzny dla danego serwera, zazwyczaj jest ustawiony na poziomie 60-120 sekund. Dość łatwo o tym zapomnieć. Dlatego jeśli aplikacja wydaje się działać poprawnie, ale dane w bazie danych się nie pojawiają proponuję sprawdzić, czy czasami transakcja nie została wycofana z powodu przekroczenia maksymalnego czasu jej trwania.

Przyjrzyjmy się teraz pierwszemu przykładowi wykorzystania interfejsu do zarządzania transakcjami z poziomu prostej aplikacji serwlet'owej operującej na dwóch bazach danych w ramach pojedynczej transakcji rozproszonej. Pominąłem tutaj właściwą obsługę wyjątków i sytuacji brzegowych, którą omówię w kolejnym przykładzie.

```
...
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.servlet.ServletException;
import javax.sql.DataSource;
import javax.transaction.UserTransaction;
...

public class SimpleTransactionDemoServlet extends HttpServlet {
    protected void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
        ...
        try {
            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");

            doSomethingInFirstDatabase(ds1);
            doSomethingInSecondDatabase(ds2);

            ut.commit();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
        ...
    }
}
```

Przeanalizujemy ten krótki przykład. Cała zabawa rozpoczyna się od pobrania z JNDI obiektu `UserTransaction`. W większości serwerów aplikacji znajduje się on pod kluczem `java:comp/UserTransaction`, ale są serwery, które udostępniają go pod innym kluczem (np. WebSphere v4 pod kluczem `jta/usertransaction`). Dlatego należy się najpierw upewnić w dokumentacji serwera, którego używamy, gdzie w JNDI można znaleźć obiekt `UserTransaction`.

Następnie rozpoczynamy transakcję za pomocą operacji `begin`, pobieramy skonfigurowane wcześniej źródła danych i wykonujemy działania na bazach danych.

Po wykonaniu operacji na bazach danych zatwierdzamy transakcję, dane trafiają na trwałe do odpowiednich baz danych. Jednym słowem lekko, łatwo i przyjemnie. Niestety nie do końca, jak to będę starał się pokazać na kolejnym przykładzie.

Niemal identycznie wygląda zarządzanie transakcjami w przypadku pisania komponentów EJB, które samodzielnie zarządzają transakcjami (*bean managed transcation*s). Jedyną różnicą jest sposób dostępu do interfejsu `UserTransaction`, zamiast pobierać go bezpośrednio z JNDI korzystamy z metody `getUserTransaction()` z interfejsu `javax.ejb.EJBContext`.

Obsługa wyjątków i sytuacji brzegowych przy samodzielnym zarządzaniu transakcjami

Niestety, przedstawiony wcześniej przykład zarządzania transakcjami poza swoją zachęcającą prostotą ma jedną zasadniczą wadę. Mianowicie w praktyce nie działa, gdyż nie obsługuje sytuacji brzegowych i wyjątkowych, które mogą się wydarzyć podczas wykonywania poszczególnych operacji. Pokazuje jedynie tzw. ścieżkę pozytywną, czyli taką, w której wszystko co sobie zamierzyliśmy wykonało się poprawnie.

Przyjrzyjmy się teraz w jaki sposób można podejść do pełnej obsługi procesu zarządzania pojedynczą transakcją:

```
Context ctx = null;
UserTransaction ut = null;

try {
    ctx = new InitialContext();
    ut = (UserTransaction)ctx.lookup("java:comp/UserTransaction");;
} catch (NamingException e) {
    throw new RuntimeException(e);
}

try {
    ut.begin();
} catch (NotSupportedException e) {
    throw new RuntimeException(e);
} catch (SystemException e) {
    throw new RuntimeException(e);
}

boolean ok = false;
Exception exception = null;

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

        doSomethingInFirstDatabase(ds1);
        doSomethingInSecondDatabase(ds2);

        ok = true;
    } catch (Exception e) {
        exception = e;
    }
} finally {
    Exception cleanupException = null;
    if (ok) {
        try {
```

```
            int transactionStatus = ut.getStatus();
            if (transactionStatus == Status.STATUS_ACTIVE) {
                ut.commit();
            } else if (transactionStatus == Status.STATUS_MARKED_ROLLBACK) {
                ut.rollback();
            } else {
                if (log.isWarnEnabled()) {
                    log.warn("Unexpected transaction status: " + ut.getStatus());
                }
            }
        } catch (SystemException e) {
            cleanupException = e;
        } catch (RollbackException e) {
            cleanupException = e;
        } catch (HeuristicMixedException e) {
            cleanupException = e;
        } catch (HeuristicRollbackException e) {
            cleanupException = e;
        }
    } else {
        try {
            int transactionStatus = ut.getStatus();
            if (transactionStatus == Status.STATUS_ACTIVE || transactionStatus ==
Status.STATUS_MARKED_ROLLBACK) {
                ut.rollback();
            } else {
                if (log.isWarnEnabled()) {
                    log.warn("Unexpected transaction status: " + ut.getStatus());
                }
            }
        } catch (SystemException e) {
            cleanupException = e;
        }
    }

    if (cleanupException != null) {
        if (exception != null) {
            throw new RuntimeException(doubleErrorMessage(exception), cleanupException);
        } else {
            throw new RuntimeException(cleanupException.getMessage(), cleanupException);
        }
    } else if (exception != null) {
        throw new RuntimeException(exception.getMessage(), exception);
    }
}
```

Prześledźmy kolejno najważniejsze elementy tej obsługi.

- Pierwszą rzeczą, która może się nie udać to pobranie obiektu `UserTransaction`. W tym przypadku nie pozostaje nam nic innego jak przerwanie działania aplikacji i rzucenie wyjątku czasu wykonania.
- Następną rzeczą, która może się nie udać, jest rozpoczęcie transakcji. Wbrew pozorom nie jest to rzadki problem. Najczęściej wynika on z próby rozpoczęcia transakcji w sytuacji, gdy już jakaś inna transakcja jest przypisana do wątku. Zazwyczaj otrzymujemy wtedy komunikat o tym, że serwer aplikacji nie wspiera transakcji zagnieżdżonych. W tym przypadku również niewiele możemy zrobić poza rzuceniem `RuntimeException`, gdyż błąd wynika albo bezpośrednio z błędu w naszej aplikacji albo zawodzi serwer aplikacji (co wbrew pozorom nie jest znowu takie wyjątkowe).
- Jeśli już uporamy się z rozpoczęciem transakcji, musimy obsłużyć sytuację, w której podczas wykonywania operacji biznesowych w ramach rozpoczętej transakcji wystąpił wyjątek. Ponieważ kontrakt współpracy z serwerem aplikacji narzuca nam konieczność zakończenia transakcji (zatwierdzenia lub wycofania) to w zasadzie jedynym wyjściem jest przechwycenie wyjątku, jego zapamiętanie, obsłużenie zakończenia transakcji i na końcu jego ponowne rzucenie dalej. Jeśli nie zrobimy tego w ten sposób transakcja pozostanie przypięta do wątku i będziemy mieli w kolejnych żądaniach wyjątek w rodzaju "Nested transactions not supported".
- Przejdźmy teraz do najtrudniejszej części, czyli obsługi zamykania transakcji. Do obsłużenia mamy trzy główne przypadki:
    - Operacje na zasobach przebiegły bez wyjątku, sama transakcja ma status `ACTIVE`. W takim przypadku zatwierdzamy transakcję (`commit`).

    - Operacje na zasobach przebiegły bez wyjątku, ale transakcja ma status `MARKED_ROLLBACK`. W takim przypadku musimy wycofać transakcję (`rollback`).
    - Podczas wykonywania operacji na zasobach wystąpił wyjątek. Wtedy niezależnie od tego czy transakcja jest w stanie `ACTIVE` czy `MARKED_ROLLBACK` musimy wycofać transakcję (`rollback`).
- Niestety podczas zatwierdzania lub wycofywania transakcji również może wystąpić wyjątek. W takim przypadku wyjątek ten musimy zapamiętać (w przykładzie zmienna `cleanupException`) a następnie na samym końcu obsługi go rzucić. Szczególna sytuacja zachodzi wówczas, gdy zarówno podczas wykonywania operacji biznesowych jak i podczas zamykania transakcji pojawią się wyjątki. Mamy wówczas do czynienia z tzw. podwójnym błędem. Należy zwrócić uwagę, żeby błędy te wzajemnie się nie maskowały powodując trudne do odszyfrowania problemy. Dlatego w przykładzie w takim przypadku w sposób specjalny konstruowany jest komunikat wyjątku tak, aby zawierał informację o wyjątku pierwotnym (metoda `doubleErrorMessage`). Przykładowa implementacja tej metody może wyglądać następująco:

```
public String doubleErrorMessage(Throwable exception) {
    StringWriter sw = new StringWriter();
    PrintWriter pw = new PrintWriter(sw);
    exception.printStackTrace(pw);
    pw.close();
    return "double error; original cause:\n[\n" +sw.toString() + "]";
}
```

Chwilę uwagi chciałbym jeszcze poświęcić wyjątkom typu `Heuristic...Exception`. Kiedy się pierwszy raz z nimi zetknąłem wydawały mi się czymś magicznym, nie bardzo przystającym do rzeczywistości. Jednak po dokładniejszym rozeznaniu się tematyce systemów transakcyjnych uświadomiłem sobie, że nie są one czymś nadzwyczajnym. Sygnalizują jedynie pewne sytuacje wyjątkowe, które mogą się wydarzyć na styku współpracy menedżera transakcji i menedżera konkretnego zasobu w ramach dwufazowego protokołu zatwierdzania. I tak na przykład wyjątek `HeuristicRollbackException` możemy zobaczyć, jeśli w pierwszej fazie zasób transakcyjny zgłosił gotowość zatwierdzenia transakcji, ale pomiędzy zakończeniem pierwszej fazy a rozpoczęciem drugiej samodzielnie podjął decyzję o wycofaniu zmian, które zaszły w tej transakcji. Dokładne omówienie przypadków związanych z tymi wyjątkami można znaleźć w [4].

Jak można zauważyć kompletna obsługa samodzielnego zarządzania transakcjami wymaga niemało wysiłku, a powstały kod jest niezbyt czytelny (mizerny stosunek prawdziwej logiki biznesowej do logiki związanej z obsługą wyjątków). Na szczęście stosunkowo łatwo można go zamknąć w pojedynczej klasie usługowej i nie powielać w innych częściach aplikacji.

Bezpośrednie korzystanie z menedżera transakcji

Korzystając jedynie z interfejsu `UserTransaction` nie jesteśmy w stanie wykorzystać wszystkich możliwości jakie daje nam model transakcji zdefiniowany przez serwer aplikacji JEE. W szczególności nie możemy zawieszać oraz wznawiać wykonania bieżącej transakcji.

W praktyce taka możliwość czasami się przydaje. Wyobraźmy sobie aplikację bankowości elektronicznej, w której chcielibyśmy logować do oddzielnej bazy danych wszystkie próby wykonywanych przez użytkowników transakcji niezależnie od tego czy zakończyły się one zatwierdzeniem czy wycofaniem. Jeśli zapis do logów umieścimy w tej samej transakcji to system będzie się zachowywał prawidłowo jedynie w przypadku zatwierdzenia transakcji. W przypadku jej wycofanie w bazie danych odpowiedzialnej za logi nic nie zobaczymy.

Taką funkcjonalność można zrealizować odwołując się bezpośrednio do menedżera transakcji, co jest jak najbardziej dopuszczane przez specyfikację JTA. Tak jak wcześniej wspomniałem w serwerze aplikacji menedżer transakcji jest reprezentowany przez interfejs `TransactionManager`, który poza metodami, które znajdziemy w interfejsie `UserTransaction` posiada jeszcze dwie dodatkowe:

- `suspend()` - pozwalającą zawiesić bieżącą transakcję - metoda zwróci wtedy obiekt `Transaction, który reprezentuje zawieszoną transakcję`.
- `resume(Transaction t)` - pozwalający wznowić zawieszoną wcześniej transakcję.

Podobnie jak w przypadku `UserTransaction` obiekt `TransactionManager` znajduje się w JNDI pod odpowiednim kluczem. Klucz ten jest specyficzny dla danego serwera aplikacji.

Poniżej przedstawiłem przykład realizujący opisaną wyżej funkcjonalność. Dla jego czytelności pominąłem obsługę wyjątków.

```
```

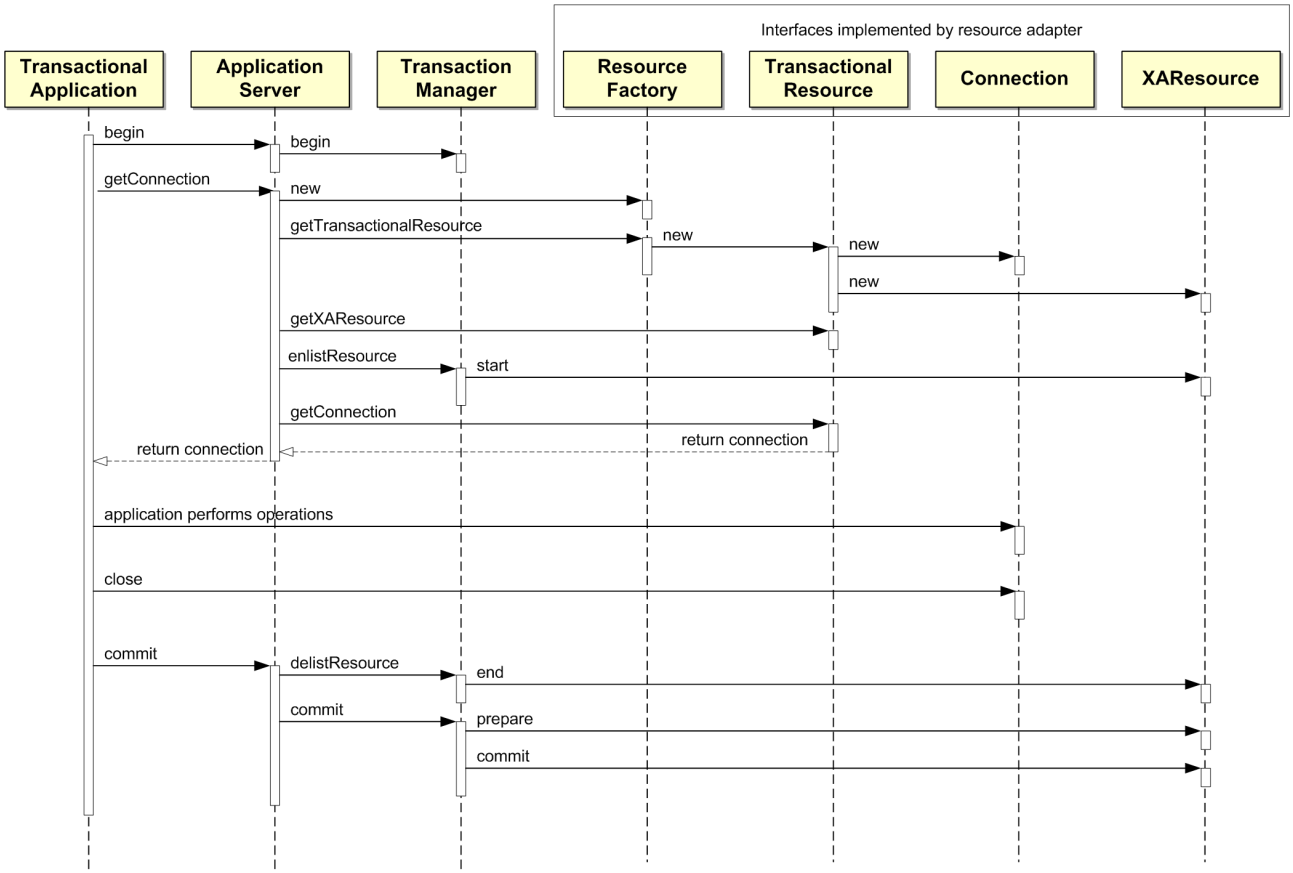
```
Context ctx = new InitialContext();
UserTransaction ut = (UserTransaction)ctx.lookup("java:comp/UserTransaction");;
TransactionManager tm = (TransactionManager)ctx.lookup("java:comp/UserTransaction");

ut.begin();

DataSource ds1 = (DataSource)ctx.lookup("java:comp/env/jdbc/TransactionDemoDS1");
// wykonujemy operacje na pierwszej bazie danych (ds1)

Transaction t = tm.suspend();
DataSource dslog = (DataSource)ctx.lookup("java:comp/env/jdbc/TransactionDemoDSLog");
// wykonujemy niezależną od otworzonej wcześniej transakcji
// operację na bazie danych (dslog)
tm.resume(t);

DataSource ds2 = (DataSource)ctx.lookup("java:comp/env/jdbc/TransactionDemoDS2");
// wykonujemy operacje na drugiej bazie danych (ds2)

ut.commit(); // lub ut.rollback() jeśli coś poszło nie tak
```

Jako ciekawostkę można zauważyć, że zarówno obiekt `UserTransaction` jak i `TransactionManager` są pobierane z pod tego samego klucza JNDI (`java:comp/UserTransaction`). Czy to jakaś magiczna sztuczka serwera aplikacji? Otóż nie, przykład ten testowałem w serwerze aplikacji JOnAS, w którym jako menedżer transakcji używany jest komponent JOTM. W tej konkretnej implementacji menedżera transakcji obiekt, który znajduje się pod wskazanym kluczem w JNDI po prostu implementuje oba interfejsy `UserTransaction` i `TransactionManager`. Jeśli spojrzymy na te oba interfejsy i pokrywające się w nich metody to takie rozwiązanie wydaje się zrozumiałe.

Powyższą funkcjonalność można również zrealizować za pomocą komponentów EJB odpowiednio sterując zachowaniem transakcyjnym poszczególnych ich metod. Przedstawię taki przykład w jednym z kolejnych artykułów.

Przebieg transakcji w serwerze aplikacji

Jako podsumowanie omawianych w tym artykule zagadnień chciałbym przedstawić całościowy obraz przebiegu pojedynczej transakcji w serwerze aplikacji JEE. Szczegółowe interakcje pomiędzy poszczególnymi elementami uczestniczącymi w transakcji przedstawia rysunek 7. Obrazuje on przebieg transakcji zarządzanej przez programistę. Przebieg transakcji zarządzanej przez kontener różni się tylko tym, że rozpoczęcie i zakończenia transakcji zamiast aplikacji wykonuje serwer aplikacji.



Rysunek 7 Przebieg transakcji w serwerze aplikacji JEE

Zacznijmy od omówienie poszczególnych obiektów uczestniczących w transakcji:

- `Transactional Application` - reprezentuje naszą aplikację, z poziomu której zarządzamy transakcją.
- `Application Server` - udostępnia środowisko zarządzania transakcjami przez udostępnienie aplikacji interfejsu `UserTransaction`.
- `Transaction Manager` - realizuje faktyczne zarządzanie transakcjami w imieniu naszej aplikacji.
- `Resource Adapter` - reprezentuje zasób transakcyjny, na którym operuje nasza aplikacja. Zasób transakcyjny z punktu widzenia zarządzania transakcjami przez serwer aplikacji musi składać się z następujących elementów:
    - `Resource Factory` - czyli obiektu umiejącego powołać do życia zasób transakcyjny.
    - `Transactional Resource` - będący podstawowym interfejsem reprezentującym zasób transakcyjny.
    - `Connection` - interfejs reprezentujący fizyczne połączenie do zasobu transakcyjnego (np. połączenie do bazy danych, serwera JMS itp.).
    - `XAResource` - interfejs umożliwiający realizację dwufazowego protokołu zatwierdzania transakcji.

Prześledźmy teraz interakcje jakie zachodzą pomiędzy poszczególnymi elementami.

- Nasza aplikacja rozpoczyna transakcję wołając `begin` na interfejsie `UserTransaction` pobranym z serwera aplikacji. Powoduje to utworzenie nowej transakcji w menedżerze transakcji (powstaje nowy kontekst transakcji) oraz skojarzenie wątku aktualnie wykonującego kod naszej aplikacji z nowoutworzoną transakcją.
- W następnym kroku nasza aplikacja musi uzyskać połączenie do zasobu na którym chce operować. Odbywa się to przez wywołanie metody `getConnection` na obiekcie, który reprezentuje zasób transakcyjny w serwerze aplikacji i który jest przez niego udostępniany (w przypadku bazy danych jest to obiekt `DataSource`). Żądanie przez aplikację dostępu do zasobu uruchamia w serwerze aplikacji całą lawinę działań:
    - Tworzona jest fabryka zasobu, a następnie za jej pomocą tworzony jest zasób transakcyjny, co w konsekwencji prowadzi do utworzenia fizycznego połączenia i obiektu `XAResource`.
    - Po pobraniu zasobu transakcyjnego jest on przypisywany do danej transakcji (`enlistResource`) oraz informowany, że rozpoczyna uczestnictwo w transakcji rozproszonej (metoda start).
    - Dopiero na samym końcu zwracany jest do aplikacji obiekt reprezentujący fizyczne połączenie do zasobu transakcyjnego.
- Następnie nasza aplikacja operuje na zasobie transakcyjnym. Po czym zamyka połączenie. Zazwyczaj wywołanie metody `close` w rzeczywistości nie wywołuje żadnego fizycznego skutku (np. zamknięcia socket'a

połączenia do bazy danych). Stanowi tylko informację dla serwera aplikacji, że aplikacja zakończyła swoje działanie na danym połączeniu.

- Po wykonaniu operacji nasza aplikacja znów za pośrednictwem `UserTransaction` zaczyna zatwierdzanie transakcji (wywołanie metody `commit`), co uruchamia następującą sekwencję działań w serwerze aplikacji:
    - Za pośrednictwem menedżera transakcji zasób transakcyjny jest informowany, że w danej transakcji nie będą już wykonywane żadne działania poza zatwierdzeniem lub wycofaniem transakcji (metoda `delistResource` w menedżerze transakcji, która z kolei wywołuje metodę `end` na `XAResource`).
    - Następnie również za pośrednictwem menedżera transakcji realizowane jest dwufazowe zatwierdzanie transakcji, co na rysunku widoczne w postaci operacji `prepare` i `commit` na obiekcie `XAResource`.

Jak widać całość nie jest taka banalna, jakby się mogło wydać po zapoznaniu się z interfejsem `UserTransacion`. Na szczęście prostota interfejsu `UserTransaction` skutecznie ukrywa przed programistą aplikacji złożone interakcje, które zachodzą we wnętrzu serwera aplikacji podczas obsługi transakcji. Niestety czasami rzeczywiste problemy, które nam się przytrafiają w konkretnych systemach korzystających często z bardzo różnych zasobów transakcyjnych, uruchomionych w różnych serwerach aplikacji (a co za tym idzie różniących się szczegółami implementacji menedżera transakcji) czy w złożonych środowiskach produkcyjnych mocno kontrastują z prostym wyglądem interfejsu `UserTransaction`. Aby pokonać te problemy znajomość szczegółów interakcji zachodzących we wnętrzu serwera aplikacji staje się niezbędna.

Na tym chciałbym zakończyć wprowadzenie w transakcje w serwerze aplikacji JEE. W następnym odcinku postaram się przedstawić szczegóły korzystania z baz danych i systemów kolejkowania w kontekście tworzenia systemów transakcyjnych na platformie JEE.

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] Jim Gray, Andreas Reuter, Transacations Processing: Concepts and Techniques, Morgan Kaufmann Publishers, 1993
[4] Mark Richards, Java Transaction Design Strategies, C4Media 2006
[5] Nuts and Bolts of Transaction Processing, http://www.theserverside.com/tt/articles/article.tss?l=Nuts-and-Bolts-of-Transaction-Processing