SOA – laboratorium nr 7

Temat*: JMS oraz Message-Driven Beans.*

Część teoretyczna.

Przesyłanie komunikatów to metoda komunikacji między elementami oprogramowania i aplikacjami. **JMS (Java Message Service)** udokumentowane na stronie *http://www.oracle.com/technetwork/java/jms/index.html* to API Javy umożliwiające aplikacjom tworzenie, wysyłanie, odbieranie i odczytywanie komunikatów.

Przesyłanie komunikatów różni się od innych standardowych protokołów, na przykład **RMI (Remote Method Invocation)** lub **HTTP (Hypertext Transfer Protocol)** w dwojaki sposób. Po pierwsze, w komunikacji pośredniczy serwer komunikatów, więc nie jest to bezpośrednia komunikacja dwukierunkowa. Po drugie, zarówno odbiorca, jak i nadawca muszą znać format komunikatu i jego cel, ale nic więcej. Różni się to od technologii ściślej wiążących odbiorcę i nadawcę, takich jak RMI, w których to aplikacja musi znać zdalnie wywoływane metody

Krótkie wprowadzenie do JMS

JMS definiuje niezależny od dostawcy (choć specyficzny dla Javy) zbiór interfejsów programistycznych do interakcji z systemami asynchronicznego przesyłania komunikatów. Takie rozwiązanie umożliwia rozproszoną komunikację luźno powiązanych elementów. Cały przesyła komunikatów to proces dwuetapowy: jeden komponent wysyła komunikat do celu, a drugi komponent odbiera go z serwera JMS.

JMS służy do asynchronicznej komunikacji(można synchronicznie odbierać komunikaty lecz jest to niezalecane) pomiędzy systemami za pomocą szeroko pojętych komunikatów(plików, preparowanych wiadomości itp). Pozwala tworzyć, wysyłać, otrzymywać i czytać te komunikaty.

W JMS istnieją dwa rodzaje celów — tematy i kolejki.

W modelu **punkt-punkt** komunikaty przesyłane są od producentów do konsumentów przy użyciu **kolejki**. Kolejka może mieć wielu odbiorców, ale konkretny komunikat otrzyma tylko jeden z nich. Pierwszy odbiorca otrzyma komunikat, a pozostali nawet się nie dowiedzą, że istniał.

Z drugiej strony, komunikat wysłany do **tematu** może być odczytany przez wielu odbiorców. Komunikat wysłany pod konkretny temat trafi do wszystkich konsumentów, którzy zgłosili chęć otrzymywania tego rodzaju komunikatów (zarejestrowali się). Nietrwały subskrybent może otrzymywać komunikaty opublikowane tylko wtedy, gdy jest **aktywny**. Subskrypcja nietrwała nie gwarantuje dostarczenia komunikatu lub też może dostarczyć komunikat więcej niż jeden raz. Subskrypcja trwała daje pewność, że konsument otrzymał komunikat dokładnie jeden raz.

Konsumpcja komunikatu, choć sam system JMS jest w pełni asynchroniczny, może odbywać się na dwa sposoby wskazane w specyfikacji JMS.

- **Synchroniczny** — subskrybent lub odbiorca jawnie pobiera komunikat z celu, wywołując metodę receive() dowolnej instancji MessageConsumer. Metoda receive()może zablokować działanie wątku, aż do momentu nadejścia komunikatu, lub też można wskazać maksymalny czas oczekiwania na komunikat.
- **Asynchroniczny** — w trybie asynchronicznym klient musi zaimplementować interfejs javax.jms.MessageListener i udostępnić metodę onMessage(). Gdy komunikat dotrze do celu, dostawca JMS dostarczy go do odbiorcy, wywołując kod metody onMessage.

Komunikat JMS składa się z nagłówka, właściwości i treści.

Nagłówki komunikatu to ściśle określone metadane opisujące komunikat, na przykład ustalające jego cel i pochodzenie. Właściwości to zestawy par klucz-wartość wykorzystywane do celów specyficznych dla aplikacji. Najczęściej używa się ich do szybkiej filtracji nadchodzących komunikatów. Treść komunikatu to właściwa zawartość komunikatu przesyłana od nadawcy do odbiorcy.

- ***Message Header***

  Predefiniowane header:

  - JMSDestination
    – JMSDeliveryMode
    – JMSMessageID
    – JMSTimestamp
    – JMSCorrelationID
    – JMSReplyTo
    – JMSRedelivered
    – JMSType

– JMSExpiration
– JMSPriority

- ***Message Properties***

- ***Message Body***

  Text message : -> javax.jms.TextMessage
  Object message : -> javax.jms.ObjectMessage.
  Bytes message : -> javax.jms.BytesMessage.
  Stream message : -> javax.jms.StreamMessage.
  Map message : -> javax.jms.MapMessage.

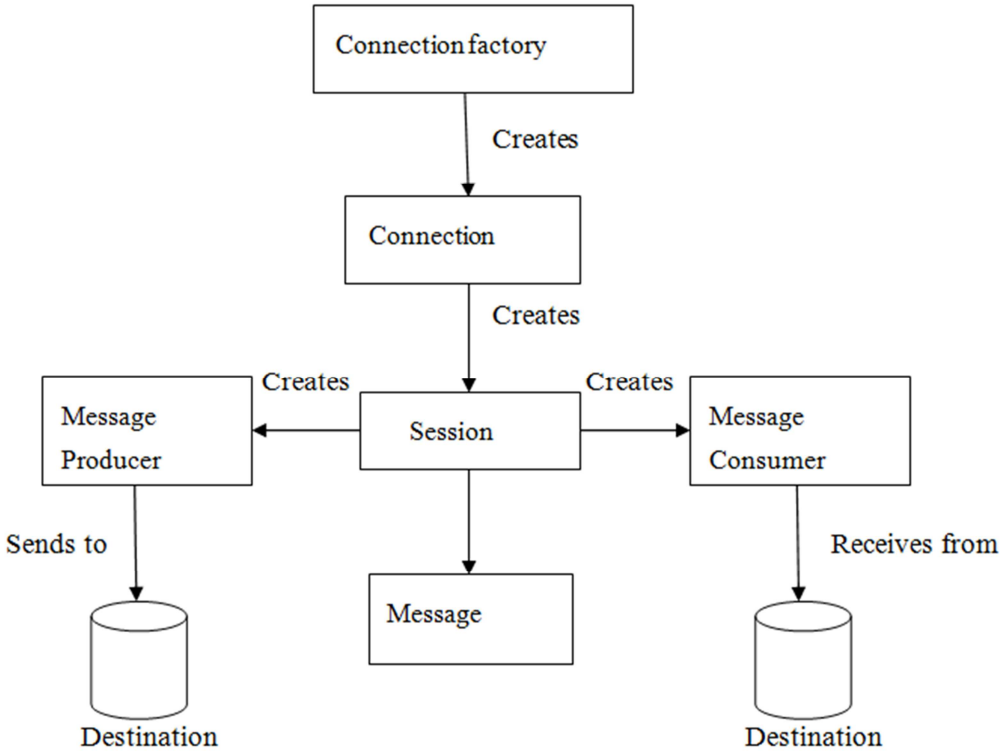
API JMS obsługuje dwa rodzaje dostarczania komunikatów, które określają, czy komunikaty zostaną zagubione, jeśli serwer JMS nie zadziała prawidłowo.

- Tryb **trwały** (stosowany domyślnie) instruuje dostawcę JMS, by podjął szczególne środki, żeby komunikat nie został utracony w przypadku błędu JMS. Dostawca JMS zapisuje otrzymany komunikat w trwałym magazynie danych.
- Tryb **nietrwały** nie wymaga od dostawcy JMS trwałego przechowywania komunikatu, więc nie gwarantuje jego dostarczenia w przypadku błędu serwera JMS.

JMS API:

Aplikacja JMS składa się z następujących elementów:

- obiektów administracyjnych — fabryk połączeń i celów,
- połączeń,
- sesji,
- producentów komunikatów,
- konsumentów komunikatów,
- komunikatów.



Obiekty zarządzające są tworzone przez JMS system administrator w JMS Provider poprzez użycie funkcjonalności aplikacji administracyjnej omawianej w lab 1. Oba obiekty są przechowywane w JNDI Directory oraz JNDI Registry.

Obiekt **fabryki połączeń** zawiera zbiór parametrów konfiguracyjnych zdefiniowanych przez administratora. Klient używa go do utworzenia połączenia z dostawcą JMS. Fabryka połączeń ukrywa przed klientem szczegóły specyficzne dla dostawcy, udostępniając wszystkie dane w postaci standardowych obiektów Javy.

```
QueueConnectionFactory queueConnFactory = (QueueConnectionFactory)
initialCtx.lookup ("PierwszaKolejka");

Queue purchaseQueue = (Queue) initialCtx.lookup ("Kolejka_nadawcza");

Queue returnQueue = (Queue) initialCtx.lookup ("Kolejka_odbiorcza");
```

Cel to komponent używany przez klienta do określenia docelowej lokalizacji dla tworzonych lub odbieranych komunikatów. W przypadku komunikacji typu **punkt-punkt (PTP)** cel nazywa się kolejką; w przypadku systemów **publikuj-subskrybuj (pub-sub)** cel nazywa się tematem.

Połączenie zawiera wirtualne połączenie z dostawcą JMS. Może ono reprezentować otwarte gniazdo TCP/IP między klientem i dostawcą usługi. Połączenie służy do utworzenia jednej lub wielu sesji.

Sesja to jednowątkowy kontekst dotyczący produkcji i konsumpcji komunikatów. Sesji używa się do tworzenia komunikatów po stronie producenta, konsumpcji komunikatów, a także obsługi samych komunikatów. Sesje szeregują wykonanie metody odbioru komunikatów i zapewniają obsługę transakcyjności, czyli wysyłanie lub odbieranie komunikatów jako niepodzielnej jednostki zadaniowej.

Producent komunikatów to obiekt utworzony przez sesję i używany do wysyłania komunikatów do konkretnej lokalizacji docelowej. W komunikacji punkt-punkt jest to obiekt implementujący interfejs QueueSender. W komunikacji publikuj-subskrybuj jest to obiekt implementujący interfejs TopicPublisher.

Konsument komunikatów to obiekt utworzony przez sesję. Służy do odbierania komunikatów wysłanych pod konkretny adres. Obiekt konsumenta umożliwia klientowi JMS wskazanie swojego zainteresowania konkretnym celem u dostawcy JMS. Dostawca JMS zarządza dostarczeniem komunikatu z celu do zarejestrowanych konsumentów. W komunikacji punkt-punkt jest to obiekt implementujący interfejs QueueReceiver. W komunikacji publikuj-subskrybuj jest to obiekt implementujący interfejs TopicSubscriber.

Tworzenie i wykorzystanie fabryk połączeń

Zadaniem fabryki połączeń jest przechowywanie parametrów połączenia umożliwiających tworzenie nowych połączeń JMS. Fabryka połączeń korzysta z **JNDI (Java Naming Directory Index)** i może być wyszukiwana przez lokalne oraz zdalne klienty, jeśli tylko obsługują odpowiednie parametry środowiska. Ponieważ fabrykę połączeń można w kodzie stosować wielokrotnie, jest to obiekt, który warto umieścić w pamięci podręcznej klienta zdalnego lub ziarna sterowanego komunikatami.

Definicje instancji fabryki połączeń znajdują się w konfiguracjach serwera nazwanych *full* i *full-ha*. Można je sprawdzić w ogólnej konfiguracji JMS, która jest dostępna w konsoli administracyjnej po przejściu do *Profile/Messaging Provider*.

Domyślnie dostępne są dwie fabryki połaczeń.

- InVmConnectionFactory — ta fabryka połączeń jest dostępna pod adresem java:/ConnectionFactory. Używa się jej, gdy serwer i klient stanowią część jednego procesu (czyli działają w jednej maszynie wirtualnej Javy).
- RemoteConnectionFactory — ta fabryka połączeń dotyczy sytuacji, w których połączenia JMS są zapewniane przez zdalny serwer. Do komunikacji używana jest Netty.

Jeśli chcemy zmienić ustawienia dowiązań JNDI, najłatwiej wykonać to w pliku konfiguracji serwera (na przykład w pliku *standalone-full.xml* dla trybu samodzielnego).

Fabrykę połączeń wstrzykuje się, tak jak każdy inny zasób Javy EE. Poniższy fragment ilustruje wstrzyknięcie fabryki połączeń do bezstanowego komponentu EJB.

```
@Stateless
public class SampleEJB
{@Resource(mappedName = "java:/ConnectionFactory")
private ConnectionFactory cf;
}
```

Aby skorzystać z podsystemu komunikatów, trzeba uruchomić serwer JBoss AS za pomocą pełnego profilu, który ten podsystem zawiera. Jeśli więc chce się uruchomić samodzielną instancję serwera zawierającą JMS, trzeba w konsoli systemowej użyć polecenia:
standalone.sh –c standalone-full.xml

Poza definicjami fabryk połączeń, trzeba się również nauczyć konfiguracji celów JMS (kolejek i tematów).
Konfigurację można przeprowadzić na kilka sposobów. Przejdziemy do zakładki *Profile* i wybierzemy podsystem *Messaging* z lewego panelu. Zaznaczymy element *Messaging Provider* i klikniemy element *View* z prawego panelu (patrz zrzut ekranu). Na górnej belce prawego panelu powinny pojawić się zakładki. Klikamy zakładkę *JMS Destinations*, która służy między innymi do konfiguracji celów JMS.

Klikamy przycisk *Add* i na następnym ekranie wpisujemy wymaganą nazwę celu i jej adres JNDI. Dodatkowo można wskazać następujące opcje.
- *Durable* — opcja ta umożliwia serwerowi JMS przechowanie komunikatu, w sytuacji gdy subskrybent jest tymczasowo niedostępny.
- *Selector* — umożliwia określenie filtru dla celu JMS

Klikamy przycisk *Save* i sprawdzamy, czy nowa kolejka znajduje się na liście celów JMS

Przedstawiona operacja spowoduje dodanie elementu do pliku konfiguracyjnego serwera.

```
<jms-destinations>
<jms-queue name="TicketQueue">
<entry name="java:jboss/jms/queue/ticketQueue"/>
<durable>false</durable>
</jms-queue>
</jms-destinations>
```

Dodanie do aplikacji ziaren sterowanych komunikatami

Ziarna sterowane komunikatami (MDB) to bezstanowe, serwerowe i transakcyjne komponenty zapewniające asynchroniczne przetwarzanie komunikatów JMS.

Jednym z najważniejszych aspektów ziaren sterowanych komunikatami jest to, że mogą konsumować i przetwarzać komunikaty w sposób współbieżny. To duża zaleta w porównaniu

z tradycyjnymi klientami JMS, które trzeba by opatrzyć dodatkowym kodem obsługującym zasoby, transakcje i bezpieczeństwo w środowisku wielowątkowym.

Kontenery MDB działają współbieżnie w sposób automatyczny, więc programista ziaren może skupić się na logice biznesowej przetwarzania komunikatów. Ziarno MDB może otrzymywać setki komunikatów z różnych aplikacji i przetwarzać je wszystkie w tym samym czasie, ponieważ poszczególne instancje ziarna będą w kontenerze działały w sposób współbieżny. Z semantycznego punktu widzenia, ziarno sterowane komunikatami traktowane jest jak ziarno EJB, podobnie jak ziarno sesyjne, ale istnieją istotne różnice. Przede wszystkim ziarno sterowane komunikatami nie posiada interfejsów komponentowych. Wynika to z faktu, iż ziarno tego typu nie jest dostępne z poziomu API Java RMI, bo reaguje tylko na asynchronicznie otrzymane komunikaty.
Podobnie jak ziarna sesyjne lub encyjne, także ziarno sterowane komunikatami ma dobrze zdefiniowany cykl życia. Cykl życia ziarna MDB składa się w zasadzie z dwóch stanów — **Nie istnieje** i **Pula gotowych metod**.

Po otrzymaniu komunikatu kontener EJB sprawdza, czy w puli dostępna jest wolna instancja MDB. Jeśli tak, JBoss przystępuje do jej użycia. Po zakończeniu działania metody onMessage()
żądanie uznaje się za zakończone, więc instancja wraca z powrotem do puli. W ten sposób uzyskuje się najszybszy czas reakcji, ponieważ żądanie obsługiwane jest bez zbędnej zwłoki związanej tradycyjnie z utworzeniem instancji.

Tworzenie ziaren sterowanych komunikatami

Dodajmy do aplikacji ziarno sterowane komunikatami, które posłuży do przechwycenia informacji o rezerwacji miejsca. W przedstawionym przykładzie po odebraniu komunikatu jedynie wyświetlimy jego zawartość. W rzeczywistości można pokusić się o znacznie bardziej rozbudowany scenariusz, na przykład powiadomienie o zakupie miejsca z zewnętrznych systemów.

W Eclipse przechodzimy do menu *File*, a następnie wybieramy *New/Other/EJB/EJB 3 Message Driven Bean*. Tworzymy nową klasę o nazwie MDBService znajdującą się w odpowiednim pakiecie. Powstanie plik z klasą zawierającą jedynie pustą metodę onMessage.

Do utworzonej klasy dodajemy konfigurację MDB w postaci adnotacji:

```
import java.util.logging.Logger;
import javax.ejb.*;
import javax.inject.Inject;
import javax.jms.*;
@MessageDriven(name = "MDBService", activationConfig = {
```

```
@ActivationConfigProperty(propertyName =
"destinationType", propertyValue = "javax.jms.Queue"),
@ActivationConfigProperty(propertyName =
"destination", propertyValue = "java:jboss/jms/queue/ticketQueue"),
@ActivationConfigProperty(propertyName =
"acknowledgeMode", propertyValue = "Auto-acknowledge") }) [1]
public class MDBService implements MessageListener {
@Inject
private Logger logger;
public void onMessage(Message message) {
TextMessage tm = (TextMessage) message;
try {
logger.info("Otrzymano komunikat "+tm.getText());
} catch (JMSException e) {
logger.error("Błąd! "+e);
}
}
}
```

Przedstawiony kod łączy MDB z celem ticketQueue dostępnym pod nazwą java:jboss/jms/queue/ticketQueue. Jedynym zadaniem komponentu jest wyświetlenie komunikatu przy użyciu java.util.Logger.

Dodanie producenta JMS

Po utworzeniu konsumenta JMS musimy jeszcze zadbać o komponent, który wygeneruje wiadomość JMS. Wykonamy ziarno CDI o zakresie aplikacji i nazwie JMSService, które zostanie wstrzyknięte do zasobów JMS.

```
import javax.annotation.Resource;
import javax.enterprise.context.ApplicationScoped;
import javax.jms.*;
import java.util.logging.Logger;
@ApplicationScoped
public class JMSService {
@Inject
private Logger logger;
@Resource(mappedName = "java:/ConnectionFactory")
private ConnectionFactory cf;
@Resource(mappedName = "java:jboss/jms/queue/ticketQueue")
private Queue queueExample;
private Connection connection;
public void sendMessage(String txt) {
try {
connection = cf.createConnection();
Session session = connection.createSession(false, Session.AUTO_
∑ACKNOWLEDGE);
MessageProducer publisher = session.createProducer(queueExample);
connection.start();
TextMessage message = session.createTextMessage(txt);
publisher.send(message);
}
catch (Exception exc) {
logger.error("Błąd! "+exc);
}
finally {
if (connection != null) {
try {
connection.close();
connection = null;
```

```
} catch (JMSException e) { logger.error(e); }
}
}
}
}
```

Usługa umożliwi informowanie o zmianach zachodzących w aplikacji. Wstrzyknięcie JMSService do ziarna BookerService umożliwi wysłanie komunikatu po rejestracji użytkownika.

```
public class BookerService implements Serializable {
@Inject JMSService jmsService;
. . . . .
public void bookSeat(Long seatId, int price) {
FacesContext fc = FacesContext.getCurrentInstance();
if (price > money) {
FacesMessage m = new FacesMessage(FacesMessage.SEVERITY_ERROR,
∑"Za mało środków!", "Rejestracja zakończona sukcesem");
fc.addMessage(null, m);
return;
}
logger.info("Rezerwuję miejsce "+seatId);
ticketService.bookSeat(seatId);
money = money - price;
FacesMessage m = new FacesMessage(FacesMessage.SEVERITY_INFO,
∑"Zarezerowano!", "Rezerwacja zakończona sukcesem");
if (fc != null)
fc.addMessage(null, m);
jmsService.sendMessage("[Komunikat JMS] Zarezerwowano miejsce "+seatId);
}
}
```

Kompilacja i wdrożenie aplikacji

Przedstawiony przykład można uruchomić jako część projektu Maven w wersji webapp-javaee6, podobnie jak to w lab 6. Jedyną dodatkową zależnością, która pojawia się w pliku *pom.xml*, jest API JBoss JMS.

```
<dependency>
<groupId>org.jboss.spec.javax.jms</groupId>
<artifactId>jboss-jms-api_1.1_spec</artifactId>
<scope>provided</scope>
</dependency>
```

Uruchamiamy JBoss AS 7 w konfiguracji zawierającej podsystem komunikatów (plik *standalonefull.xml*).

standalone.sh –c standalone-full.xml

Wdrażamy aplikację przy użyciu środowiska Eclipse lub narzędzia Maven i przechodzimy pod adres *http://localhost:8080/ticket-agency-jms/*.

Zadanie do realizacji

Zad 1. Napisz prostą aplikacje typu lista tematyczna. Możliwe jest tworzenie nowych list tematycznych. W ramach istniejącej listy wysyłamy komunikaty do wszystkich subskrybentów lub powinna być możliwość wskazania tylko wybranego.

Zad 2. Napisz aplikacje pozwalającą na rejestracje nowych użytkowników (firm) przez zewnętrzny system wysyłania komunikatów JMS. Zrealizować aplikację klienta (np. działu wysyłania reklam do

firm) aby pełniła rolę odbiorcy informacji o komunikatach JMS przesyłanych przez system w przypadku rejestracji nowej firmy. Rejestrację nowej firmy zrealizować jako prostą aplikację war z formularzem dodawania danych firmy. Wykorzystać wzorzec delegata biznesowego hermetyzującego kod obsługi JMS klienta.

Zad3. Rozszerz projekt biblioteka z lab 6 o otrzymywane potwierdzenia wykonywania każdej z operacji. Użyj do tego celu MDB. Możliwe jest również dodawania nowych pozycji książkowych do oferty biblioteki. W takim przypadku wysyłane są komunikaty powiadamiające o nowej pozycji do wszystkich użytkowników, którzy w procesie rejestracji zdefiniowali taka chęć otrzymywania powiadomień.