

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 <https://javaee.github.io/tutorial/jms-concepts.html#BNCDQ> 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 przesył 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.

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.

Aplikacja JMS składa się z następujących elementów:

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

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.

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`.

Część praktyczna – konfigurowanie dostępu do JMS.

Aby rozpocząć tworzenie aplikacji wykorzystującej technologie JMS najpierw musimy przygotować infrastrukturę tj. Kolejki i Topiki do których nadawca będzie pisał a odbiorca czytał wiadomości.

Niestety w aktualnej wersji WildFly Subsystem obsługujący wymianę komunikatów znajduje się poza standardem. Oznacza to że nie jest zdefiniowany w domyślnej wersji pliku standalone.xml. Na szczęście istnieje drugi plik standalone-full.xml, który zawiera odpowiednie zapisy.

Proszę nadpisać plik standalone plikiem standalone-full. Sprawdzić czy w nowym pliku znajduje się moduł obsługujący messaging.

Innym sposobem jest uruchomienie serwera z opcją -c i jawne podaniem plikiem konfiguracyjnym (w naszym przypadku **standalone-full.xml**)

Proszę zwrócić uwagę na komunikaty wyświetlane przy starcie serwera aplikacyjnego. Jednym z nich jest informacja, że uruchomiony został serwer obsługi kolejek który jest domyślnie skonfigurowany z WildFly.

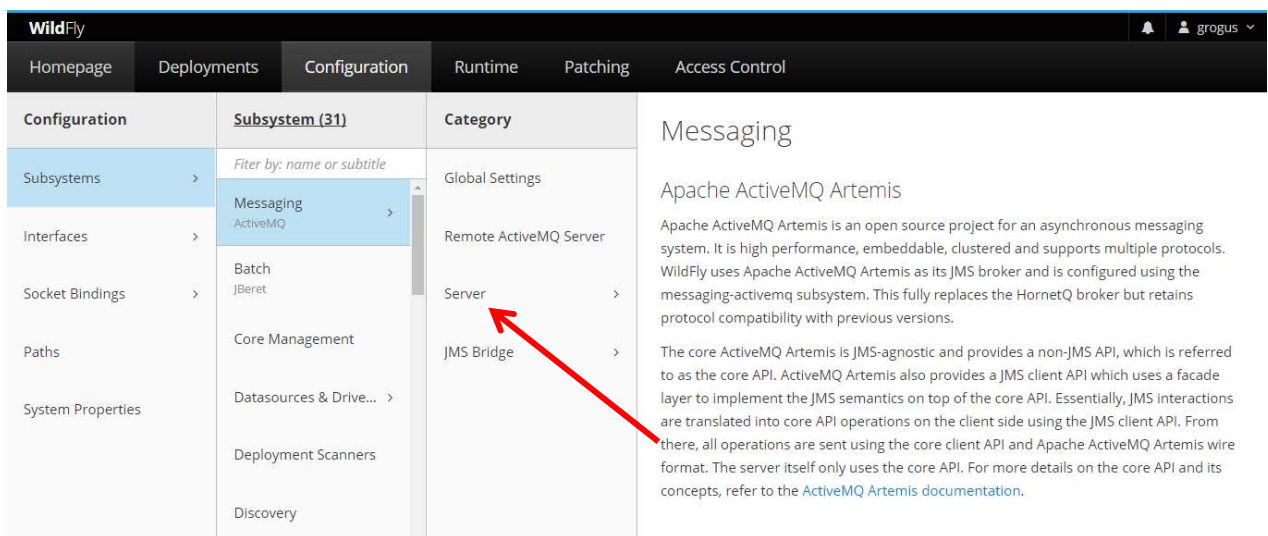
```
ad Pool -- 81) AMQ221034: Waiting indefinitely to obtain live lock
20:18:32,614 INFO [org.apache.activemq.artemis.core.server] (ServerService Thre
ad Pool -- 81) AMQ221035: Live Server Obtained live lock
20:18:32,980 INFO [org.jboss.ws.common.management] (MSC service thread 1-3) JBW
S022052: Starting JBossWS 5.2.4.Final (Apache CXF 3.2.7)
20:18:34,910 INFO [org.wildfly.extension.messaging-activemq] (MSC service threa
d 1-3) WFLYMSGAMQ0016: Registered HTTP upgrade for activemq-remoting protocol ha
ndled by http-acceptor acceptor
20:18:34,909 INFO [org.wildfly.extension.messaging-activemq] (MSC service threa
d 1-1) WFLYMSGAMQ0016: Registered HTTP upgrade for activemq-remoting protocol ha
ndled by http-acceptor-throughput acceptor
20:18:34,914 INFO [org.wildfly.extension.messaging-activemq] (MSC service threa
d 1-4) WFLYMSGAMQ0016: Registered HTTP upgrade for activemq-remoting protocol ha
ndled by http-acceptor-throughput acceptor
20:18:34,917 INFO [org.wildfly.extension.messaging-activemq] (MSC service threa
d 1-2) WFLYMSGAMQ0016: Registered HTTP upgrade for activemq-remoting protocol ha
ndled by http-acceptor acceptor
20:18:35,754 INFO [org.apache.activemq.artemis.core.server] (ServerService Thre
ad Pool -- 81) AMQ221007: Server is now live
20:18:35,756 INFO [org.apache.activemq.artemis.core.server] (ServerService Thre
ad Pool -- 81) AMQ221001: Apache ActiveMQ Artemis Message Broker version 2.6.3.j
bossorg-00014 [default, nodeID=cha7882b-5ee0-11e9-b32f-881c20524153]
20:18:35,976 INFO [org.wildfly.extension.messaging-activemq] (ServerService Thr
ead Pool -- 81) WFLYMSGAMQ0002: Bound messaging object to jndi name java:jboss/e
xported/jms/RemoteConnectionFactory
20:18:36,068 INFO [org.wildfly.extension.messaging-activemq] (ServerService Thr
ead Pool -- 85) WFLYMSGAMQ0002: Bound messaging object to jndi name java:/Connec
tionFactory
20:18:36,400 INFO [org.jboss.as.connector.deployment] (MSC service thread 1-4)
WFLYJCA0007: Registered connection factory java:/JmsXA
20:18:36,454 INFO [org.jboss.as.connector.deployment] (MSC service thread 1-1)
WFLYJCA0011: Unbound JCA ConnectionFactory [java:/JmsXA]
20:18:36,466 INFO [org.jboss.as.connector.deployment] (MSC service thread 1-2)
WFLYJCA0119: Unbinding connection factory named java:/JmsXA to alias java:jboss/
DefaultJMSConnectionFactory
20:18:36,868 INFO [org.apache.activemq.artemis.ra] (MSC service thread 1-4) AMQ
151007: Resource adaptor started
```

Krokiem następnym jest konfiguracja infrastruktury. Podobnie jak przy okazji konfiguracji JPA możemy to zrobić na 3 sposoby:

1. Za pomocą Web Admin Console
2. CLI
3. Ręcznie modyfikując plik standalone.xml

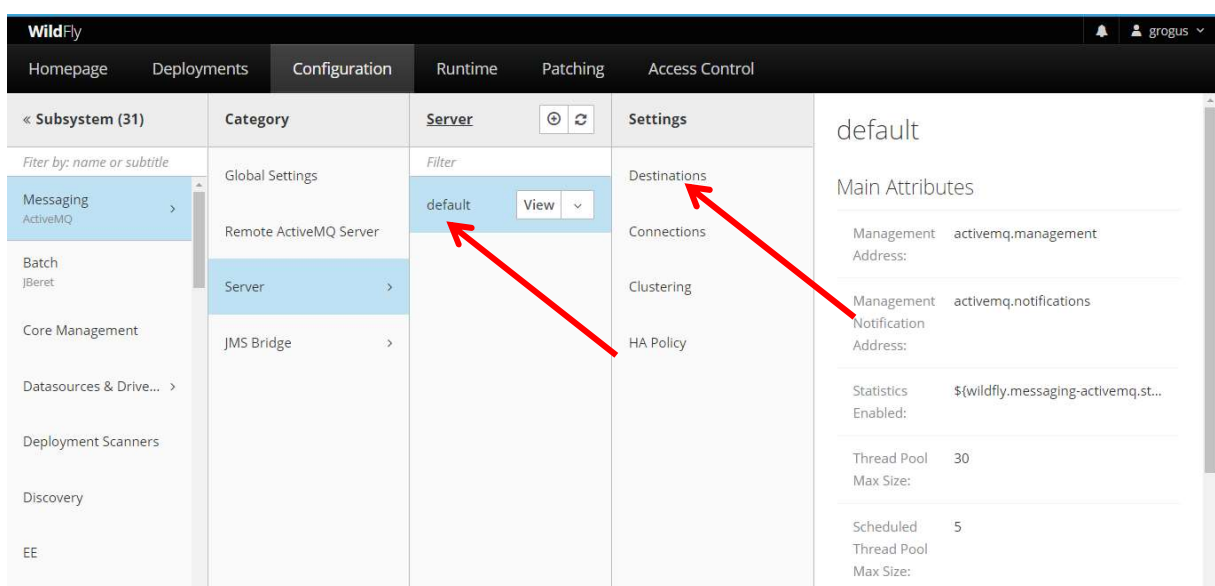
1. Konfiguracja za pomocą Web Admin Console

W sekcji Configuration znajdź podsystem Messaging



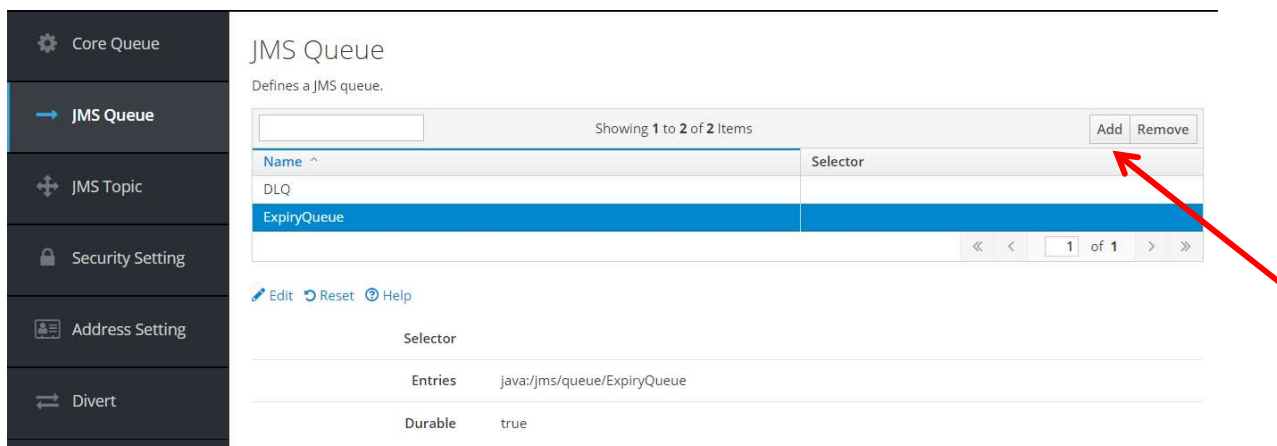
The screenshot shows the WildFly Web Admin Console interface. The top navigation bar includes 'Homepage', 'Deployments', 'Configuration', 'Runtime', 'Patching', and 'Access Control'. The 'Configuration' tab is active. On the left, the 'Subsystems (31)' list is shown, with 'Messaging' (ActiveMQ) selected. The main content area displays the 'Messaging' subsystem configuration. It includes a 'Global Settings' section, a 'Remote ActiveMQ Server' section, and a 'Server' section. A red arrow points to the 'Server' section. The 'Server' section shows a 'JMS Bridge' configuration. The right sidebar contains the 'Messaging' title and a description of Apache ActiveMQ Artemis.

Konfigurować będziemy domyślny serwer wbudowany w WildFly. Jak będzie można zauważyć konfigurować można także dowolny zewnętrzny serwer obsługujący kolejki komunikatów.



The screenshot shows the WildFly Web Admin Console interface, specifically the 'Server' configuration page for the 'default' server in the 'Messaging' subsystem. The top navigation bar is the same as the previous screenshot. The left sidebar shows the 'Subsystems (31)' list, with 'Messaging' (ActiveMQ) selected. The main content area displays the 'Server' configuration. It includes a 'Filter' section, a 'Settings' section, and a 'default' section. A red arrow points to the 'default' section. The 'default' section shows the 'Main Attributes' configuration, including 'Management Address', 'Management Notification Address', 'Statistics Enabled', 'Thread Pool Max Size', and 'Scheduled Thread Pool Max Size'. A red arrow points to the 'Destinations' section in the 'Settings' area.

Wybieramy sekcję Destination w celu stworzenia nowej kolejki oraz topica, którego będziemy używali w naszej aplikacji testowej.



JMS Queue

Defines a JMS queue.

Showing 1 to 2 of 2 Items

Name ^	Selector
DLQ	
ExpiryQueue	

« < 1 of 1 > »

Edit Reset Help

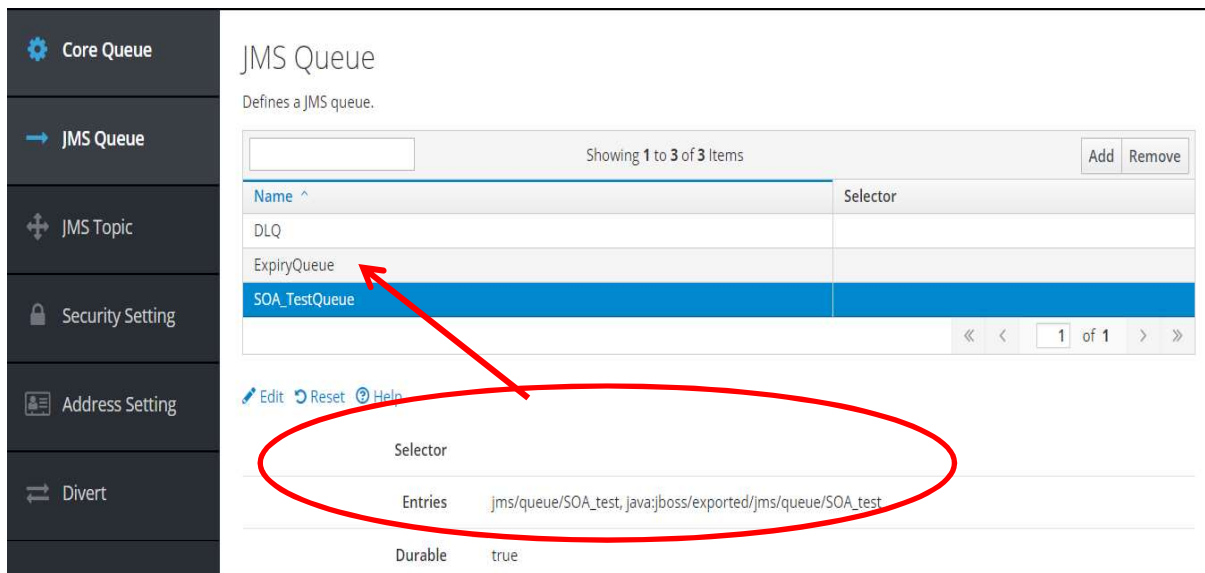
Selector

Entries java:/jms/queue/ExpiryQueue

Durable true

Dodajemy nową kolejkę o nazwie SOA_TestQueue

v



JMS Queue

Defines a JMS queue.

Showing 1 to 3 of 3 Items

Name ^	Selector
DLQ	
ExpiryQueue	
SOA_TestQueue	

« < 1 of 1 > »

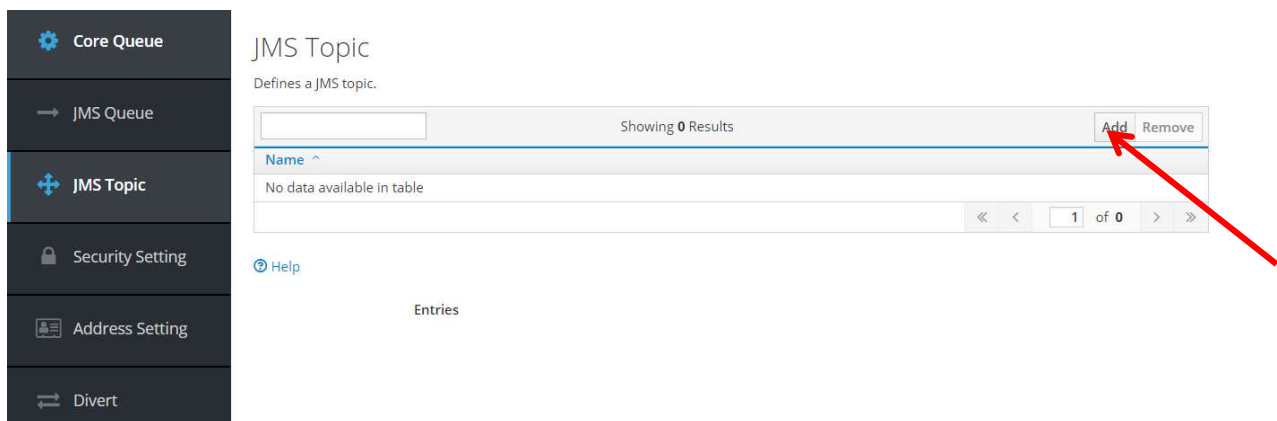
Edit Reset Help

Selector

Entries jms/queue/SOA_test, java:jboss/exported/jms/queue/SOA_test

Durable true

Teraz czas na nowy topic



JMS Topic

Defines a JMS topic.

Showing 0 Results

Name ^
No data available in table

« < 1 of 0 > »

Help

Entries

Wynikiem tych operacji jest modyfikacja pliku standalone.xml w sekcji

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:6.0">
```

```

<jms-queue name="SOA_TestQueue" entries="jms/queue/SOA_test java:jboss/exported/jms/queue/SOA_test" durable="true"/>
<jms-topic name="SOA_TestTopic" entries="jms/topic/SOA_Test java:jboss/exported/jms/topic/SOA_Test"/>

```

Alternatywnym sposobem jest konfiguracja za pomocą CLI:

Przez CLI:

Dodanie kolejki JMS:

```
[standalone@localhost:9999/] jms-queue add --queue-address= SOA_Test
Queue --entries=java:/jms/queue/SOA_test,java:/jboss/exported/jms/qu
eue/SOA_test
```

Add JMS Topic:

```
[standalone@localhost:9999/] jms-topic add --topic-address=SOA_TestT
opic --entries=java:/jms/topic/SOA_Test,java:/jboss/exported/jms/top
ic/SOA_Test
```

Drugi adres JNDI `"java:/jboss/exported/jms/[destination]"` jest wykorzystywany w przypadku łączenia z zdalnym klientem JMS

Poprzez ręczną modyfikację pliku standalone.xml:

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:6.0">
  <server name="default">
```

```

    . . . . .
    <jms-queue name="SOA_Test" entries="java:/jms/queue/SOA_test
java:/jboss/exported/jms/queue/SOA_test"/>

```

```

    <jms-topic name="SOA_TestTopic" entries="java:/jms/topic/SOA_Test
java:/jboss/exported/jms/topic/SOA_Test"/>

```

```

</server>
</subsystem>

```

Aktualny stan podsystemu JMS można zobaczyć w sekcji Runtime.

The screenshot shows the WildFly management console interface. The top navigation bar includes 'Homepage', 'Deployments', 'Configuration', 'Runtime', 'Patching', and 'Access Control'. The 'Runtime' tab is selected. On the left, there are two panels: 'Server' and 'Monitor'. The 'Server' panel shows a list of servers, with 'grzes..' selected. The 'Monitor' panel shows various monitoring categories, with 'Messaging' selected. The main area displays the 'Destination (4)' list, which includes 'DLQ', 'ExpiryQueue', 'SOA_TestQueue', and 'SOA_TestTopic'. The 'SOA_TestQueue' is highlighted with a red circle. On the right, the details for 'SOA_TestQueue' are shown, including its JNDI Names, Queue Address, and other attributes. A red arrow points to the 'JNDI Names' field, which contains 'jms/queue/SOA_test, java:jboss/...'.

Widać w niej aktualna ilość kolejek oraz adresy za pomocą których możesz odwoływać się do kolejek z poziomu aplikacji. Dodatkowo można zobaczyć aktualny stan każdej z kolejek z ilością komunikatów włącznie.

Server: defa... View

Destination (4): Filter by: name, type, deployment

- DLQ
- ExpiryQueue
- SOA... View
- SOA_TestTopic

Messages

Messages Added:	0
Message Count:	0
Delivering Count:	0
Scheduled Count:	0
Consumer Count:	0

Podobnie jest z Topic. Mamy możliwość usunąć poszczególnych subskrybentów oraz podglądać ilość przesłanych komunikatów.

Server: defa... View

Destination (4): Filter by: name, type, deployment

- DLQ
- ExpiryQueue
- SOA_TestQueue
- Drop Subscriptions

SOA_TestTopic Refresh

JMS Topic

Main Attributes

Topic Address:	jms.topic.SOA_TestTopic
JNDI Names:	jms/topic/SOA_Test, java:jboss/e...
Temporary:	false

Messages

Message Count:	0
Durable	0

Destination (4)

Filter by: name, type, deployment

→ DLQ

→ ExpiryQueue

→ SOA_TestQueue

... Drop Subscriptions

Count:

Messages Added:	0
Delivering Count:	0

Subscriptions

Subscription Count:	0
Durable Subscription Count:	0
Non Durable Subscription Count:	0

Ze względu na to że będziemy odwoływać się do adresów JNDI – przypominam stosowane konwencje.

Monitor

EJB >

IO >

JAX-RS >

JNDI View

JPA >

JNDI

Provides an overview of the local JNDI namespace. The Java EE platform specification defines the following JNDI contexts:

- `java:comp` - The namespace is scoped to the current component (i.e. EJB)
- `java:module` - Scoped to the current module
- `java:app` - Scoped to the current application
- `java:global` - Scoped to the application server

In addition to the standard namespaces, WildFly also provides the following two global namespaces:

- `java:jboss`
- `java:/`

Please note that only entries within the `java:jboss/exported` context are accessible over remote JNDI. For web deployments `java:comp` is aliased to `java:module`, so EJB's deployed in a war do not have their own comp namespace.

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.

Domyślnie dostępne są dwie fabryki połączeń.

- 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.

Tworzenie aplikacji:

Wymagane zależności:

```
<dependency>
  <groupId>org.jboss.spec.javax.jms</groupId>
  <artifactId>jboss-jms-api_2.0_spec</artifactId>
  <version>1.0.0.Final</version>
</dependency>
```

Oraz

```
<!-- https://mvnrepository.com/artifact/org.wildfly/wildfly-jms-client-bom -->
<!-- https://mvnrepository.com/artifact/org.wildfly/wildfly-jms-client-bom -->
<dependency>
  <groupId>org.wildfly</groupId>
  <artifactId>wildfly-jms-client-bom</artifactId>
  <version>10.1.0.Final</version>
  <type>pom</type>
</dependency>
```

Przykłady kodu:

Wersja klasyczna zużyciem interfejsu JSM_1.1

Wysyłanie wiadomości:

```
@Resource(mappedName="java:/ConnectionFactory")
private static ConnectionFactory cf;
@Resource(mappedName=" java:/PrzykładowaKolejka")
Private static Queue queue;

Connection conn = cf.createConnection();
Session session = conn.createSession(false, Session.AUTO_ACKNOWLEDGE);

MessageProducer producer = session.createProducer(queue);
TextMessage msg = session.createTextMessage();
msg.setText("Komunikat testowy");
producer.send(msg);
```

Odbiór komunikatu z wykorzystaniem komponentu MDB

```
@MessageDriven(mappedName=" java:/PrzykładowaKolejka")
public class MyMessageBean implements MessageListener {
    public void onMessage(Message msg) {
        TextMessage txtMsg = null;
        try {
            if (msg instanceof TextMessage) {
                txtMsg = (TextMessage) msg;
                String txt = txtMsg.getText();
            }
        } catch (JMSEException e) {...}
    }
}
```

// Przykład bardziej rozbudowany

```
import javax.annotation.Resource;
import javax.jms.*;

public class JMSService {

    @Resource(mappedName = "java:/queue/SOA_Test")
    private Queue queueExample;

    @Resource(mappedName = "java:/JmsXA")
    private ConnectionFactory cf;

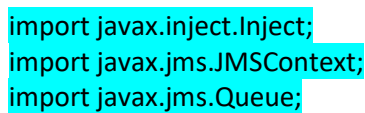
    private Connection connection;
    public void sendMessage(String txt) {

        try {
            connection = cf.createConnection();
            Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
            MessageProducer publisher = null;

            publisher = session.createProducer(queueExample);

            connection.start();

            TextMessage message = session.createTextMessage(txt);
            publisher.send(message);
        }
        catch (Exception exc) {
            exc.printStackTrace();
        }
        finally {
```

[illegible]

```
@Resource(mappedName = "java:/queue/SOA_Test")
private Queue queueTest;
```

```
@Inject
JMSContext context;
```

```

public void sendMessage(String txt) {
    try {
        context.createProducer().send(queueTest, txt);
    }
    catch (Exception exc) {
        exc.printStackTrace();
    }
}
}
}

```

Obiekt **JMSContext** jest nowym elementem wprowadzonym przez specyfikację JMS 2.0. Zastąpił on znane z JMS 1.1 API: obiekty **Connection** oraz **Session**.

Domyślnie **JMSContext** używa domyślnej obiektu **JMSConnectionFactory**, który jest powiązany z "java:/ConnectionFactory". W przypadku gdy chcemy lub musimy skorzystać z innego obiektu można użyć adnotacji **@JMSConnectionFactory** do wskazania innej fabryki połączeń:

```

@Inject
@JMSConnectionFactory("java:/InnyConnectionFactory")
JMSContext context;

```

Odbieranie komunikatów

```

import javax.inject.Inject;
import javax.jms.JMSContext;
import javax.jms.Queue;

```

```

public class QueueReceiver {

```

```

    @Inject
    private JMSContext context;

```

```

    @Resource(mappedName = "java:/queue/SOA_Test")
    Queue myQueue;

```

```

    public String receiveMessage() {
        String message = context.createConsumer(myQueue).receiveBody(String.class);
        if (message == null)
            message = "Nic nie ma w kolejce";
        return message;
    }
}

```

Zadanie do realizacji

Zad 1. Napisz prostą aplikację typu forum tematyczne. W ramach forum możliwe jest tworzenie nowych list tematycznych. Użytkownicy mogą subskrybować się do dowolnej ilości list tematycznych. W ramach istniejącej listy wysyłamy komunikaty do wszystkich subskrybentów lub powinna być możliwość wskazania tylko wybranego do którego chcemy napisać.

Zad 2. Rozszerz projekt biblioteka z lab 6 o otrzymywane potwierdzenia wykonywania każdej z operacji. Użyj do tego celu MDB. Możliwe jest również dodawanie nowych pozycji książkowych do oferty biblioteki. W takim przypadku wysyłane są komunikaty powiadamiające o nowej pozycji do wszystkich użytkowników, którzy w procesie rejestracji zdefiniowali taką chęć otrzymywania powiadomień. Podobnie w przypadku zwrotu książki, wszystkie osoby które chciały pożyczyć książkę gdy była nie dostępna mają otrzymać powiadomienia. Przyjmijmy, że w naszej bibliotece każdy tytuł liczy tylko jedną pozycję.