Java Message Service

dzień 4

v3.0

Plan

1. Java Message Service (JMS)
2. Struktura wiadomości JMS
3. Apache ActiveMQ
4. Komunikacja w JMS
5. JMS Point-To-Point
6. JMS Publish / Subscribe
7. JMS i Spring

Java Message Service - JMS

Java Message Service (JMS)

Java Message Service (JMS) to pośrednik w przesyłaniu wiadomości pomiędzy dwoma lub więcej aplikacjami

API Java Message Service definiuje standard przesyłania wiadomości i umożliwia aplikacjom Java tworzenie, wysyłanie, odbieranie i odczytywanie wiadomości JMS

Java Message Service (JMS)

JMS charakteryzuje się tym że mechanizmy wykonywane przez JMS są:

- luźno powiązane, czyli jeden komponent aplikacji może być wykonywany/testowany niezależnie od innego
- wysyłanie i odbieranie wiadomości odbywa się niezależnie od siebie
- JMS określa się jako niezawodny mechanizm komunikacji

Java Message Service (JMS)

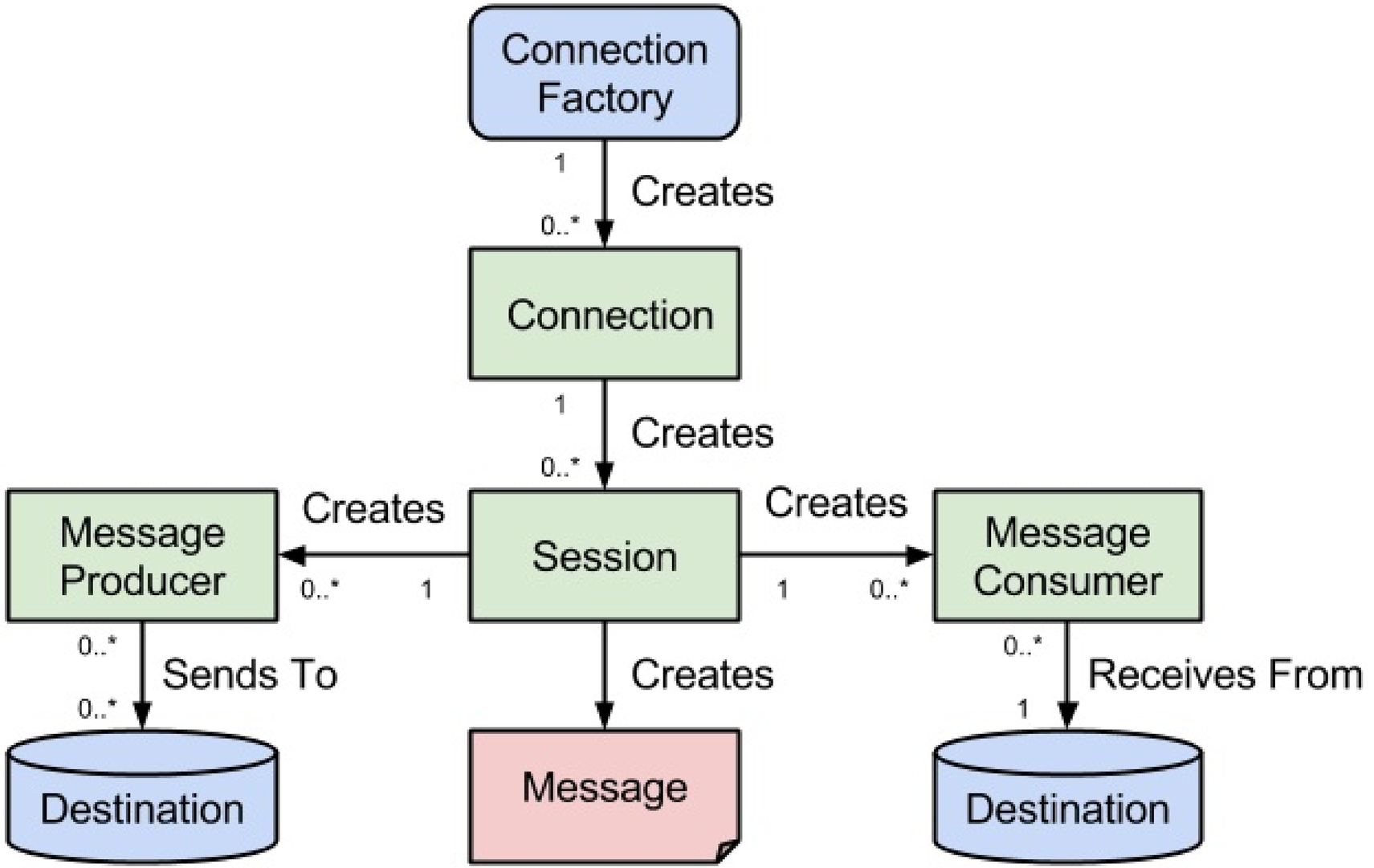
Aby korzystać z JMS musimy wykorzystać **dostawcę(ang. provider)** który wykonuje operacje na wiadomościach za pomocą JMS API

Dostawca **jest pośrednikiem** pomiędzy wysyłaniem a odbieraniem wiadomości poprzez JMS

Najbardziej znani dostawcy JMS to np. Apache ActiveMQ, Rabbit MQ, WebSphere MQ lub SonicMQ

W późniejszym czasie poznamy jak zainstalować ActiveMQ

Na kolejnym slajdzie przedstawiony jest schemat JMS



JMS - ConnectionFactory

Podstawą komunikacji w JMS jest obiekt **`ConnectionFactory`**. Tego obiektu używa się do utworzenia połączenia do dostawcy

```
ConnectionFactory connFact =
      new ActiveMQConnectionFactory(ActiveMQConnection.DEFAULT_BROKER_URL);
```

Metody klasy Fabryki które tworzą obiekt ConnectionFactory ustawiają parametry takie jak np. adres URL dostawcy

Parametr **`DEFAULT_BROKER_URL`** domyślnie ma domyślnie ustawioną wartość **`tcp://localhost:61616`**

JMS - Connection

Utworzony obiekt ConnectionFactory jest używany w dalszej części do utworzenia obiektu połączenia **`Connection`**

Obiekt **`Connection`** jest właśnie obiektem połączeniowym utrzymującym komunikację z dostawcą JMS.

Ważne aby przed zakończeniem aplikacji zamknąć obiekt połączeniowy, w przeciwnym wypadku zasoby dostawcy JMS nie zostaną zwolnione.

```
Connection connection = connFact.createConnection();
connection.close();
```

JMS - Session

Kolejnym ważnym obiektem który jest tworzony to obiekt **`Session`**

Obiekt Session daje możliwość do tworzenia i odbierania wiadomości, oraz działa na zasadzie transakcji grupując zestaw operacji wysyłki i odbioru wiadomości w atomowe jednostki pracy.

Uwaga! Zamknięcie obiektu Connection zamyka również obiekt sesji

```
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
```

JMS - Destination

Obiekt **`Destination`** jest miejscem docelowym do którego wysyłane są wiadomości i z którego te wiadomości są odczytywane

Istnieją dwa rodzaje miejsc docelowych(które szczegółowo omówimy później):

- **`kolejki(ang. queues)`** - reprezentowane przez interfejs **`Queue`**
- **`tematy(ang. topics)`** - reprezentowane przez interfejs **`Topic`**

```
Destination destination = session.createQueue(destinationName);
```

lub

```
Destination destination = session.createTopic(destinationName);
```

JMS - MessageProducer

Obiekt **`MessageProducer`** jest tworzony przez obiekt Session i jest używany do wysyłki wiadomości do **`miejsca docelowego(ang. Destination)`**

```
MessageProducer messageProducer = session.createProducer(destination);
```

JMS - MessageConsumer

Obiekt **`MessageConsumer`** jest obiektem tworzonym przez obiekt Session i służy do odbioru wiadomości z miejsca docelowego

Po utworzeniu obiektu MessageConsumer staje się on aktywny i można go używać do odbierania wiadomości

Uwaga! Odbiór wiadomości nie rozpocznie się dopóki nie zostanie wywołana metoda **`start()`** na obiekcie **`Connection`**

```
MessageConsumer messageConsumer = session.createConsumer(destination);
```

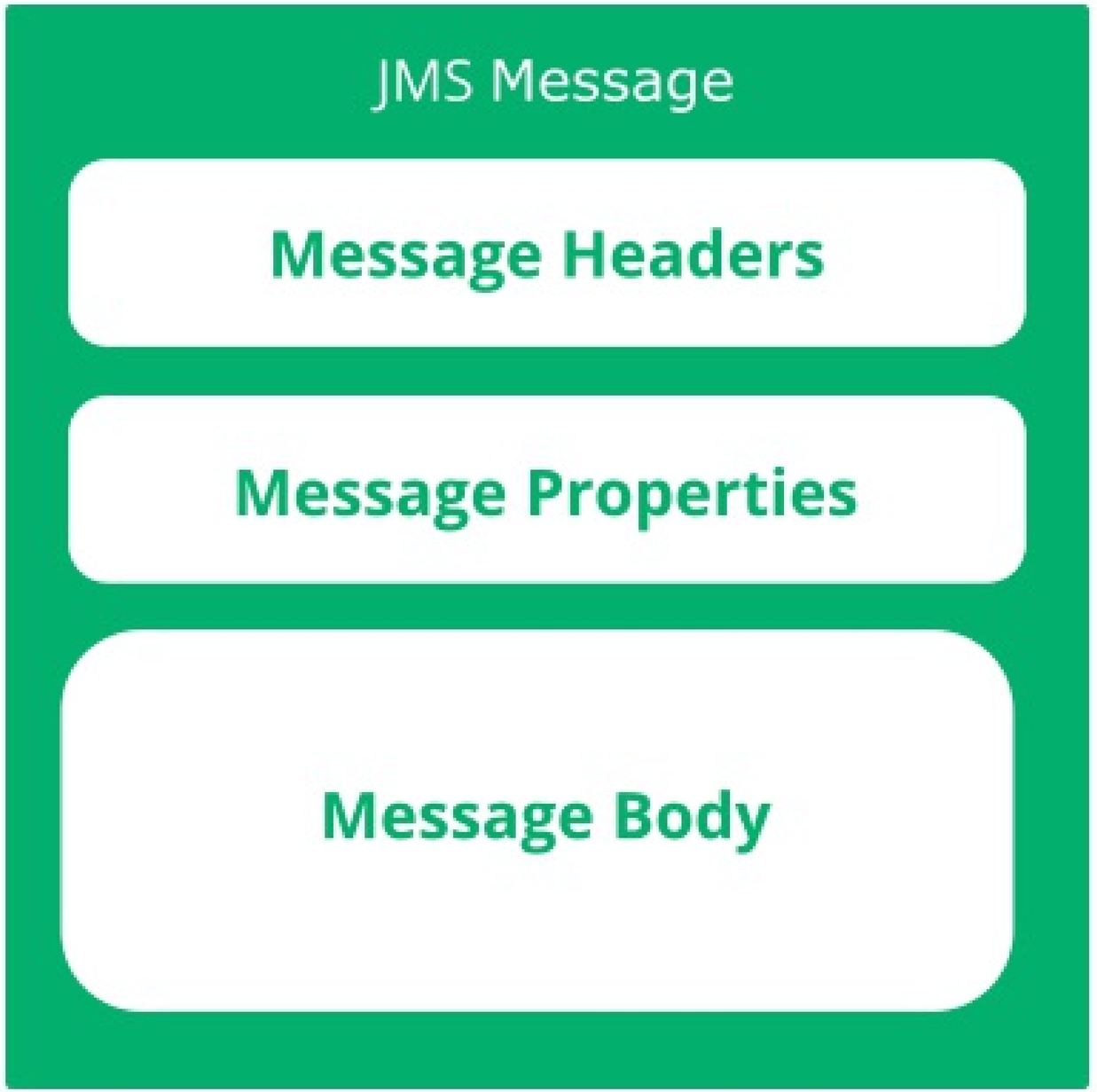
Struktura wiadomości JMS

Struktura wiadomości JMS

Podstawowa struktura wiadomości JMS składa się z trzech części:

- nagłówków
- parametrów
- zawartości

W następnym slajdzie przedstawiona jest graficzna postać struktury wiadomości JMS



Nagłówki JMS

`Nagłówki wiadomości JMS` zawierają zestaw zdefiniowanych pól które muszą być przekazane w każdej wiadomości JMS

Większość wartości przesyłanych w nagłówkach jest ustawiane przez dostawcę JMS kiedy wiadomość przekazywana jest do kolejki albo tematu

Wartości z nagłówków używane są zarówno przez aplikacje, jak i przez dostawcę do identyfikacji wiadomości

Nagłówki JMS

Poniżej przedstawiona jest lista pól które przekazywane są w nagłówkach i ich krótki opis:

- **JMSDestination** - zwraca obiekt Destination opisujący gdzie wiadomość jest kierowana
- **JMSDeliveryMode** - opisują sposób dostarczania NON_PERSISTENT lub PERSISTENT
- **JMSExpiration** - zwraca czas wygaśnięcia wiadomości
- **JMSPriority** - określa priorytet wiadomości
- **JMSMessageID** - zawiera wygenerowany ID identyfikujący wiadomość
- **JMSTimestamp** - zwraca czas wysłania wiadomości
- **JMSCorrelationID** - może powiązać jedną wiadomość z drugą
- **JMSReplyTo** - zawiera miejsce docelowe gdzie odpowiedź powinna być skierowana
- **JMSType** - zawiera typ wysyłanej wiadomości
- **JMSRedelivered** - zwraca wartość boolean czy wiadomość ma być ponownie dostarczana w przypadku braku potwierdzenia jej odbioru

Parametry wiadomości JMS

Dodatkowe parametry mogą być zawarte w przesyłanej wiadomości JMS jeżeli potrzeba innych wartości niż te zdefiniowane w nagłówkach

Parametry są opcjonalne i przechowywane zą jako zestaw par klucz-wartość

Występują trzy typy parametrów które możemy przekazać w wiadomości:

- **związane z aplikacją** - związane ze specyficznymi ustawieniami aplikacji do której jest wysyłana wiadomość
- **związane z dostawcą** - ich nazwy najczęściej zaczynają się prefiksem "JMS_", a następnie nazwą dostawcy
- **standardowe** - ich nazwy zaczynają się od "JMSX" i jest to np. JMSXUserid

Zawartość JMS

`Główna zawartość wiadomości JMS(ang. Message body)` zawiera właściwe informacje które są wymieniane między aplikacjami za pomocą tej technologii

JMS API definiuje **pięć typów zawartości wiadomości** które umożliwiają przesyłanie i odbieranie danych w różnej formie

JMS jest jedynie **specyfikacją** a nie podaje implementacji, daje to elastyczność w dobieraniu dostawcy JMS natomiast w aplikacji wykorzystywany jest jednolity interfejs

Co ciekawe niektóre implementacje JMS dodały własne typy wiadomości(np. SonicMQ dodał typ MultipartMessage)

Typy wiadomości w JMS

Główne typy wiadomości JMS to:

- **TextMessage** - jeżeli wysyłamy dane typu String
- **MapMessage** - zestaw par klucz-wartość
- **BytesMessage** - strumień bitów, najczęściej służy do kodowania formatu wiadomości
- **StreamMessage** - strumień wartości prostych(np. int), możliwy do odczytania sekwencyjnie
- **ObjectMessage** - zwykły obiekt

Wiadomości w JMS

JMS API dostarcza różnych metod do tworzenia wiadomości danego typu. Np. aby utworzyć i wysłać wiadomość typu TextMessage można użyć metody **`createTextMessage()`** oraz **`setText()`** (dokładną konstrukcję omówimy w dalszej części):

```
TextMessage message = session.createTextMessage();
message.setText(msg_text); // zmienna msg_text jest typu String
producer.send(message); // wysłanie wiadomości
```

Wiadomości w JMS

Aby odebrać wiadomość należy wykorzystać obiekty typu Message który należy jeszcze rzutować na odpowiedni typ danych który chcemy odczytać

Można skorzystać z metody **`getText()`** aby pobrać zawartość wiadomości typu TextMessage

```
Message message = consumer.receive();
if(message instanceof TextMessage) {
  TextMessage message = (TextMessage) message;
  System.out.println("Reading message: " + message.getText());
} else {
  // To Do
}
```

Apache ActiveMQ

Apache ActiveMQ

Apache ActiveMQ to oprogramowanie Open Source napisane w Javie które oferuje wsparcie dla różnych technologii takich jak JMS, REST czy WebSocket. My będziemy używali ActiveMQ do pracy z JMS

Pierwsze, co należy zrobić, to ściągnąć pliki instalacyjne ActiveMQ z oficjalnej strony internetowej projektu: http://activemq.apache.org/download.html

Następnie wybieramy najbardziej aktualną wersję znajdującą się przy nagłówku "Latest Releases". W czasie pisania materiałów jest to wersja **ActiveMQ 5.15.2 Release**

Po tym kroku powinniśmy uzyskać widok z wyborem wersji dla konkretnych systemów operacyjnych. W naszym przykładzie zainstalujemy ActiveMQ w systemie **Ubuntu**

ActiveMq - wybór wersji

ActiveMQ 5.15.2 Release

Apache ActiveMQ 5.15.2 includes several resolved issues and bug fixes.

Getting the Binary Distributions

Description	Download Link	*Verify*
Windows Distribution	apache-activemq-5.15.2-bin.zip	ASC, MD5, SHA512
Unix/Linux/Cygwin Distribution	apache-activemq-5.15.2-bin.tar.gz	ASC, MD5, SHA512

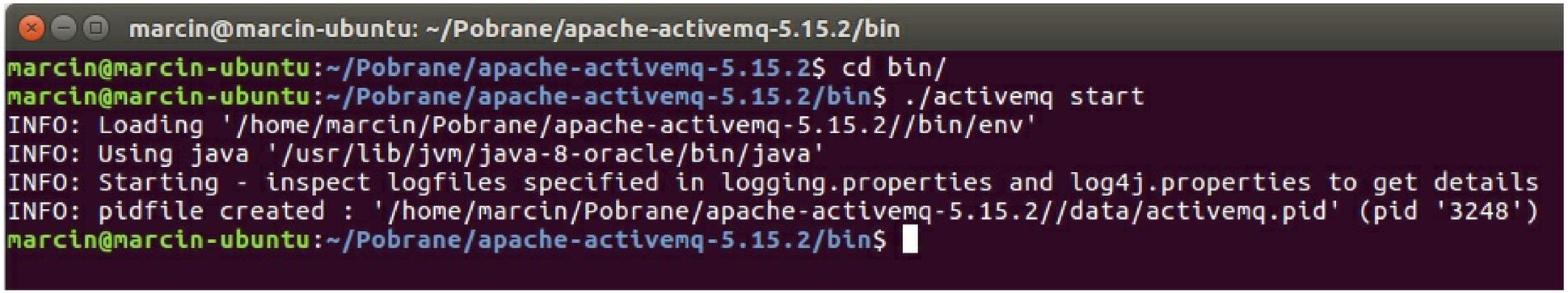
ActiveMq - wypakowanie i start

Po ściągnięciu pliku i wypakowaniu zawartości ściągniętego archiwum, katalog z plikami ActiveMQ powinien zawierać różne katalogi jak bin, conf lib itp.

Następnie należy przejść za pomocą terminalu do katalogu bin. Aby wystartować ActiveMQ wykonaj polecenie **./activemq start**

Po wykonaniu tego polecenia w konsoli powinny pojawić się komunikaty podobne do tych z następnego slajdu

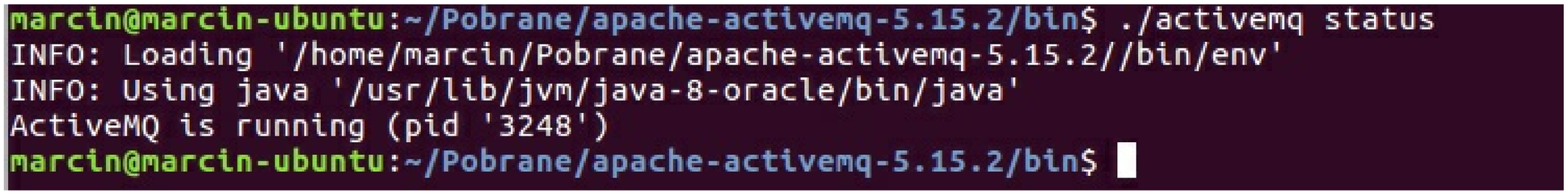
ActiveMq startowanie



ActiveMq - status

Aby potwierdzić poprawność wystartowania ActiveMQ lub w każdej chwili podczas korzystania z ActiveMQ sprawdzić jego status możemy wykonać w katalogu z plikami ActiveMQ, a następnie wchodząc do katalogu bin komendę **./activemq status**

W naszym przypadku po wykonaniu poprzedniej komendy wyświetli się informacja potwierdzająca uruchomienie ActiveMQ taka jak na zdjęciu w następnym poniżej



ActiveMq - zatrzymanie

Kolejna z przydatnych komend ActiveMQ to komenda służąca do zatrzymania wcześniej uruchomionego procesu. Aby tego dokonać należy w tym samym katalogu co dwie poprzednie komendy wykonać komendę **./activemq stop**

Po wykonaniu powyższej komendy powinniśmy otrzymać rezultat podobny do przedstawionego na kolejnym slajdzie

Cała lista komend dla ActiveMQ które możemy wykonać za pomocą terminalu jest dostępna pod adresem: http://activemq.apache.org/unix-shell-script.html#UnixShellScript-Functionaloverview

ActiveMq zatrzymywanie

```
marcin@marcin-ubuntu:~/Pobrane/apache-activemq-5.15.2/bin$ ./activemq stop
INFO: Loading '/home/marcin/Pobrane/apache-activemq-5.15.2//bin/env'
INFO: Using java '/usr/lib/jvm/java-8-oracle/bin/java'
INFO: Waiting at least 30 seconds for regular process termination of pid '3248' :
Java Runtime: Oracle Corporation 1.8.0_151 /usr/lib/jvm/java-8-oracle/jre
  Heap sizes: current=62976k  free=61992k  max=932352k
    JVM args: -Xms64M -Xmx1G -Djava.util.logging.config.file=logging.properties -Djava.security.auth.login.config=/home
apache-activemq-5.15.2//conf/login.config -Dactivemq.classpath=/home/marcin/Pobrane/apache-activemq-5.15.2//conf:/home/
pache-activemq-5.15.2//../lib/: -Dactivemq.home=/home/marcin/Pobrane/apache-activemq-5.15.2/ -Dactivemq.base=/home/marc
e-activemq-5.15.2/ -Dactivemq.conf=/home/marcin/Pobrane/apache-activemq-5.15.2//conf -Dactivemq.data=/home/marcin/Pobra
mq-5.15.2//data
Extensions classpath:
  [/home/marcin/Pobrane/apache-activemq-5.15.2/lib,/home/marcin/Pobrane/apache-activemq-5.15.2/lib/camel,/home/marcin/P
tivemq-5.15.2/lib/optional,/home/marcin/Pobrane/apache-activemq-5.15.2/lib/web,/home/marcin/Pobrane/apache-activemq-5.1
ACTIVEMQ_HOME: /home/marcin/Pobrane/apache-activemq-5.15.2
ACTIVEMQ_BASE: /home/marcin/Pobrane/apache-activemq-5.15.2
ACTIVEMQ_CONF: /home/marcin/Pobrane/apache-activemq-5.15.2/conf
ACTIVEMQ_DATA: /home/marcin/Pobrane/apache-activemq-5.15.2/data
Connecting to pid: 3248
.Stopping broker: localhost
. TERMINATED
```

ActiveMq - administracja

Po zainstalowaniu i uruchomieniu ActiveMQ mamy dostęp do **webowego interfejsu administracyjnego** ActiveMQ który dostępny jest pod adresem http://localhost:8161/

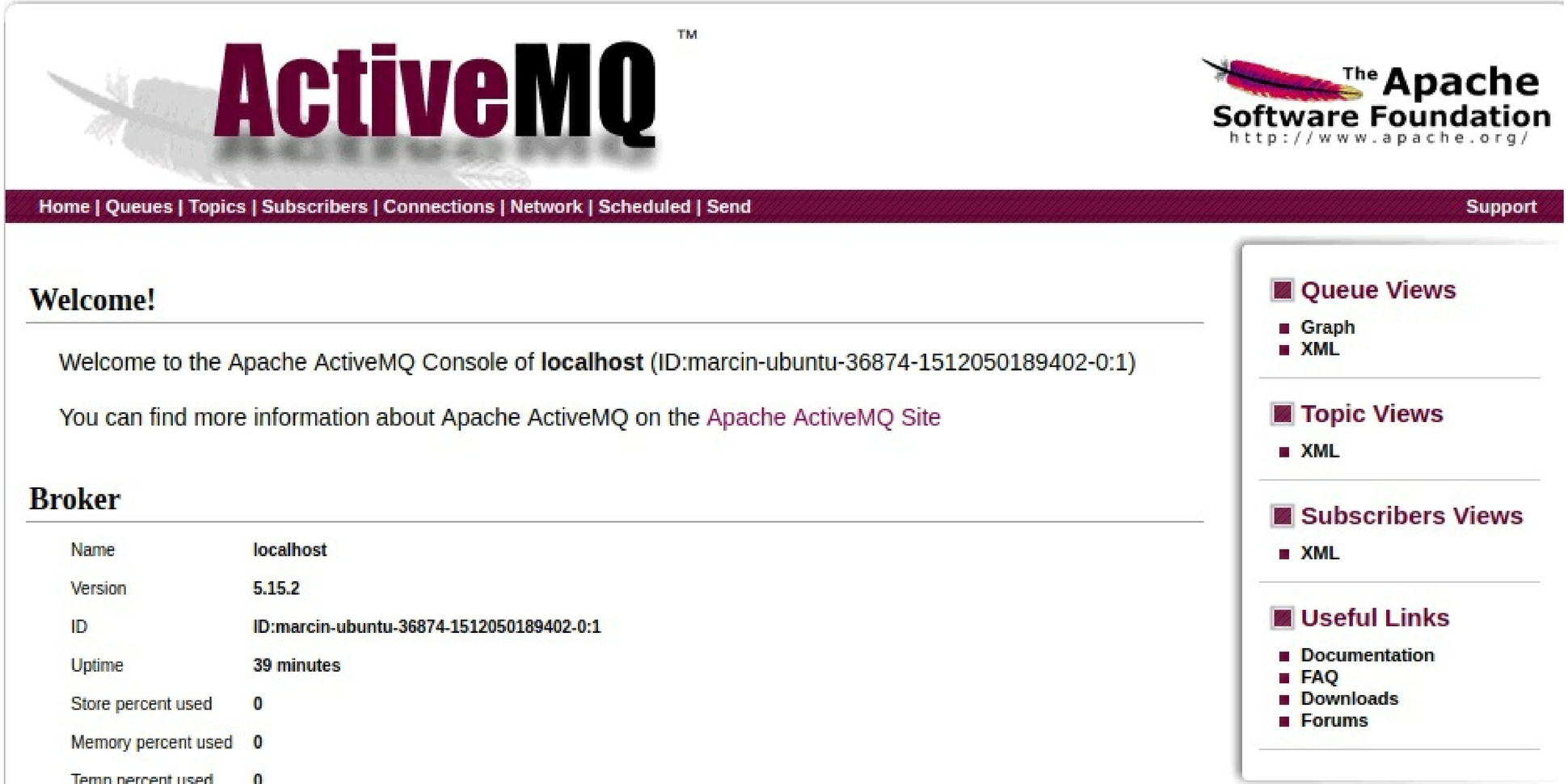
Jeżeli wszystko zostało pomyślnie zainstalowane i uruchomione powinniśmy uzyskać obraz podobny do tego na kolejnym slajdzie

ActiveMq - administracja

Po uzyskaniu dostępu do powyższego panelu administracyjnego, należy skonfigurować połączenie naszego ActiveMQ i w tym celu klikamy w link **Manage ActiveMQ broker**

W oknie które się pojawi wpisujemy jako login oraz hasło: **admin**. Po potwierdzeniu wprowadzonych danych, zostaniemy przekierowani do głównego ekranu ActiveMQ w którym zobaczymy szczegóły naszego połączenia oraz będziemy mieli dodatkowe opcje do konfiguracji kolejek czy tematów

Widok jaki powinniśmy otrzymać znajduje się na następnym slajdzie. Tym sposobem dochodzimy również do końca instalacji i konfiguracji ActiveMQ



Zadania

Wykonaj instalację Apache ActiveMQ na własnym komputerze

Komunikacja w JMS

Komunikacja w JMS

Dwa główne typy komunikacji w JMS które omówimy szczegółowo to:

- Point-To-Point (PTP)
- Publish/Subscribe

JMS Point-To-Point

JMS Point-to-Point

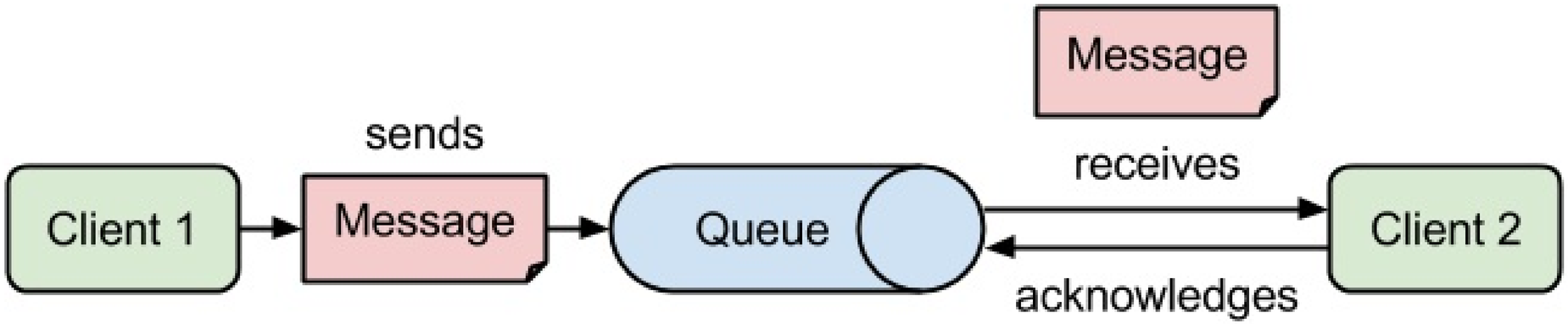
Wymiana wiadomości typu Point-to-Point zakłada trzy główne składniki w wymianie wiadomości:

- klient wysyłający wiadomości do kolejki - producent
- kolejka która przechowuje wiadomości
- klient odbierający wiadomości - konsument

W ogólnym założeniu wiadomość jest wysyłana przez producenta do kolejki i z tej kolejki pobierana przez konsumenta

Po odebraniu wiadomości z kolejki przez konsumenta, czynność ta może zostać potwierdzona automatycznie lub mechanizm ten można zaimplementować samodzielnie

Schemat JMS Point-to-Point - schemat



JMS Point-to-Point

Na podstawie powyższego schematu, cechy charakterystyczne które można wskazać to:

- każda wiadomość ma tylko jednego konsumenta
- wysyłający i odbierający wiadomości nie mają zależności czasowych, oznacza to że odbiorca może pobrać wiadomość z kolejki niezależnie od tego czy dany odbiorca istniał w momencie wysyłania wiadomości do kolejki
- odbiorca przesyła **potwierdzenie(ang. acknowledgement)** pozytywnego odczytania wiadomości. Automatyczne wysyłanie potwierdzenia odbywa się poprzez ustawienie parametru **`AUTO_ACKNOWLEDGE`** na obiekcie sesji, natomiast ustawiając **`CLIENT_ACKNOWLEDGE`** informujemy że trzeba samodzielnie potwierdzić otrzymanie wiadomości

Aby pokazać jak wygląda w praktyce wykorzystanie poznanej wiedzy, na kolejnych slajdach zobaczymy przykład kodu wykorzystującego wysyłanie wiadomości w sposób PTP

JMS Point-to-Point

Na początek należy upewnić się że mamy zainstalowanego i uruchomionego Apache ActiveMQ(instalacja opisywana wcześniej)

Następnie trzeba dołączyć do naszego projektu wymagane biblioteki, aby sprawnie tego dokonać wystarczy użyć do tego **Maven** i w naszym pliku **pom.xml** dodać zależność dotyczącą ActiveMQ:

```
<dependency>
  <groupid>org.apache.activemq</groupid>
  <artifactid>activemq-all</artifactid>
  <version>5.15.2</version>
</dependency>
```

JMS Point-to-Point - przykład

Tworzymy klasę producenta - czyli klasę wysyłającą wiadomości do kolejki

```
public class Producer {
  private String clientId;
  private Connection connection;
  private Session session;
  private MessageProducer messageProducer;
  public void closeConnection() throws JMSException {
    connection.close();
  }
```

JMS Point-to-Point - przykład

Tworzymy klasę producenta - czyli klasę wysyłającą wiadomości do kolejki

```
public class Producer {
  private String clientId;
  private Connection connection;
  private Session session;
  private MessageProducer messageProducer;
  public void closeConnection() throws JMSException {
    connection.close();
  }
```

metoda zamykająca połączenie

JMS Point-to-Point - przykład

```
public void create(String clientId, String queueName) throws JMSException {
   this.clientId = clientId;
   ConnectionFactory connectionFactory = new ActiveMQConnectionFactory(
   ActiveMQConnection.DEFAULT_BROKER_URL);
   connection = connectionFactory.createConnection();
   connection.setClientID(clientId);
   session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
   Queue queue = session.createQueue(queueName);
   messageProducer = session.createProducer(queue);
 }
```

JMS Point-to-Point - przykład

```
public void create(String clientId, String queueName) throws JMSException {
    this.clientId = clientId;
    ConnectionFactory connectionFactory = new ActiveMQConnectionFactory(
    ActiveMQConnection.DEFAULT_BROKER_URL);
    connection = connectionFactory.createConnection();
    connection.setClientID(clientId);
    session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
    Queue queue = session.createQueue(queueName);
    messageProducer = session.createProducer(queue);
  }
```

metoda wywoływana po utworzeniu klasy

JMS Point-to-Point - przykład

```
public void create(String clientId, String queueName) throws JMSException {
    this.clientId = clientId;
    ConnectionFactory connectionFactory = new ActiveMQConnectionFactory(
    ActiveMQConnection.DEFAULT_BROKER_URL);
    connection = connectionFactory.createConnection();
    connection.setClientID(clientId);
    session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
    Queue queue = session.createQueue(queueName);
    messageProducer = session.createProducer(queue);
  }
```

utworzenie obiektu ConnectionFactory

JMS Point-to-Point - przykład

```
public void create(String clientId, String queueName) throws JMSException {
    this.clientId = clientId;
    ConnectionFactory connectionFactory = new ActiveMQConnectionFactory(
    ActiveMQConnection.DEFAULT_BROKER_URL);
    connection = connectionFactory.createConnection();
    connection.setClientID(clientId);
    session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
    Queue queue = session.createQueue(queueName);
    messageProducer = session.createProducer(queue);
  }
```

utworzenie połączenia za pomocą obiektu ConnectionFactory

JMS Point-to-Point - przykład

```
public void create(String clientId, String queueName) throws JMSException {
    this.clientId = clientId;
    ConnectionFactory connectionFactory = new ActiveMQConnectionFactory(
    ActiveMQConnection.DEFAULT_BROKER_URL);
    connection = connectionFactory.createConnection();
    connection.setClientID(clientId);
    session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
    Queue queue = session.createQueue(queueName);
    messageProducer = session.createProducer(queue);
  }
```

ustawienie id klienta do identyfikacji

JMS Point-to-Point - przykład

```
public void create(String clientId, String queueName) throws JMSException {
    this.clientId = clientId;
    ConnectionFactory connectionFactory = new ActiveMQConnectionFactory(
    ActiveMQConnection.DEFAULT_BROKER_URL);
    connection = connectionFactory.createConnection();
    connection.setClientID(clientId);
    session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
    Queue queue = session.createQueue(queueName);
    messageProducer = session.createProducer(queue);
  }
```

utworzenie sesji za pomocą obiektu connection

JMS Point-to-Point - przykład

```
public void create(String clientId, String queueName) throws JMSException {
    this.clientId = clientId;
    ConnectionFactory connectionFactory = new ActiveMQConnectionFactory(
    ActiveMQConnection.DEFAULT_BROKER_URL);
    connection = connectionFactory.createConnection();
    connection.setClientID(clientId);
    session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
    Queue queue = session.createQueue(queueName);
    messageProducer = session.createProducer(queue);
  }
```

utworzenie kolejki za pomocą obiektu session

JMS Point-to-Point - przykład

```
public void create(String clientId, String queueName) throws JMSException {
    this.clientId = clientId;
    ConnectionFactory connectionFactory = new ActiveMQConnectionFactory(
    ActiveMQConnection.DEFAULT_BROKER_URL);
    connection = connectionFactory.createConnection();
    connection.setClientID(clientId);
    session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
    Queue queue = session.createQueue(queueName);
    messageProducer = session.createProducer(queue);
  }
```

utworzenie producenta za pomocą obiektu session

JMS Point-to-Point - przykład

```
 public void sendName(String name) throws JMSException {
  TextMessage textMessage = session.createTextMessage(name);
  messageProducer.send(textMessage);
  System.out.println(clientId + ": sent message: " + name);
  }
}
```

JMS Point-to-Point - przykład

```
 public void sendName(String name) throws JMSException {
   TextMessage textMessage = session.createTextMessage(name);
   messageProducer.send(textMessage);
   System.out.println(clientId + ": sent message: " + name);
   }
}
```

metoda wysyłająca wiadomość - w tym przypadku podane imię

JMS Point-to-Point - przykład

```
 public void sendName(String name) throws JMSException {
  TextMessage textMessage = session.createTextMessage(name);
  messageProducer.send(textMessage);
  System.out.println(clientId + ": sent message: " + name);
  }
}
```

utworzenie obiektu TextMessage za pomocą obiektu sesji

JMS Point-to-Point - przykład

```
 public void sendName(String name) throws JMSException {
  TextMessage textMessage = session.createTextMessage(name);
  messageProducer.send(textMessage);
  System.out.println(clientId + ": sent message: " + name);
  }
}
```

wysłanie wiadomości za pomocą obiektu producenta

JMS Point-to-Point - przykład

```
 public void sendName(String name) throws JMSException {
  TextMessage textMessage = session.createTextMessage(name);
  messageProducer.send(textMessage);
  System.out.println(clientId + ": sent message: " + name);
  }
}
```

wypisanie wiadomości w konsoli

JMS Point-to-Point - przykład

Tworzymy klasę konsumenta - czyli klasę pobierającą wiadomości z kolejki

```
public class Consumer {
  private String clientId;
  private Connection connection;
  private Session session;
  private MessageConsumer messageConsumer;
  public void closeConnection() throws JMSException {
    connection.close();
  }
}
```

JMS Point-to-Point - przykład

```
public void create(String clientId, String queueName) throws JMSException {
    this.clientId = clientId;
    ConnectionFactory connectionFactory = new ActiveMQConnectionFactory(
    ActiveMQConnection.DEFAULT_BROKER_URL);
    connection = connectionFactory.createConnection();
    connection.setClientID(clientId);
    session = connection.createSession(false, Session.CLIENT_ACKNOWLEDGE);
    Queue queue = session.createQueue(queueName);
    messageConsumer = session.createConsumer(queue);
    connection.start();
}
```

JMS Point-to-Point - przykład

```
public void create(String clientId, String queueName) throws JMSException {
    this.clientId = clientId;
    ConnectionFactory connectionFactory = new ActiveMQConnectionFactory(
    ActiveMQConnection.DEFAULT_BROKER_URL);
    connection = connectionFactory.createConnection();
    connection.setClientID(clientId);
    session = connection.createSession(false, Session.CLIENT_ACKNOWLEDGE);
    Queue queue = session.createQueue(queueName);
    messageConsumer = session.createConsumer(queue);
    connection.start();
}
```

tworzenie obiektu konsumenta za pomocą obiektu session

JMS Point-to-Point - przykład

```
public void create(String clientId, String queueName) throws JMSException {
    this.clientId = clientId;
    ConnectionFactory connectionFactory = new ActiveMQConnectionFactory(
    ActiveMQConnection.DEFAULT_BROKER_URL);
    connection = connectionFactory.createConnection();
    connection.setClientID(clientId);
    session = connection.createSession(false, Session.CLIENT_ACKNOWLEDGE);
    Queue queue = session.createQueue(queueName);
    messageConsumer = session.createConsumer(queue);
    connection.start();
}
```

wykonanie metody start() na obiekcie połączenia - obowiązkowe do poprawnego działania

```
 public String getMessage(int timeout, boolean acknowledge) throws JMSException {
    String receivedMessage = "no message";
    Message message = messageConsumer.receive(timeout);
    if(message != null) {
      TextMessage textMessage = (TextMessage) message;
      String text = textMessage.getText();
      if(acknowledge) {
        message.acknowledge();
        System.out.println(clientId + ": message acknowledged");
      } else {
        System.out.println(clientId + ": message not acknowledged");
      }
      receivedMessage = "Hello " + text + "!";
    } else {
      System.out.println(clientId + " - no message received");
    }
    return receivedMessage;
  }
}
```

```
public String getMessage(int timeout, boolean acknowledge) throws JMSException {
    String receivedMessage = "no message";
    Message message = messageConsumer.receive(timeout);
    if(message != null) {
      TextMessage textMessage = (TextMessage) message;
      String text = textMessage.getText();
      if(acknowledge) {
        message.acknowledge();
        System.out.println(clientId + ": message acknowledged");
      } else {
        System.out.println(clientId + ": message not acknowledged");
      }
      receivedMessage = "Hello " + text + "!";
    } else {
      System.out.println(clientId + " - no message received");
    }
    return receivedMessage;
  }
}
```

metoda pobierająca wiadomości

```
public String getMessage(int timeout, boolean acknowledge) throws JMSException {
    String receivedMessage = "no message";
    Message message = messageConsumer.receive(timeout);
    if(message != null) {
      TextMessage textMessage = (TextMessage) message;
      String text = textMessage.getText();
      if(acknowledge) {
        message.acknowledge();
        System.out.println(clientId + ": message acknowledged");
      } else {
        System.out.println(clientId + ": message not acknowledged");
      }
      receivedMessage = "Hello " + text + "!";
    } else {
      System.out.println(clientId + " - no message received");
    }
    return receivedMessage;
  }
}
```

odebranie wiadomości za pomocą metody receive()

```
public String getMessage(int timeout, boolean acknowledge) throws JMSException {
    String receivedMessage = "no message";
    Message message = messageConsumer.receive(timeout);
    if(message != null) {
      TextMessage textMessage = (TextMessage) message;
      String text = textMessage.getText();
      if(acknowledge) {
        message.acknowledge();
        System.out.println(clientId + ": message acknowledged");
      } else {
        System.out.println(clientId + ": message not acknowledged");
      }
      receivedMessage = "Hello " + text + "!";
    } else {
      System.out.println(clientId + " - no message received");
    }
    return receivedMessage;
  }
}
```

rzutowanie wiadomości na typ TextMessage

```
public String getMessage(int timeout, boolean acknowledge) throws JMSException {
    String receivedMessage = "no message";
    Message message = messageConsumer.receive(timeout);
    if(message != null) {
      TextMessage textMessage = (TextMessage) message;
      String text = textMessage.getText();
      if(acknowledge) {
        message.acknowledge();
        System.out.println(clientId + ": message acknowledged");
      } else {
        System.out.println(clientId + ": message not acknowledged");
      }
      receivedMessage = "Hello " + text + "!";
    } else {
      System.out.println(clientId + " - no message received");
    }
    return receivedMessage;
  }
}
```

sprawdzenie czy do wiadomości ma być wysłane potwierdzenie odczytania

```
public String getMessage(int timeout, boolean acknowledge) throws JMSException {
    String receivedMessage = "no message";
    Message message = messageConsumer.receive(timeout);
    if(message != null) {
      TextMessage textMessage = (TextMessage) message;
      String text = textMessage.getText();
      if(acknowledge) {
        message.acknowledge();
        System.out.println(clientId + ": message acknowledged");
      } else {
        System.out.println(clientId + ": message not acknowledged");
      }
      receivedMessage = "Hello " + text + "!";
    } else {
      System.out.println(clientId + " - no message received");
    }
    return receivedMessage;
  }
}
```

ręczne wysłanie potwierdzenia metodą acknowledge()

JMS Point-to-Point - przykład

Wykorzystanie utworzonych klas w metodzie main:

```
public static void main(String[] args) throws JMSException {
        Producer producer = new Producer();
        producer.create("producer-name", "pointtopoint.q");
        Consumer consumer = new Consumer();
        conscumer.create("consumer-name", "pointtopoint.q");
        producer.sendName("Test Name");
        String receivedName = conscumer.getMessage(1000, true);
        System.out.println("Name from producer: " + receivedName);
        producer.closeConnection();
        consumer.closeConnection();
}
```

JMS Point-to-Point - przykład

Wykorzystanie utworzonych klas w metodzie main:

```
public static void main(String[] args) throws JMSException {
        Producer producer = new Producer();
        producer.create("producer-name", "pointtopoint.q");
        Consumer consumer = new Consumer();
        conscumer.create("consumer-name", "pointtopoint.q");
        producer.sendName("Test Name");
        String receivedName = conscumer.getMessage(1000, true);
        System.out.println("Name from producer: " + receivedName);
        producer.closeConnection();
        consumer.closeConnection();
}
```

połączenie do kolejki pointtopoint.q

JMS Point-to-Point - przykład

Wykorzystanie utworzonych klas w metodzie main:

```
public static void main(String[] args) throws JMSException {
        Producer producer = new Producer();
        producer.create("producer-name", "pointtopoint.q");
        Consumer consumer = new Consumer();
        conscumer.create("consumer-name", "pointtopoint.q");
        producer.sendName("Test Name");
        String receivedName = conscumer.getMessage(1000, true);
        System.out.println("Name from producer: " + receivedName);
        producer.closeConnection();
        consumer.closeConnection();
}
```

wysłanie wiadomości przez producenta

JMS Point-to-Point - przykład

Wykorzystanie utworzonych klas w metodzie main:

```
public static void main(String[] args) throws JMSException {
        Producer producer = new Producer();
        producer.create("producer-name", "pointtopoint.q");
        Consumer consumer = new Consumer();
        conscumer.create("consumer-name", "pointtopoint.q");
        producer.sendName("Test Name");
        String receivedName = conscumer.getMessage(1000, true);
        System.out.println("Name from producer: " + receivedName);
        producer.closeConnection();
        consumer.closeConnection();
}
```

odebranie wiadomości przez konsumenta

JMS Point-to-Point - przykład

Możliwe jest również wykorzystanie interfejsu **`MessageListener`** w klasie odbierającej wiadomości, wtedy klasa ta będzie automatycznie odbierała wiadomości które przychodzą do danej kolejki

Aby wykorzystać interfejs MessageListener nasza klasa dobierająca wiadomości z kolejki musi:

- zaimplementować interfejs **`MessageListener`**
- nadpisać publiczną metodę **`onMessage()`** typu void tego interfejsu

Nadpisywana metoda onMessage jako parametr przyjmuje obiekt wiadomości typu **`Message`** i obsługę tego obiektu można potraktować tak samo jak we wcześniej napisanej metodzie getMessage()

Zadania

JMS Publish / Subscribe

JMS Publish/Subscribe

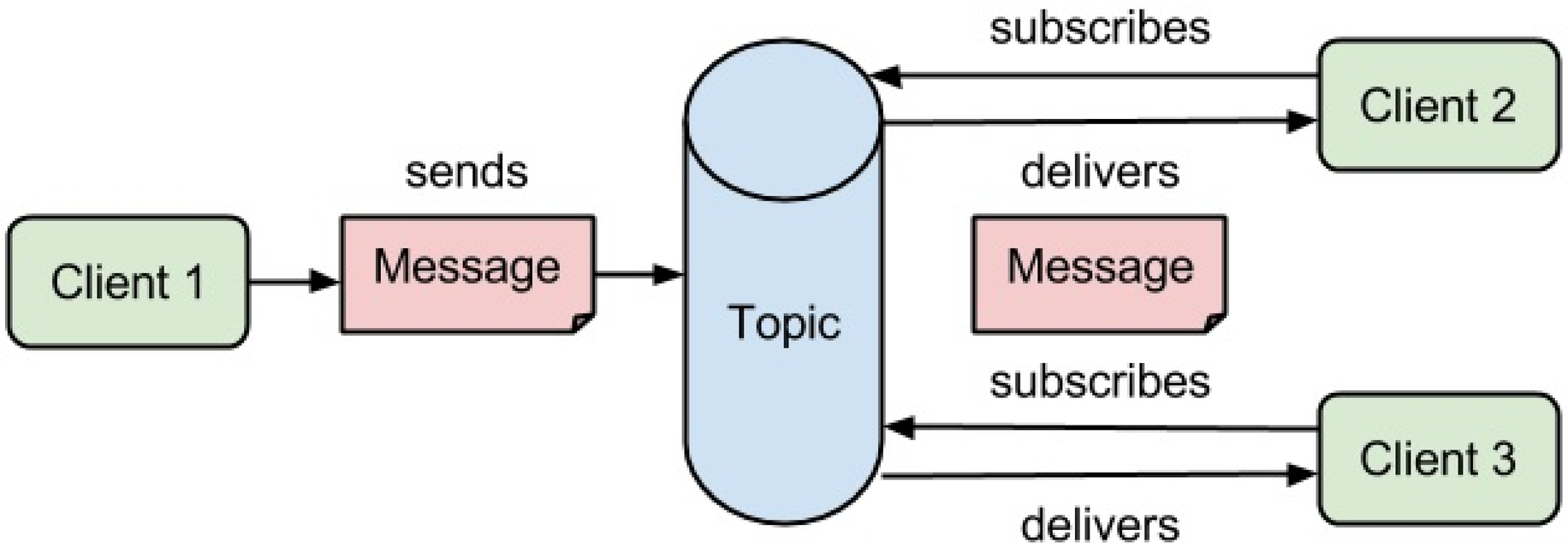
Drugim najważniejszym typem komunikacji za pomocą JMS jest model **Publish/Subscribe**

W tym modelu aplikacja wysyłająca wiadomości, znany z mechanizmu Point-To-Point producent określany jest w tym modelu jako **publikator(ang. Publisher)** wysyła wiadomości do wspomnianego wcześniej **tematu(ang. Topic)**

Druga strona odbierająca wiadomości którą w przypadku modelu Point-To-Point nazywaliśmy konsumentem będzie teraz określana jako **subskrybent(ang. Subscriber)**

Dzieje się tak ponieważ klasa odbierająca wiadomości korzysta jako jedna z wielu z danego źródła czyli w tym przypadku z tematu, co można porównać schematem działania do subskrypcji znanej z komunikacji mailowej

JMS Publish/Subscribe - schemat



JMS Publish/Subscribe

Model Publish/Subscribe charakteryzuje się tym że:

- każda wysyłana wiadomość przez producenta może mieć **wielu odbiorców**
- producenci i odbiorcy wiadomości w przeciwieństwie do modelu Point-To-Point mają zależność czasową, czyli odbiorca wiadomości z danego tematu może otrzymać dane wiadomości jeżeli istniał przed wysłaniem danej wiadomości do tematu
- Tak więc jeżeli subskrybent nie istnieje w momencie wysyłania wiadomości na dany temat nie może już po utworzeniu otrzymać "zaległych" wiadomości
- praktyczny przykład kodu wykorzystujący mechanizm Publish/Subscribe zostanie przedstawiony na kolejnych slajdach

JMS Publish/Subscribe

Tworzymy klasę publikatora:

```
public class Publisher {
  private String clientId;
  private Connection connection;
  private Session session;
  private MessageProducer messageProducer;
  public void closeConnection() throws JMSException {
    connection.close();
  }
```

JMS Publish/Subscribe

```
public void create(String clientId, String topicName) throws JMSException {
   this.clientId = clientId;
   ConnectionFactory connectionFactory = new ActiveMQConnectionFactory(
     ActiveMQConnection.DEFAULT_BROKER_URL);
   connection = connectionFactory.createConnection();
   connection.setClientID(clientId);
   session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
   Topic topic = session.createTopic(topicName);
   messageProducer = session.createProducer(topic);
 }
```

JMS Publish/Subscribe

```
public void create(String clientId, String topicName) throws JMSException {
   this.clientId = clientId;
   ConnectionFactory connectionFactory = new ActiveMQConnectionFactory(
     ActiveMQConnection.DEFAULT_BROKER_URL);
   connection = connectionFactory.createConnection();
   connection.setClientID(clientId);
   session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
   Topic topic = session.createTopic(topicName);
   messageProducer = session.createProducer(topic);
 }
```

tworzymy obiekt typu Topic czyli temat

JMS Publish/Subscribe

```
public void sendName(String name) throws JMSException {
    TextMessage textMessage = session.createTextMessage(name);
    messageProducer.send(textMessage);
    System.out.println(clientId + " - sent name: " + name);
}
```

JMS Publish/Subscribe

Tworzymy klasę subskrybenta:

```
public class Subscriber {
    private String clientId;
    private Connection connection;
    private Session session;
    private MessageConsumer messageConsumer;
    public void closeConnection() throws JMSException {
        connection.close();
    }
}
```

JMS Publish/Subscribe

```
public void create(String clientId, String topicName) throws JMSException {
    this.clientId = clientId;
    ConnectionFactory connectionFactory = new ActiveMQConnectionFactory(
        ActiveMQConnection.DEFAULT_BROKER_URL);
    connection = connectionFactory.createConnection();
    connection.setClientID(clientId);
    session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
    Topic topic = session.createTopic(topicName);
    messageConsumer = session.createConsumer(topic);
    connection.start();
}
```

JMS Publish/Subscribe

```
public String getName(int timeout) throws JMSException {
    String receivedName = "no name was sent";
    Message message = messageConsumer.receive(timeout);
    if (message != null) {
      TextMessage textMessage = (TextMessage) message;
      String text = textMessage.getText();
      receivedName = "Hello " + text + "!";
    } else {
      System.out.println(clientId + ": no message received");
    }
    return receivedName;
}
```

Wykorzystanie wcześniej utworzonych klas w metodzie main:

```
public static void main(String[] args) throws JMSException {
  Publisher publisher = new Publisher();
  publisher.create("publisher-name", "publishsubscribe.t");
  Subscriber subscriber1 = new Subscriber();
  subscriber1.create("subscriber1-name", "publishsubscribe.t");
  Subscriber subscriber2 = new Subscriber();
  subscriber2.create("subscriber2-name", "publishsubscribe.t");
  publisher.sendName("Test Name");
  String recName1 = subscriber1.getName(1000);
  String recdName2 = subscriber2.getName(1000);
  System.out.println("Received name1: " + recName1 + ", name2: " + recdName2);
  // zamknięcie obiektów publikatora oraz subskrybentów
}
```

Wykorzystanie wcześniej utworzonych klas w metodzie main:

```
public static void main(String[] args) throws JMSException {
  Publisher publisher = new Publisher();
  publisher.create("publisher-name", "publishsubscribe.t");
  Subscriber subscriber1 = new Subscriber();
  subscriber1.create("subscriber1-name", "publishsubscribe.t");
  Subscriber subscriber2 = new Subscriber();
  subscriber2.create("subscriber2-name", "publishsubscribe.t");
  publisher.sendName("Test Name");
  String recName1 = subscriber1.getName(1000);
  String recdName2 = subscriber2.getName(1000);
  System.out.println("Received name1: " + recName1 + ", name2: " + recdName2);
  // zamknięcie obiektów publikatora oraz subskrybentów
}
```

utworzenie dwóch subskrybentów dla tematu publishsubscribe.t

Wykorzystanie wcześniej utworzonych klas w metodzie main:

```
public static void main(String[] args) throws JMSException {
  Publisher publisher = new Publisher();
  publisher.create("publisher-name", "publishsubscribe.t");
  Subscriber subscriber1 = new Subscriber();
  subscriber1.create("subscriber1-name", "publishsubscribe.t");
  Subscriber subscriber2 = new Subscriber();
  subscriber2.create("subscriber2-name", "publishsubscribe.t");
  publisher.sendName("Test Name");
  String recName1 = subscriber1.getName(1000);
  String recdName2 = subscriber2.getName(1000);
  System.out.println("Received name1: " + recName1 + ", name2: " + recdName2);
  // zamknięcie obiektów publikatora oraz subskrybentów
}
```

Wykorzystanie wcześniej utworzonych klas w metodzie main:

```
public static void main(String[] args) throws JMSException {
  Publisher publisher = new Publisher();
  publisher.create("publisher-name", "publishsubscribe.t");
  Subscriber subscriber1 = new Subscriber();
  subscriber1.create("subscriber1-name", "publishsubscribe.t");
  Subscriber subscriber2 = new Subscriber();
  subscriber2.create("subscriber2-name", "publishsubscribe.t");
  publisher.sendName("Test Name");
  String recName1 = subscriber1.getName(1000);
  String recdName2 = subscriber2.getName(1000);
  System.out.println("Received name1: " + recName1 + ", name2: " + recdName2);
  // zamknięcie obiektów publikatora oraz subskrybentów
}
```

utworzenie dwóch subskrybentów dla tematu publishsubscribe.t

Zadania

JMS i Spring Boot

JMS w Spring Boot

Do tej pory używaliśmy JMS w zwykłych aplikacjach Java. W większości przypadków JMS będzie częścią większych systemów które mogą opierać się o Spring, dlatego na kolejnych slajdach poznamy jak w praktyce wykorzystać poznaną technologię JMS i zastosować ją w Spring Boot.

JMS w Spring Boot

W naszym przykładzie napiszemy prostą aplikację która wykorzysta JMS i dwie główne klasy które utworzymy **Producer** i **Consumer**.

Aplikacja będzie przyjmowała parametr przekazywany przez adres przeglądarki który zostanie przesłany do kolejki w **ActiveMQ**.

Po wejściu na odpowiedni adres w przeglądarce, nastąpi automatyczne odebranie wiadomości która znajduje się w kolejce.

Zmiany będzie można obserwować logując się do panelu administracyjnego ActiveMQ tak jak robiliśmy to przy instalacji ActiveMQ.

Utworzenie projektu

Na początek utworzymy nowy projekt Spring Boot lub skorzystamy z projektu który wykorzystywany był wcześniej. Ważne aby zawierał on zależności:

```
<dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-jms</artifactId>
</dependency>
<dependency>
        <groupId>org.apache.activemq</groupId>
        <artifactId>activemq-broker</artifactId>
</dependency>
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Konfiguracja application.properties

Zależnie od instalacji ActiveMQ konfiguracja ustawień naszego brokera oraz haseł odbywa się poprzez nadpisanie parametrów:

```
spring.activemq.broker-url=tcp://localhost:61616
spring.activemq.user=admin
spring.activemq.password=admin
```

W przypadku punktu docelowego, w naszym przypadku kolejki do której nasza aplikacja będzie wysyłała wiadomości i z której będzie je odczytywała, wartość tą ustalamy poprzez parametr:

```
jms.queue.destination="queue-name"
```

JMS w Spring Boot

W przypadku zwykłej aplikacji Java wykorzystującej JMS aby wysyłać lub odbierać wiadomości musieliśmy utworzyć obiekty typu provider, session i wykorzystywać wprost wszystkie mechanizmy które oferuje JMS API.

W przypadku Spring Boot wszystko staje się o wiele łatwiejsze ponieważ mamy do dyspozycji **JmsTemplate** czyli klasę pomocniczą która oszczędza nam dużo pracy i pisania powtarzalnego kodu.

Klasa JmsTemplate jest automatycznie tworzona przez Spring na podstawie poprzednio ustawionej konfiguracji i za pomocą jej metod będziemy wysyłać i odbierać wiadomości

Klasa JmsConsumer

```
@Component
public class JmsConsumer {
        @Autowired
        JmsTemplate jmsTemplate;

        @Value("${jms.queue.destination}")
        String destinationQueue;

        public String receive(){
                return (String)jmsTemplate.receiveAndConvert(destinationQueue);
        }
}
```

Klasa JmsConsumer

```
@Component
public class JmsConsumer {
        @Autowired
        JmsTemplate jmsTemplate;

        @Value("${jms.queue.destination}")
        String destinationQueue;

        public String receive(){
                return (String)jmsTemplate.receiveAndConvert(destinationQueue);
        }
}
```

Oznaczamy klasę JmsConsumer jako komponent

Klasa JmsConsumer

```
@Component
public class JmsConsumer {
        @Autowired
        JmsTemplate jmsTemplate;

        @Value("${jms.queue.destination}")
        String destinationQueue;

        public String receive(){
                return (String)jmsTemplate.receiveAndConvert(destinationQueue);
        }
}
```

Automatycznie dołączamy klasę JmsTemplate

Klasa JmsConsumer

```
@Component
public class JmsConsumer {
        @Autowired
        JmsTemplate jmsTemplate;

        @Value("${jms.queue.destination}")
        String destinationQueue;

        public String receive(){
                return (String)jmsTemplate.receiveAndConvert(destinationQueue);
        }
}
```

Odczytujemy wartość parametru zapisanego wcześniej w application.properties

Klasa JmsConsumer

```
@Component
public class JmsConsumer {
        @Autowired
        JmsTemplate jmsTemplate;

        @Value("${jms.queue.destination}")
        String destinationQueue;

        public String receive(){
                return (String)jmsTemplate.receiveAndConvert(destinationQueue);
        }
}
```

Wykorzystujemy metodę receiveAndConvert do pobrania wiadomości z kolejki

Klasa JmsConsumer

```
@Component
public class JmsConsumer {
        @Autowired
        JmsTemplate jmsTemplate;

        @Value("${jms.queue.destination}")
        String destinationQueue;

        public String receive(){
                return (String)jmsTemplate.receiveAndConvert(destinationQueue);
        }
}
```

Argumentem metody będzie nazwa kolejki z której chcemy pobrać wiadomość

Tworzenia kontrolera

Aby za pomocą odpowiedniego adresu wpisywanego w przeglądarce wysyłać i odbierać wiadomości utworzymy nowy kontroler w którym skorzystamy z utworzonego wcześniej interfejsu JmsClient.

Poza tym utworzymy dwa adresy które będą wykonywały konkretne akcje:

- **`@RequestMapping("/produce")`** - po wejściu na ten adres i dodania parametru "msg" wiadomość w nim zawarta zostanie wysłana do przeglądarki
- **`@RequestMapping("/receive")`** - po wejściu na ten adres zostanie odczytana wiadomość z kolejki

Wyniki działania zostaną przedstawione bezpośrednio w przeglądarce a kod kontrolera zobaczymy na kolejnym slajdzie.

Tworzenia kontrolera

```
@RestController
public class WebController {
        @Autowired
        JmsClient jsmClient;

        @RequestMapping(value="/produce")
        public String produce(@RequestParam("msg")String msg){
                jsmClient.send(msg);
                return "Done";
        }

        @RequestMapping(value="/receive")
        public String receive(){
                return jsmClient.receive();
        }
}
```

Tworzenia kontrolera

```
@RestController
public class WebController {
        @Autowired
        JmsClient jsmClient;

        @RequestMapping(value="/produce")
        public String produce(@RequestParam("msg")String msg){
                jsmClient.send(msg);
                return "Done";
        }

        @RequestMapping(value="/receive")
        public String receive(){
                return jsmClient.receive();
        }
}
```

Oznaczamy klasę jako @RestController czyli @Controller oraz @ResponseBody

Tworzenia kontrolera

```
@RestController
public class WebController {
        @Autowired
        JmsClient jsmClient;

        @RequestMapping(value="/produce")
        public String produce(@RequestParam("msg")String msg){
                jsmClient.send(msg);
                return "Done";
        }

        @RequestMapping(value="/receive")
        public String receive(){
                return jsmClient.receive();
        }
}
```

Pod tym adresem będziemy wysyłać wiadomość dodając ją w parametrze

Tworzenia kontrolera

```
@RestController
public class WebController {
        @Autowired
        JmsClient jsmClient;

        @RequestMapping(value="/produce")
        public String produce(@RequestParam("msg")String msg){
                jsmClient.send(msg);
                return "Done";
        }

        @RequestMapping(value="/receive")
        public String receive(){
                return jsmClient.receive();
        }
}
```

Pod tym adresem odczytamy wiadomość z kolejki

Uruchomienie aplikacji

Możemy uruchomić napisaną aplikację podobnie jak każdą inną aplikację tworzoną w Spring Boot, a następnie warto zalogować się do panelu administracyjnego ActiveMQ aby na bieżąco śledzić wiadomości jakie trafiają do kolejki.

Dla przypomnienia domyślnym adresem administracyjnym dla ActiveMQ jest **localhost:8161/admin/** a domyślny login oraz hasło to **admin**

Aby wysłać wiadomość za pomocą utworzonych klas oraz kontrolera wprowadzamy w przeglądarce np. **`http://localhost:8080/produce?msg=test`** natomiast aby wyświetlić wiadomość znajdującą się w kolejce wprowadzamy w przeglądarce **`http://localhost:8080/receive`**

Jest to dobry start do dalszego rozwijania aplikacji np. o przesyłanie obiektów i automatyczne bindowanie ich za pomocą **jackson-databind**

Klasa JmsProducer

```
@Component
public class JmsProducer {
        @Autowired
        JmsTemplate jmsTemplate;

        @Value("${jms.queue.destination}")
        String destinationQueue;

        public void send(String msg){
                jmsTemplate.convertAndSend(destinationQueue, msg);
        }
}
```

Klasa JmsProducer

```
@Component
public class JmsProducer {
        @Autowired
        JmsTemplate jmsTemplate;

        @Value("${jms.queue.destination}")
        String destinationQueue;

        public void send(String msg){
                jmsTemplate.convertAndSend(destinationQueue, msg);
        }
}
```

Wykorzystujemy metodę convertAndSend do wysłania wiadomości

Klasa JmsProducer

```
@Component
public class JmsProducer {
        @Autowired
        JmsTemplate jmsTemplate;

        @Value("${jms.queue.destination}")
        String destinationQueue;

        public void send(String msg){
                jmsTemplate.convertAndSend(destinationQueue, msg);
        }
}
```

Pierwszym argumentem metody jest nazwa kolejki do której będzie wysłana wiadomość

Klasa JmsProducer

```
@Component
public class JmsProducer {
        @Autowired
        JmsTemplate jmsTemplate;

        @Value("${jms.queue.destination}")
        String destinationQueue;

        public void send(String msg){
                jmsTemplate.convertAndSend(destinationQueue, msg);
        }
}
```

Drugim parametrem metody jest wiadomość do wysłania

Interfejs JmsClient

Tworzymy interfejs JmsClient posiadający dwie metody:

```
public interface JmsClient {
        public void send(String msg);
        public String receive();
}
```

Na następnym slajdzie zobaczymy implementację powyższego interfejsu

Klasa JmsClientImpl implementacja JmsClient

```
@Service
public class JmsClientImpl implements JmsClient{
        @Autowired
        JmsConsumer jmsConsumer;
        @Autowired
        JmsProducer jmsProducer;
        @Override
        public void send(String msg) {
                jmsProducer.send(msg);
        }
        @Override
        public String receive() {
                return jmsConsumer.receive();
        }
}
```

Klasa JmsClientImpl implementacja JmsClient

```
@Service
public class JmsClientImpl implements JmsClient{
        @Autowired
        JmsConsumer jmsConsumer;
        @Autowired
        JmsProducer jmsProducer;
        @Override
        public void send(String msg) {
                jmsProducer.send(msg);
        }
        @Override
        public String receive() {
                return jmsConsumer.receive();
        }
}
```

Oznaczamy klasę adnotacją Service

Klasa JmsClientImpl implementacja JmsClient

```
@Service
public class JmsClientImpl implements JmsClient{
        @Autowired
        JmsConsumer jmsConsumer;
        @Autowired
        JmsProducer jmsProducer;
        @Override
        public void send(String msg) {
                jmsProducer.send(msg);
        }
        @Override
        public String receive() {
                return jmsConsumer.receive();
        }
}
```

Dołączamy wcześniej utworzoną klasę JmsConsumer

Klasa JmsClientImpl implementacja JmsClient

```
@Service
public class JmsClientImpl implements JmsClient{
        @Autowired
        JmsConsumer jmsConsumer;
        @Autowired
        JmsProducer jmsProducer;
        @Override
        public void send(String msg) {
                jmsProducer.send(msg);
        }
        @Override
        public String receive() {
                return jmsConsumer.receive();
        }
}
```

Dołączamy wcześniej utworzoną klasę JmsProducer

Klasa JmsClientImpl implementacja JmsClient

```
@Service
public class JmsClientImpl implements JmsClient{
        @Autowired
        JmsConsumer jmsConsumer;
        @Autowired
        JmsProducer jmsProducer;
        @Override
        public void send(String msg) {
                jmsProducer.send(msg);
        }
        @Override
        public String receive() {
                return jmsConsumer.receive();
        }
}
```

Implementjemy metodę wysyłającą wiadomość do kolejki

Klasa JmsClientImpl implementacja JmsClient

```
@Service
public class JmsClientImpl implements JmsClient{
        @Autowired
        JmsConsumer jmsConsumer;
        @Autowired
        JmsProducer jmsProducer;
        @Override
        public void send(String msg) {
                jmsProducer.send(msg);
        }
        @Override
        public String receive() {
                return jmsConsumer.receive();
        }
}
```

Implementjemy metodę odbierającą wiadomość z kolejki

Tworzenia kontrolera

Aby za pomocą odpowiedniego adresu wpisywanego w przeglądarce wysyłać i odbierać wiadomości utworzymy nowy kontroler w którym skorzystamy z utworzonego wcześniej interfejsu JmsClient.

Poza tym utworzymy dwa adresy które będą wykonywały konkretne akcje:

- **`@RequestMapping("/produce")`** - po wejściu na ten adres i dodania parametru "msg" wiadomość w nim zawarta zostanie wysłana do przeglądarki
- **`@RequestMapping("/receive")`** - po wejściu na ten adres zostanie odczytana wiadomość z kolejki

Wyniki działania zostaną przedstawione bezpośrednio w przeglądarce a kod kontrolera zobaczymy na kolejnym slajdzie.

Tworzenia kontrolera

```
@RestController
public class WebController {
        @Autowired
        JmsClient jsmClient;

        @RequestMapping(value="/produce")
        public String produce(@RequestParam("msg")String msg){
                jsmClient.send(msg);
                return "Done";
        }

        @RequestMapping(value="/receive")
        public String receive(){
                return jsmClient.receive();
        }
}
```

Tworzenia kontrolera

```
@RestController
public class WebController {
        @Autowired
        JmsClient jsmClient;

        @RequestMapping(value="/produce")
        public String produce(@RequestParam("msg")String msg){
                jsmClient.send(msg);
                return "Done";
        }

        @RequestMapping(value="/receive")
        public String receive(){
                return jsmClient.receive();
        }
}
```

Oznaczamy klasę jako @RestController czyli @Controller oraz @ResponseBody

Tworzenia kontrolera

```
@RestController
public class WebController {
        @Autowired
        JmsClient jsmClient;

        @RequestMapping(value="/produce")
        public String produce(@RequestParam("msg")String msg){
                jsmClient.send(msg);
                return "Done";
        }

        @RequestMapping(value="/receive")
        public String receive(){
                return jsmClient.receive();
        }
}
```

Pod tym adresem będziemy wysyłać wiadomość dodając ją w parametrze

Tworzenia kontrolera

```
@RestController
public class WebController {
        @Autowired
        JmsClient jsmClient;

        @RequestMapping(value="/produce")
        public String produce(@RequestParam("msg")String msg){
                jsmClient.send(msg);
                return "Done";
        }

        @RequestMapping(value="/receive")
        public String receive(){
                return jsmClient.receive();
        }
}
```

Pod tym adresem odczytamy wiadomość z kolejki

Uruchomienie aplikacji

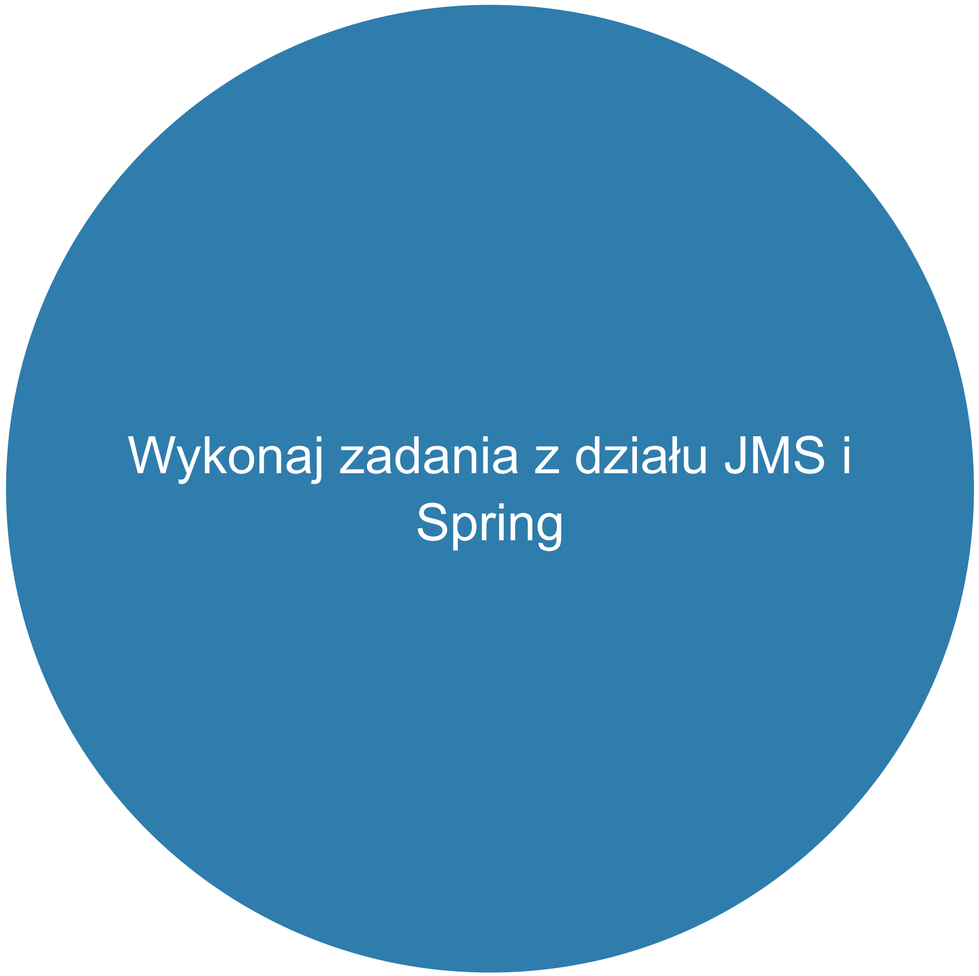
Możemy uruchomić napisaną aplikację podobnie jak każdą inną aplikację tworzoną w Spring Boot, a następnie warto zalogować się do panelu administracyjnego ActiveMQ aby na bieżąco śledzić wiadomości jakie trafiają do kolejki.

Dla przypomnienia domyślnym adresem administracyjnym dla ActiveMQ jest **localhost:8161/admin/** a domyślny login oraz hasło to **admin**

Aby wysłać wiadomość za pomocą utworzonych klas oraz kontrolera wprowadzamy w przeglądarce np. **`http://localhost:8080/produce?msg=test`** natomiast aby wyświetlić wiadomość znajdującą się w kolejce wprowadzamy w przeglądarce **`http://localhost:8080/receive`**

Jest to dobry start do dalszego rozwijania aplikacji np. o przesyłanie obiektów i automatyczne bindowanie ich za pomocą **jackson-databind**

Zadania



KONIEC