JMS with ActiveMQ

Planning

- Introducing ActiveMQ
- Building JMS applications
- Using ActiveMQ features
- ActiveMQ, Spring and JMS
- ActiveMQ configuration

Apache ActiveMQ

- Open-source message broker
  - Developed at the Apache Software Foundation
    - Very liberal software license
  - Active community
  - Commercial support available

Apache ActiveMQ

- Some features:
  - Support for JMS, J2EE, Spring, ...
  - Cross-language support
  - Advanced features
  - Multiple deployment topologies and failover options

Apache ActiveMQ

- Support for Apache Camel
  - EIP-based integration
  - Configured through
    - Java DSL
    - Spring XML
    - Scala DSL
  - Large set of components available (over 70)
    - File, FTP, HTTP, web services, JMS, JPA, Atom, Google App Engine, HDFS, HL7, IRC, JCR, ...

Apache ActiveMQ

- Installation procedure
  - Unzip/untar the distribution
  - Run bin/activemq
- Managing the broker
  - Using JMX
  - Using the web console

Planning

- Introducing ActiveMQ
- Building JMS applications
- Using ActiveMQ features
- ActiveMQ, Spring and JMS
- ActiveMQ configuration

Building JMS applications

- Java Message Service
- Connecting to the broker
- Sending messages
- Receiving messages
- Exercise
- JMS messages
- Message selectors
- Durable subscriptions

Java Message Service

- Common API to interact with MOM
- Versions
  - JMS 1.0.2
  - JMS 1.1
    Single APIs to work with both messaging domains

Java Message Service

- Messaging domains
  - Point-to-point messaging domain
    aka queues
  - Publish/subscribe messaging domain
    aka topics

Java Message Service

- ConnectionFactory
- Destination
- Connection
- Session
- MessageConsumer and MessageListener
- MessageProducer

Connecting to the broker

- ConnectionFactory, Connection and Session

```
ConnectionFactory factory =
       new ActiveMQConnectionFactory("tcp://localhost:61616");
Connection connection = factory.createConnection();
Session session =
       connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
```

Connecting to the broker

- ConnectionFactory
  - Sometimes stored in JNDI as a managed object
  - ActiveMQ specific implementation
    - Requires broker URL
    - Optionally set broker specific options (user, password, sync/async send, ...)

Connecting to the broker

- Connection
  - Created from ConnectionFactory
    - createConnection()
    - createConnection(String username, String password)
  - Thread-safe
  - start()/stop()
  - ExceptionListener

Connecting to the broker

- Session
  - Created from Connection.createSession()
    - true (transacted) or false
    - acknowledgeMode:
      - AUTO_ACKNOWLEDGE
      - CLIENT_ACKNOWLEDGE
      - DUPS_OK_ACKNOWLEDGE
  - commit()/rollback()/recover()
  - Single-threaded context for interaction with broker

Sending messages

- MessageProducer
  - allows setting message options for all messages
    - delivery mode (persistent or not)
    - time to live
    - priority
  - send() methods

Sending messages

- Sending messages to a queue
  - Use Session to create
    - Destination
    - MessageProducer
    - Messages

```
Destination destination = session.createQueue(QUEUE_NAME);
MessageProducer producer = session.createProducer(destination);

for (int i = 0; i < 1000; i++) {
    Message message = session.createTextMessage("Message # " + i);
    producer.send(message);
}
```

Sending messages

- Example when using JMS transactions

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

Destination destination = session.createQueue(QUEUE_NAME);
MessageProducer producer = session.createProducer(destination);

for (int i = 0; i < 1000; i++) {
    Message message = session.createTextMessage("Message # " + i);
    producer.send(message);
}

session.commit();
```

Sending messages

- Send messages to a topic
  - JMS 1.1: Unified API for both domains, only one line different

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

Destination destination = session.createTopic(TOPIC_NAME);
MessageProducer producer = session.createProducer(destination);

for (int i = 0; i < 1000; i++) {
    Message message = session.createTextMessage("Message # " + i);
    producer.send(message);
}
```

Receiving messages

- Receiving a message synchronously
  - receive(), receive(int timeout), receiveNoWait()
  - Build loop yourself to receive multiple messages

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

Destination destination = session.createQueue(QUEUE_NAME);
MessageConsumer consumer = session.createConsumer(destination);

TextMessage message = (TextMessage) consumer.receive();
LOGGER.info("Received: " + message.getText());
```

Receiving messages

- Receiving message asynchronously
  - Register MessageListener with MessageConsumer
  - start() the connection

```
Destination destination = session.createQueue(QUEUE_NAME);
MessageConsumer consumer = session.createConsumer(destination);

consumer.setMessageListener(new MessageListener() {
    public void onMessage(Message message) {
        // handle message here
    }
});

connection.start();
```

Exercise

- Send and receive messages
  - Using a queue
  - Using a topic
- Build an asynchronous receiver
- Start multiple receivers
  - What happens with a queue?
  - What happens with a topic?

JMS Messages

- Body types
  - Message
  - BytesMessage
  - MapMessage
  - ObjectMessage
  - StreamMessage
  - TextMessage

JMS Messages

- Set by the client's send() method
  - JMSDeliveryMode, JMSDestination, JMSExpiration, JMSMessageID, JMSPriority, JMSTimestamp
- Optionally set by the client
  - JMSCorrelationID, JMSReplyTo, JMSType
- Optionally set by the broker
  - JMSRedelivered

JMS Messages

- Set by the application
- Allowed value types:
  - Java primitives
  - String
  - Object
- getter/setter methods for every type, e.g.
  - getBooleanProperty(name), setBooleanProperty(name, value)

JMS Messages

- Some property names are defined in the spec
  - Prefixed with JMSX
    - JMSXAppID, JMSXConsumerTXID, JMSXDeliveryCount, JMSXGroupID, JMSXGroupSeq, JMSXProducerTXID, JMSXRcvTimestamp, JMSXState and JMSXUserID
- Vendor-specific property names
  - Prefixed with JMSX_<vendor>

Message Selectors

- Allow client to filter the messages it receives
- Uses an SQL-like expression syntax

Message Selectors

- Syntax summary
  - Identifiers: header or property names
  - Literals: TRUE/FALSE and numbers (5, -10, +15.548, -20.3E5)
  - Operators
    - NOT, AND, OR
    - IN, BETWEEN, IS NULL, LIKE
    - +, -, *, /,
    - =, >, >=, <, <=, <>

Message Selectors

- Example: consumer side

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

Destination destination = session.createTopic("news.it.sofware");
MessageConsumer consumer =
       session.createConsumer(destination, "Language in ('cy', 'ga')");
```

Message Selectors

- Example: producer side

```
Message message = session.createTextMessage("Helo o newyddion TG");
message.setStringProperty("Language", "cy");
producer.send(message);

message = session.createTextMessage("Dia duit ó IT nuachta");
message.setStringProperty("Language", "ga");
producer.send(message);

message = session.createTextMessage("Hello from IT News!");
message.setStringProperty("Language", "en");
producer.send(message);
```

Durable subscriptions

- Durable subscription
  - For topics
  - Receive messages sent while consumer was unavailable
  - Subscription name
- Durable versus persistent

Durable subscriptions

- Creating a durable subscription
  - Assign a client ID to the connection
  - Use createDurableSubscription
    - Specify topic
    - Specify subscription name

Durable subscriptions

- Example

```
connection = getConnectionFactory().createConnection();
connection.setClientID("nagios");

Session session =
          connection.createSession(true, Session.AUTO_ACKNOWLEDGE);

Topic topic = session.createTopic("system.events.errors");
TopicSubscriber subscriber =
          session.createDurableSubscriber(topic, "subscription");
```

Planning

- Introducing ActiveMQ
- Building JMS applications
- Using ActiveMQ features
- ActiveMQ, Spring and JMS
- ActiveMQ configuration

Using ActiveMQ Features

- Embedding the broker
- Wildcards
- Composite destinations
- Virtual topics

Embedding the broker

- ActiveMQ can be embedded
  - Unit testing
  - Ensure application is independent of remote broker
  - Improve performance by using vm:// transport
- Can be created in plain Java code or using Spring XML

Embedding the broker

- Using the BrokerService

```
BrokerService broker = new BrokerService();
broker.setPersistent(false);
broker.start();
```

```
BrokerService broker = new BrokerService();

broker.addNetworkConnector(
                "static:(tcp://remote1:61616,tcp://remote2:61616)");
broker.setPersistenceAdapter(new KahaPersistenceAdapter());
broker.addConnector("tcp://localhost:61616");

broker.start();
```

Wildcards

- Destination hierarchies
  - Delimeter .
    news.it
    news.it.software
    news.it.software.apache
    news.sports
- Wildcard characters
  - > matches one or more trailing elements
  - * matches one element

Wildcards

- Consuming from multiple destinations
  - Use wildcards in createTopic/createQueue

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

Destination destination = session.createQueue("*.msgs.anova");
MessageConsumer consumer = session.createConsumer(destination);
```

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

Destination destination = session.createTopic("news.it.>");
MessageConsumer consumer = session.createConsumer(destination);
```

Composite destinations

- Allow sending to multiple destinations
  - , as the separator
  - when mixing queues and topics
    - prefix queue:// indicates queue
    - prefix topic:// indicates topic

Composite destinations

- Examples

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

Destination destination =
       session.createQueue("private.msgs.anova,private.msgs.abis");
MessageProducer producer = session.createProducer(destination);
```

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

Destination destination =
       session.createTopic("events.application,queue://users.logon");
MessageConsumer consumer = session.createConsumer(destination);
```

Virtual topics

- Combine semantics of topics and queue
  - Deliver message to multiple consumers
  - Use a pool of consumers to handle messages

Virtual topics

- Naming convention based
  - Topic name
    - starts with VirtualTopic.
  - Queue name
    - starts with Consumer.<name>
    - ends with topic name
- Naming convention is configurable

Virtual topics

- Example
  - Topic name is
    - VirtualTopic.News
  - Queue names are
    - Consumer.Editor.VirtualTopic.News
    - Consumer.Typeset.VirtualTopic.News
    - Consumer.Archive.VirtualTopic.News
  - Message sent to VirtualTopic.News will be received on each of the 3 queues

Exercise

- Create an embedded broker
  - in the JUnit test
  - in the RemoteClient class and connect it to your own broker
- Create a queue/topic hierarchy
  - Create a producer that sends messages to multiple destinations
  - Create a consumer that receives messages from multiple destinations

Planning

- Introducing ActiveMQ
- Building JMS applications
- Using ActiveMQ features
- ActiveMQ, Spring and JMS
- ActiveMQ configuration

ActiveMQ, Spring and JMS

- ActiveMQ namespace
- JmsTemplate
- Message-driven POJO

ActiveMQ namespace

- ActiveMQ namespace
  - Provides easy way to define ActiveMQ beans

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:amq="http://activemq.apache.org/schema/core"
       xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://activemq.apache.org/schema/core
        http://activemq.apache.org/schema/core/activemq-core.xsd">

</beans>
```

ActiveMQ namespace

- Example: Define an embedded broker and a matching ConnectionFactory

```
<amq:broker persistent="false"/>

<amq:connectionFactory id="connectionFactory"
                       brokerURL="vm://localhost"/>
```

ActiveMQ namespace

- Example: Define destinations

```
<amq:queue id="events" physicalName="example.events" />

<amq:topic id="controlbus" physicalName="example.controlbus" />

<amq:queue id="all">
  <amq:compositeDestinations>
    <ref bean="events"/>
    <ref bean="controlbus"/>
  </amq:compositeDestinations>
</amq:queue>
```

JmsTemplate

- Helper class to simplify JMS access code
  - Send and receive message
  - Supports message conversion
    - String <-> TextMessage
    - Map <-> MapMessage
    - byte[] <-> BytesMessage
    - Object <-> ObjectMessage
  - Support message selectors
  - Dynamic destination creation

JmsTemplate

- Usually created in Spring XML file

```
<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
  <property name="connectionFactory" ref="connectionFactory" />
  <property name="defaultDestination" ref="events"/>
</bean>

<bean id="topicTemplate"
                        class="org.springframework.jms.core.JmsTemplate">
  <property name="connectionFactory" ref="connectionFactory" />
  <property name="defaultDestinationName" value="example.topic"/>
  <property name="pubSubDomain" value="true"/>
</bean>
```

JmsTemplate

- Example: Sending messages

```
template.send(new MessageCreator() {
    public Message createMessage(Session session) throws JMSException {
        String message =
              String.format("Message sent at %tc", new Date());
        return session.createTextMessage(message);
    }
});

template.convertAndSend("example.second.queue", "A plain String");
```

JmsTemplate

- Example: Receiving messages

```
// receive from default destination
message = template.receive();

// receive and convert from another queue
String text =
          (String) template.receiveAndConvert("example.second.queue");

// using a message selector on the default destination
message = template.receiveSelected("Language IN ('cy', 'ga', 'gd')");
```

Message-drive POJO

- Asynchronous reception of messages
- Two implementations
  - SimpleMessageListenerContainer
  - DefaultMessageListenerContainer
- POJO implements MessageListener

Message-driven POJO

- Example: the POJO

```
public class MessageReceiver implements MessageListener {

    public void onMessage(Message message) {
        try {
            if (message instanceof TextMessage) {
                String text = ((TextMessage) message).getText();
                LOGGER.info("Received" + text)
            }
        } catch (JMSException e) {
            LOGGER.warn("Error receiving JMS message", e);
        }
    }
}
```

Message-driven POJO

- Example: DMLC Configuration

```
<bean id="receiver" class="be.anova.course.activemq.MessageReceiver" />

<bean class="o.s.jms.listener.DefaultMessageListenerContainer">
  <property name="connectionFactory" ref="connectionFactory"/>
  <property name="destinationName" value="example.first.queue"/>
  <property name="messageListener" ref="receiver" />
</bean>

<bean class="o.s.jms.listener.DefaultMessageListenerContainer">
  <property name="connectionFactory" ref="connectionFactory"/>
  <property name="destination" ref="queue"/>
  <property name="messageListener" ref="receiver" />
</bean>
```

Exercise

- Skeleton application uses
  - A control bus topic
  - An event queue
- Set up
  - Embedded ActiveMQ broker with ConnectionFactory
  - JmsTemplate to send event messages
  - DefaultMessageListenerContainer configs to receive event/controls bus messages

Planning

- Introducing ActiveMQ
- Building JMS applications
- Using ActiveMQ features
- ActiveMQ, Spring and JMS
- ActiveMQ configuration

ActiveMQ Configuration

- Configuring persistence
- Master-slave configurations
- Network of brokers
- Client failover handling

Configuring persistence

- AMQ Message Store
- KahaDB Message Store
- (Replicated) LevelDB Message Store
- JDBC Message Store
- Memory Message Store

Configuring persistence

- AMQ Message Store
  - Default message store up to 5.3
    - File-based, transactional store
    - Easy to use and set-up

```
<broker persistent="yes">
  <persistenceAdapter>
    <amqPersistenceAdapter directory="activemq-data"/>
  </persistenceAdapter>
</broker>
```

Configuring persistence

- KahaDB Message Store
  - Default message store since 5.3
    - File-based, transactional store
    - Improved recovery after kill/crash

```
<broker xmlns="http://activemq.apache.org/schema/core">
  <persistenceAdapter>
    <kahaDB directory="activemq-data"/>
  </persistenceAdapter>
</broker>
```

Configuring persistence

- LevelDB Message Store
  - Available since 5.8
    - Faster than KahaDB
    - Master/slave and replication through Zookeeper (since 5.9)

```
<broker xmlns="http://activemq.apache.org/schema/core">

  <persistenceAdapter>
    <levelDB directory="activemq-data"/>
  </persistenceAdapter>

</broker>
```

Configuring persistence

- JDBC Message Store
  - Sample configuration for Apache Derby

```
<broker xmlns="http://activemq.apache.org/schema/core">
  <persistenceAdapter>
    <jdbcPersistenceAdapter dataSource="#derby-ds"/>
  </persistenceAdapter>
</broker>

<bean id="derby-ds" class="org.apache.commons.dbcp.BasicDataSource"
                                                destroy-method="close">
  <property name="driverClassName"
            value="org.apache.derby.jdbc.ClientDriver"/>
  <property name="url"
            value="jdbc:derby://localhost/activemq;create=true"/>
  <property name="poolPreparedStatements" value="true"/>
</bean>
```

Configuring persistence

- Memory Message Store
  - No persistence in long-term storage
  - All persistent messages are only kept in memory
  - Configured by setting persistent to false on broker

```
<broker xmlns="http://activemq.apache.org/schema/core"
        persistent="false">

</broker>
```

Master-slave configuration

- Master-slave configuration
  - Pure master-slave
  - Shared database
  - Shared file-system

Master-slave configuration

- Pure master-slave
  - 2 brokers
  - Each broker has own message store
  - All state is replicated from master to slave

Master-slave configuration

- Configuration
  - Only requires configuration of the slave broker
    - Add <masterConnector/>
      - remoteURI
      - userName/password when required
    - Options to control slave/master behavior on failure

```
<broker   >
  <services>
     <masterConnector remoteURI="tcp://master.anova.be:62001" />
  </services>
</broker>
```

Master-slave configuration

- Benefits
  - Easy to configure
  - No external shared resources required
- Drawbacks
  - Only allows for one slave broker
  - Requires restart of master broker on failure

Master-slave configuation

- Shared database
  - Multiple brokers
  - Configured with same JDBC Message Store
  - Uses database lock to determine master
    - First broker gets lock and becomes master
    - Other brokers become slave

Master-slave configuration

- Benefits
  - Only JDBC Message Store config, no additional config required
  - More than two brokers possible
- Drawbacks
  - Generally slower than the other message stores
  - Requires an enterprise database system (to ensure high availability)

Master-slave configuration

- Shared file system
  - Multiple brokers
  - Configured with the default or KahaDB store on a shared file system
  - Uses file system lock to determine master
    - First broker gets file lock and becomes master
    - Other brokers become slave

Master-slave configuration

- Configuration
  - Configure KahaDB or Default ActiveMQ with a shared directory

```
<broker xmlns="http://activemq.apache.org/schema/core">
  <persistenceAdapter>
    <amqPersistenceAdapter directory="/mnt/shared/activemq-data"/>
  </persistenceAdapter>
</broker>

<broker xmlns="http://activemq.apache.org/schema/core">
  <persistenceAdapter>
    <kahaDB directory="/mnt/shared/activemq-data"/>
  </persistenceAdapter>
</broker>
```

Master-slave configuration

- Benefits
  - Only message store configuration,
    no additional cluster configuration required
  - Ability to use high-performance journal
  - More than two brokers possible
- Drawbacks
  - Requires availability of either
    - SAN
    - Distributed file system with locking semantics

Network of brokers

- Features
- Network discovery
- Static configuration
- Configuration options

Network of brokers

- Features:
  - Connect brokers together
  - Store and forward
  - Many configuration options:
    - One-way or duplex
    - Static configuration or using discovery
    - Static or dynamic selection of destinations

Network of brokers

- Network discovery
  - Search for brokers using multicast or rendezvous
  - Choose unique group name

```
<broker xmlns="http://activemq.apache.org/schema/core">
  <networkConnectors>
      <networkConnector uri="multicast://default?group=brokers"/>
  </networkConnectors>

  <transportConnectors>
      <transportConnector name="openwire" uri="tcp://0.0.0.0:61616"
                          discoveryUri="multicast://default?group=brokers"/>
  </transportConnectors>
</broker>
```

Network of brokers

- Static configuration
  - Specify list of URIs to connect to
  - URI parameters to specify reconnect behavior

```
<broker xmlns="http://activemq.apache.org/schema/core">

  <networkConnectors>
    <networkConnector
            uri="static:(tcp://broker1:61616,tcp://broker2:61616)" />
  </networkConnectors>

</broker>
```

Network of brokers

- Some other network connector options
  - Duplex
  - Exclude/include destinations

```
<networkConnectors>
  <networkConnector duplex="true" uri="...">
    <staticallyIncludedDestinations>
      <queue physicalName="management.>"/>
      <topic physicalName="global.*"/>
     </staticallyIncludedDestinations>
     <dynamicallyIncludedDestinations>
         ...
     </dynamicallyIncludedDestinations>
     <excludedDestinations>...</excludedDestinations>
  </networkConnector>
</networkConnectors>
```

Client failover handling

- Failover transport
- Sample configuration

Client failover handling

- Failover transport
  - Adds reconnect logic to any other transport
  - Allows connecting to one or multiple brokers
    - 5.4+ - broker provides cluster update info to clients
  - Configurable:
    - timeout for sending
    - reconnect attempts, delay, exponential back-off, ...
    - random connecting for load balancing

Client failover handling

- Failover transport
  - Connect to broker1 and broker2

    ```
    failover:(tcp://broker1:61616,tcp://broker2:61616)
    ```

  - Connect to broker1 by default
    when broker1 is unavailable, connect to broker2

    ```
    failover:(tcp://broker1:61616,tcp://broker2:61616)?randomize=false
    ```

  - Connect to broker1, block send for maximum 5 seconds when broker is unavailable

    ```
    failover:(tcp://broker1:61616)?timeout=5000
    ```

Exercise

- Set up a network of brokers with all participants
- Set up master/slave
  - Using pure master/slave
  - Using a shared database (Derby will be provided)

Planning

- Introducing ActiveMQ
- Building JMS applications
- Using ActiveMQ features
- ActiveMQ, Spring and JMS
- ActiveMQ configuration

Thanks

Questions?