

Architecture of Enterprise Applications 2

Messaging

2021/9/17

Haopeng Chen

REliable, INtelligent and Scalable Systems Group (REINS)

Shanghai Jiao Tong University

Shanghai, China

<http://reins.se.sjtu.edu.cn/~chenhp>

e-mail: chen-hp@sjtu.edu.cn

- Contents
 - Messaging in Java EE Applications
 - What is a Messaging?
 - What is JMS API?
 - JMS Programming Model
 - Kafka
 - Introduction
 - Quick start
- Objective
 - 能够根据系统需求，分析后端适用于异步通信机制的业务场景，并设计并实现基于消息中间件的实现方案

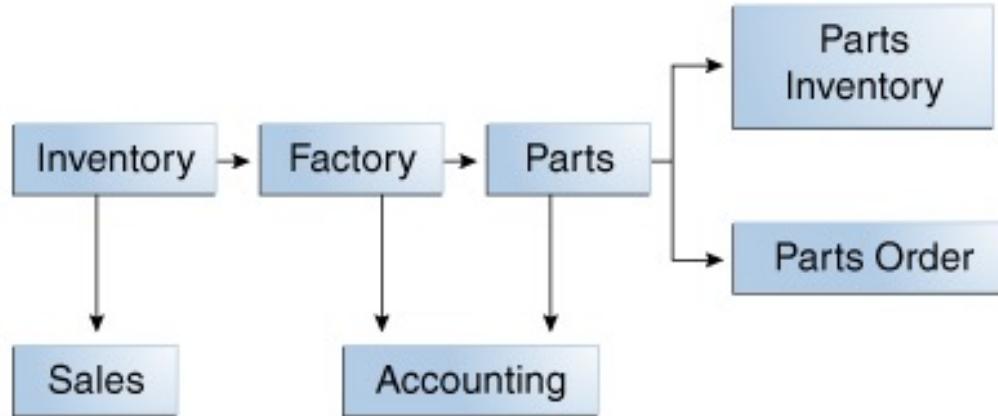
What Is Messaging?

- Messaging is a method of communication between software components or applications.
 - A messaging system is a peer-to-peer facility:
 - A messaging client can send messages to, and receive messages from, any other client.
 - Each client connects to a messaging agent that provides facilities for creating, sending, receiving, and reading messages.
- Messaging enables distributed communication that is **loosely coupled**.

What Is the JMS API?

- The Java Message Service is a Java API that allows applications to create, send, receive, and read messages.
- JMS enables communication that is not only loosely coupled but also:
 - **Asynchronous**: A receiving client does not have to receive messages at the same time the sending client sends them. The sending client can send them and go on to other tasks; the receiving client can receive them much later.
 - **Reliable**: A messaging provider that implements the JMS API can ensure that a message is delivered once and only once. Lower levels of reliability are available for applications that can afford to miss messages or to receive duplicate messages.

When Can We Use Messaging?



- The JMS API in the Java EE platform has the following features.
 - Application clients, Enterprise JavaBeans (EJB) components, and web components can send or synchronously receive a JMS message. Application clients can in addition set a message listener that allows JMS messages to be delivered to it asynchronously by being notified when a message is available.
 - Message-driven beans, which are a kind of enterprise bean, enable the asynchronous consumption of messages in the EJB container. An application server typically pools message-driven beans to implement concurrent processing of messages.
 - Message send and receive operations can participate in Java Transaction API (JTA) transactions, which allow JMS operations and database accesses to take place within a single transaction.

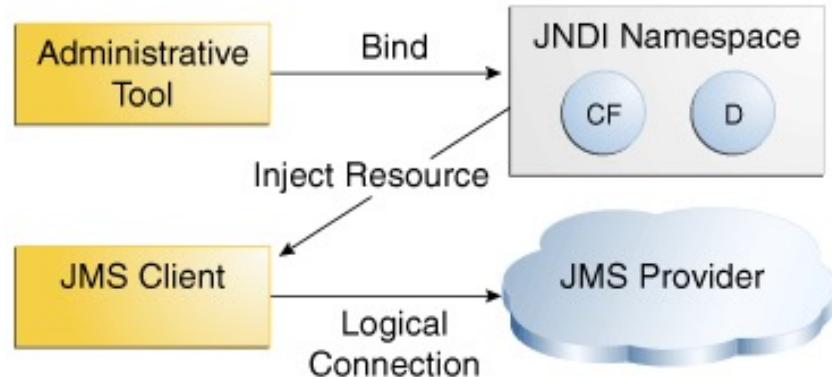
- Applications produce or consume messages
- The format of message is elastic, including three parts:
 - A header
 - Properties (optional)
 - A body (optional)

- Includes some pre-defined fields
 - JMSDestination (S)
 - JMSDeliveryMode (S)
 - JMSMessageID (S)
 - JMSTimestamp (S)
 - JMSCorrelationID (C)
 - JMSReplyTo (C)
 - JMSRedelivered (P)
 - JMSType (C)
 - JMSExpiration (S)
 - JMSPriority (S)
- Clients can not extend the fields

- Includes some pre-defined fields
 - JMSXUserID (S)
 - JMSXAppID (S)
 - JMSXDeliveryCount (S)
 - JMSXGroupID (C)
 - JMSXGroupSeq (C)
 - JMSXProducerTXID (S)
 - JMSXConsumerTXID (S)
 - JMSXRcvTimestamp (S)
 - JMSXState (P)
- Clients can extend the fields
 - Property name: follows the rule of naming selectors
 - Property value: boolean, byte, short, int, long, float, double, String

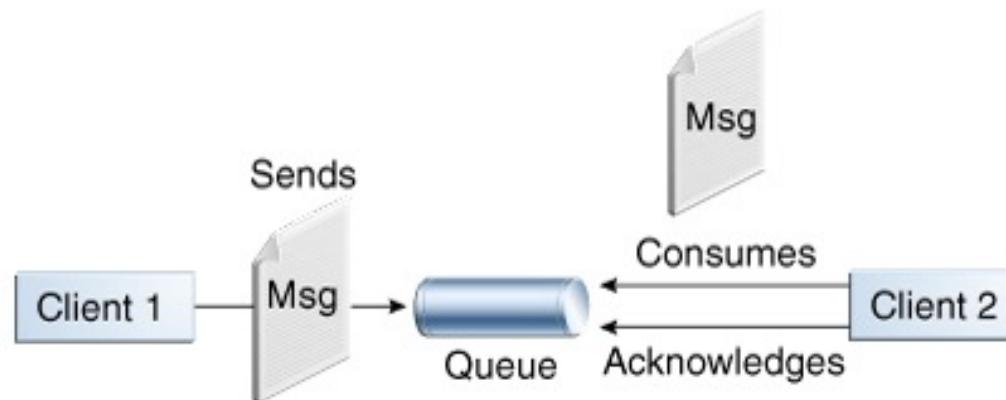
Message Type	Body Contains
TextMessage	A java.lang.String object (for example, the contents of an XML file).
MapMessage	A set of name-value pairs, with names as String objects and values as primitive types in the Java programming language. The entries can be accessed sequentially by enumerator or randomly by name. The order of the entries is undefined.
BytesMessage	A stream of uninterpreted bytes. This message type is for literally encoding a body to match an existing message format.
StreamMessage	A stream of primitive values in the Java programming language, filled and read sequentially.
ObjectMessage	A Serializable object in the Java programming language.
Message	Nothing. Composed of header fields and properties only. This message type is useful when a message body is not required.

JMS API Architecture



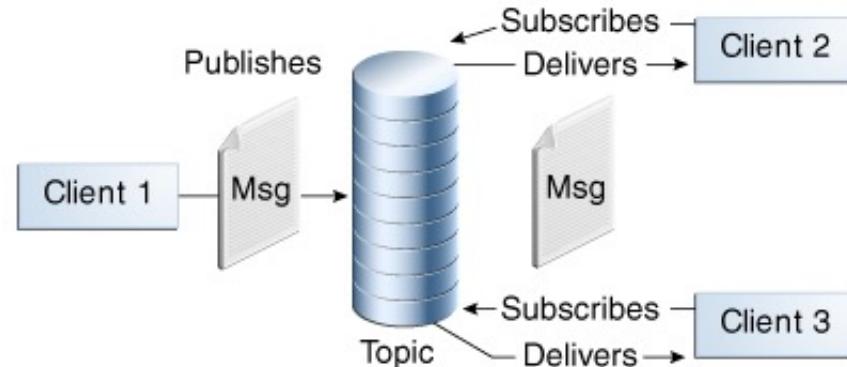
Messaging Styles

- A **point-to-point** (PTP) product or application is built on the concept of message **queues**, **senders**, and **receivers**.
 - Each message has only one consumer.
 - The receiver can fetch the message whether or not it was running when the client sent the message.

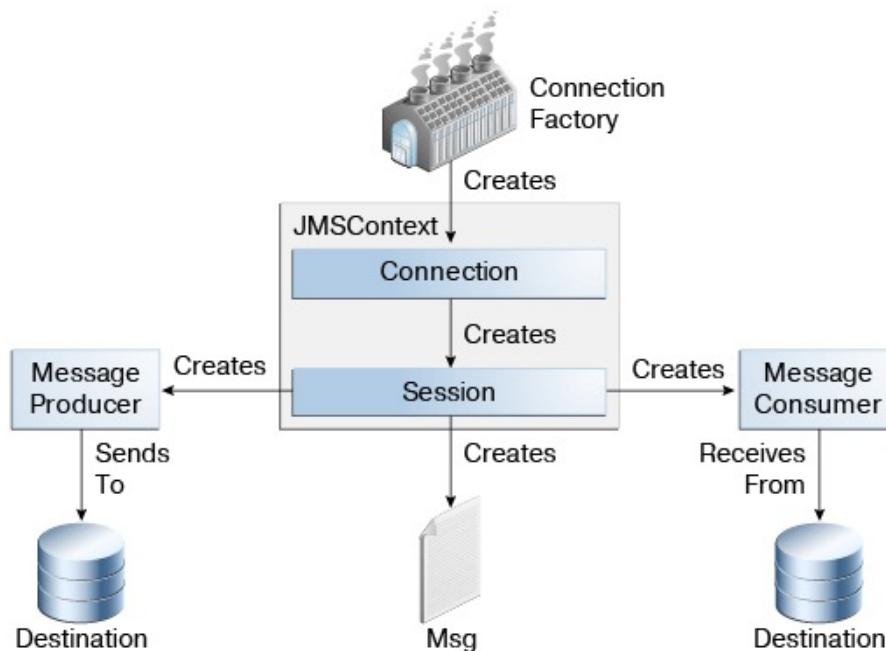


Messaging Styles

- In a **publish/subscribe** (pub/sub) product or application, clients address messages to a **topic**, which functions somewhat like a bulletin board. Publishers and subscribers can dynamically publish or subscribe to the topic.
 - Each message can have multiple consumers.
 - A client that subscribes to a topic can consume only messages sent *after* the client has created a subscription, and the consumer must continue to be active in order for it to consume messages.



The JMS API Programming Model



- Get a reference to `ConnectionFactory`
- A **connection factory** is the object a client uses to create a connection to a provider.

```
import javax.naming.*;  
import javax.jms.*;  
QueueConnectionFactory queueConnectionFactory;  
Context messaging = new InitialContext();  
queueConnectionFactory = (QueueConnectionFactory)  
    messaging.lookup("QueueConnectionFactory");
```

- or

```
@Resource(lookup = "java:comp/DefaultJMSConnectionFactory")  
private static ConnectionFactory connectionFactory;
```

- A **destination** is the object a client uses to specify the target of messages it produces and the source of messages it consumes.
 - In the PTP messaging style, destinations are called queues.
 - In the pub/sub messaging style, destinations are called topics.

```
Queue queue;  
queue = (Queue)messaging.lookup("theQueue");  
Topic topic;  
topic = (Topic)messaging.lookup("theTopic");
```

- Or

```
@Resource(lookup = "jms/MyQueue")  
private static Queue queue;  
@Resource(lookup = "jms/MyTopic")  
private static Topic topic;
```

- Get a Connection and a Session
- A **connection** encapsulates a virtual connection with a JMS provider.
- A **session** is a single-threaded context for producing and consuming messages.
 - You normally create a session (as well as a connection) by creating a JMSContext object

```
JMSContext context = connectionFactory.createContext();
```

- Create a Producer or a Consumer

```
try (JMSContext context = connectionFactory.createContext()) {  
    JMSProducer producer = context.createProducer();  
    ...  
    context.createProducer().send(dest, message);
```

- Or

```
try (JMSContext context = connectionFactory.createContext()) {  
    JMSConsumer consumer = context.createConsumer(dest);  
    ...  
    Message m = consumer.receive();  
    Message m = consumer.receive(1000);
```

- Create a message

```
TextMessage message = context.createTextMessage();
message.setText(msg_text); // msg_text is a String
context.createProducer().send(message);
```

```
Message m = consumer.receive();
if (m instanceof TextMessage) {
    String message = m.getBody(String.class);
    System.out.println("Reading message: " + message);
} else {
    // Handle error or process another message type
}
```

- JMS Message Listener
- A message listener is an object that acts as an asynchronous event handler for messages.

```
Listener myListener = new Listener();  
consumer.setMessageListener(myListener);
```

Listener:

```
void onMessage(Message inMessage)
```

- Selection of messages
- Producer:

```
String data;  
TextMessage message;  
message = session.createTextMessage();  
message.setText(data);  
message.setStringProperty("Selector", "Technology");
```

- Selection of messages
- Consumer:

```
String selector;
```

```
selector = new String(" (Selector = 'Technology') ");
```

```
JMSConsumer consumer = context.createConsumer(dest, selector);
```

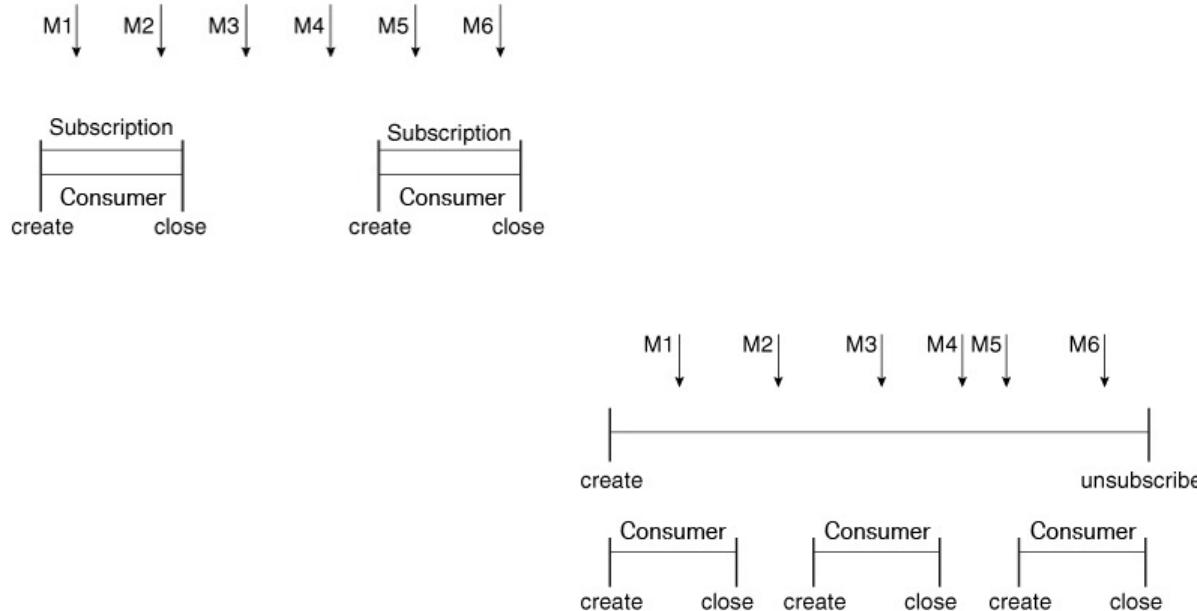
- Durable subscription

```
String subName = "MySub";  
JMSConsumer consumer = context.createDurableConsumer(myTopic, subName);
```

```
consumer.close();  
context.unsubscribe(subName);
```

```
JMSConsumer consumer =  
    context.createSharedDurableConsumer(topic, "MakeItLast");
```

JMS programming model



- JMS Browser
 - allows you to browse the messages in the queue and display the header values for each message.

```
QueueBrowser browser = context.createBrowser(queue);
```

- JMS Exception Handling
 - The root class for all checked exceptions in the JMS API is **JMSEException**

- IntelliJ IDEA 创建消息驱动Bean - 接收JMS消息
 - <https://www.cnblogs.com/yangyquin/p/5346104.html>
- Wildfly Messaging Configuration
 - <https://docs.jboss.org/author/display/WFLY/Messaging+configuration>



Welcome to WildFly

Your WildFly instance is running.

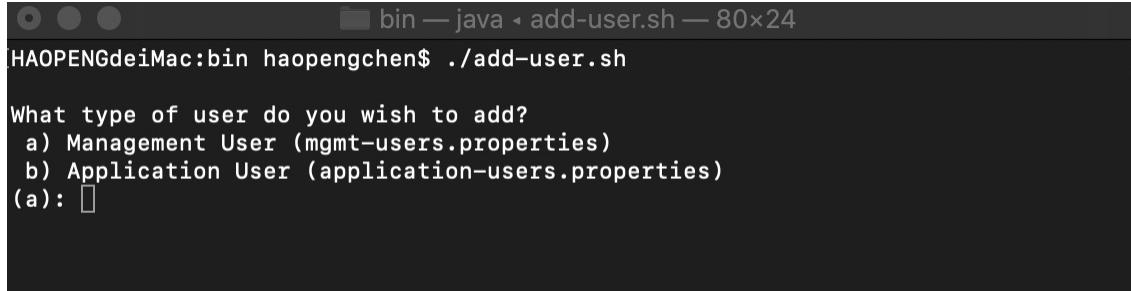
[Documentation](#) | [Quickstarts](#) | [Administration Console](#)

[WildFly Project](#) | [User Forum](#) | [Report an issue](#)

[JBoss Community](#)

- Add User

- `$WILDFLY_HOME/bin/add-user.sh`



```
bin — java — add-user.sh — 80x24
HAOPENGdeiMac:bin haopengchen$ ./add-user.sh

What type of user do you wish to add?
a) Management User (mgmt-users.properties)
b) Application User (application-users.properties)
(a):
```

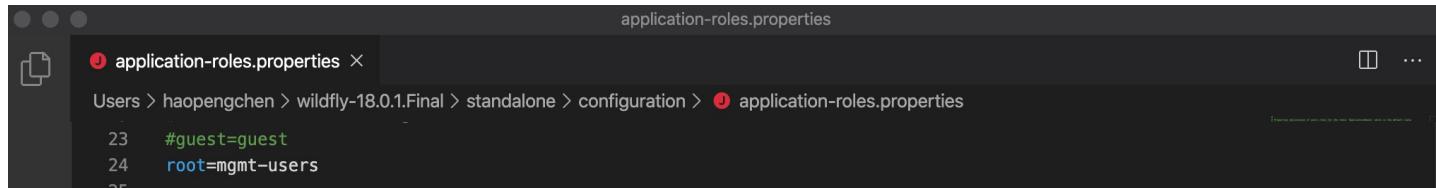
- Add a **mgmt-user**
 - For example: username: `root`, password: `wildfly2020!`
 - In `$WILDFLY_HOME/standalone/configuration/mgmt-users.properties`



```
mgmt-users.properties

27 #admin=2a0923285184943425d1f53ddd58ec7a
28 root=88447ca5938dca6ed9a3156ed77e6579
```

- Assign a role to the new user
 - In `/standalone/configuration/applications-roles.properties`



The screenshot shows a terminal window with the title "application-roles.properties". The path is "Users > haopengchen > wildfly-18.0.1.Final > standalone > configuration > application-roles.properties". The content of the file is as follows:

```
23 #guest=guest
24 root=mgmt-users
25
```

- Add JMS Server & Destination in `standalone.xml`

```
<extension module="org.wildfly.extension.messaging-activemq"/>
<!-- messaging server --&gt;
&lt;subsystem xmlns="urn:jboss:domain:messaging-activemq:8.0"&gt;
    &lt;server name="default"&gt;
        &lt;statistics enabled="${wildfly.messaging-activemq.statistics-enabled:${wildfly.statisti
        &lt;security-setting name="#"&gt;
            &lt;role name="mgmt-users" send="true" consume="true" create-non-durable-queue="true
        &lt;/security-setting&gt;
    &lt;!-- Sample Queue &amp; Topic --&gt;
    &lt;jms-queue name="HelloWorldMDBQueue" entries="java:jboss/exported/jms/queue/HelloWorldMDBQueue"/&gt;
    &lt;jms-topic name="HelloWorldMDBTopic" entries="java:jboss/exported/jms/topic/HelloWorldMDBTopic"/&gt;</pre>
```

A JMS Sample

- Run Wildfly Server, and check the JMS conf on console

The screenshot shows the HAL Management Console interface at `localhost:9990/console/index.html#jndi`. The navigation bar includes links for Back, Server, Standalone Server, Monitor, and JNDI. The main area is divided into two sections: 'JNDI Tree' on the left and 'Details' on the right.

JNDI Tree: This section displays a hierarchical tree of JNDI entries under 'Java Contexts'. A red box highlights the 'jms' entry under 'java:jboss/exported'. The expanded 'jms' node contains three entries: 'RemoteConnectionFactory', 'topic' (which further expands to 'HelloWorldMDBTopic'), and 'queue' (which further expands to 'HelloWorldMDBQueue').

Level 1	Level 2	Level 3	
Java Contexts	java:		
Java Contexts	java:jboss		
Java Contexts	java:jboss/exported	RemoteConnectionFactory	
Java Contexts	java:jboss/exported	topic	HelloWorldMDBTopic
Java Contexts	java:jboss/exported	queue	HelloWorldMDBQueue

Details: This section is currently empty, displaying the message 'No details available'.

A JMS Sample - Queue

JMSProducer.java

```
public class JMSProducer {  
    private static final Logger log = Logger.getLogger(JMSProducer.class.getName());  
    private static final String DEFAULT_MESSAGE = "这是JMS信息.....";  
    private static final String DEFAULT_CONNECTION_FACTORY = "jms/RemoteConnectionFactory";  
    private static final String DEFAULT_DESTINATION = "/jms/queue/HelloWorldMDBQueue";  
    private static final String DEFAULT_MESSAGE_COUNT = "10";  
    private static final String DEFAULT_USERNAME = "root";  
    private static final String DEFAULT_PASSWORD = "reins2011!";  
    private static final String INITIAL_CONTEXT_FACTORY = "org.wildfly.naming.client.WildFlyInitialContextFactory";  
    private static final String PROVIDER_URL = "http-remoting://localhost:8080";  
    public static void main(String[] args) throws Exception {  
        Context context=null;  
        Connection connection=null;  
        try {  
            final Properties env = new Properties();  
            env.put(Context.INITIAL_CONTEXT_FACTORY, INITIAL_CONTEXT_FACTORY);  
            // 该KEY的值为初始化Context的工厂类,JNDI驱动的类名  
            env.put(Context.PROVIDER_URL, PROVIDER_URL);  
            // 该KEY的值为Context服务提供者的URL.命名服务提供者的URL  
            env.put(Context.SECURITY_PRINCIPAL, DEFAULT_USERNAME);  
            env.put(Context.SECURITY_CREDENTIALS, DEFAULT_PASSWORD);//应用用户的登录名,密码.  
            // 获取到InitialContext对象.  
            context = new InitialContext(env);  
            Destination destination = (Destination) context.lookup(DEFAULT_DESTINATION);  
        } catch (Exception e) {  
            log.error("Error occurred while creating JMS producer.", e);  
        }  
    }  
}
```

A JMS Sample - Queue

JMSProducer.java

```
// 创建JMS连接、会话、生产者和消费者
connection = connectionFactory.createConnection(DEFAULT_USERNAME, DEFAULT_PASSWORD);
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
MessageProducer producer = session.createProducer(destination);
connection.start();
int count = Integer.parseInt(DEFAULT_MESSAGE_COUNT);
// 发送特定数目的消息
TextMessage message = null;
for (int i = 0; i < count; i++) {
    message = session.createTextMessage(DEFAULT_MESSAGE);
    producer.send(message);
}
// 等待30秒退出
CountDownLatch latch = new CountDownLatch(1);
latch.await(30, TimeUnit.SECONDS);
} catch (Exception e) { log.severe(e.getMessage()); throw e;
} finally {
    if (context != null) { context.close(); }
    if (connection != null) { connection.close(); }
}
```

A JMS Sample - Queue

JMSConsumer.java

```
public class JMSConsumer {  
    private static final Logger log = Logger.getLogger(JMSConsumer.class.getName());  
    private static final String DEFAULT_CONNECTION_FACTORY = "jms/RemoteConnectionFactory";  
    private static final String DEFAULT_DESTINATION = "/jms/queue/HelloWorldMDBQueue";  
    private static final String DEFAULT_USERNAME = "root";  
    private static final String DEFAULT_PASSWORD = "reins2011!";  
    private static final String INITIAL_CONTEXT_FACTORY = "org.wildfly.naming.client.WildFlyInitialContextFactory";  
    private static final String PROVIDER_URL = "http-remoting://localhost:8080";  
    private static final int WAIT_COUNT = 5;  
  
    public static void main(String[] args) throws Exception {  
        ConnectionFactory connectionFactory = null;  
        Connection connection = null;  
        Session session = null;  
        MessageConsumer consumer = null;  
        Destination destination = null;  
        TextMessage message = null;  
        Context context = null;  
        try {  
            final Properties env = new Properties();  
            env.put(Context.INITIAL_CONTEXT_FACTORY, INITIAL_CONTEXT_FACTORY);  
            env.put(Context.PROVIDER_URL, PROVIDER_URL);  
            env.put(Context.SECURITY_PRINCIPAL, DEFAULT_USERNAME);  
            env.put(Context.SECURITY_CREDENTIALS, DEFAULT_PASSWORD);  
            context = new InitialContext(env);  
            connectionFactory = (ConnectionFactory) context.lookup(DEFAULT_CONNECTION_FACTORY);  
            destination = (Destination) context.lookup(DEFAULT_DESTINATION);  
        } catch (Exception e) {  
            log.error("Error occurred while creating JMSConsumer object", e);  
        }  
    }  
}
```

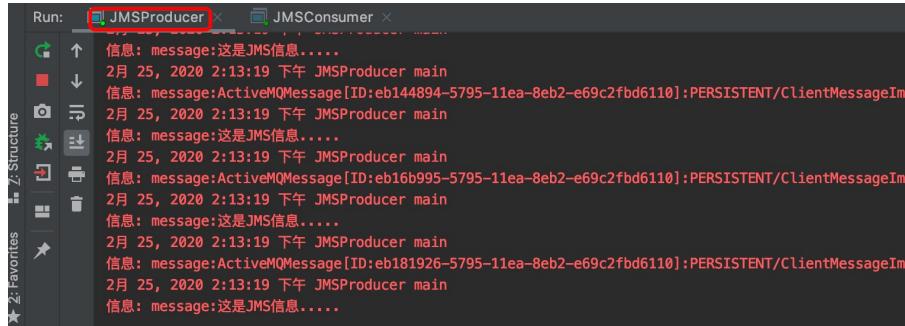
A JMS Sample - Queue

JMSConsumer.java

```
connection = connectionFactory.createConnection(DEFAULT_USERNAME, DEFAULT_PASSWORD);
session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
consumer = session.createConsumer(destination);
connection.start();
// 等待30秒退出
CountDownLatch latch = new CountDownLatch(1);
log.info("开始从JBoss端接收信息-----");
int i = 0;
for (; i < WAIT_COUNT; i++) {
    if (message != null) {
        log.info("接收到的消息的内容:" + message.getText());
        i = 0;
    }
    log.info("开始从JBoss端接收信息-----");
    message = (TextMessage) consumer.receive(5000);
    latch.await(1, TimeUnit.SECONDS);
}
} catch (Exception e) { log.severe(e.getMessage()); throw e;
} finally {
    if (context != null) { context.close(); }
    if (connection != null) { connection.close(); }
}
```

A JMS Sample - Queue

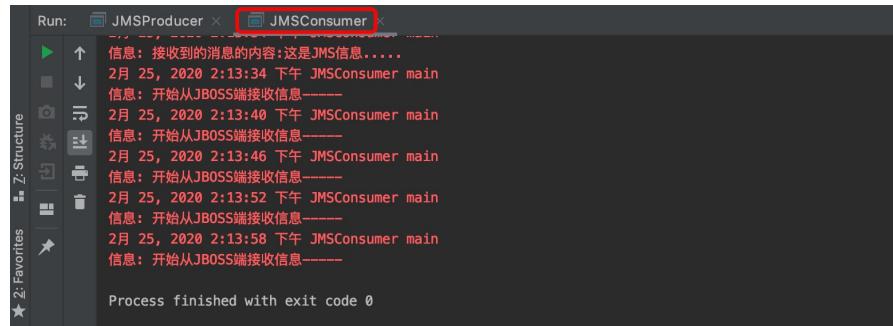
- Run **JMSProducer**



The screenshot shows a terminal window titled "Run: JMSProducer". The log output is as follows:

```
信息: message:这是JMS信息.....  
2月 25, 2020 2:13:19 下午 JMSProducer main  
信息: message:ActiveMQMessage[ID:eb144894-5795-11ea-8eb2-e69c2fb6110]:PERSISTENT/ClientMessageIm  
2月 25, 2020 2:13:19 下午 JMSProducer main  
信息: message:这是JMS信息.....  
2月 25, 2020 2:13:19 下午 JMSProducer main  
信息: message:ActiveMQMessage[ID:eb16b995-5795-11ea-8eb2-e69c2fb6110]:PERSISTENT/ClientMessageIm  
2月 25, 2020 2:13:19 下午 JMSProducer main  
信息: message:这是JMS信息.....  
2月 25, 2020 2:13:19 下午 JMSProducer main  
信息: message:ActiveMQMessage[ID:eb181926-5795-11ea-8eb2-e69c2fb6110]:PERSISTENT/ClientMessageIm  
2月 25, 2020 2:13:19 下午 JMSProducer main  
信息: message:这是JMS信息.....
```

- Run **JMSConsumer**



The screenshot shows a terminal window titled "Run: JMSProducer" with the consumer tab selected. The log output is as follows:

```
信息: 接收到的消息的内容:这是JMS信息.....  
2月 25, 2020 2:13:34 下午 JMSConsumer main  
信息: 开始从JBoss端接收信息-----  
2月 25, 2020 2:13:40 下午 JMSConsumer main  
信息: 开始从JBoss端接收信息-----  
2月 25, 2020 2:13:46 下午 JMSConsumer main  
信息: 开始从JBoss端接收信息-----  
2月 25, 2020 2:13:52 下午 JMSConsumer main  
信息: 开始从JBoss端接收信息-----  
2月 25, 2020 2:13:58 下午 JMSConsumer main  
信息: 开始从JBoss端接收信息-----  
  
Process finished with exit code 0
```

A JMS Sample - Topic

JMSPub.java

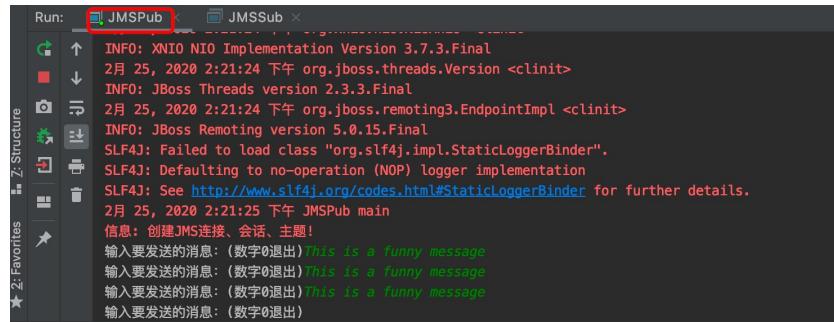
```
System.out.print("输入要发送的消息: (数字0退出)");
line = msgStream.readLine();
if (line != null && line.trim().length() != 0) {
    TextMessage textMessage = session.createTextMessage();
    textMessage.setText(line);
    textMessage.setStringProperty("Selector", "Funny");
    producer.send(textMessage);
    quitNow = line.equalsIgnoreCase("0");
}
```

JMSSub.java

```
String selector;
selector = new String("(Selector = 'Funny')");
consumer = session.createConsumer(topic, selector);
consumer.setMessageListener(new javax.jms.MessageListener() {
    public void onMessage(Message message) {
        try {
            TextMessage tm = (TextMessage) message;
            System.out.println("接收到的消息内容: " + tm.getText().toString());
            System.out.println("JMS目的地: " + tm.getJMSPublication());
            System.out.println("JMS回复: " + tm.getJMSReplyTo());
            System.out.println("JMS消息ID号: " + tm.getJMSMessageID());
            System.out.println("是否重新接收: " + tm.getJMSRedelivered());
        } catch (JMSException e1) {
            e1.printStackTrace();
        }
    }
});
```

A JMS Sample - Topic

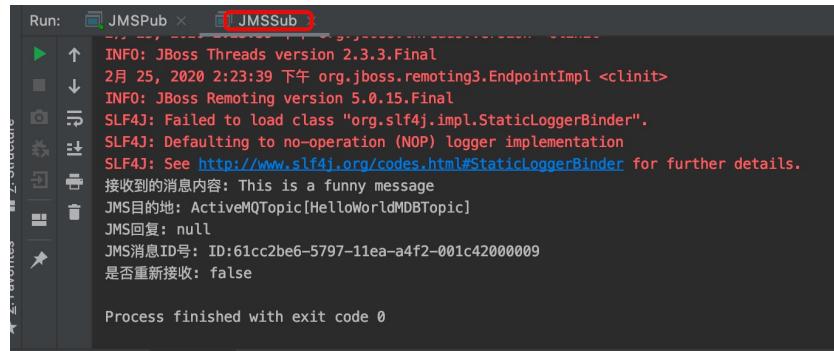
- Run JMSPub



The terminal window shows the output of the JMSPub application. It includes system logs and the application's main loop where it prompts for message input.

```
INFO: XNIO NIO Implementation Version 3.7.3.Final
2月 25, 2020 2:21:24 下午 org.jboss.threads.Version <clinit>
INFO: JBoss Threads version 2.3.3.Final
2月 25, 2020 2:21:24 下午 org.jboss.remoting3.EndpointImpl <clinit>
INFO: JBoss Remoting version 5.0.15.Final
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
2月 25, 2020 2:21:25 下午 JMSPub main
信息： 创建JMS连接、会话、主题！
输入要发送的消息：(数字0退出)This is a funny message
输入要发送的消息：(数字0退出)This is a funny message
输入要发送的消息：(数字0退出)This is a funny message
输入要发送的消息：(数字0退出)
```

- Run JMSSub



The terminal window shows the output of the JMSSub application. It includes system logs and the application's main loop where it prints received messages.

```
INFO: JBoss Threads version 2.3.3.Final
2月 25, 2020 2:23:39 下午 org.jboss.remoting3.EndpointImpl <clinit>
INFO: JBoss Remoting version 5.0.15.Final
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
接收到的消息内容: This is a funny message
JMS目的地: ActiveMQTopic[HelloWorldMDBTopic]
JMS回复: null
JMS消息ID号: ID:61cc2be6-5797-11ea-a4f2-001c42000009
是否重新接收: false

Process finished with exit code 0
```

Messaging with JMS

- Spring provides the means to publish messages to any POJO (Plain Old Java Object).

Email.java

```
public class Email {  
  
    private String to;  
    private String body;  
  
    public Email() {  
    }  
  
    public Email(String to, String body) {  
        this.to = to;  
        this.body = body;  
    }  
  
    public String getTo() {  
        return to;  
    }  
  
    public void setTo(String to) {  
        this.to = to;  
    }  
}
```

```
    public String getBody() {  
        return body;  
    }  
  
    public void setBody(String body) {  
        this.body = body;  
    }  
  
    @Override  
    public String toString() {  
        return String.format("Email{to=%s, body=%s}", getTo(), getBody());  
    }  
}
```

- This POJO is quite simple, containing two fields (**to** and **body**), along with the presumed set of getters and setters.

Messaging with JMS

- Message Receiver

Receiver.java

```
package org.reins.demo;
```

```
import org.springframework.jms.annotation.JmsListener;
import org.springframework.stereotype.Component;
```

```
@Component
```

```
public class Receiver {
```

```
    @JmsListener(destination = "mailbox")//, containerFactory = "myFactory")
```

```
    public void receiveMessage(Email email) {
```

```
        System.out.println("Received <" + email + ">");
```

```
}
```

```
}
```

Messaging with JMS

- Send and receive JMS messages with Spring

Application.java

```
package org.reins.demo;
@SpringBootApplication
@EnableJms
public class Application {
    @Bean
    public JmsListenerContainerFactory<?> myFactory(@Qualifier("jmsConnectionFactory") ConnectionFactory
                                                    connectionFactory, DefaultJmsListenerContainerFactoryConfigurer configurer) {
        DefaultJmsListenerContainerFactory factory = new DefaultJmsListenerContainerFactory();
        // This provides all boot's default to this factory, including the message converter
        configurer.configure(factory, connectionFactory);
        // You could still override some of Boot's default if necessary.
        return factory;
    }
}
```

@Bean // Serialize message content to json using TextMessage

```
public MessageConverter jacksonJmsMessageConverter() {
    MappingJackson2MessageConverter converter = new MappingJackson2MessageConverter();
    converter.setTargetType(MessageType.TEXT);
    converter.setTypeIdPropertyName("_type");
    return converter;
}
```

Messaging with JMS

- Send and receive JMS messages with Spring

Application.java

```
public static void main(String[] args) {  
    // Launch the application  
    ConfigurableApplicationContext context = SpringApplication.run(Application.class, args);  
  
    JmsTemplate jmsTemplate = context.getBean(JmsTemplate.class);  
  
    // Send a message with a POJO - the template reuse the message converter  
    System.out.println("Sending an email message.");  
    jmsTemplate.convertAndSend("mailbox", new Email("info@example.com", "Hello"));  
}  
  
}
```

Messaging with JMS

- Send and receive JMS messages with Spring

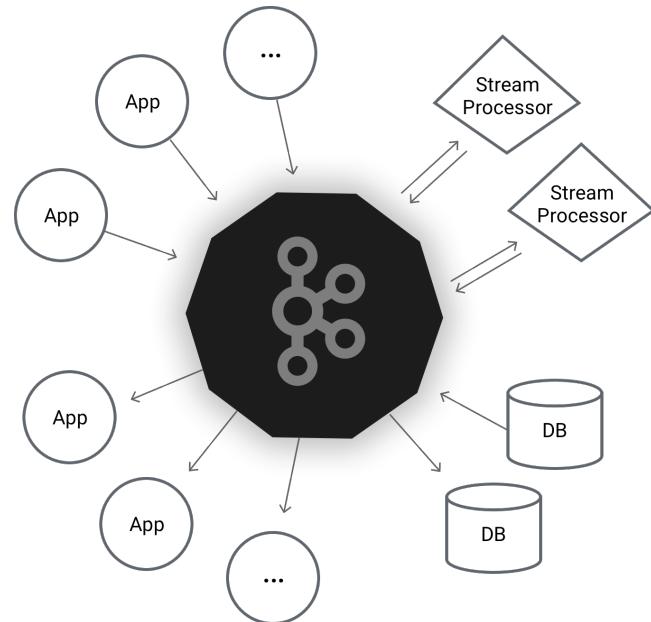
MsgController.java

```
@RestController
public class MsgController {
    @Autowired
    WebApplicationContext applicationContext;

    @GetMapping(value = "/msg")
    public void findOne() {
        JmsTemplate jmsTemplate = applicationContext.getBean(JmsTemplate.class);

        // Send a message with a POJO - the template reuse the message converter
        System.out.println("Sending an email message.");
        jmsTemplate.convertAndSend("mailbox", new Email("info@example.com", "Hello Msg"));
    }
}
```

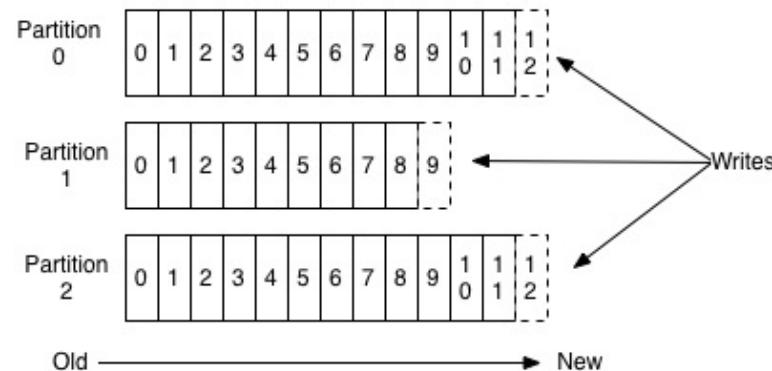
- <http://kafka.apache.org>
- **PUBLISH & SUBSCRIBE**
 - Read and write streams of data like a messaging system.
- **PROCESS**
 - Write scalable stream processing applications that react to events in real-time.
- **STORE**
 - Store streams of data safely in a distributed, replicated, fault-tolerant cluster.



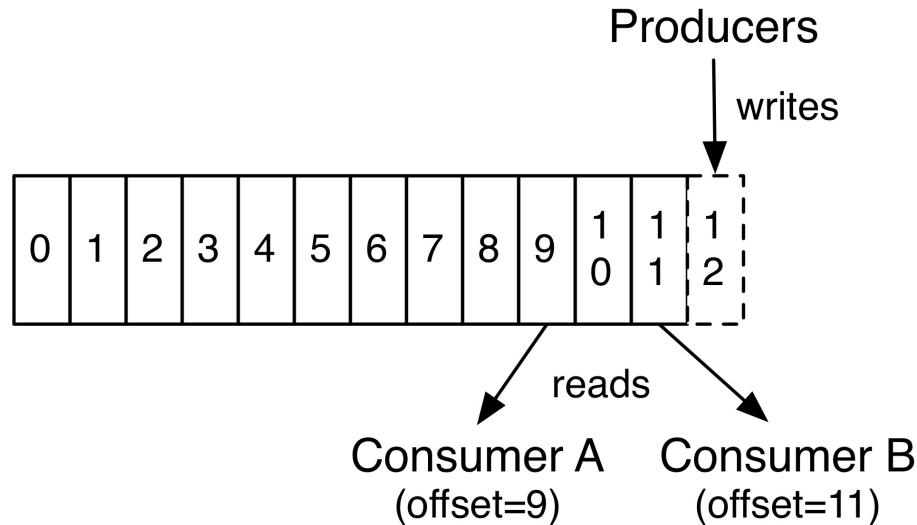
- **Apache Kafka® is a distributed streaming platform.**
- Kafka has four core APIs:
 - The [Producer API](#) allows an application to **publish** a stream of records to one or more Kafka topics.
 - The [Consumer API](#) allows an application to **subscribe** to one or more topics and **process** the stream of records produced to them.
 - The [Streams API](#) allows an application to act as a **stream processor**, **consuming** an input stream from one or more **topics** and **producing** an output stream to one or more output **topics**, effectively transforming the input streams to output streams.
 - The [Connector API](#) allows building and running reusable producers or consumers that **connect Kafka topics to existing applications or data systems**. For example, a connector to a relational database might capture every change to a table.

- Topics and log
 - A topic is a category or feed name to which records are published. Topics in Kafka are always multi-subscriber; that is, a topic can have zero, one, or many consumers that subscribe to the data written to it.
 - For each topic, the Kafka cluster maintains a **partitioned log** that looks like this:

Anatomy of a Topic



- The Kafka cluster durably persists all published records—**whether or not they have been consumed**—using a configurable retention period.



- Quickstart
 - <https://kafka.apache.org/quickstart>

STEP 1: GET KAFKA

Download the latest Kafka release and extract it:

```
1 | $ tar -xzf kafka_2.13-2.8.0.tgz
2 | $ cd kafka_2.13-2.8.0
```

STEP 2: START THE KAFKA ENVIRONMENT

NOTE: Your local environment must have Java 8+ installed.

Run the following commands in order to start all services in the correct order:

```
1 # Start the ZooKeeper service
2 # Note: Soon, ZooKeeper will no longer be required by Apache Kafka.
3 $ bin/zookeeper-server-start.sh config/zookeeper.properties
```

Open another terminal session and run:

```
1 # Start the Kafka broker service
2 $ bin/kafka-server-start.sh config/server.properties
```

Once all services have successfully launched, you will have a basic Kafka environment running and ready to use.

STEP 3: CREATE A TOPIC TO STORE YOUR EVENTS

Kafka is a distributed *event streaming platform* that lets you read, write, store, and process [events](#) (also called *records* or *messages* in the documentation) across many machines.

Example events are payment transactions, geolocation updates from mobile phones, shipping orders, sensor measurements from IoT devices or medical equipment, and much more. These events are organized and stored in [topics](#). Very simplified, a topic is similar to a folder in a filesystem, and the events are the files in that folder.

So before you can write your first events, you must create a topic. Open another terminal session and run:

```
$ bin/kafka-topics.sh --create --topic quickstart-events --bootstrap-server localhost:9092
```

All of Kafka's command line tools have additional options: run the `kafka-topics.sh` command without any arguments to display usage information. For example, it can also show you [details such as the partition count](#) of the new topic:

```
1 | $ bin/kafka-topics.sh --describe --topic quickstart-events --bootstrap-server localhost:9092
2 | Topic:quickstart-events PartitionCount:1 ReplicationFactor:1 Configs:
3 |   Topic: quickstart-events Partition: 0 Leader: 0 Replicas: 0 Isr: 0
```

STEP 4: WRITE SOME EVENTS INTO THE TOPIC

A Kafka client communicates with the Kafka brokers via the network for writing (or reading) events. Once received, the brokers will store the events in a durable and fault-tolerant manner for as long as you need—even forever.

Run the console producer client to write a few events into your topic. By default, each line you enter will result in a separate event being written to the topic.

```
1 | $ bin/kafka-console-producer.sh --topic quickstart-events --bootstrap-server localhost:9092
2 | This is my first event
3 | This is my second event
```

You can stop the producer client with `Ctrl-C` at any time.

STEP 5: READ THE EVENTS

Open another terminal session and run the console consumer client to read the events you just created:

```
1 | $ bin/kafka-console-consumer.sh --topic quickstart-events --from-beginning
2 | This is my first event
3 | This is my second event
--bootstrap-server localhost:9092
```

You can stop the consumer client with `Ctrl-C` at any time.

Feel free to experiment: for example, switch back to your producer terminal (previous step) to write additional events, and see how the events immediately show up in your consumer terminal.

Because events are durably stored in Kafka, they can be read as many times and by as many consumers as you want. You can easily verify this by opening yet another terminal session and re-running the previous command again.

- Access Kafka with Java

- To use Producer API and Consumer API, you can use the following maven dependency:

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>2.8.0</version>
</dependency>
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>2.8.0</version>
</dependency>
```

KafkaPro.java

```
public class KafkaPro {  
    public static void main(String[] args) throws Exception {  
        Properties props = new Properties();  
        props.put("bootstrap.servers", "localhost:9092");  
        props.setProperty("transactional.id", "my-transactional-id");  
  
        Producer<String, String> producer = null;  
  
        try {  
            producer = new KafkaProducer<String, String>(props, new StringSerializer(), new StringSerializer());  
            producer.initTransactions();  
  
            producer.beginTransaction();  
            for (int i = 0; i < 100; i++)  
                producer.send(new ProducerRecord<>("test", Integer.toString(i), "Message " + Integer.toString(i)));  
            producer.commitTransaction();  
        } catch (ProducerFencedException e) {  
            producer.close();  
        } catch (OutOfOrderSequenceException e) {  
            producer.close();  
        } catch (AuthorizationException e) {  
            producer.close();  
        } catch (KafkaException e) {  
            producer.abortTransaction();  
        }  
        producer.close();  
    }  
}
```

KafkaCon.java

```
public class KafkaCon {  
    public static void main(String[] args) throws Exception {  
        Properties props = new Properties();  
        props.setProperty("bootstrap.servers", "localhost:9092");  
        props.setProperty("group.id", "test");  
        props.setProperty("enable.auto.commit", "true");  
        props.setProperty("auto.commit.interval.ms", "1000");  
        props.setProperty("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");  
        props.setProperty("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");  
  
        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);  
        consumer.subscribe(Arrays.asList("test"));  
  
        while (true) {  
            ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));  
            for (ConsumerRecord<String, String> record : records)  
                System.out.printf("offset = %d, key = %s, value = %s%n", record.offset(), record.key(), record.value());  
        }  
    }  
}
```

Apache Kafka

The screenshot shows an IDE interface with two main panes. The left pane displays a project structure for 'SE343_5_KafkaSample' containing Java files like 'KafkaCon.java' and 'KafkaPro.java', and configuration files like 'pom.xml'. The right pane shows a terminal window running 'kafka_2.12-2.4.0' with log output. Below the terminal is a 'Run' tab with two entries: 'KafkaCon' and 'KafkaPro'. The 'KafkaCon' entry is highlighted with a red box and has a dropdown menu open over it, showing a list of offsets for a specific consumer group. The offsets listed are: offset = 797, key = 87, value = Message 87; offset = 798, key = 88, value = Message 88; offset = 799, key = 89, value = Message 89; offset = 800, key = 90, value = Message 90; offset = 801, key = 91, value = Message 91; offset = 802, key = 92, value = Message 92; offset = 803, key = 93, value = Message 93; offset = 804, key = 94, value = Message 94; offset = 805, key = 95, value = Message 95; offset = 806, key = 96, value = Message 96; offset = 807, key = 97, value = Message 97; offset = 808, key = 98, value = Message 98; offset = 809, key = 99, value = Message 99.

```
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerRecord;
```

```
[2020-02-25 18:37:35,643] INFO maxSessionTimeout set to 60000 (org.apache.zookeeper.server.ZooKeeperServer)
[2020-02-25 18:37:35,644] INFO Created server with tickTime 3000 minSessionTimeout 60000 maxSessionTimeout 60000 datadir /tmp/zookeeper/version-2 snapdir /tmp/zookeeper/version-2 (org.apache.zookeeper.server.ZooKeeperServer)
[2020-02-25 18:37:35,554] INFO Using org.apache.zookeeper.server.NIOServerCnxnFactory as server connection factory (org.apache.zookeeper.server.ServerCnxnFactory)
[2020-02-25 18:37:35,557] INFO Configuring NIO connection handler with 10s sessionless connection timeout, 2 selector threads(s), 16 worker threads, and 64 KB direct buffers (org.apache.zookeeper.server.NIOServerCnxnFactory)
[2020-02-25 18:37:35,562] INFO binding to port 0.0.0.0/0.0.0.0:2181 (org.apache.zookeeper.server.NIOServerCnxnFactory)
[2020-02-25 18:37:35,577] INFO zookeeper.snapshotsizeFactor = 0.33 (org.apache.zookeeper.server.ZKDatabase)
[2020-02-25 18:37:35,581] INFO Reading snapshot /tmp/zookeeper/version-2/snapshot.14d (org.apache.zookeeper.server.persistence.FileSnapshot)
[2020-02-25 18:37:35,617] INFO Snapshottting: 0x28d to /tmp/zookeeper/version-2/snapshot.28d (org.apache.zookeeper.server.persistence.FileTxnSnapLog)
[2020-02-25 18:37:35,637] INFO Using checkIntervalMs=60000 maxPerMinute=10000 (org.apache.zookeeper.server.ContainerManager)
[2020-02-25 18:37:39,389] INFO Creating new log file: log.28e (org.apache.zookeeper.server.persistence.FileTxnLog)
```

```
for (int i = 0; i < 100; i++)
    producer.send(new ProducerRecord<( topic: "test", Integer)
```

```
[2020-02-25 18:37:52,188] INFO [GroupDataManager brokerId=0] Finished loading offsets and group metadata from __consumer_offsets-48 in 21 milliseconds. (kafka.coordinator.group.GroupDataManager)
[2020-02-25 18:38:07,128] INFO [TransactionCoordinator id=0] Initialized transactionId my-transactional-id with producerId 10000 and producer epoch 5 on partition __transaction_state-46 (kafka.coordinator.transaction.TransactionCoordinator)
[2020-02-25 18:38:09,701] INFO [TransactionCoordinator id=0] Initialized transactionId my-transactional-id with producerId 10000 and producer epoch 6 on partition __transaction_state-46 (kafka.coordinator.transaction.TransactionCoordinator)
[2020-02-25 18:38:20,790] INFO [GroupCoordinator 0]: Preparing to rebalance group test in state PreparingRebalance with old generation 2 (__consumer_offsets-48) (reason: Adding new member consumer-test-1-a867b457-id2c-4339-88e5-7013323f4998 with group instanceNone) (kafka.coordinator.group.GroupCoordinator)
[2020-02-25 18:38:20,794] INFO [GroupCoordinator 0]: Stabilized group test generation 3 (__consumer_offsets-48) (kafka.coordinator.group.GroupCoordinator)
[2020-02-25 18:38:20,800] INFO [GroupCoordinator 0]: Assignment received from leader for group test for generation 3 (kafka.coordinator.group.GroupCoordinator)
[2020-02-25 18:38:42,452] INFO [TransactionCoordinator id=0] Initialized transactionId my-transactional-id with producerId 10000 and producer epoch 7 on partition __transaction_state-46 (kafka.coordinator.transaction.TransactionCoordinator)
```

作业一

- 请你在大二开发的E-Book系统的基础上，完成下列任务：
 1. 优化服务层设计，确定有状态与无状态服务的运用方式，要求至少分别选择一个服务设计成有状态的和无状态的，并观察它们的差异，确认你的设计合理。
 2. 编写一个JMS程序，用来接收并异步地处理订单，具体功能与现有E-Book中的下订单相同，具体流程为：
 - 1) 用户通过Web前端下订单到服务器，你的服务器端程序 A(Controller) 应该先反馈给用户订单已接收；
 - 2) A 将订单数据发送到“Order”队列；
 - 3) 服务器端另一个程序 B(Service) 一直监听Order队列，一旦读到消息，就立刻进行处理，将数据写入数据库。
 - 你应该将上述功能集成到你的E-Book系统中，如果你无法将上述功能集成到你的E-Book系统中，可以单独建立工程实现，但是会适当扣分。
 - 请将你编写的相关代码整体压缩后上传，请勿压缩整个工程提交。
- 评分标准：
 1. 选择的有状态服务和无状态服务合理，代码编写正确（2分）
 2. JMS功能实现正确无误（3分）

- Messaging with JMS
 - <https://spring.io/guides/gs/messaging-jms/>
- Apache Kafka Quick Start
 - <https://kafka.apache.org/quickstart>
- Java Message Service Concepts
 - <https://javaee.github.io/tutorial/jms-concepts.html>



Thank You!