**Pasi de urmat**

* Tot trebuie dependenta:

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-activemq</artifactId>  
</dependency>

* Atentie! In Docker instalam anume ActiveMQ-Artemis!

docker run -d --name activemq-artemis -p 61616:61616 -p 8161:8161 -e ARTEMIS\_USER=admin -e ARTEMIS\_PASSWORD=admin makyo/activemq-artemis

Cream Bean pentru ConnectionFactory:@Bean  
public ConnectionFactory connectionFactory() throws JMSException {  
 ActiveMQConnectionFactory connectionFactory = new ActiveMQConnectionFactory("tcp://localhost:61616");  
 connectionFactory.setUser("admin");  
 connectionFactory.setPassword("admin");  
  
  
  
 return connectionFactory;  
}  
  
@Bean  
public CommandLineRunner commandLineRunner(JmsTemplate jmsTemplate){  
 return (cl) -> {  
 jmsTemplate.convertAndSend("MyQueue.Q","Hello");  
 };  
}

si avem un commandLineRunner.

convertAndSend() - converteste mesajul in text si il trimite la queue. Daca asa queue nu exista, se creaza

**Consumarea mesajelor**

@Component  
public class JmsConsumer {  
 @JmsListener(destination = "MyQueue.Q")  
 public void consume(String payload){  
 System.*out*.println("Message received: "+payload);  
 }  
}

* Acest consumer va prelua mesajul de la aceasta Queue de indata ce va exista unul.
* Am putea pune ca tip de return un String, ca sa oferim un mesaj de raspuns la cel interceptat, dar vom primi exceptie, caci el ActiveMQ nu stie in ce queue sa pune mesajul returnat de metoda

Acum, daca din consola la ActiveMQ vom trimite un mesaj din MyQueue.Q, el deodata va fi interceptat de @JmsListener

**Trimiterea unui mesaj de raspuns**

@JmsListener(destination = "MyQueue.Q")  
@SendTo("ResponseQueue.Q")  
public String consume(String payload){  
System.*out*.println("Message received: "+payload);  
return "Hello from application";  
}

@SendTo("Queue") - mesajul din return e trimis la Queue

Putem crea si propriile haders:

@JmsListener(destination = "MyQueue.Q")  
@SendTo("ResponseQueue.Q")  
public Message<String> consume(String payload){  
System.*out*.println("Message received: "+payload);  
return MessageBuilder.*withPayload*("A response message").setHeader("Test\_Header","A message").build();  
}

withPayload() - setam textul

setHeader("header","text")

Putem pune orice obiect in <>

**Mesaj de raspuns fara anotatie**

@JmsListener(destination = "MyQueue.Q")  
public JmsResponse<String> consume(String payload){  
System.*out*.println("Message received: "+payload);  
return JmsResponse.*forQueue*("Message from response queue","ResponseQueue.Q");  
}

Asta e util caci numele la queue ar putea gen veni din baza de date si nu il putem pune in anotatie

**Atentie! Cand un queue nu e gasit, mereu e creat!!!!**

**De ce SpringEvent nu e asa bun**

* Problema e ca cu String Event, daca e trimis un eveniment si aplicatia da crash pana a receptiona si face ceva cu eventul, eventul e pierdut
* Spring Event e un In Memory event
* Cu JMS, eventurile raman in queue si de indata ce aplicatia porneste, ea va primi si prelucra evenimentele din queue

**Mesaje tranzactionale**

@Bean  
public PlatformTransactionManager transactionManager() throws JMSException {  
 return new JmsTransactionManager(connectionFactory());  
}

Si apoi pur si simplu punem @Transactional unde ne trebuie

Desi @Transactional e pentru JPA, noi am creat un bean cu numele transactionManager de alt tip, si el va face tranzactii la nivel de event

* Acum, cand vom trimite mai multe mesage dintr-o metoda @Transctional, ele fie vor fi trimise toate deodata, fie nici unul.

@Component  
public class MyService {  
 @Autowired  
 @Lazy  
 private JmsTemplate jmsTemplate;  
  
 @Transactional  
 public void addMessageInQueue() throws InterruptedException {  
 jmsTemplate.convertAndSend("MyQueue.Q","Message1");  
 Thread.*sleep*(5000);  
 jmsTemplate.convertAndSend("MyQueue.Q","Message2");  
 }  
}

Ambele vor fi trimise odata peste 5 secunde!