01

ObjectMessage e DLQ

Transcrição

Começando deste ponto? Você pode fazer o DOWNLOAD (https://s3.amazonaws.com/caelum-online-public/jms/stages/jms-stage-cap7.zip) completo do projeto do capítulo anterior e continuar seus estudos a partir deste capítulo. Neste capítulo vamos importar algumas classes que você pode baixar aqui (https://s3.amazonaws.com/caelum-online-public/jms/jms-modelo.zip).

Introdução

No último capítulo, falamos sobre Selectors, que são os critérios de consumo das nossas mensagens. O consumidor só receberá a mensagem se a mesma atender aos critérios, às condições definidas.

Nesse capítulo, o nosso foco é a mensagem em si, como ter mais controle sobre ela, além conseguirmos manipulá-la melhor.

Na classe `TesteProdutorTopico` , criamos uma mensagem, que nada mais é do que um XML:

```
// código anterior comentado

Message message = session.createTextMessage("<pedido><id>333</id></pedido>");

// código posterior comentado
```

É muito comum utilizar o XML na integração, pois é um formato conhecido em todas as plataformas, tem validadores, várias ferramentas que trabalham com esse formato. Mas esse XML, normalmente, é criado a partir de um objeto, e não como foi feito, colocado como String. Há apis que ajudam a converter um objeto para XML.

Vamos modificar a forma como esse XML é criado.

XML a partir de uma string? Porque não de um objeto?

Então o ideal seria a nossa mensagem ser criada a partir de um objeto, e não de uma string:

```
// código anterior comentado

String xml = ???;

Message message = session.createTextMessage(xml);

// código posterior comentado
```

Mas o que deveria ficar no lugar das interrogações?

Primeiramente, baixe [aqui][2] e extraia o zip disponibilizado. Nele, há algumas classes, vamos colocá-las no projeto. Essas classes nada mais são do que um modelo, o que antes foi visto como XML, através de uma string, agora são classes.

Como agora temos uma classe `Pedido`, podemos usá-la para termos um objeto a ser manipulado. Vamos criar um pedido através do método `geraPedidoComValores()`, da classe `PedidoFactory`:

```
// código anterior comentado

public Pedido geraPedidoComValores() {

    Pedido pedido = new Pedido(2459, geraData("22/12/2016"),
        geraData("23/12/2016"), new BigDecimal("145.98"));

    Item motoG = geraItem(23,"Moto G");
    Item motoX = geraItem(51,"Moto X");

    pedido.adicionaItem(motoX);
    pedido.adicionaItem(motoG);

    return pedido;

}

// código posterior comentado
```

Esse método gera um pedido com os itens Moto G e Moto X, além do seu código, datas de compra e entrega, e o valor.

Com esse método, criamos um pedido:

```
// código anterior comentado

Pedido pedido = new PedidoFactory().geraPedidoComValores();

String xml = ???;

Message message = session.createTextMessage(xml);

// código posterior comentado
```

Construindo o XML

Agora que temos o objeto `pedido` em mãos, precisamos gerar um XML baseado nele. Para isso, usaremos o **JAXB** (**J**ava **A**pi for **X**ML **B**inding), que é uma biblioteca que nos ajuda nesse *binding* do objeto com o XML.

A partir dessa classe, podemos gerar um XML através do método `marshal`, pois temos um objeto e queremos um XML, caso fosse o contrário, de um XML para um objeto, usaríamos o método `unmarshal`.

Esse método `marshal` recebe um `jaxbObject` e um `writter`, o `jaxbObject` será o nosso pedido, e o `writer`, um `StringWriter`, pois quero ter acesso a ele em qualquer parte do código. E com o `writer` em mãos, conseguimos extrair o XML a partir do método `toString()`, com isso finalmente iremos substituir as interrogações do XML:

```
// código anterior comentado

Pedido pedido = new PedidoFactory().geraPedidoComValores();
```

```
StringWriter writer = new StringWriter();

JAXB.marshal(pedido, writer);

String xml = writer.toString();

Message message = session.createTextMessage(xml);

// código posterior comentado
```

Essas linhas já criam o XML a partir do objeto `pedido` ! Mas não é apenas isso, o JAXB precisa de uma configuração inicial, precisamos dizer a ele qual é o elemento raíz do XML, que no nosso caso é o pedido, então lá na classe `Pedido` essa configuração é feita:

```
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Pedido implements Serializable {
    // ...
}
```

A anotação `@XmlRootElement` diz para o JAXB que a classe `Pedido` é o elemento raíz do XML, e a anotação `@XmlAccessorType(XmlAccessType.FIELD)` diz para o JAXB construir o XML a partir dos **atributos** da classe `Pedido` , pois o seu padrão é construir o XML a partir dos **`getters`** e **`setters`** . Como a classe `Pedido` não tem `getters` e `setters` para todos os atributos, essa segunda anotação foi necessária.

Além disso, dizemos que o atributo `itens` seja representado no XML pelo elemento de mesmo nome, através da anotação `@XmlElementWrapper(name="itens")` e que cada elemento dessa coleção se torne um item no XML, através da anotação `@XmlElement(name="item")` :

```
// ...

@XmlElementWrapper(name="itens")
@XmlElement(name="item")
private Set<Item> itens = new LinkedHashSet<>();

//...
```

Vamos voltar à classe `TesteProdutorTopico` e testar o código que nós fizemos, podemos até colocar um `System.out.println(xml)` para verificar o XML criado. Basta executá-la e vemos que realmente o XML é gerado:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<pedido>
    <codigo>2459</codigo>
    <dataPedido>2016-12-22T00:00:00-02:00</dataPedido>
    <dataPagamento>2016-12-23T00:00:00-02:00</dataPagamento>
    <valorTotal>145.98</valorTotal>
    <itens>
        <item>
            <id>51</id>
            <nome>Moto X</nome>
```

```
        </item>
        <item>
            <id>23</id>
            <nome>Moto G</nome>
        </item>
    </itens>
</pedido>
```

Podemos testar também o recebimento do XML, executando a classe `TesteConsumidorTopicoComercial` e vemos que o pedido é recebido!

Até aqui não tivemos nenhuma novidade em relação ao JMS, apenas tornamos o XML um pouco mais flexível, mas ele continua sendo uma string, um `TextMessage` .

Enviando diretamente um objeto, ao invés de XML

Como já foi falado, o XML é muito comum na integração, mas no nosso caso temos um produtor em Java e um consumidor em Java. Então será que é realmente necessário gerar esse XML? Precisamos mesmo que o produtor serialize um objeto para XML, e envie a mensagem para o consumidor deserializar esse XML e criar novamente o objeto?

Podemos enviar diretamente o objeto! Sem precisar do serializar e deserializar um XML. Então vamos alterar novamente a classe `TesteProdutorTopico` , primeiro comentando as linhas que geram o XML:

```
// código anterior comentado

Pedido pedido = new PedidoFactory().geraPedidoComValores();

// StringWriter writer = new StringWriter();
// JAXB.marshal(pedido, writer);
// String xml = writer.toString();

Message message = session.createTextMessage(xml); // Não temos mais o xml, e agora?

// código posterior comentado
```

Como não temos mais o XML, não temos mais um `TextMessage` . Enviaremos no lugar do XML, um objeto, então temos um `ObjectMessage` . E que objeto que iremos enviar? O nosso pedido!

```
// código anterior comentado

Pedido pedido = new PedidoFactory().geraPedidoComValores();

Message message = session.createObjectMessage(pedido);

// código posterior comentado
```

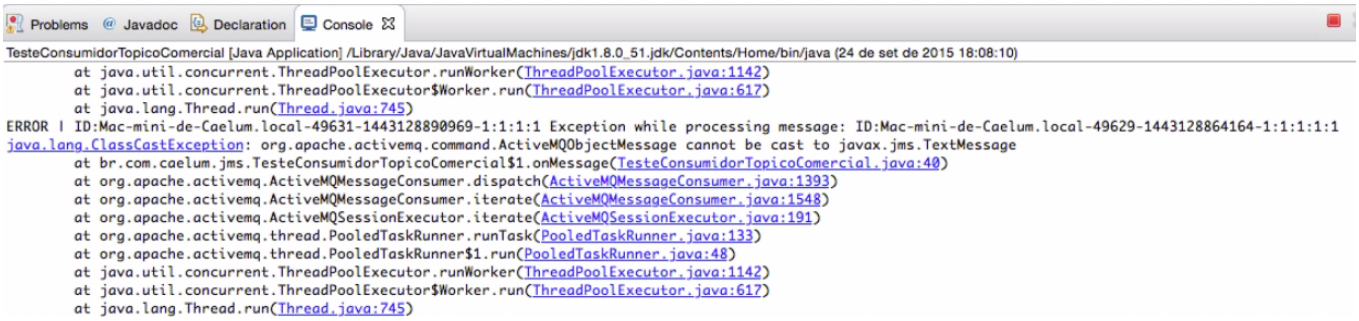
Repare que o método `createObjectMessage` necessita de um objeto serializável, então esse código só funciona porque o pedido é um objeto serializável, dizemos isso ao Java ao implementar a classe `java.io.Serializable` na classe `Pedido` .

O Java vai serializar esse objeto, acessar seus dados, seu código, data de pagamento, etc, e os transformará em um fluxo de bits e bytes. Ou seja, a classe em si não será enviada pelo ActiveMQ, apenas os dados do objeto. Por isso a classe `Pedido`

precisa implementar essa interface `Serializable`, e não só a classe `Pedido`, a serialização exige que todos o seus filhos também implementem essa interface, como a classe `Item`.

Executando o produtor, vemos que não deu erro, o que significa (na teoria) que a mensagem foi enviada ao ActiveMQ. Se a mensagem foi enviada, vamos executar o consumidor e fazê-lo receber essa mensagem.

Executamos e...:



Deu erro! Não só um, mas seis erros!

Quando o recebimento da mensagem falha

Visualizando as exceções, vemos que todas são iguais, são todas `ClassCastException`. Olhando a mensagem com um pouco mais de atenção, percebe-se que em `TesteConsumidorTopicoComercial`, que é o consumidor, tentamos fazer um *cast* para `TextMessage`, mas nossa mensagem não é mais um `TextMessage`, e sim um `ObjectMessage`! Descobrimos a causa do erro! O consumidor recebeu a mensagem, e a passou para o `MessageListener`:

```
// código anterior omitido

consumer.setMessageListener(new MessageListener() {

@Override
public void onMessage(Message message) {

    TextMessage textMessage = (TextMessage)message;//vilão

    try {
        System.out.println(textMessage.getText());
    } catch (JMSException e) {
        e.printStackTrace();
    }
}

});

// código posterior omitido
```

Era esperado um `TextMessage`, que não foi recebido, por isso a mensagem não foi confirmada. Então a mensagem foi enviada novamente ao ActiveMQ, que a guardou, mas antes de guardá-la ele tentou entregar essa mesma mensagem seis vezes, por isso recebemos seis exceções.

Ajustes de configuração

Atenção: A partir da versão 5.12.2 do ActiveMQ é preciso configurar explicitamente quais pacotes podem ser deserializados. Sem ter essa configuração você receberá um exceção na hora de consumir uma `ObjectMessage`. A exceção indica um problema de segurança:

```
Caused by: java.lang.ClassNotFoundException: Forbidden class br.com.caelum.modelo.Pedido! This class
```

Isto acontece pois o ActiveMQ se tornou mais rígido e exige que você configura explicitamente quais pacote são permitidas na deserialização::

```
public class TesteConsumidorTopicoComercial {

  public static void main(String[] args) {
    System.setProperty("org.apache.activemq.SERIALIZABLE_PACKAGES","java.lang,br.com.caelum.modelo");
    //resto do codigo omitido
```

Isso permite deserializar qualquer objeto de uma classe dos pacotes `java.lang` ou `br.com.caelum.modelo`. Caso queira permitir todos os pacotes, coloque:

```
System.setProperty("org.apache.activemq.SERIALIZABLE_PACKAGES","*");
```

Mais infos no site do ActiveMQ: LINK (http://activemq.apache.org/objectmessage.html)

Recebendo ObjectMessage

Então vamos resolver esse problema, não estamos mais recebendo um `TextMessage` e sim um `ObjectMessage`, e esse `ObjectMessage` nada mais é que um pedido. Para testar vamos imprimir seu código:

```
// código anterior omitido

consumer.setMessageListener(new MessageListener() {

@Override
public void onMessage(Message message) {

    ObjectMessage objectMessage = (ObjectMessage)message;

    try {
        Pedido pedido = (Pedido) objectMessage.getObject();
        System.out.println(pedido.getCodigo());
    } catch (JMSException e) {
        e.printStackTrace();
    }
}

});

// código posterior omitido
```

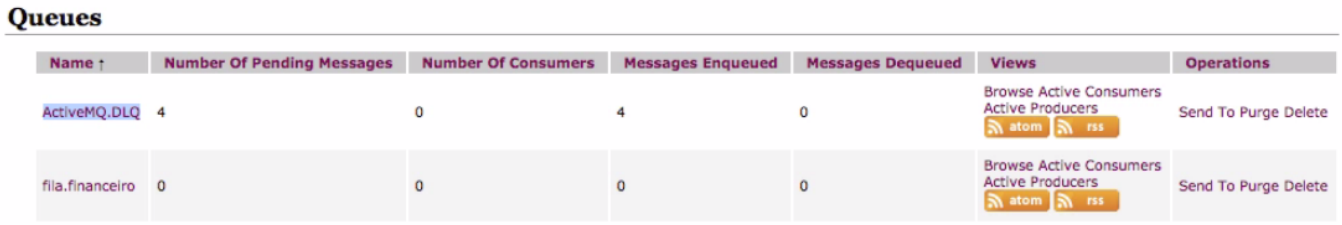
Tentativas de entrega da mensagem

Vamos executar a classe e não deu erro! Porém, ele não imprimiu o código do pedido, ou seja, ele não recebeu a mensagem. O que aconteceu é que o ActiveMQ tentou entregar a mensagem mais seis vezes, mas quantas vezes ele vai tentar entregar a mensagem que, aparentemente, não pode ser processada? Cada MOM, e o ActiveMQ não é diferente, define um limite de reentrega, e o padrão do ActiveMQ são seis vezes. Mas depois dessas seis tentativas, o que ele faz com a mensagem?

Conhecendo a DLQ

Por padrão, o ActiveMQ pega essa mensagem, a qual chamamos de mensagem venenosa, tira essa mensagem do tópico, ou da fila, e coloca essa mensagem em uma fila especial de mensagens que não podem ser entregues, de mensagens venenosas.

Podemos ver isso no console de administração:



A fila `ActiveMQ.DLQ` (**D**ead **L**etter **Q**ueue) foi criada, é essa fila que recebe as mensagens que não puderam ser entregues. Repare que temos justamente uma mensagem que não pôde ser entregue, que é o nosso pedido, o `ObjectMessage` .

Então, para vermos a mensagem ser entregue, vamos enviá-la novamente! Basta executar novamente o produtor (classe `TesteProdutorTopico`) e vemos que o código do pedido é impresso! Logo, a mensagem foi recebida pelo consumidor.