11

Usando acknowledge

Transcrição

No último video, vimos que há mensagens que podem não ser processadas quando há alguma falha. Neste caso, o MOM fica com a mensagem e não sabe o que fazer com ela. Para resolver esse problema foi criado o conceito de `Dead Letter Queue`, uma fila para as mensagens que não puderam ser entregues.

Aprendemos também que acessamos a DLQ alterando nosso arquivo de propriedades do ActiveMQ fazendo com que ele crie uma fila automaticamente com essas mensagens, inclusive, por ser uma fila, nada nos impede de consumi-la.

Nesse video vamos ver detalhes sobre a entrega da mensagem. Para motivar melhor, pense que a entrega pode falhar por vários motivos inclusive por uma condição na lógica de negocio. Pode ser que não queremos confirmar o recebimento da mensagem pois alguma lógica de negócio não permite. Ou seja, não acontece uma exceção como `ClassCastExceptoin` e sim apenas um problema na lógica. Nesse caso pode ser útil ter controle programático sobre o recebimento da mensagem.

Para ver esses detalhes, vamos abrir nossa classe `TesteConsumidorFila`:

```
// TesteConsumidorFila.java

// código anterior omitido
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
// código posterior omitodo
```

O primeiro ponto é deixar de usar o `AUTO_ACKNOWLEDGE`, porque seremos nós os responsáveis pela confirmação. Para isso, usamos `CLIENT_ACKNOWLEDGE`:

```
// TesteConsumidorFila.java

// código anterior omitido
Session session = connection.createSession(false, Session.CLIENT_ACKNOWLEDGE);
// código posterior omitido
```

Usando acknowledge

Agora, em algum momento, precisa indicar que recebemos a mensagem. No nosso `MessageListener` precisamos confirmar programaticamente o recebimento da mensagem:

```
// TesteConsumidorFila.java

// código anterior omitido
consumer.setMessageListener(new MessageListener() {
    // código posterior omitido
    try {
        message.acknowledge(); // fazendo programaticamente
        // código posterior omitido
    } catch(JMSException e) {
```

```
            e.printStackTrace();
        }
    });
```

Com o ActiveMQ rodando, vamos alterar `TesteProdutorFila` alterando o ID da mensagem:

```
// TesteProdutorFila.java
// código anterior omitido
Message message = session.createTextMessage("<pedido><id>12</id></pedido>");
// código posterior omitido
```

Vamos rodará agora `TesteProdutorFila` , enviando um pedido. Em seguida, vamos rodar `TesteConsumidorFila` .

Veremos que a mensagem foi consumida, inclusive se rodarmos novamente nosso `TesteConsumidorFila` veremos que nada será consumido, pois a mensagem já foi consumida antes.

Agora, vamos voltar para nosso `TesteProdutorFila` e mudar o ID do nosso pedido para 13, número mágico do azar para alguns. Vamos pensar que essa mensagem não foi processada ou por algum erro ou porque algum lógica não foi atendida. Podemos comentar a linha que realiza o método `acknowledge()` .

Vamos executar o `TesteProdutorFila` e logo em seguida o `TesteConsumidorFila` . Se rodarmos 100 vezes, não importa, como não fizemos `acknowledge()` ao consumir a mensagem, ela continuará na fila. Inclusive podemos ver através do console de administrar do ActiveMQ que a mensagem ainda continua lá. A fila `financeira` terá ainda a mensagem que ainda não foi entregue. O MOM não conseguirá apagar essa mensagem enquanto não houver um `acknowledge()` realizado por um consumidor.

Trabalhando com Session commit e rollback

Uma coisa que pode chamar atenção é que só existe um método de confirmação, não há um desconfirmar. Podemos pensar no contexto de uma transação que o `acknowledge()` é o `commit()` , mas qual seria o equivalente ao conceito de `rollback()` ? Não há um `unacknowledge()` . No entanto a classe `Session` pode ajudar mas devemos primeiro dizer que queremos um comportamento transacional, justamente para termos os métodos já conhecidos de uma transação.

```
// TesteConsumidorFila.java

// código anterior omitido
// mudando de false para true e usando SESSION_TRANSACTED
Session session = connection.createSession(true, Session.SESSION_TRANSACTED);
// código posterior omitodo
```

Não basta só colocar só true ou `SESSION_TRANSACTED` , tem que ser os dois, por mais redundante que isso possa parecer.

Agora, o grande lance é usamos o objeto `Session` que possui os métodos `commit()` e `rollback()` . Vamos começar fazendo um `commit()` :

```
// TesteConsumidorFila.java

// código anterior omitido
consumer.setMessageListener(new MessageListener() {
```

```
        // código posterior omitido
        try {
            session.commit(); // novidade

        } catch(JMSException e) {
            e.printStackTrace();
        }
    });
```

Agora, só rodar o `TesteConsumidorFila`. Veremos que ele consumiu a mensagem. Se rodarmos novamente, não haverá mais nenhuma mensagem.

É muito mais comum usar o `Sesson.SESSION_TRANSACTED` porque na maior parte das vezes queremos fazer `commit()` ou `rollback()`. Ainda há mais uma vantagem da sessão transacional que é não apenas confirmar o recebimento, mas participar de uma transação maior (transação XA). Por exemplo, quando queremos acessar um banco de dados junto ao JMS. É claro, essa transação teria que ser global, envolvendo outras.

E que tal agora testarmos um rollback? Só alterar o `session.commit()` para o `session.rollback()` em `TesteConsumidorFila`. Vamos continuar enviando o pedido de número 13.

No console, vemos que ele tenta reentregar a mensagem seis vezes, isto é, ele faz aquele `redelivery`. Pelo administrador web do ActiveMQ conseguimos ver a mensagem lá no DLQ.

Para saber mais: Session.DUPS_OK_ACKNOWLEDGE

Há ainda uma outra configuração, a `Session.DUPS_OK_ACKNOWLEDGE` que indica para nosso MOM lidar com mensagem duplicadas. Porém, os mais importantes no dia a dia são os `Session.SESSION_TRANSACTED` e o `Session.CLIENT_ACKNOWLEDGE`.

Agora é ir para os exercícios e praticar.