# O conceito do JMS

Para se entender o Java JMS há três conceitos básicos que precisam ser explicados, são eles:

* O que é *Messaging*?
* O que é a API JMS?
* Conceitos Básicos do JMS

## O que é *Messaging*?

*Messaging* (do inglês *message*, ‘mensagem’) é um serviço usado para comunicar diferentes programas ou subprogramas. É um sistema baseado em *peer-to-peer*, onde cada cliente pode enviar mensagens ou receber mensagens de qualquer outro cliente e há um provedor de mensagens que possibilita a criação, envio, recepção e leitura de cada mensagem.

A comunicação no *messaging* é construída baseada no conceito de *loosely coupled* que prega que tanto o destinatário quanto o remetente precisam saber apenas o formato da mensagem e o destino de envio dela e mais nenhuma outra informação, como a hora em que a mensagem foi enviada ou se ambas as partes estão online, permitindo bastante flexibilidade.

É importante notar que o *messaging* embora possa parecer, é bastante diferente de tecnologias como RMI (*tightly coupled*), a qual agrega muitas informações além do destino e do formato ou e-mail *(eletronic mail*), que é uma comunicação entre pessoas e pessoas ou pessoas e máquinas, enquanto o *messaging* comunica apenas programas ou subprogramas.

## O que é a API JMS

O JMS, ou *Java Message Service* (Serviço de Mensagens do Java, numa tradução livre) é uma API do Java produzida pela Sun e suas associadas, que define um conjunto de interfaces e semânticas que possibilita que aplicativos Java se comuniquem como outros componentes de software através da criação, envio, recepção e leitura de mensagens.

Além de ser *loosely coupled* por se valer do messaging o JMS também é marcado por:

* Assíncrono: o provedor entrega as mensagens ao cliente assim que elas chegam sem a necessidade de haver um pedido de envio.
* Confiável: a API garante que a mensagem será entregue uma vez e apenas uma vez.

O JMS tem suas peculiaridades, mas são nas seguintes situações que ele costuma ser o escolhido pela indústria no lugar de APIs *tightly coupled* (como o RMI):

* O provedor quer que um componente não dependa das informações dos outros componentes, para uma melhor manutenção;
* O provedor precisa que o aplicativo funcione mesmo que haja outros que não estejam ativos ou funcionais;
* O modelo de negócio prefere um componente que envie informação a outros softwares e continue funcionando, mesmo sem receber uma resposta imediata deles.

## Conceitos básicos do JMS

O JMS funciona levando em conta três conceitos básicos:

* A Arquitetura da API;
* O Domínio das Mensagens;
* O Consumo das Mensagens.

### A arquitetura da API

É composta pelos:

* Provedor JMS: sistema mensageiro que efetua o funcionamento das interfaces JMS e dá os devidos controle e suporte administrativo.
* Cliente JMS: programas ou subprogramas escritos em Java, que consomem e/ou produzem mensagens.
* Mensagens: objetos que comunicam informação entre os clientes JMS.
* Objetos administrados: são objetos pré-configurados criados por um administrador, para o uso dos clientes, há dois tipos deles, os destinos e as fábricas de conexão.
* Clientes nativos: programas que usam o cliente de mensagens da API nativa do produto em questão ao invés da API JMS. É possível que programas feitos antes do JMS sejam modificados e trabalhem com os 2 tipos de clientes citados acima.

### O domínio das mensagens

Há duas possibilidades de domínio, *ponto-a-ponto* e *publicar/assinar* sendo que a API JMS pode trabalhar com apenas uma delas, ou ambas separadamente, ou até com as duas em conjunto.

#### Ponto-a-ponto

No ponto-a-ponto a aplicação lida com os conceitos de filas, remetentes e receptores de mensagens, onde cada mensagem é endereçada a uma fila específica e cada cliente, a fim de receber, deve retirar a suas mensagens da fila relacionada a essa função. As mensagens continuarão na fila até que expirem ou que sejam consumidas.



Imagem 1 – Esquema do ponto-a-ponto

Características do ponto-a-ponto:

* Cada mensagem tem apenas um consumidor.
* Um remetente e um destinatário não têm limite de tempo. O destinatário pode receber a mensagem independentemente se ele estava ativos quando ela foi enviada.
* O destinatário avisa quando recebe uma mensagem com sucesso.

#### Publicar/assinar

Nesse modelo mensagens são endereçadas a um tópico. Editores (que publicam) e assinantes são geralmente anônimos e podem publicar ou assinar dinamicamente na hierarquia do conteúdo. O sistema se encarrega de distribuir as mensagens quem vêm dos vários editores de um tópico para os vários assinantes. Tópicos guardam mensagens apenas o tempo necessário até que elas sejam entregues aos assinantes.



Imagem 2 – Sistema de publicar/assinar

Características do sistema de publicar/assinar:

* Cada mensagem tem múltiplos consumidores.
* Assinantes e editores têm um limite de tempo. Um cliente que assine um tópico pode consumir apenas mensagens publicadas após a sua assinatura, e o assinante deve continuar ativo para que possa consumir mensagens.

Obs. A API pode permitir certa flexibilidade nesse limite de tempo, de modo que há o conceito de *assinaturas duráveis*, as quais permitem o envio de mensagens a assinantes inativos.

### O consumo das mensagens

Sistemas baseados em *messaging* são naturalmente assíncronos, já que não há uma barreira definitiva de tempo impedindo o consumo ou a produção de uma mensagem. O JMS ainda assim, utiliza o termo para se referir a uma situação mais específica. Há duas maneiras de se consumir uma mensagem.

* Síncrona: o assinante ou o destinatário explicitamente recupera a mensagem através de um método *receive*. Método esse que pode bloquear até que a mensagem chegue ou até que o tempo dela se esgote, dado um limite.
* Assíncrona: o cliente pode registrar um *message listener* com o consumidor. Ele é similar a um *event listener* assim, sempre que a mensagem chega ao seu destino o provedor JMS entrega a mensagem chamando o método *onMessage* do *listener*, o qual age sobre os conteúdos da mensagem.

# O funcionamento interno do JMS

Os componentes básicos para a construção de uma aplicação JMS consiste em:

* Objetos administráveis: fábrica de conexão e destino
* Conexão
* Sessões
* Produtores de mensagem
* Consumidores de mensagem
* Mensagens



**Imagem 3 – modelo de componentes para a API JMS**

Assim que a conexão é criada pela fábrica de conexões, ela cria também a sessão referente a um produtor e um consumidor de mensagens para realizarem comunicações.

## Objetos administráveis

O destino e a fábrica de conexão são duas partes de uma aplicação JMS que são melhores mantidas administrativamente do que programadas, e por isso a tecnologia implementada para controlar essas partes difere muito de uma aplicação para outra. O gerenciamento desses objetos pertence a tarefas administrativas de cada provedor de serviços.

Os clientes JMS acessam esses objetos por interfaces portáveis necessitando de poucas ou nenhuma mudança. Esses objetos administrativos podem ser organizados no SDK do J2EE, com a ferramenta *j2eeadmin*.

### Fábricas de conexão

Uma fábrica de conexão é o objeto usado para que um cliente crie uma conexão com um provedor. Este objeto encapsula os parâmetros de configuração da conexão definidos pelo administrador, no SDK do J2EE já existem 2 interfaces de fábricas de conexão pré configuradas, *TopicConnectionFactory* e *QueueConnectionFactory.*

Assim é possível criar conexões com o SDK do J2EE usando as fábricas de conexão padrões com os seguintes comandos:

j2eeadmin -addJmsFactory *jndi\_name* queue

j2eeadmin -addJmsFactory *jndi\_name* topic

Ao início do programa JMS é uma boa prática executar a localização da fábrica de conexão, para isso pode-se instanciar o objeto *InitialContext* para fazer essa localização da seguinte maneira:

Context ctx = new InitialContext();

QueueConnectionFactory queueConnectionFactory =

(QueueConnectionFactory) ctx.lookup("QueueConnectionFactory");

TopicConnectionFactory topicConnectionFactory =

(TopicConnectionFactory) ctx.lookup("TopicConnectionFactory");

### Destinos

Um destino é o objeto que o cliente utiliza para especificar o alvo das mensagens produzidas e a fonte das mensagens que são recebidas. Dependendo do domínio, o destino assume nomes diferentes. Para domínios *Point to Point (*PTP*)* o destino se chama *queue,* e para ser criado usa-se o seguinte comando:

j2eeadmin -addJmsDestination *queue\_name* queue

Já para o domínio de mensagens pub/sub, os destinos são chamados de *topics*, e para cria-los usa-se o comando:

j2eeadmin -addJmsDestination *topic\_name* topic

Assim como as fábricas, é uma boa prática fazer a localização dos destinos, tanto para *queue* quanto *topic*, os comandos são:

Topic myTopic = (Topic) ctx.lookup("MyTopic");

Queue myQueue = (Queue) ctx.lookup("MyQueue");

## Conexões

Uma conexão encapsula uma conexão virtual com um provedor JMS e pode representar um socket TCP/IP aberto entre o cliente e o provedor de serviços. A conexão cria uma ou mais sessões e pode ser vista de duas maneiras, assim como a fábrica de conexão, sendo implementada com o *QueueConnect* ou usando a interface TopicConnection:

QueueConnection queueConnection = queueConnectionFactory.createQueueConnection();

TopicConnection topicConnection = topicConnectionFactory.createTopicConnection();

Quando uma aplicação completa seu ciclo, é preciso fechar todas as conexões criadas com o método *close()*:

queueConnection.close(); topicConnection.close();

Uma falha no fechamento da conexão pode fazer com que recursos não sejam liberados pelo provedor JMS.

## Sessões

Uma sessão é usada para criar produtores e consumidores de mensagens e as próprias mensagens. Ela produz um contexto transacional que agrupa mensagens a enviar e mensagens recebidas numa unidade atômica de trabalho.

Assim como as conexões, a sessão pode ser implementada no *QueueSession* ou utilizada com a interface *TopicSession*, que pode ser instanciado da seguinte maneira:

TopicSession topicSession = topicConnection.createTopicSession(false,

Session.AUTO\_ACKNOWLEDGE);

Onde o primeiro argumento significa que a sessão é não transacionada, o segundo argumento significa que a sessão automaticamente verificará se a mensagem foi entregue com sucesso.

Para o *QueueSession*:

QueueSession queueSession = queueConnection.createQueueSession(true, 0);

Onde o primeiro argumento significa que a sessão é transacionada, o segundo argumento significa que a sessão não verificará se a mensagem foi entregue com sucesso.

## Produtores de Mensagem

É um objeto criado por uma sessão e é usado para enviar uma mensagem para um destino. Enquanto o formato PTP implementa a interface *QueueSender* o formato pub/sub implementa a interface *TopicPublisher*.

Por exemplo, você utiliza o *QueueSession* para criar um objeto de envio e um *TopicSession* para criar um publicador da seguinte maneira:

QueueSender queueSender = queueSession.createSender(myQueue);

TopicPublisher topicPublisher = topicSession.createPublisher(myTopic);

É possível criar um produtor não identificado, especificando o argumento como *null*, e assim pode-se aguardar a especificação do destino que será enviada a mensagem para então realizar a publicação da mesma.

Com o método *send(message)* é possível enviar mensagens:

queueSender.send(message);

E o método para publicar:

topicPublisher.publish(message);

## Consumidores de Mensagem

É um objeto criado por uma sessão e é usado para receber mensagens enviadas a um destino. O provedor JMS gerencia a entrega das mensagens de um destino para o consumidor registrado do destino.

QueueReceiver queueReceiver = queueSession.createReceiver(myQueue);

TopicSubscriber topicSubscriber = topicSession.createSubscriber(myTopic);

Quando o consumidor é criado, o mesmo fica ativo e já pode ser usado para receber mensagens, para torná-lo inativo deve ser usado o método *close* para o *QueueReceiver* ou para um *TopicSubscriber*. O método *receive* consome a mensagem de maneira síncrona, pode ser usado a qualquer momento após o método *start*.

### Message Listeners

*Message Listeners* é um objeto que atua como um gerenciador de eventos assíncrono com mensagens. Implementa a interface *MessageListener* que contém o método *onMessage,* que define as ações a serem tomadas quando uma mensagem chega. O *messageListener* pode ser registrado com o método *setMessageListener:*

TopicListener topicListener = new TopicListener();

topicSubscriber.setMessageListener(topicListener);

É muito importante registrar primeiro o *messageListener* antes de chamar o método *start* para começar a receber mensagens, se o método *start* se iniciar antes poderão haver mensagens não registradas. Qualquer exceção deve ser lidada com o método *onMessage*. A sessão usada para criar o consumidor da mensagem serializa a execução de todos os *message listeners* registrados na sessão. A qualquer momento apenas uma sessão do *message listener* está executando.

### Seletores de Mensagem

Se sua aplicação de mensagens precisa filtrar as mensagens que recebe, você pode usar a API JMS *message selector*, que permite o consumidor da mensagem especificar as mensagens que está interessado. Os seletores de mensagem define o trabalho de filtrar as mensagens para o provedor JMS e não para a aplicação.

Portanto, com os métodos *createReceiver, createSubscriber* e *createDurableSubscriber* é possível especificar um *message selector* como um argumento no momento em que se cria o consumidor da mensagem, e assim, o consumidor recebe apenas mensagens em que o cabeçalho e suas propriedades conferem com o seletor. Não se pode usar o seletor para verificar o texto do corpo de uma mensagem.

## Mensagens

As mensagens no JMS têm um formato básico simples e bastante flexível que permite que sejam criadas mensagens que conferem com outros formatos de aplicações não-JMS e até em plataformas heterogêneas.

Uma mensagem JMS possui três partes:

* Cabeçalho
* Propriedades (opcional)
* Corpo (opcional)

### Cabeçalho

O cabeçalho da mensagem JMS contém um número de campos predefinidos que contém valores, que são usados por clientes e provedores, para identificar e rotear mensagens. Alguns campos do cabeçalho são definidos pelo cliente, enquanto outros são definidos automaticamente pelo método *send* ou *publish*.

**Campos e definição de valores**

|  |  |
| --- | --- |
| **Nome do campo** | **Definido por** |
| JMSDestination | Método *send* ou *publish* |
| JMSDeliveryMode | Método *send* ou *publish* |
| JMSExpiration | Método *send* ou *publish* |
| JMSPriority | Método *send* ou *publish* |
| JMSMessageID | Método *send* ou *publish* |
| JMSTimestamp | Método *send* ou *publish* |
| JMSCorrelationID | Cliente |
| JMSReplyTo | Cliente |
| JMSType | Cliente |
| JMSRedelivered | Provedor JMS |

### Propriedades de uma mensagem

Se for preciso é possível criar e definir propriedades para uma mensagem que não esteja satisfatória com os campos do cabeçalho. Propriedades podem ser usadas para prover compatibilidade com outros sistemas de mensagem. O uso das propriedades é opcional.

### Corpos de mensagem

Existem 5 formatos ou tipos de corpo para a API JMS, assim dados podem ser enviados de diferentes formas que sejam compatíveis com uma vasta gama de aplicações.

**Tipos de mensagem JMS**

|  |  |
| --- | --- |
| **Tipo** | **Conteúdo** |
| TextMessage | Um objeto Java.lang.String, por exemplo, o conteúdo de uma linguagem de marcação. |
| MapMessage | Um grupo com pares de nome/valor, com nomes em objeto String e valores como tipos primitivos da linguagem Java. Pode ser acessado sequencialmente com enumerador ou randomicamente por nome. A ordem dos registros é indefinida. |
| BytesMessage | Um fluxo de bytes não interpretados. Esse tipo de mensagem é para literalmente para conferir um corpo com um formato de mensagem já existente. |
| StreamMessage | Um fluxo de valores primitivos na linguagem Java, não vazio e lido sequencialmente. |
| ObjectMessage | Um objeto Serializable na linguagem Java. |
| Message | Nada. Composto de cabeçalho e propriedades apenas. Esse tipo de mensagem é útil quando um corpo não é necessário. |

Para criar e enviar uma mensagem com corpo *TextMessage* para um *queue* é feito o seguinte:

TextMessage message = queueSession.createTextMessage();

message.setText(msg\_text); // msg\_text is a String

queueSender.send(message);

Ao final, uma mensagem chega como um objeto *Message* genérico e precisa ser feito o *cast* para o tipo apropriado, pode-se usar o método *get*:

Message m = queueReceiver.receive();

if (m instanceof TextMessage) {

TextMessage message = (TextMessage) m;

System.out.println("Reading message: " + message.getText());

} else {

// Handle error

}

## Tratamento de Exceção

A classe base para exceções lançadas pela API JMS é a *JMSException.* Porém ela lida de maneira genérica com a exceção e para fins mais específicos podem ser usadas suas subclasses:

•IllegalStateException

• InvalidClientIDException

• InvalidDestinationException

• InvalidSelectorException

• JMSSecurityException

• MessageEOFException

• MessageFormatException

• MessageNotReadableException

• MessageNotWriteableException

• ResourceAllocationException

• TransactionInProgressException

• TransactionRolledBackException