



# Padrões de

# Integração de

# Sistemas

## com aplicações em Java

Helder da Rocha  
[www.agonavis.com.br](http://www.agonavis.com.br)  
4a. revisão. 06.08.2017

## Termos de uso

Este tutorial contém material (texto, código, imagens) produzido por Helder da Rocha entre dezembro de 2013 e agosto de 2017 e pode ser usado de acordo com os termos da licença *Creative Commons BY-SA (Attribution-ShareAlike)* descrita em <http://creativecommons.org/licenses/by-sa/3.0/br/legalcode>.

Citações, imagens e código de terceiros, com fonte indicada (ex: exemplos, diagramas) podem ter termos de uso diferentes.

Ícones, diagramas e tradução do texto do “problema” e “solução” de cada padrão do catálogo EIP foram reusados de acordo com os termos da licença CC-BY, descrita no site [www.eaipatterns.com](http://www.eaipatterns.com).

Código usado em exemplos e exercícios disponíveis nos repositórios *GitHub* de autoria de Helder da Rocha ou Argo Navis são software livre e têm licença de uso Apache 2.0.

R672g Rocha, Helder Lima Santos da, 1968-

*Padrões de Integração de Sistemas com Aplicações em Java. Quarta Revisão.*

275p. 21cm x 29.7cm. PDF.

Documento criado em 13 de novembro de 2011.

Primeira revisão: 26 de janeiro de 2013. Segunda revisão: 31 de agosto de 2015.

Terceira revisão: 11 de outubro de 2015. Quarta revisão: 06 de agosto de 2017.

1. Telecomunicações – Mensageria.
2. Java (*Linguagem de programação de computadores*) – Arquitetura de Sistemas de Mensageria.
3. Arquitetura de Sistemas de Mensageria (*Engenharia de Software*).
3. Padrões de Design (*Engenharia de Software*).
- I. Título.

CDD 005.7'136

# Conteúdo

<b>Capítulo 1: Introdução .....</b>	<b>1</b>
O que é integração.....	2
Tecnologias Java para integração.....	3
Tecnologias para representação de dados .....	5
Padrões .....	7
Resumo .....	9
<b>Capítulo 2: Integração .....</b>	<b>2</b>
Estilos de Integração.....	3
(1) Transferência de arquivos (File Transfer).....	3
(2) Banco de dados compartilhado (Shared Database).....	5
(3) RPC (Remote Procedure Call).....	6
(4) Mensageria (Messaging) .....	7
<b>Capítulo 3: Mensageria .....</b>	<b>11</b>
(5) Canal de Mensagens (Message Channel) .....	12
(6) Mensagem (Message) .....	15
(7) Dutos e filtros (Pipes and Filters) .....	21
(8) Roteador de mensagens (Message Router) .....	25
(9) Tradutor de mensagens (Message Translator).....	28
(10) Terminal de Mensageria (Messaging Endpoint).....	32
Padrões de gerenciamento do sistema .....	36
Frameworks.....	36
Revisão.....	44
<b>Capítulo 4: Canais .....</b>	<b>45</b>
(11) Canal Ponto-a-Ponto (Point-to-Point Channel).....	47
(12) Canal Publicar-Inscrever (Publish-Subscribe Channel) .....	53
(13) Canal de Tipo de Dados (Datatype Channel) .....	60
(14) Canal de Mensagens Inválidas (Invalid Message Channel).....	63
(15) Canal de Mensagens Não-Entregues (Dead Letter Channel).....	66
(16) Entrega Garantida (Guaranteed Delivery) .....	69
(17) Adaptador de Canal (Channel Adapter) .....	71
(18) Ponte de Mensageria (Messaging Bridge) .....	76
(19) Barramento de Mensagens (Message Bus) .....	78
Revisão.....	79
<b>Capítulo 5: Mensagens .....</b>	<b>80</b>
(20) Mensagem-comando (Command Message).....	82
(21) Mensagem-documento (Document Message) .....	85
(22) Mensagem-evento (Event Message) .....	87

(23) Requisição-resposta (Request-Reply).....	91
(24) Endereço de Resposta (Return Address).....	98
(25) Identificador de Correlação (Correlation Identifier).....	99
(26) Sequência de Mensagens (Message Sequence) .....	101
(27) Prazo de Validade (Message Expiration) .....	109
(28) Indicador de Formato (Format Indicator) .....	110
Revisão.....	112
<b>Capítulo 6: Roteamento .....</b>	<b>113</b>
(29) Roteador baseado em conteúdo (Content-Based Router) (CBR).....	115
(30) Filtro de mensagens (Message Filter).....	124
(31) Roteador dinâmico (Dynamic Router).....	130
(32) Lista de receptores (Recipient List) .....	136
(33) Divisor (Splitter).....	143
(34) Agregador (Aggregator).....	149
(35) Re-sequenciador (Resequencer) .....	153
(36) Processador de Mensagens Compostas (Composed Message Processor) (CMP) .....	158
(37) Espalha-Recolhe (Scatter-Gather) .....	160
(38) Lista de circulação (Routing Slip) .....	161
(39) Gerente de Processos (Process Manager) .....	163
(40) Corretor de mensagens (Message Broker) .....	164
Revisão.....	166
<b>Capítulo 7: Transformação.....</b>	<b>168</b>
(41) Envelope (Envelope Wrapper).....	169
(42) Enriquecedor de conteúdo (Content Enricher) .....	174
(43) Filtro de conteúdo (Content Filter) .....	177
(44) Recibo de bagagem (Claim Check).....	180
(45) Normalizador (Normalizer).....	183
(46) Modelo de dados canônico (Canonical Data Model) .....	185
Revisão.....	186
<b>Capítulo 8: Endpoints .....</b>	<b>188</b>
(47) Gateway de mensageria (Messaging Gateway).....	190
(48) Mapeador de mensageria (Messaging Mapper).....	197
(49) Cliente transacional (Transactional Client).....	201
(51) Consumidor de sondagem (Polling Consumer) .....	207
(50) Consumidor ativado por eventos (Event-Driven Consumer) .....	210
(52) Consumidores concorrentes (Competing Consumers).....	213
(53) Despachante de mensagens (Message Dispatcher) .....	215
(54) Consumidor seletivo (Selective Consumer) .....	219
(55) Assinante durável (Durable Subscriber) .....	222
(56) Receptor idempotente (Idempotent Receiver) .....	226
(57) Ativador de serviço (Service Activator) .....	230
Revisão.....	235
<b>Capítulo 9: Gerenciamento .....</b>	<b>236</b>
(58) Barramento de controle (Control Bus).....	237
(59) Desvio (Detour) .....	243
(60) Escuta (Wire Tap) .....	245
(61) Histórico da Mensagem (Message History) .....	249
(62) Repositório de mensagens (Message Store) .....	251
(63) Proxy inteligente (Smart Proxy).....	253

(64) Mensagem de Teste (Test Message) .....	255
(65) Purificador de canal (Channel Purger) .....	258
Revisão.....	259
<b>Apêndice A: Ambiente.....</b>	<b>261</b>
Apache ActiveMQ.....	261
Apache Camel.....	263
Spring Integration.....	265
Ferramentas do Eclipse .....	267
<b>Apêndice B: Referências.....</b>	<b>269</b>
Livros .....	269
Especificações .....	270
Artigos .....	270
Documentação, tutoriais e referências diversas.....	270

# Prefácio

Este texto foi elaborado como material didático de apoio ao curso *Padrões de Integração de Sistemas com aplicações em Java*, que tem como objetivo introduzir e discutir os 65 padrões de integração de sistemas [EIP] do catálogo Gregor Hohpe e Bobby Woolf: *Enterprise Integration Patterns – Designing, Building and Deploying Messaging Solutions* (Addison-Wesley, 2003), apresentando exemplos usando APIs e frameworks na linguagem Java.

O texto explora os padrões mais importantes de cada grupo, com exemplos e exercícios em Java (usando *JMS*, *EJB* e *Web Services*) e frameworks populares que implementam esses padrões: *Apache Camel* e *Spring Integration*.

Este não é um curso sobre *JMS*, *Camel* ou *Spring Integration*! Exemplos e exercícios usando esses frameworks são apresentados para demonstrar como os padrões podem ser implementados na prática. O objetivo é explorar conceitualmente cada padrão da forma como é apresentado no catálogo [EIP], mostrando soluções e destacando diferenças nas implementações e arquiteturas, não se prendendo a nenhuma implementação específica. Tanto o Camel como o Spring Integration e outras implementações dos padrões adotam soluções diferentes não apenas em relação à arquitetura, mas também em relação a terminologia, conceitos e definições. Conhecer bem os padrões do catálogo [EIP], no entanto, ajuda a entender melhor como esses frameworks implementaram suas soluções.

Ao concluir este curso você deverá ter condições de

- Identificar um padrão de integração pelo nome e problema/solução.
- Dentro de um grupo de padrões similares, selecionar quais podem ser usados como solução para um dado problema de integração, apontar prós e contras e indicar possíveis implementações.
- Analisar um sistema que necessita de integração e descrever uma solução baseada em padrões de integração.
- Aplicar o conhecimento sobre padrões no design e implementação de uma solução de integração usando APIs Java (como JMS), frameworks de integração de sistemas como o Camel ou Spring Integration, ou ainda ESBs como Mule, TIBCO, Talend, WSO2 e outros.

Apesar do foco deste curso ser arquitetura, exemplos em código são fundamentais para ilustrar a aplicação prática dos padrões. A execução dos exemplos ilustram aspectos da aplicação dos padrões que não seriam possíveis apenas com texto e ilustrações. Os exemplos são fornecidos principalmente em JMS, mas também há exemplos executáveis em Apache Camel e Spring Integration para a maior parte dos padrões. Para executar os exemplos e exercícios você deve ter um computador Linux, Mac ou PC com uma IDE Java (qualquer uma que suporte Maven), ambiente Java 8, Maven 3+ e Apache ActiveMQ, além de um acesso direto à Internet para que o Maven possa baixar dependências adicionais (dependências do Java EE, JMS, Camel, Spring, etc. serão obtidas via Maven).

Nem todos os exemplos de código mostrados são executáveis. Eles podem ser fragmentos de exemplos executáveis maiores. Às vezes trechos são omitidos ou destacados para maior clareza e para dar ênfase o assunto que está sendo discutido. Os exemplos e fragmentos foram criados para ilustrar a construção e comportamento dos padrões. Eles não representam necessariamente soluções úteis. Com exceção de alguns poucos fragmentos extraídos da documentação do Camel e Spring Integration, a maior parte dos fragmentos de código foram retirados de código executável que está disponível no repositório GitHub deste curso em:

<https://github.com/argonavisbr/EIP-Course>

Para tirar maior proveito deste material você deve saber programar em Java e estar familiarizado com as APIs Java EE: JMS, EJB, JPA, servlets (as aplicações a serem integradas usam essas tecnologias e outras).

# Capítulo 1

# Introdução

A integração de sistemas é uma tarefa necessária. Não existe uma única aplicação que faça tudo, que trate de todos os aspectos de um negócio, e não vale a pena criar tal aplicação. Para realizar tarefas diferentes, ainda que relacionadas, geralmente executamos aplicações diferentes, cada qual especializada em resolver eficientemente um problema específico. Mas essas aplicações precisam trabalhar juntas para que possam suportar processos de negócio que envolvem todas elas, compartilhar dados, regras, processos. Por isso precisam ser *integradas*.

A integração de aplicações não é simples. A maior parte das aplicações não foi construída pensando em integração, e mesmo as que foram, geralmente precisam adaptar formatos, dados, interfaces. A integração precisa ser eficiente, confiável e segura. Além disso, é importante que a solução seja independente de sistema, conecte as aplicações com acoplamento fraco e seja tolerante a mudanças, permitindo que diferentes aplicações integradas evoluam sem afetar outras que participam da integração. É um problema que envolve muitos desafios. Dentre as várias soluções possíveis para cada problema de integração há vantagens e desvantagens que precisam ser consideradas em contexto. Portanto, a construção de soluções é uma tarefa complexa.

Uma forma de lidar com essa complexidade é aprender com a experiência dos que já testaram e avaliaram muitas soluções possíveis. *Padrões de design* são uma técnica usada para documentar experiência e conhecimento. Os *padrões de integração de sistemas* refletem as melhores soluções resultantes da experiência acumulada de profissionais resolvendo problemas recorrentes de integração. As soluções apresentadas na forma de padrões buscam demonstrar soluções genéricas para problemas comuns encontrados na integração.

O catálogo [EIP], usado como referência principal para este texto, relaciona 65 padrões. Mas todos os frameworks que implementam os padrões de integração oferecem mecanismos e componentes adicionais. Alguns deles são inclusive padrões descritos em outros catálogos como [PEAA], [POSA], [GoF], [SDP]. Este texto concentra-se apenas nos padrões listados no catálogo [EIP] que focam principalmente em mensageria.

## O que é integração

Integração de sistemas é fazer vários sistemas trabalharem juntos, de forma coordenada. Dentro do escopo considerado neste curso, a integração envolve uma comunicação entre máquinas (*Business to Business*).

Um sistema *integrado* não é a mesma coisa que um sistema *distribuído* (embora seja possível substituir um sistema distribuído por uma solução de integração que realize as mesmas tarefas). Um sistema distribuído é uma aplicação que tem partes fortemente acopladas, na qual uma parte não funciona sem a outra e que geralmente se comunica de forma síncrona. Sistemas que participam de uma integração são aplicações independentes que podem rodar sozinhas, mas operam de forma coordenada e com baixo acoplamento. A integração permite que cada uma se concentre em uma funcionalidade.

A maior parte da comunicação entre aplicações em um sistema integrado é realizada de forma *assíncrona*. Na comunicação síncrona um cliente chama um serviço e precisa esperar até o fim da execução da operação remota para que possa continuar seu processamento. A chamada bloqueia o *thread* e não continua até receber uma resposta. Na comunicação assíncrona o cliente chama o serviço e não espera resposta. Pode nunca receber uma resposta porque não precisa, ou porque ela chegará como uma notificação em outra parte da aplicação. A comunicação assíncrona permite comunicação com acoplamento muito baixo.

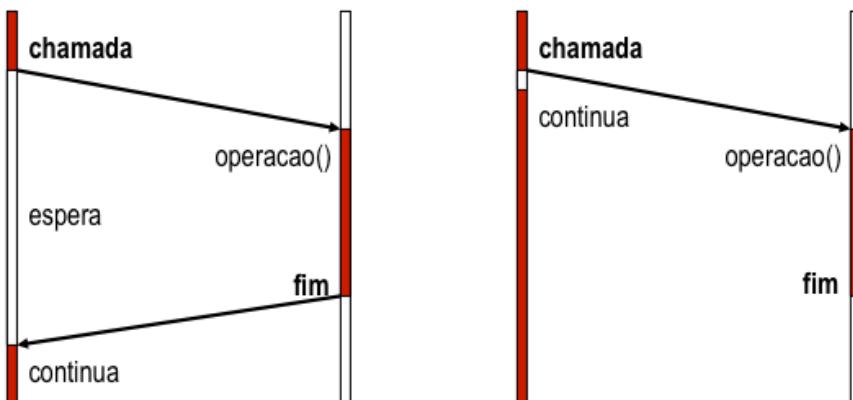


Figura 1 Comunicação síncrona e assíncrona

Soluções como RPC (*Remote Procedure Call*) ou RMI (*Remote Method Invocation*) utilizam comunicação síncrona. A comunicação assíncrona é alcançada com mensageria.

Típicos cenários de integração incluem portais de informação (que precisam concentrar dados de vários lugares em um lugar só), replicação de dados (quando aplicações precisam copiar dados umas das outras) e compartilhamento de processos e funcionalidades.

Um sistema integrado pode ser construído combinando *serviços*. Processos de negócios distribuídos e SOAs estão em um caminho intermediário entre sistemas distribuídos e integrados. A chamada de um serviço pode ser vista como integração entre aplicações.

## Tecnologias Java para integração

Existem várias tecnologias Java que podem ser usadas para integrar sistemas. No *Java Standard Edition* (Java SE) há pacotes para comunicação com o sistema de arquivos, acesso a banco de dados, comunicação em rede via sockets, URLs, comunicação RMI/RPC via objetos distribuídos, além de pacotes para processamento de XML. Java 8 oferece ainda um pacote de *streams* que podem ser usados para processamento eficiente de grandes quantidades de dados. Nos pacotes do *Java Enterprise Edition* (Java EE), implementados por servidores de aplicação, há soluções de alto-nível como *Enterprise JavaBeans (EJB)* que oferece objetos gerenciados para serviços síncronos, *Servlets* e *JavaServer Faces* que oferecem uma API para aplicações Web, *JAX-WS* com implementação de Web Services SOAP, *JAX-RS* com Web Services REST e *Java Message Service (JMS)*, que oferece uma interface para sistemas de mensageria.

Os pacotes *java.io*, *java.nio* e *java.net* contém mecanismos gerais de entrada e saída síncrona (blocking IO) e assíncrona (non-blocking IO) para arquivos, sockets e recursos acessíveis via URLs e soquetes de rede. Um típico acesso síncrono *java.io* utiliza-se de *InputStreams*, *OutputStreams*, *Readers* e *Writers*. Com *java.nio* pode-se construir arquiteturas assíncronas de *Dutos* e *Filtros* em baixo nível tratando arquivos e soquetes como canais. As classes *Socket/SocketChannel* e *File/FileChannel* abstraem arquivos e soquetes e podem ser usadas para construir terminais de acessos a portas de rede e arquivos, em soluções de integração.

Java SE também oferece uma API de acesso a bancos de dados através da interface JDBC, pela qual pode-se construir DAOs ou camadas de persistência. Java EE adiciona o framework JPA que traz suporte a persistência transparente e automática de objetos. Ambas podem ser usadas como pontos de integração, já que bancos de dados podem ser compartilhados entre aplicações.

Java RMI é uma API de objetos distribuídos que pode usar protocolos de comunicação Java ou CORBA (IIOP), permitindo a integração de sistemas através de adaptadores de interface (aplicações Java) ou IDLs (aplicações de linguagens diferentes através de um ORB). É uma solução síncrona normalmente usada em aplicações distribuídas, mas que pode ser usada como solução de integração.

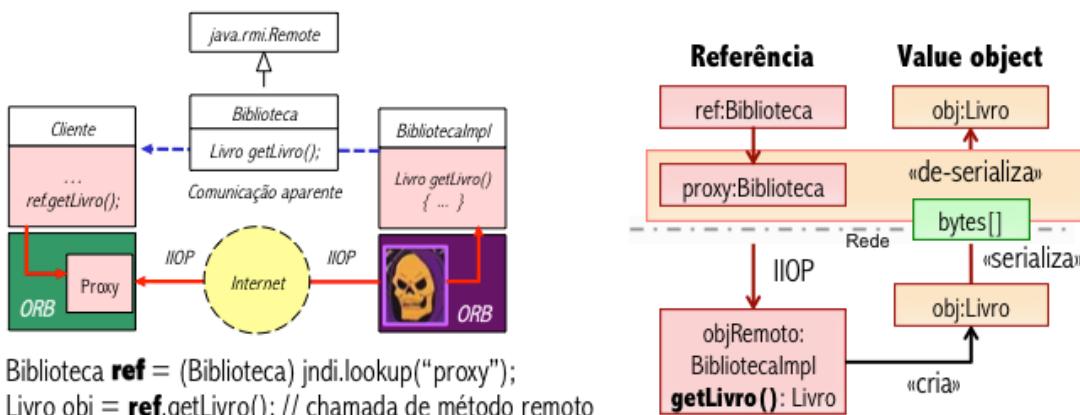


Figura 2 Arquitetura Java RMI (*java.rmi*)

Java RMI é solução de baixo nível se comparada a EJB (*Enterprise JavaBeans*), que fornece objetos gerenciados que podem ser distribuídos remotamente. Um EJB é uma classe Java simples (POJO) que é tratada como componente, se configurada em container específico, fornecido por um servidor de aplicações Java EE. O container controla o ciclo de vida do objeto e permite interceptar todos os seus métodos, incluindo seu contexto transacional e outros aspectos com interesses ortogonais configurados via anotações ou XML. EJBs podem ser síncronos (*Session Beans*) ou assíncronos (*Message-Driven Beans*) e geralmente são usados para modelar serviços. Os serviços síncronos também podem ser stateless (*Stateless* e *Singleton Session Beans*) ou stateful (*Stateful Session Beans*), embora seja mais comum (e recomendado) a preferência por componentes stateless para serviços, usando entidades construídas via JPA para guardar estado. EJBs síncronos podem oferecer serviços via JVM, IIOP, REST e SOAP, e EJBs assíncronos são receptores de canais de mensageria via JMS, portanto oferecem vários pontos de acesso que podem ser usados em integração.

As APIs JAX-WS e JAX-RS são frequentemente usadas com EJB e JPA e permitem disponibilizar serviços através de pontos de acesso HTTP.

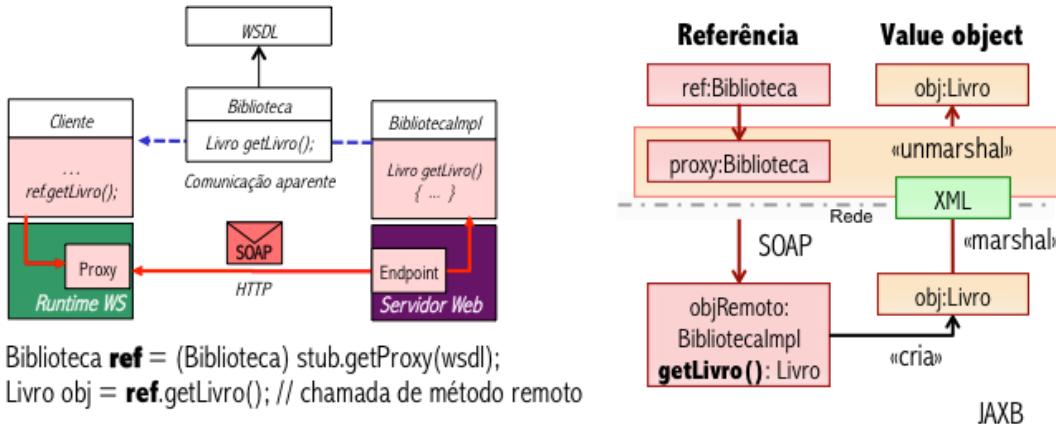


Figura 3 Arquitetura JAX-WS (SOAP Web Services)

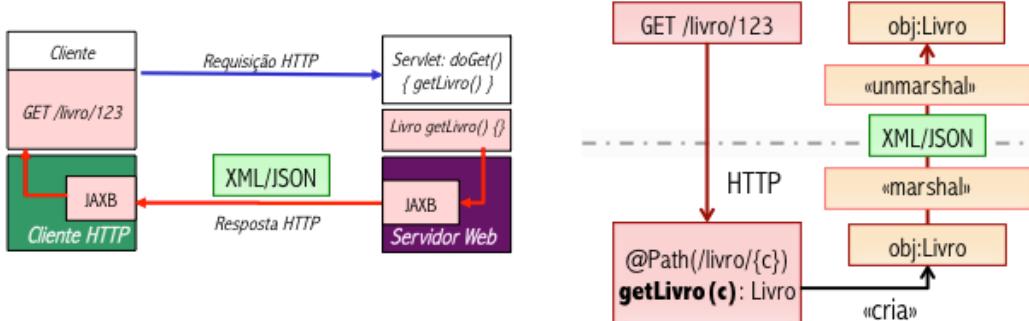


Figura 4 Arquitetura JAX-RS (RESTful Web Services)

Finalmente, usando a API *Java Message Service* (JMS), suportada por qualquer servidor de aplicações Java EE, é possível interligar serviços através de canais de mensageria. JMS é uma das principais tecnologias usadas em soluções de integração, e é suportada não apenas por servidores Java EE, mas pelos principais e mais populares servidores de mensageria do mercado. A maior parte dos padrões do catálogo [EIP] podem ser implementados através de soluções usando apenas Java e JMS.

As tecnologias mencionadas até aqui fazem parte das *distribuições padrão* do Java. Mas além delas é importante mencionar o *Spring*, que oferece alternativas para várias das soluções do Java EE. Objetos no Spring são automaticamente gerenciados, como são os componentes EJB (e provavelmente todos os objetos do Java no futuro, quando o CDI for finalmente incorporado ao Java SE). Spring pode e é usado por todos os frameworks que implementam os padrões integração de sistemas, não apenas com o Spring Integration, mas também com Camel e Mule.

## Tecnologias para representação de dados

Integração não consiste apenas em conectar um recurso a um canal de acesso. É preciso muitas vezes lidar com os dados que estão em formatos diferentes e incompatíveis. Uma aplicação abre um arquivo esperando um XML e recebe um CSV, ou um objeto Java serializado, ou um formato binário proprietário. É preciso transformá-lo. Existe uma categoria inteira de padrões de integração relacionados à *transformação de dados*, que pode revelar-se uma tarefa muito complexa, principalmente se envolver formatos proprietários ou binários. Isto é cada vez mais raro. Uma grande parte dos dados que fluem pelos canais que integram sistemas são formatos de *texto*, e os mais populares, hoje, provavelmente são formatos abertos como XML e JSON.

Tanto XML como JSON estruturam e representam dados como uma árvore com uma única raiz. É comum usar essas tecnologias para representar objetos. É típico o mapeamento de classes a esquemas, associado ao mapeamento de objetos (instanciados das classes) a documentos (produzidos pelos esquemas).

Considere, por exemplo, a seguinte estrutura de classes

```
01.      class Telefone {  
02.          int ddd;  
03.          int numero;  
04.      }  
05.  
06.      class Pessoa {  
07.          int numero;  
08.          String nome;  
09.          Telefone[] telefones = new Telefone[3];  
10.      }
```

que poderia ser mapeada a um XSD ou um esquema JSON. O trecho abaixo ilustra a criação de instâncias dessas classes em Java:

```
1.      Pessoa pessoa = new Pessoa();  
2.      pessoa.numero = 145;  
3.      pessoa.nome = "Jeeves";  
4.      pessoa.telefone[0] = new Telefone();  
5.      pessoa.telefone[0].ddd = 11;  
6.      pessoa.telefone[0].numero = 34567890;
```

Para compartilhar essa informação com outras aplicações em outro sistema e linguagem, poderíamos usar uma representação em XML (`application/xml`), por exemplo:

```
1. <pessoa xmlns="http://pessoa" numero="145">
2.   <nome>Jeeves</nome>
3.   <telefone>
4.     <ddd>11</ddd>
5.     <numero>34567890</numero>
6.   </telefone>
7. </pessoa>
```

ou em JSON (`application/json`), da forma:

```
1. {
2.   "@numero": 145,
3.   "nome": "Jeeves",
4.   [ "telefone": {
5.     "ddd": 11, "numero": 34567890
6.   } ]
7. }
```

Muitas das abstrações de dados e objetos que existem para bancos de dados relacionais também existem para XML e JSON. A maior parte dos bancos de dados NoSQL, por exemplo, representam objetos em formato JSON. Existem várias APIs que convertem JSON em Java e XML em Java, além de pacotes que fazem o mapeamento automático., como o JAXB, que faz parte do Java SE e permite que documentos XML sejam usados transparentemente como objetos em aplicações Java.

JAXB é para o XML o que o JPA é para o JDBC. Mapeia classes a esquemas XSD e objetos a documentos XML. JAXB é uma API de interfaces. Uma das implementações de JAXB, o EclipseLink MOXy, permite mapear não apenas XML via JAXB, mas também JSON. Ainda não é padrão, mas o mapeamento automático JSON-Java está prometido para o próximo lançamento do Java (9) ou do Java EE (8). Esse mapeamento é usado automaticamente em aplicações JAX-RS, que geralmente aceitam representações de objetos tanto como JSON como XML.

Há atualmente duas APIs no Java EE (7) para processamento e produção de JSON, e várias outras para processamento de XML. Para XML, JAXB é a que fornece o maior nível de abstração, já que realiza mapeamento transparente entre Java e XML. Outras APIs do Java SE para processamento XML incluem SAX, que é ideal para leitura, pesquisa e extração de dados de documentos XML muito grandes (lê o documento como um stream e não guarda na memória), e o DOM (Document Object Model), que carrega um documento XML inteiro na memória e o representa como uma árvore. O DOM pode ser usado para ler, construir e processar XML.

Além de DOM e SAX, que são APIs de baixo nível e que adotam uma interface padrão determinadas por entidades independentes (como o W3C), existe uma API de streaming similar ao SAX, porém mais eficiente, chamada StAX (Streaming API for XML) e uma API para transformação XSLT chamada TrAX (Transformation API for XML). A API TrAX permite que regras de transformação sejam escritas em XML usando a linguagem funcional XSLT, compiladas e reutilizadas para diversos documentos

similares. XSLT é uma ferramenta útil na construção de processadores que realizam transformação de dados, quando os dados de entrada estão em formato XML. XSLT usa XPath para localizar e extrair informações do XML. O XPath também pode ser usado em Java através da API JAXP.

Para transformar dados que não estão estruturados nem em JSON nem em XML, pode-se recorrer a bibliotecas externas. Existem bibliotecas para extrair dados e transformar formatos CSV, HTML e até mesmo Excel, Word e PDF. A linguagem Java possui nativamente bibliotecas para manipular bytes individuais, encoding Base64 e outros recursos para processamento binário. Também existem diversos recursos na linguagem para extrair e transformar dados em Strings, usando desde posicionamento de caracteres a expressões regulares.

Além das APIs padrão, pacotes de terceiros podem ser usados para auxiliar na transformação de dados. Exemplos são pacotes do Apache Commons e componentes fornecidos pelo Camel e Spring. Um processador de transformação também pode recorrer a um serviço externo para realizar uma transformação.

## Padrões

Um padrão representa uma *decisão de design* e as considerações que justificam essa decisão. A engenharia de software tem visto nos últimos anos o surgimento de vários *catálogos de padrões*, organizados em uma “linguagem de padrões”, baseada no exemplo pioneiro do livro *A Pattern Language*, de Christopher Alexander, que descreve padrões para arquitetura e engenharia civil. As três mais populares referências na engenharia de software talvez sejam os catálogos *Design Patterns* [GoF], *Pattern Oriented Software Architecture* [POSA] e *Patterns of Enterprise Application Architecture* [PEAA].

Uma linguagem de padrões apresenta padrões de forma interligada, mostrando a relação entre eles e com os problemas onde são aplicados, servindo de guia para auxiliar no design de sistemas complexos. Uma linguagem de padrões é uma técnica eficaz e testada para documentar a experiência e conhecimento adquiridos de forma que seja mais facilmente compreendida e reusada por outros.

Um padrão não é apenas uma receita de bolo do tipo “se você tem este problema, faça isto”. Cada padrão apresenta um problema e uma solução, mas também discute em detalhes por que tal problema é difícil de solucionar, quais as razões para que se adote determinada solução, por que essa solução é melhor que outra, e ainda em que condições outras soluções poderiam ser mais adequadas. Os padrões estão relacionados entre si e às vezes representam até mesmo soluções opostas, que seriam adotadas em contextos diferentes.

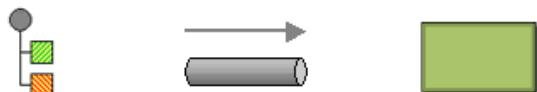
Um padrão é um *conceito abstrato*. Alguns padrões representam conceitos bem simples que não são representáveis apenas por código, diagramas UML ou componentes. Outros abstraem uma solução complexa que pode ser representada por toda a arquitetura que está sendo considerada. Por exemplo, o padrão *Mensagem-Evento* é implementado como uma mensagem qualquer, mas o padrão refere-se à *forma como a mensagem é usada* dentro de uma aplicação. O padrão *Entrega Garantida* refere-se a um *aspecto de configuração* que é aplicado a um sistema, e padrões complexos como *Barramento de Mensageria*, ou *Corretor de Mensagens* são microarquiteturas que podem ser usados para representar

soluções completas contendo vários outros padrões mais simples, APIs, frameworks e o próprio servidor usado.

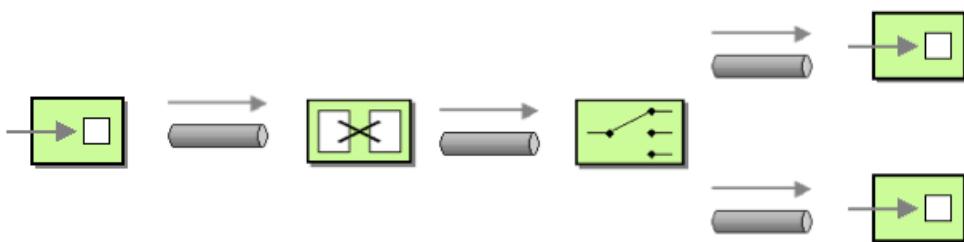
Não existe uma forma universal de apresentar padrões, mas em geral os catálogos que os descrevem destacam pelo menos o *nome, problema, solução, considerações e exemplos*.

O *nome* é um dos aspectos mais importantes de um padrão. Atribuir um nome a uma arquitetura complexa ou a um conceito muito abstrato permite a discussão em níveis mais elevados. Os nomes em uma linguagem de padrões formam um vocabulário para a descrição de soluções.

O catálogo de padrões de integração de sistemas [EIP] também inclui um *ícone* para representar a maior parte dos padrões, que permite a representação gráfica de soluções. Os ícones contém mais informação do que seria eficiente usando uma linguagem existente, como UML, e se tornaram um padrão adotado em ferramentas e frameworks. Os ícones, assim como os *diagramas* e *enunciados* de problemas e soluções foram liberados para uso (licença *Creative Commons*) pelos autores do catálogo [EIP]. Existem três tipos de ícones. Um para mensagens, outro para canais e outro para componentes (processadores) em geral:



Uma solução, portanto, pode ser descrita graficamente combinando vários ícones. O exemplo abaixo ilustra a integração de duas aplicações, usando os padrões *Terminal de Mensageria*, *Mensagem*, *Canal*, *Roteador*, *Tradutor*.



É uma solução de alto nível. A mesma solução poderia ser melhor detalhada, especificando o tipo de mensagem usada, tipos de canais, roteadores, etc.

O *enunciado do problema* em forma de pergunta curta permite avaliar se o padrão em questão é um candidato ao problema de integração que precisa ser solucionado. A *solução* descreve de uma maneira geral como o padrão soluciona o problema enunciado. Uma *descrição* muitas vezes é necessária, e ela geralmente contém um ou mais diagramas ou *esboço da solução*. O padrão pode ainda conter exemplos de *código*, e deve conter uma ou mais seções que discutam o *contexto* onde o problema e a solução são válidos, descrevendo, por exemplo, consequências, vantagens e desvantagens da adoção da solução, e outros padrões que são similares ou que estão de alguma forma *relacionados*.

Os padrões do catálogo [EIP] são organizados em uma estrutura de árvore, também adotada neste livro. Dos quatro padrões básicos, que representam estilos de integração, apenas o padrão (4) *Mensageria* é explorado. Ele agrupa outros seis padrões que abstraem aspectos da mensageria. Cada

um desses seis padrões-base são explorados em capítulos separados do catálogo [EIP]. Este livro os organiza de forma similar (mas no curso adotamos uma ordem diferente, com base na sua função.) No capítulo 3 haverá uma visão geral desssa estrutura.

## Resumo

Este capítulo introduziu o tema de integração de sistemas, algumas tecnologias Java usadas na integração e a importância dos padrões na construção de soluções de integração. A partir do próximo capítulo começaremos a explorar alguns padrões básicos de integração e veremos como implementá-los usando APIs e frameworks.

# Capítulo 2

# Integração

A integração de sistemas consiste em fazer aplicações diferentes trabalharem juntas para obter um conjunto coeso de funcionalidades. O objetivo é que funcionem de forma coordenada como se tivessem sido projetadas para isto, ou que operem como se fossem uma única aplicação distribuída. Como alcançar esse objetivo mantendo as aplicações independentes e fracamente acopladas?

A complexidade da integração depende de como as aplicações envolvidas interagem com o mundo externo. Se elas são fechadas, usam formatos e protocolos proprietários, rodam em dispositivos exclusivos desconectados da rede, pode ser muito difícil. No outro extremo estão as aplicações que oferecem uma API, que exportam e importam dados de formatos diversos e que oferecem muitos pontos de conexão. Mesmo que as aplicações tenham sido projetadas pensando em integração oferecendo APIs e usando padrões abertos, ainda existem desafios a serem enfrentados em uma integração. Fatores externos como a qualidade da rede, orquestração e sincronização de operações, transações, autorização, autenticação e segurança na transferência de dados são também questões que precisam ser levadas em conta.

Uma integração pode ser tão simples quanto uma aplicação produzir um arquivo e outra ler e processar esse arquivo. Mas o que acontece se uma das aplicações já estiver ocupada processando outros arquivos? E se ela não tiver permissão para ler o arquivo? Como ela vai saber que o arquivo foi criado, ou alterado? A transferência do arquivo ocorre pela rede? Qual o protocolo? As plataformas são diferentes? O que acontece se a rede cair quando o arquivo ainda não terminou de ser transferido? E o encoding?

Toda integração tem um custo, e esse custo pode não valer a pena. Nem sempre a automação de um processo é a melhor solução. Há também aplicações que não precisam ser integradas. Se uma aplicação não precisa colaborar com outra, você não precisa de integração.

Considerando que a integração seja necessária, o próximo passo é escolher um estilo ou estratégia para integração. A *transferência de arquivos*, mencionada acima, é uma opção. Outras opções são *banco de dados compartilhado* e a *comunicação em rede*, síncrona ou assíncrona. Pode ser que todas as opções sejam soluções viáveis. Nesse caso é preciso avaliar qual solução é a melhor dentro dos cenários em que a integração será usada.

Alguns critérios que devem ser levados em consideração ao se escolher uma estratégia de integração incluem o nível de acoplamento necessário, tecnologias necessárias, latência e tamanho dos dados, o que será compartilhado (dados e/ou funcionalidades), se a comunicação será remota, se a comunicação precisa ser confiável.

## Estilos de Integração

Quatro estilos de integração são classificados como *padrões* no catálogo [EIP]:

- (1) Transferência de arquivos (File Transfer)
- (2) Banco de dados compartilhados (Shared Database)
- (3) RMI / RPC (Remote Procedure Invocation)
- (4) Mensageria (Messaging)

Esses quatro padrões oferecem soluções diferentes para o mesmo problema: “como integrar aplicações”. Os contextos são similares, mas cada padrão propõe uma solução mais sofisticada (e mais complexa). Eles devem ser avaliados dentro do contexto do problema e levando em consideração os critérios e necessidades da solução. Estes padrões são o ponto de partida para se escolher uma solução de integração.

### (1) Transferência de arquivos (File Transfer)

#### Ícone



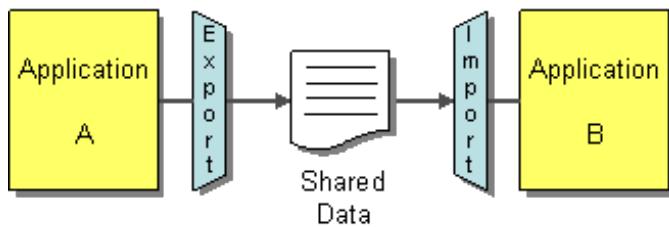
#### Problema

“Como integrar múltiplas aplicações para que possam trabalhar juntas e trocar informações?”

#### Solução

“Faça com que cada aplicação produza arquivos que contenham a informação que a outra aplicação precisa consumir. Integradores assumem a responsabilidade de transformar os arquivos em diferentes formatos. Produza os arquivos em intervalos regulares de acordo com a natureza do negócio.”

## Diagrama



## Descrição

O processo consiste de exportar os dados para um arquivo, gravar o arquivo, transferir o arquivo para outro lugar onde a segunda aplicação pode ler o arquivo e importar seus dados.

É fundamental na comunicação via transferência de arquivos a concordância de todas as aplicações integradas a respeito do *formato* de dados usado. Em geral deve haver um *esquema* que descreva a estrutura do arquivo e que possa ser usado por todos os participantes da solução de integração (ex: XML, JSON, XSD, etc.)

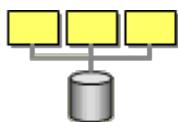
O arquivo está sendo usado efetivamente como uma *Mensagem*(6), ou seja, para *comunicação*, portanto a existência dele depois que a comunicação aconteceu é algo que precisa ser levado em consideração. Ele deve ser apagado? Ele é transferido para algum lugar específico (pasta, porta)? Como o receptor toma conhecimento da sua chegada? Em que momento ele pode considerado “consumido” pelo receptor da mensagem? Ele recebe alguma marcação? Como garantir que ele chegou intacto? Se for apagado, quem apaga o arquivo? O remetente? O receptor? O sistema? Como? Quando? O que fazer se ele estiver corrompido? Essas são algumas das questões que precisam ser tratadas nesse tipo de solução de integração.

Uma das vantagens desta solução é o baixo nível de acoplamento. Mas a solução por si só não garante que os dados estarão sempre atualizados, nem que a estrutura dos arquivos será sempre válida. É preciso implementar controles extras para isto, que poderá aumentar o custo da solução. Esta solução também não é adequada a ambientes distribuídos.

Embora limitada, às vezes uma solução como esta é suficiente para um problema de integração que não precise de uma solução mais complexa. Por exemplo, uma aplicação que grava um arquivo com uma contagem, que é lido por outra aplicação que mostra a contagem na tela. Se um ou outro arquivo for perdido pelo leitor, a contagem dará um salto mas não haverá maiores problemas com a aplicação. Esta solução começa a ficar complexa quando o arquivo precisar ser modificado concorrentemente por várias aplicações.

## (2) Banco de dados compartilhado (Shared Database)

### Ícone



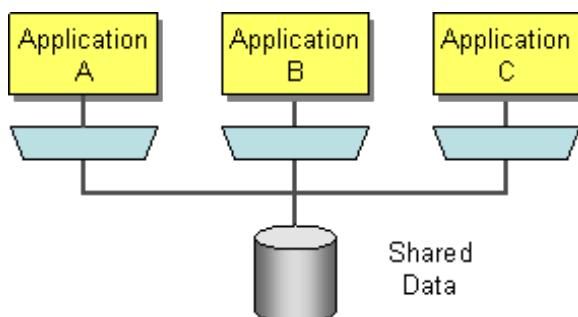
### Problema

“Como integrar múltiplas aplicações para que possam trabalhar juntas e trocar informações?”

### Solução

“Integre aplicações fazendo com que guardem seus dados em um único Banco de Dados Compartilhado (Shared Database), e defina o esquema do banco de dados para lidar com todas as necessidades das diferentes aplicações.”

### Diagrama



### Descrição

Uma das desvantagens da transferência de arquivos é sua latência, que pode causar a obsolescência dos dados. Outra questão é a dificuldade de controlar o formato dos arquivos. Para contornar os problemas podemos introduzir uma estrutura (ex: XSDs para documentos XML), incluir meios para garantir essa estrutura (validadores), lidar com acessos concorrentes, etc. Quando todos os problemas que surgirão para garantir essa estrutura forem solucionados, teremos praticamente criado um sistema de banco de dados.

Um banco de dados compartilhado garante uma estrutura comum, padronizada e obrigatória. Garante também atualização constante. Um banco de dados é projetado para lidar com acesso concorrente, níveis de isolamento, transações, etc. sendo uma solução mais completa quando esses atributos forem necessários. Pode-se usar um banco de dados relacional ou soluções alternativas como bancos de dados XML ou NoSQL.

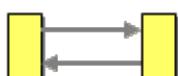
Uma das principais dificuldades desta solução é a elaboração de um design eficiente. Um esquema genérico costuma ser muito complexo, e se as necessidades mudam com frequência, é necessário ou mudar a estrutura do banco ou adicionar camadas extras que interceptam chamadas, o que pode ter impacto na eficiência. Outra questão é o custo do compartilhamento de dados, que é síncrono e

envolve controle transacional. Se um banco é compartilhado por uma grande quantidade de clientes para leitura e gravação, pode tornar-se um gargalo de performance crítico.

Assim como a transferência de arquivos, o foco desta solução é o compartilhamento de *dados* (embora os dados armazenados e compartilhados também possam ser usados para representar comandos e dispararem a execução de *funcionalidades* nas aplicações integradas.)

### (3) RPC (Remote Procedure Call)

#### Ícone



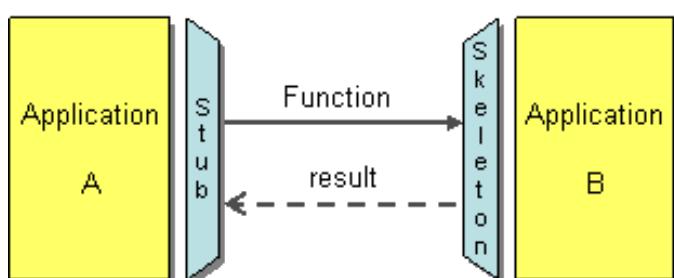
#### Problema

“Como integrar múltiplas aplicações para que possam trabalhar juntas e trocar informações?”

#### Solução

“Desenvolva cada aplicação como um objeto de larga escala ou componente com dados encapsulados. Forneça uma interface para permitir que outras aplicações interajam com a aplicação que está executando.”

#### Diagrama



#### Descrição

Um dos problemas com o banco de dados compartilhado é a falta de encapsulamento. As aplicações que compartilham os dados têm acesso a mais dados do que precisam. O nível de granularidade das permissões geralmente é insuficiente para que as aplicações tenham a quantidade necessária de acesso para realizar a integração e não efeitos colaterais indesejados. O fato de haver uma ampla interface aos dados também aumenta o acoplamento do sistema. Alterações na estrutura dos dados irão afetar todo o sistema.

O ideal é que os dados sejam privados, e que as operações de alteração dos dados sejam encapsuladas em funções que são chamadas remotamente. Usar uma Chamada de Procedimento Remoto (RPC – Remote Procedure Call, ou RMI - Remote Method Invocation) permite que os dados sejam acessados

apenas por aplicações locais, que *exportam suas interfaces* de maneira que clientes remotos poderão chamar funções, que serão executadas localmente. Assim, uma função pode interceptar o acesso aos dados controlando como, quando, onde e por quem os dados serão alterados. Essa estratégia também permite que *funcionalidades* que não envolvem dados também sejam compartilhadas. Por exemplo, operações *stateless* ou ainda operações que realizam a execução controlada de outras operações.

Diversas tecnologias implementam RPC/RMI. Exemplos incluem CORBA, COM/DCOM, .NET Remoting, Java RMI, Java RMI sobre IIOP, Web Services SOAP. Componentes como EJB Session Beans são baseados em comunicação síncrona e podem implementar RPC.

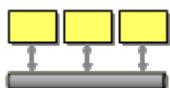
RPC é uma solução *síncrona* para compartilhamento de funcionalidades, ou seja, o cliente chama uma operação remota e espera ela terminar antes de continuar sua tarefa.

A forma como a comunicação síncrona é implementada difere quando o acesso é local ou remoto. Em ambientes Java, a comunicação entre dois componentes pode usar ou não RPC. Se o acesso é *local*, ou acontece a cópia de dados (passagem por valor) ou de referências (endereço dos dados). Se o acesso é *remoto*, a referência é um objeto, um stub, que precisa ser *serializado* e enviado pela rede. Em algumas soluções, como RMI e EJB, usa-se um proxy que garante que o código para acesso local ou remoto sejam idênticos, embora persistam diferenças na implementação.

Uma das principais desvantagens do RMI/RPC é o nível de acoplamento, que é forte. Os clientes precisam conhecer as interfaces de acesso, os tipos de dados dos parâmetros e retorno, os nomes das operações. Em ambientes CORBA ou WSDL, um cliente pode gerar um proxy de acesso dinamicamente, mas uma vez gerado seu acoplamento é forte. Se a interface do serviço for alterada, o cliente precisará gerar um novo proxy.

## (4) Mensageria (Messaging)

### Ícone



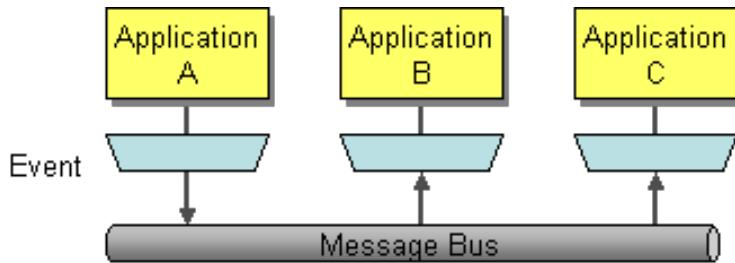
### Problema

“Como integrar múltiplas aplicações para que possam trabalhar juntas e trocar informações?”

### Solução

“Use Mensageria (Messaging) para transferir pacotes de dados com frequência, imediatamente, de forma confiável e sincronizada usando formatos customizados.”

## Diagrama



## Descrição

Com (1) *Transferência de Arquivos* é possível o compartilhamento de dados com baixo acoplamento, porém sem mecanismos para garantir que os dados serão sempre atualizados nem que as respostas aconteçam em tempo hábil. Um (2) *Banco de Dados Compartilhado* oferece meios para resolver essas e outras questões, mas acopla tudo ao banco de dados e não oferece encapsulamento adequado. (3) *RPC* permite que aplicações compartilhem funcionalidade, garante encapsulamento mas ao custo de um alto acoplamento. As interfaces precisam ser conhecidas e as aplicações que se comunicam precisam estar ativas no momento da comunicação. Uma solução que garante baixo acoplamento, notificação, confiabilidade da comunicação (mesmo que as aplicações não estejam ativas), com compartilhamento de dados e funcionalidades é (4) *Mensageria*.

Em relação aos outros estilos de comunicação, a Mensageria oferece diversas vantagens, por exemplo:

- Maior rapidez e menos latência que a transferência de arquivos;
- Maior encapsulamento que banco de dados compartilhado;
- Mais confiabilidade, mais escalabilidade e menos acoplamento que RPC.

O serviço de mensageria é proporcionado por um servidor que coordena a *mediação* da comunicação por mensagens. Esse servidor é às vezes chamado de *Message Queue* (MQ) ou *Message-Oriented Middleware*, ou MOM. Existem MOMs de vários fabricantes. Alguns dos mais populares incluem o JBossMQ / HornetMQ, ActiveMQ, IBM MQ, Spring AMQP. Um MOM fornece a infraestrutura básica para a mensageria (conexões e canais), e age como mediador na comunicação entre aplicações.

Um sistema de mensageria possui quatro componentes essenciais:

- A (6) *Mensagem (Message)*, que é basicamente um *envelope* que encapsula dados; contém um corpo onde os dados (*payload*) é transportado, e um *cabeçalho* que contém propriedades com informações sobre roteamento, qualidade do serviço, metadados, etc.
- Clientes, que podem ser *Produtores* (remetentes de mensagens) ou *Consumidores* (receptores de mensagens). Produtores enviam Mensagens para (5) *Canais*. Consumidores retiram Mensagens de Canais. As aplicações que são integradas via Mensageria conectam-se a Consumidores ou Produtores através de (10) *Terminais de Mensageria (Messaging Endpoint)*.

- (4) *Canais de Mensagens (Messaging Channel)*, que são endereços usados para comunicação. Canais são *destinos* compartilhados entre Clientes que desejam trocar Mensagens.
- O *Servidor de Mensageria (MOM)*, que fornece os Canais e realiza a mediação da comunicação entre Clientes.

Canais e conexões são tipicamente criados, configurados e instanciados pelas ferramentas administrativas do servidor de mensageria usado. Aplicações geralmente *localizam* conexões e canais, depois usam e reusam durante a operação. O acesso aos serviços de um servidor de mensageria pode ser feito através de suas ferramentas proprietárias, ou através de APIs como o Java Message Service (JMS) que se comunicam com o servidor através de um provedor de serviços (driver),

O MOM pode ser um serviço distribuído que cuida da transferência das mensagens em rede. As responsabilidades das aplicações em relação ao MOM são apenas duas: 1) enviar mensagens *para* os canais, e 2) ler mensagens *dos* canais. O MOM cuida da entrega e da qualidade do serviço, que pode ser configurado no servidor. O MOM nada sabe sobre o conteúdo da mensagem, fica sob a responsabilidade das aplicações que compartilham dados.

Nem sempre mensageria é a melhor solução de integração. Existem cenários nos quais a comunicação precisa ser síncrona. Em outros, o custo de reordenar mensagens que chegam fora de ordem pode ser excessivo. Em alguns casos, o ideal pode ser escolher outro estilo de integração, mas muitas vezes os benefícios da mensageria podem superar os custos.

Algumas dificuldades encontradas em sistemas de mensageria incluem:

- *Depuração complexa*: é bem mais difícil depurar uma aplicação não sequencial, assíncrona e com um fluxo de controle determinado por eventos. O catálogo [EIP] apresenta diversos padrões voltados à monitoração, testes e depuração buscam diminuir essa dificuldade.
- *Não há garantia de entrega*: pode-se garantir que uma mensagem será entregue, mas não se sabe quando. É comum mensagens chegarem fora de ordem. Alguns padrões lidam com essas questões incluem (16) *Entrega Garantida* e (35) *Resequenciador*.
- *Comunicação assíncrona*: que é um dos principais motivos para se usar a mensageria, pode ser um problema se algumas aplicações que serão integradas precisarem de uma comunicação síncrona. Padrões como (23) *Requisição-Resposta* e (57) *Ativador de Serviço* fazem uma ponte entre os domínios síncrono e assíncrono, diminuindo essa desvantagem.
- *Dados grandes*: a mensageria pode ser ineficiente para transferir vários dados de uma só vez, porque pode ser necessário repartir os dados em pequenas partes gerando mais overhead (necessidade de replicar cabeçalhos reordenar as mensagens, etc.)

## Resumo

Quatro estilos usados para realizar integração entre aplicações:

- (1) Transferência de arquivos (File Transfer)
- (2) Banco de dados compartilhados (Shared Database)
- (3) RMI / RPC (Remote Procedure Invocation)
- (4) Mensageria (Messaging)

# Capítulo 3

# Mensageria

Esta seção trata dos conceitos fundamentais da mensageria: a *mensagem*, que contém os dados a serem transmitidos; o *canal*, usado pelos componentes para enviar e receber mensagens; e os *terminais* (endpoints) que funcionam como entradas e saídas de dados. Explora também a *arquitetura* usada para conectar esses componentes, além de componentes especiais para *roteamento* de mensagens e *transformação* de dados.

Existem três componentes fundamentais em qualquer sistema de mensageria:

- *Mensagem* (Message)
- *Canal* de mensageria (Message Channel)
- *Terminal* de mensageria (Messaging Endpoint)

O transporte de dados é realizado pela *mensagem*, uma estrutura de dados que age como um envelope encapsulando os dados em um componente padrão que pode ser transportado pelo sistema.

O *canal* é o meio onde a mensagem é transportada. É um destino virtual que une dois terminais, permitindo que troquem mensagens entre si.

Os *terminais* podem ser de três tipos: produtores, consumidores ou *filtros*. Produtores enviam mensagens para um canal, consumidores retiram imagens do canal e filtros têm um ou mais canais de entrada, e um ou mais canais de saída.

O termo *filtro*, nesse contexto, não representa um componente que necessariamente filtra dados, mas um componente qualquer que possui entrada e saída (na prática, pode ser implementado combinando um consumidor e um produtor.) O nome filtro é usado por causa do padrão *Dutos e Filtros (Pipes and Filters)*, que descreve a arquitetura usada na construção de soluções de integração. Um filtro é um componente inserido no meio de uma rota que pode processar a mensagem recebida ou simplesmente redirecioná-la sem alterações. O catálogo EIP, usado como principal referência para este texto, assim descreve dois tipos de componentes que têm canais na entrada e na saída:

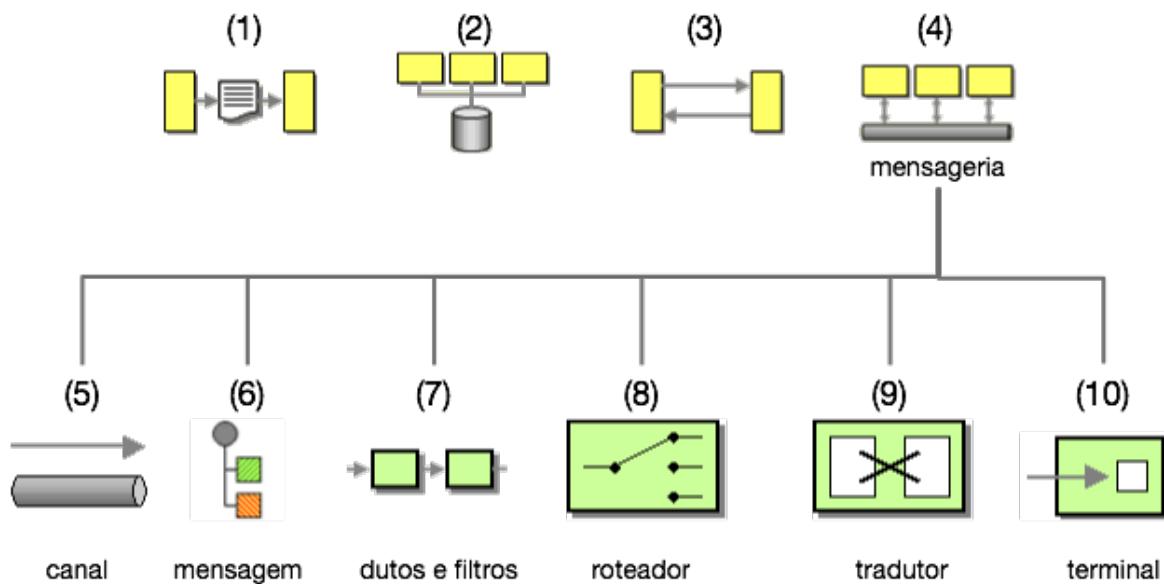
- *Roteador* de mensagens (Message Router)

- *Tradutor* de mensagens (Message Translator)

Um *roteador* é um componente que altera ou define a rota por onde uma mensagem irá circular. Pode filtrar mensagens indesejadas, distribuir mensagens em canais diferentes, partir uma mensagem em partes menores, agregar mensagens em uma única mensagem, etc.

Um *tradutor* não altera rotas, mas pode processar o conteúdo de uma mensagem transformando seus dados, mudar seu formato, acrescentar ou remover informações.

Esses conceitos: mensagem, canal, terminal, roteador, tradutor e arquitetura dutos e filtros são classificados no catálogo EIP como *padrões raiz de Mensageria* (4) ilustrados abaixo:



Dos 55 padrões restantes, 47 padrões estão relacionados a cada um dos padrões raiz. Neste capítulo estudaremos os três padrões fundamentais: (6) *Mensagem*, (5) *Canal* e (10) *Terminal*, e faremos uma breve introdução à (7) *Arquitetura de Dutos e Filtros*, (8) *Roteadores* e (9) *Tradutores*.

## (5) Canal de Mensagens (Message Channel)

### Ícone



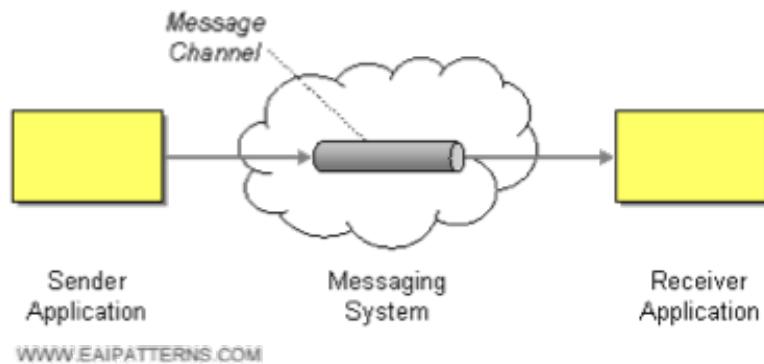
### Problema

“Como pode uma aplicação comunicar-se com outra aplicação usando mensageria?”

## Solução

“Conectar as aplicações usando um *Canal de Mensagens (Message Channel)*, onde uma aplicação grava informação no canal e a outra lê informação do canal.”

## Diagrama



## Descrição

A comunicação entre aplicações em um sistema de mensageria é realizada através de *canais*, destinos virtuais que recebem mensagens de um ou mais remetentes e as disponibiliza para um ou mais destinatários. Sistemas de mensageria permitem a criação e configuração de diferentes tipos de canais.

É necessário criar os canais *antes* que seja possível usar um sistema de mensageria porque eles devem ser *conhecidos* pelas aplicações e componentes que irão se comunicar através deles. Tipicamente canais são criados através de ferramentas proprietárias do sistema, ou, no caso de frameworks, configurados previamente antes que a aplicação seja iniciada. Promovem assim o baixo acoplamento ao isolarm remetentes e destinatários, que podem se comunicar sem se conhecerem.

Canais são *destinos virtuais*. Os detalhes de como são implementados não é relevante para o uso do sistema. O mais importante é como são *referenciados* pelos terminais que produzem ou consomem mensagens. Dependendo do sistema, terão um *identificador*, um *nome*, uma *referência*. Em *Spring Integration* canais recebem um nome (*id*) que é usado como referência pelos componentes que querem se comunicar através deles. Em JMS, um canal é injetado usando DI (*Dependency Injection*) ou localizado usando JNDI (*Java Naming and Directory Interface*) e associado a uma instância de objeto, usado pelos componentes. *Apache Camel* referencia canais em JMS, sistemas de arquivos e outros destinos através de URIs.

Um canal não transmite qualquer tipo de dados, apenas dados reconhecidos pelo sistema de mensageria como *mensagens*. Uma mensagem possui um formato específico do sistema de mensageria usado. Portanto, para que dados sejam enviados para um canal, é preciso antes encapsulá-los em uma (6) *Mensagem*.

É preciso ter pelo menos um canal para que haja comunicação. Uma típica solução de integração usa diversos canais. Às vezes uma mesma solução pode ser implementada com diferentes quantidades de canais, com impactos de performance, acoplamento e outros *trade-offs*.

A comunicação usando canais é geralmente de dois tipos: *ponto a ponto* (um remetente envia, um destinatário consome) ou *difusão* (um remetente envia, vários destinatários consomem). Esses dois domínios de comunicação são representados por dois padrões relacionados a canais: (11) *Canal Ponto-a-Ponto (Point-to-Point Channel)* e (12) *Canal de Difusão (Publish-Subscribe Channel)*.

Alguns padrões são relacionados ao tipo de mensagem recebida por um canal:

- (13) *Canal de Tipo-de-Dados (Datatype Channel)*: um canal que permite apenas mensagens de um determinado tipo.
- (14) *Canal de Mensagens Inválidas (Invalid Message Channel)*: canal usado como repositório de mensagens que não puderam ser processadas por não serem válidas.
- (15) *Canal de Mensagens Não-Entregues (Dead-Letter Channel)*: canal para onde são redirecionadas mensagens que não puderam ser enviadas.

O padrão (17) *Adaptador de Canal (Channel Adapter)* é uma combinação de Terminal + Adaptador + Canal permitindo a interface com o mundo externo. Por exemplo, um Adaptador de Canal que lê dados do sistema de arquivos poderia ser usado para encapsular arquivos em uma mensagem e publicá-la em um canal. Este padrão foi classificado no catálogo EIP como um padrão de (5) Canal, pois contém um canal, mas também poderia ser classificado como um padrão de (10) *Terminal de Mensageria*, já que também contém um terminal.

Outros padrões descrevem arquiteturas usando canais: (18) *Ponte de Mensageria (Messaging Bridge)* é uma coleção integrada de *Adaptadores de Canais* que permite a integração de sistemas de mensageria diferentes, permitindo que compartilhem mensagens, canais e outros componentes, mas pode também representar uma única rota que realiza a comunicação entre dois sistemas. (19) *Barramento de Mensageria (Messaging Bus)* descreve uma solução completa de integração centrada em um barramento capaz de distribuir mensagens entre remetentes e destinatários.

## Aplicações

Os canais são *mediadores* na comunicação entre componentes de um sistema de mensageria.

### Canal de Mensageria em Java (JMS)

Em JMS, o padrão *Message Channel* é representado pela interface *javax.jms.Destination*. Um *Destination* deve ser instanciado e configurado através das ferramentas do servidor de mensageria e disponibilizado como um serviço localizável ou injetável. Uma vez obtida uma instância local, ele é usado para inicializar um *MessageProducer* ou *MessageConsumer* que pode, respectivamente, enviar e receber mensagens neste canal.

O trecho de código abaixo mostra como usar JMS para acessar um destino tipo *Queue* registrado em JNDI com o nome “*simple-p2p-channel*” e enviar uma *Mensagem* simples contendo uma linha de texto para ele:

```
01. Context ctx = new InitialContext();
```

```

02. ConnectionFactory factory =
        (ConnectionFactory) ctx.lookup("ConnectionFactory");
03. Destination destination = (Queue) ctx.lookup("simple-p2p-channel");
04. Connection con = factory.createConnection();
05. con.start();
06. Session session = con.createSession(false, Session.AUTO_ACKNOWLEDGE);
07. MessageProducer producer = session.createProducer(destination);
08. TextMessage message = session.createTextMessage("Hello World!");
09. producer.send(message);

```

## Canal de Mensageria em Apache Camel

Camel não implementa explicitamente o padrão (5) *Canal de Mensageria*, mas apenas provê uma interface para acessá-los através de objetos chamados de *Endpoints* (*org.apache.Camel.Endpoint*) e *Componentes* (*org.apache.camel.Component*). Um canal pode ser fornecido por uma infraestrutura de mensageria externa (ex: *JMSEndpoint*), pelo sistema operacional (ex: *FileEndpoint*), ou ainda através de um componente nativo (ex: *Direct*, *SEDA*, *VM*) e acessado através de uma URI (ex: *direct:canal1*, *jms:queue:canal2*, *file:///tmp*). A URI identifica o endpoint e o componente que fornece o canal, que pode ser usado no Camel para enviar ou receber mensagens.

A construção de uma rota em Camel parte de um canal e termina em outro. Os canais, informados através de suas URIs, podem ser incluídos na configuração da rota através dos métodos *to()* e *from()*, de um *RouteBuilder* que fornece uma DSL (*Domain-Specific Language*) para Java. A rota também pode ser expressa usando outros DSLs, como XML (usando tags *<to>* e *<from>* em arquivo de contexto *Spring*).

O trecho de código abaixo pode ser usado para consumir as mensagens enviadas à fila chamada no ambiente JMS de “test-queue” e imprimir o conteúdo da mensagem na saída padrão (o código completo e executável deste e de outros exemplos está disponível no repositório GitHub):

```

01. CamelContext context = new DefaultCamelContext();
02. context.addRoutes(new RouteBuilder() {
03.     @Override
04.     public void configure() throws Exception {
05.         from("jms:queue:test-queue")
06.             .to("stream:out");
07.     }
08. });
09. context.start();

```

## Exemplo em Spring Integration

*Spring Integration* implementa canais de Mensageria através da interface *MessageChannel* (*org.springframework.integration.MessageChannel*) que declara o comportamento de envio de mensagens. O *recebimento* de uma mensagem pode ser *síncrono* ou *assíncrono*, e isto é definido por duas outras interfaces: *PollableChannel* (canal que precisa ser sondado por mensagens) e *SubscribableChannel* (canal no qual pode-se subscrever para ser notificado quando uma mensagem

chegar). As interfaces são implementadas em vários diferentes tipos de canais, normalmente configurados no estilo Spring através de beans em tags XML aos quais são mapeados.

No XML de integração usa-se o tag `<channel>` para representar um canal. Ele é implementado pelo próprio Spring e não requer nenhum sistema externo de mensageria. Existem várias configurações possíveis para canais de mensagens em *Spring Integration*. Cada configuração pode representar uma implementação diferente. O default para o tag `<channel>` é representar um *DirectChannel*, que é uma implementação de *SubscribableChannel* com semântica *ponto-a-ponto*. Ele funciona de forma similar ao componente *direct*: do *Camel*.

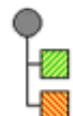
Para integrar com canais JMS é preciso usar um *adaptador* ou uma implementação de canal JMS fornecida pelo componente *Spring Integration JMS* (que possui um namespace próprio).

O exemplo abaixo possui dois (10) *Terminais* e um (5) *Canal*. O *Canal* é local ao Spring e identificado com o ID “*jms-example*”, referenciado pelos dois componentes através do atributo *channel*. Um dos Terminais está conectado ao destino JMS “*test-queue*” e configurado através do elemento `<message-driven-channel-adapter>`, que enviará mensagem para o Canal “*jms-example*” sempre que uma mensagem chegar no destino JMS “*test-queue*”. Consumindo as mensagens do canal “*jms-example*” está outro (10) *Terminal*, configurado como `<stdout-channel-adapter>`, que redireciona as mensagens recebidas à saída padrão. Esta configuração, portanto, faz o mesmo que a rota implementada usando *Camel* acima.

```
<int:channel id="jms-example"/>
<int-jms:message-driven-channel-adapter
    destination-name="test-queue"
    channel="jms-example"/>
<int-stream:stdout-channel-adapter channel="jms-example"/>
```

## (6) Mensagem (Message)

### Ícone



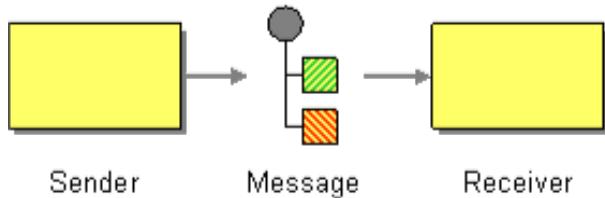
### Problema

“Como é que duas aplicações conectadas por um canal de mensagens podem trocar informação?”

### Solução

“Empacote a informação dentro de uma *Mensagem (Message)*, um registro de dados que o sistema de mensageria pode transmitir através de um canal de mensagens.”

## Diagrama



## Descrição

Uma mensagem permite que duas aplicações conectadas por um canal troquem informações. As informações são encapsuladas na mensagem, que é um pacote de dados que tem um formato compatível com o sistema de mensageria. Para que dados possam ser transmitidos em um canal, é preciso encapsulá-los em uma ou mais mensagens. Depois, quando as mensagens forem recebidas, o seu conteúdo deve ser extraído para que os dados possam ser recuperados. Os dados encapsulados por uma mensagem podem ser de qualquer tipo.

Mensagens possuem duas partes essenciais:

- *Cabeçalho* (propriedades do sistema, propriedades da aplicação, meta-informação, dados)
- *Corpo* (dados e anexos)

O *cabeçalho* possui um formato reconhecido pelo sistema de mensageria (geralmente propriedades no formato *nome: valor*). Podem conter informação, meta-informação (sobre o conteúdo), propriedades usadas pela aplicação para roteamento, transformação e outros serviços, além de propriedades do sistema usados no endereçamento e serviços padrão.

O *corpo* contém dados que são ignorados pelo sistema de mensageria, mas que geralmente são transformados em um formato compatível com o canal (*marshalling*) para a transmissão, e depois decifrados (*unmarshalling*) quando a mensagem for processada.

O *conteúdo* da mensagem é ignorado pelo sistema de mensageria, que funciona de forma análoga ao serviço postal. O envelope adere ao padrão dos correios, e contém cabeçalhos com propriedades do sistema (destinatário, remetente, endereço de entrega, CEP, declaração de conteúdo) e possivelmente cabeçalhos adicionais que não são usados pelos correios, mas têm utilidade em outros sistemas (departamento, sala, telefone, etc.) O conteúdo é ignorado, mas às vezes precisa ser declarado, ou ser transportado em mensagens separadas se extrapolar os limites permitidos (ex: livros transportados em várias caixas).

Uma mensagem pode ter diferentes finalidades. Alguns padrões descrevem esses cenários.

- Se ela encapsula um comando que deve ser executado remotamente, é uma (20) *Mensagem-Comando* (*Command Message*).
- Mensagens cuja principal função é transportar informação, sem nenhuma função especial, são (21) *Mensagens-Documento* (*Document Message*).

- Algumas mensagens servem apenas para notificação, e seu conteúdo é menos importante que o evento do seu recebimento. São (22) *Mensagens-Evento (Event Message)*.
- Às vezes, quando uma mensagem é enviada, espera-se uma resposta. Pode ser um aviso de recebimento (evento), ou o resultado da execução de um comando (documento). Esse cenário é representado pelo padrão (23) *Requisição-Resposta (Request-Reply)*. O endereço para onde a resposta é enviada é representado pelo padrão (24) *Endereço de Resposta (Return Address)*.

Outros padrões descrevem cabeçalhos e arquiteturas relacionadas à construção de mensagens.

- (25) *Identificador de Correlação (Correlation Identifier)* é usado para identificar um conjunto de mensagens que se relacionam de alguma forma.
- (26) *Sequência de Mensagens (Message Sequence)* descreve como enviar e receber dados que foram fragmentados através de diversas mensagens.
- (27) *Prazo de Validade (Message Expiration)* inclui um cabeçalho ou mecanismo para indicar mensagens vencidas, que não devem mais ser enviadas.
- (28) *Indicador de Formato (Format Indicator)* descreve o conteúdo da mensagem, que pode ser usado para validar ou rotear a mensagem.

## Aplicações

Uma mensagem é essencial para encapsular informações que precisam ser transmitidas através de um canal de mensageria.

### Mensagem em Java (JMS)

O padrão Mensagem é representado em Java através da interface `javax.jms.Message`. É a interface raiz para várias mensagens mais especializadas. `Message` não possui corpo, apenas propriedades e cabeçalhos. As versões especializadas fornecem um corpo que pode ser texto (`TextMessage`), stream (`StreamMessage`), `java.util.Map` (`MapMessage`), objeto serializado (`ObjectMessage`) ou bytes (`BytesMessage`).

Uma mensagem em JMS possui três partes:

- *Cabeçalhos (Header)* – são propriedades (pares nome/valor) reservadas pelo sistema usadas pelo sistema de mensageria para roteamento e identificação de mensagens. Existem métodos em `Message` e sub-interfaces para ler (e às vezes gravar) valores dos cabeçalhos (ex: `getMessageID()`). Cabeçalhos começam com as três letras “`JMS`”, por exemplo `“JMSCorrelationID”`, `“JMSType”`, `“JMSSessageID”`.
- *Propriedades* – são pares nome/valor onde o nome é um `String` e o valor pode ser um objeto ou tipo primitivo. Um produtor pode armazenar dados em propriedades arbitrariamente definidas ao criar ou editar uma mensagem. Essas propriedades podem ser lidas pelos receptores das mensagens e normalmente são usadas como metadados quando a mensagem tem um corpo. Uma mensagem muito simples sem corpo pode usar apenas propriedades para

transferir dados. Propriedades reservadas pelo sistema começam com JMSX, mas não possuem métodos específicos para leitura e gravação (o método `getJMSXPropertyNames()` de `ConnectionMetaData` pode ser usado para descobrir as propriedades reservadas usadas em uma conexão.)

- *Corpo* – presente apenas nas subinterfaces de `Message`. O corpo depende do tipo da mensagem e os métodos para gravar e recuperar os dados refletem essas propriedades. Por exemplo, um `TextMessage` possui métodos `getText()` e `setText()` para ler e gravar o corpo da mensagem que contém texto.

Uma *Mensagem* pode ser criada automaticamente e enviada a partir de um *contexto* em JMS 2.0 (o contexto cria um produtor cujo método `send()` recebe o conteúdo da mensagem). Mas o suporte a JMS 2.0 ainda é raro entre os principais fabricantes de MOMs.

Em JMS 1.1 é preciso usar uma *sessão*. Uma vez criada ela pode ser enviada por *Produtores* (`MessageProducer`) através do seu método `send()`:

```
TextMessage message = session.createTextMessage("Hello World!");
message.setStringProperty("category", "greeting");
message.setStringProperty("content-type", "text/plain");
producer.send(message);
```

Mensagens são retornadas pelo método `receive()` de `MessageConsumer`, que implementa acesso *síncrono* como (51) *Consumidor de Sondagem (Polling Consumer)* ou recebidas como *notificação* no método `onMessage()` da interface `MessageListener`, que implementa acesso *assíncrono* como (50) *Consumidor Ativado por Eventos (Event Driven Consumer)*:

```
01.  public class AsyncConsumer implements MessageListener {
02.      @Override
03.      public void onMessage(Message message) {
04.          TextMessage tm = (TextMessage)message;
05.          String category = tm.getStringProperty("category");
06.          String contents = tm.getText();
07.          ...
08.      }
09.  }
```

## Mensagem em Apache Camel

Camel implementa este padrão através de duas interfaces: `org.apache.camel.Message` e `org.apache.camel.Exchange`. Um ou mais objetos `Message` podem ser obtidos a partir de um `Exchange`. Se `Exchange` representar um *par requisição-resposta* ele contém dois objetos `Message` (cada um deles obtidos pelos métodos `getIn()` e `getOut()`), caso contrário contém apenas um (método `getIn()`).

Existem várias formas de criar um `Exchange`. O exemplo abaixo ilustra como fazer isto usando um `DefaultExchange`:

```
01.  CamelContext context = new DefaultCamelContext();
```

```

02. Exchange exchange = new DefaultExchange(context);
03. Message message = exchange.getIn();
04. message.setBody("Hello World!");
05. message.setHeader("category", "greeting");
06. message.setHeader("content-type", "text/plain");

```

Vários componentes recebem mensagens. Um processador que imprime um log de mensagens que passam por ele poderia imprimir os dados da mensagem da seguinte forma:

```

01. public class LoggingProcessor implements Processor {
02.     @Override
03.     public void process(Exchange exchange) throws Exception {
04.         System.out.println("Category: "
05.             + exchange.getIn().getHeader("category"));
06.         System.out.println("Contents: "
07.             + exchange.getIn().getBody());

```

## Mensagem em Spring Integration

Spring Integration representa o padrão Mensagem através da interface *org.springframework.messaging.Message*, que é um empacotador genérico para qualquer objeto Java. Uma Message consiste de duas partes: *corpo* (payload) e *cabeçalho* (header).

Um *MessageBuilder* é usado para criar mensagens através de uma API fluente. Por exemplo, para criar uma mensagem contendo um texto simples e alguns cabeçalhos, pode-se usar:

```

01. Message<String> message =
02.     MessageBuilder.withPayload("Hello World")
03.         .setHeader("category", "greeting")
04.         .setHeader("content-type", "text/plain")
05.         .build();

```

A mensagem pode ser recebida por componentes para processamento. Por exemplo, em um tradutor de mensagens o método *transform()* recebe uma mensagem para transformar:

```

01. public Message transform(Message message) {
02.     String text = message.getPayload().toString();
03.     String header = (String)message.getHeaders().get("category");
04.     ...
05. }

```

## (7) Dutos e filtros (Pipes and Filters)

### Ícone



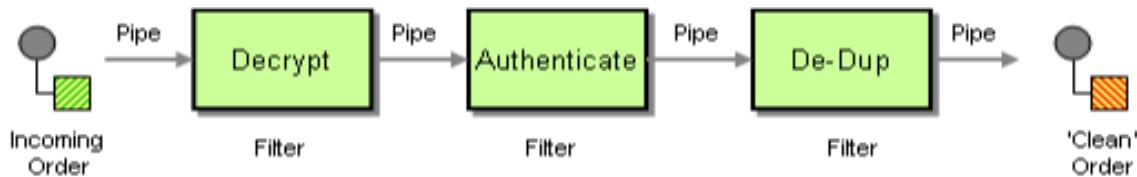
### Problema

“Como realizar processamento complexo em uma mensagem mantendo independência e flexibilidade?”

### Solução

“Use o estilo arquitetônico *Dutos e Filtros (Pipes and Filters)* para dividir uma tarefa de processamento maior em uma sequência de passos menores e independentes (*Filtros*) que são conectados por canais (*Dutos*).”

### Diagrama



### Descrição

*Dutos e Filtros (Pipes and Filters)* é um padrão de arquitetura. É descrito no catálogo *Pattern Oriented Software Architecture, Vol. 1*, de Frank Buschmann et al. [POSA], e tem semelhanças com *Intercepting Filter* (Padrões J2EE, Alur et al.) [J2EE] e *Decorator* (Padrões de Design, Gamma et al.) [GoF]. Basicamente descreve um sistema no qual componentes que filtram, aumentam ou transformam dados são conectados em série, dividindo um processamento complexo em etapas diferentes.

*Dutos e Filtros*, no catálogo EIP, difere um pouco desses outros padrões pois inclui também tarefas de roteamento de mensagens, que não transformam os dados e permitem processamento paralelo. Nesse contexto, o filtro é um componente intermediário, como um roteador ou tradutor, e o duto é um canal.

Usando a arquitetura *Dutos e Filtros* pode-se desenhar rotas de integração com etapas de roteamento e transformação, distribuindo responsabilidades e facilitando a criação de componentes, que poderão ser reusados. Uma possível desvantagem é o uso maior de canais, que pode requerer transformação da mensagem (*marshalling* e *unmarshalling*) em cada componente.

O padrão Dutos e Filtros suporta processamento paralelo em estilo *pipeline* (componentes conectados em série e executados em paralelo). Como todo processamento é assíncrono, assim que um filtro termina de processar uma mensagem, ele já pode receber outra. Nessa arquitetura, *Consumidores Concorrentes (Competing Customers)* podem ser usados para consumir as mensagens que são lançadas

em um canal assim que elas estiverem disponíveis. Rotas paralelas que depois são reunidas usando um *Agregador (Aggregator)* podem fazer com que as mensagens cheguem fora de ordem. Se a ordem for importante, pode-se usar um *Re-sequenciador (Resequencer)* para reordená-las.

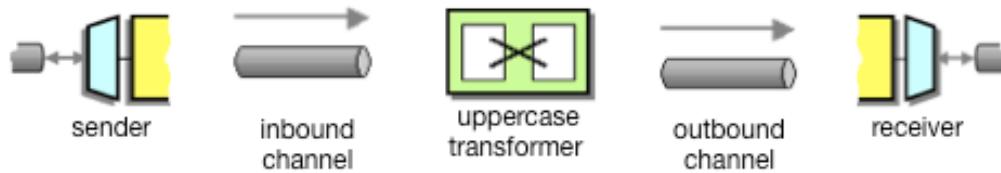
## Aplicações

*Dutos e Filtros* é um padrão de arquitetura que permite que tarefas em uma rota de integração sejam distribuídas para componentes dedicados, interligados por canais, permitindo maior reuso, menor acoplamento e maior flexibilidade. É usado em soluções utilizando padrões de mensageria, e na construção de padrões compostos.

### Dutos e Filtros em Java (JMS)

Não há uma implementação de Dutos e Filtros em JMS. O padrão deve ser construído interligando produtores (*javax.jms.MessageProducer*) e consumidores (*javax.jms.MessageConsumer*) através de canais (*javax.jms.Destination*). Cada produtor e consumidor constrói um objeto de sessão ou de contexto que está associado a um canal específico. A rota formada por um ou mais canais aparecendo como destinos de produtores e consumidores representa uma arquitetura de Dutos e Filtros.

Considere a seguinte rota que utiliza arquitetura Dutos e Filtros:



Para implementar essa arquitetura em JMS, precisamos de uma classe para criar e executar o produtor, uma classe para criar e executar o receptor, dois canais configurados no MOM e uma classe para ler a mensagem, processá-la e enviá-la para outro canal. O exemplo abaixo ilustra os dois primeiros componentes da rota (*sender* e *inbound-channel*):

```

01.     public static void main(String[] args) throws Exception {
02.         Context ctx = new InitialContext();
03.         ConnectionFactory factory = ...
04.         Connection con = ...
05.         Session session = ...
06.         Queue queue = (Queue) ctx.lookup("inbound-channel");
07.         MessageProducer sender = session.createProducer(queue);
08.         TextMessage message = session.createTextMessage("Hello World!");
09.         sender.send(message);
10.     }
  
```

A segunda parte da rota envolve dois threads: um para o *uppercase-transformer*, que consome as mensagens da fila *inbound-channel*, para onde o *sender* envia mensagens; e outro thread para o *receiver*, que consome as mensagens da fila *outbound-channel*, para onde o *uppercase-transformer* envia as mensagens transformadas:

```

01. public static void main(String[] args) throws Exception {
02.         Context ctx = new InitialContext();
03.         ConnectionFactory factory = ...;
04.         Connection con = factory.createConnection();
05.         final Session trSession = con.createSession(...);
06.         final Session receiverSession = con.createSession(...);
07.         final Queue inQueue = (Queue) ctx.lookup("inbound-channel");
08.         final Queue outQueue = (Queue) ctx.lookup("outbound-channel");
09.
10.        MessageConsumer trIn = trSession.createConsumer(inQueue);
11.        trIn.setMessageListener(new MessageListener() {
12.            public void onMessage(Message message) { // recebe msg de inQueue
13.                try {
14.                    TextMessage inMessage = (TextMessage) message;
15.                    String trContents = inMessage.getText().toUpperCase();
16.                    TextMessage out = trSession.createTextMessage(trContents);
17.                    MessageProducer trOut = trSession.createProducer(outQueue);
18.                    trOut.send(out); // envia mensagem para outQueue
19.                } catch (JMSEException e) {
20.                    e.printStackTrace();
21.                }
22.            }
23.        });
24.
25.        MessageConsumer receiver = receiverSession.createConsumer(outQueue);
26.        receiver.setMessageListener(new MessageListener() {
27.            public void onMessage(Message message) { // recebe msg de outQueue
28.                TextMessage finalMessage = (TextMessage) message;
29.                System.out.println("Received message: " + finalMessage);
30.            }
31.        });
32.
33.        con.start();
34.    }

```

Execute o *ReceiverExample* primeiro. Os threads ficarão esperando que mensagens cheguem às suas filas. Quando o *SenderExample* executar, ele enviará uma mensagem para o *inbound-channel*, que será lida pelo *TranslatorExample*, que fará a transformação e enviará a mensagem transformada para o *outbound-channel*, que é lida pelo *ReceiverExample* que imprimirá o resultado na saída:

```
35. HELLO WORLD!
```

Um cenário típico da arquitetura *Dutos e Filtros* consiste de rotas mais complexas, que são muito mais fáceis de implementar usando um framework como *Camel* ou *Spring Integration*.

## Dutos e Filtros em Apache Camel

O padrão *Dutos e Filtros* em *Camel* pode ser implementado através de vários trechos *from()*-*to()* sucessivos, cada um usando como *Endpoint* *from()* de partida, o *Endpoint* *to()* de destino do trecho

anterior. Em *Camel* isto é uma *rota*. No exemplo abaixo (no contexto do método *configure()* de um *RouteBuilder*) a mensagem foi enviada para o “*inbound-queue*”

```

01. // Trecho 1 Sender -> inbound-channel
02. ProducerTemplate template = context.createProducerTemplate
03. template.sendBody("jms:queue:inbound-queue"), "Hello World!";
04.
05. // Trecho 2 inbound-channel -> uppercase-transformer -> outbound-channel
06. from("jms:queue:inbound-queue").process(new Processor() {
07.     public void process(Exchange exchange) {
08.         Message in = exchange.getIn();
09.         in.setBody(in.getBody(String.class).toUpperCase());
10.    }
11. }).to("jms:queue:outbound-queue");
12.
13. // Trecho 3 - outbound-channel -> Receiver
14. from("jms:queue:outbound-queue").to("stream:out");

```

A construção típica de rotas em *Camel* nem sempre é uma implementação rigorosa de *Dutos e Filtros*, pois os componentes podem ser conectados diretamente, sem canais intermediários:

```

01. from("jms:queue:inbound-queue").process(new Processor() {
02.     public void process(Exchange exchange) {
03.         Message in = exchange.getIn();
04.         in.setBody(in.getBody(String.class).toUpperCase());
05.    }
06. }).to("stream:out");

```

## Dutos e Filtros em Spring Integration

Para *Spring Integration* qualquer componente que se comunica com um canal, seja ele um roteador, tradutor ou terminal é um *Filtro*, e qualquer canal é um *Duto*. *Spring Integration* chama todos esses componentes de *Endpoints* (mesmo os que não se comunicam com o mundo externo), portanto a arquitetura Dutos e Filtros é automaticamente realizada ao conectar componentes a canais.

```

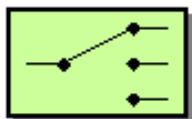
01. <int:channel id="inbound-channel" />
02. <int:transformer input-channel="inbound-channel"
03.                 output-channel="outbound-channel">
04.     <bean class="br.com.argonavis.si.examples.UppercaseTransformer" />
05. </int:transformer>
06. <int:channel id="outbound-channel" />
07. <int-stream:stdout-channel-adapter channel="outbound-channel"/>

```

O exemplo anterior não usa filas JMS, mas canais nativos do Spring. Para que esse exemplo interaja com os mesmos canais JMS que os exemplos anteriores, seria necessário usar Adaptadores de Canais para JMS na entrada e saída, mapeando os canais do Spring aos destinos JMS correspondentes.

## (8) Roteador de mensagens (Message Router)

### Ícone



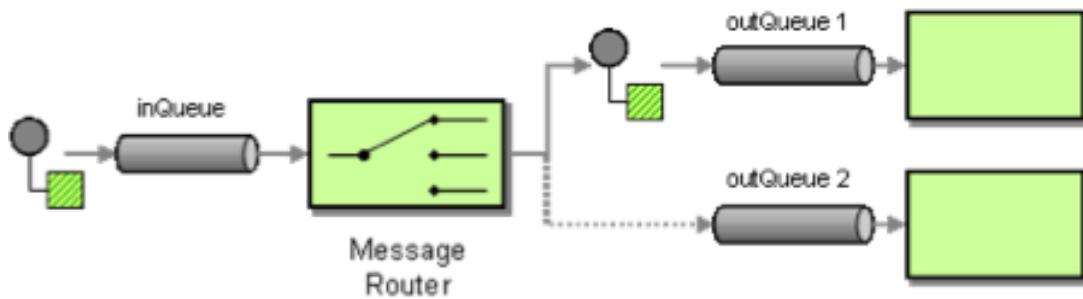
### Problema

“Como desacoplar passos individuais de processamento de forma que mensagens possam ser passadas para diferentes filtros dependendo de uma série de condições?”

### Solução

“Use um filtro especial, um *Roteador de Mensagens (Message Router)* que consome uma Mensagem de um Canal de Mensagens e publique-a em outro Canal de Mensagens, dependendo de uma série de condições.”

### Diagrama



### Descrição

Quando há várias aplicações interligadas que são conectadas por mais de um canal, é necessário determinar o caminho (a rota) que a mensagem precisa seguir até o seu destino. Se o destinatário não for acessível diretamente, é necessário calcular o melhor caminho dentro da rede de canais que a mensagem precisará seguir até o destino. Os componentes intermediários que interligam os canais são (8) *Roteadores de Mensagens (Message Router)*.

Se dados foram distribuídos em várias mensagens, é possível que elas sigam por caminhos diferentes e cheguem fora de ordem.

O *Roteador de Mensagens*, dentro da arquitetura *Dutos e Filtros*, é um filtro que determina em qual canal será depositada uma mensagem. Um conjunto de roteadores age como um *Chain of Responsibility* (Padrão GoF), onde cada componente decide o que será executado em seguida. Um roteador nunca transforma o conteúdo da mensagem, mas pode modificar o seu cabeçalho onde ficam as informações de endereçamento, dividir seu conteúdo em múltiplas mensagens, ou combinar o conteúdo de várias mensagens em uma só..

Roteadores podem ser incluídos transparentemente em uma rota de integração, mas precisam conhecer os destinos para onde irão enviar mensagens. Cada roteador precisa conhecer o caminho até o destino final ou até o próximo roteador. Essa informação pode ser obtida dinamicamente.

Vários padrões existem para descrever soluções usando roteadores.

Um (29) *Roteador Baseado em Conteúdo (Content-Based Router)* analisa o conteúdo de uma mensagem ou seu cabeçalho para decidir em qual canal irá depositar mensagens. O roteador pode também distribuir suas mensagens com base em uma lista estática, como em (32) *Lista de Receptores (Recipient List)*. Em vez de distribuir mensagens em múltiplos canais, um (30) *Filtro de Mensagens (Message Filter)* pode ser usado para descartar mensagens indesejadas.

Existem roteadores como o (33) *Divisor (Splitter)* que dividem uma mensagem em múltiplas partes, distribuindo cada parte em um canal, e também o (34) *Agregador (Aggregator)* que faz o inverso.

A maior parte dos roteadores são estáticos, mas (31) *Roteador Dinâmico (Dynamic Router)* descreve um roteador que decide para onde enviar mensagens em tempo de execução. Muitos roteadores são *stateless*, mas alguns, como o (35) *Re-sequenciador (Resequencer)*, que reordena mensagens, precisam ser *stateful* para lembrar das mensagens já processadas.

Usando o padrão de arquitetura *Dutos e Filtros* é possível construir um (36) *Processador de Mensagens Compostas (Composed Message Processor)* que combina roteadores, divisores e agregadores para dividir o processamento de uma mensagem por vários canais. Também são classificados entre os padrões de roteamento as microarquiteturas (38) *Lista de Circulação (Routing Slip)*, (39) *Gerente de Processos (Process Manager)* e (40) *Corretor de Mensagens (Message Broker)* que representam estruturas ainda mais elaboradas baseadas em *Dutos e Filtros*.

## Aplicações

Um roteador é necessário para distribuir mensagens em canais diferentes, estabelecendo rotas diferentes para mensagens entre o remetente e o destinatário final das mensagens.

### Roteador de Mensagens em Java (JMS)

Roteadores de Mensagens podem ser implementados em Java através de uma estrutura condicional que envie uma mensagem a um destino ou outro dependendo de certas condições. O exemplo abaixo redireciona mensagens para a fila *goodChannel* ou *evilChannel* dependendo do valor do cabeçalho "nature" (é um Roteador Baseado em Conteúdo):

```

01.  public class GoodEvilRouter implements MessageListener {
02.      @Override
03.      public void onMessage(Message message) {
04.          Context ctx = new InitialContext();
05.          Destination goodChannel = (Destination) ctx.lookup("good-queue");
06.          Destination evilChannel = (Destination) ctx.lookup("evil-queue");
07.          String type = message.getStringProperty("nature");
08.          if (nature != null && nature.equals("good")) { // to good-queue
09.              routeMessage(goodChannel, message);

```

```

10.         } else if (nature != null && type.equals("evil")) { // to evil-queue
11.             routeMessage(evilChannel, message);
12.         }
13.     }
14.     public void routeMessage(Destination destination, Message message) {
15.         ...
16.         MessageProducer producer = session.createProducer(destination);
17.         producer.send(message);
18.     }
19. }
```

Para funcionar, o roteador precisa ser configurado para consumir mensagens da fila “mixed-queue”, onde as mensagens são depositadas:

```

01. ...
02. Destination mixedChannel = (Destination) ctx.lookup("mixed-queue");
03. MessageConsumer receiver = receiverSession.createConsumer(mixedChannel);
04. receiver.setMessageListener(new GoodEvilRouter());
```

## Roteador de Mensagens em Apache Camel

Camel implementa vários diferentes tipos de roteadores. O exemplo mostrado em JMS acima, que representa um *Roteador Baseado em Conteúdo (Content-Based Router)*, pode ser configurado em Java DSL através do método *choice()*:

```

01. from("jms:queue:mixed-queue")
02.     .choice()
03.         .when(header("nature").equals("good"))
04.             .to("jms:queue:good-queue")
05.         .when(header("nature").equals("evil"))
06.             .to("jms:queue:evil-queue")
07.     .end();
```

## Roteador de Mensagens em Spring Integration

O tag *<router>* é usado para configurar vários tipos de roteadores. Outros tags trazem roteadores especializados e pré-configurados. O exemplo abaixo usa um *Roteador Baseado em Conteúdo (Content-Based Router)* para analisar um cabeçalho e determinar para qual fila uma mensagem enviada para a fila “mixed-queue” será enviada baseado no seu valor:

```

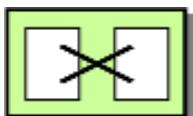
01. <int:header-value-router
02.     input-channel="mixed-queue"
03.     header-name="nature">
04.         <int:mapping value="good" channel="good-queue" />
05.         <int:mapping value="evil" channel="evil-queue" />
06.     </int:header-value-router>
```

Os canais do exemplo acima, assim como os cabeçalhos das mensagens são implementações internas do Spring. Para que esse componente se comunique com os mesmos canais usados nos exemplos Camel e JMS, é preciso que seus canais de entrada e saída sejam integrados a destinos JMS usando um

*Adaptador de Canal (JMS Channel Adapter)*, e que sejam copiados as propriedades das mensagens JMS para os cabeçalhos das mensagens do Spring.

## (9) Tradutor de mensagens (Message Translator)

### Ícone



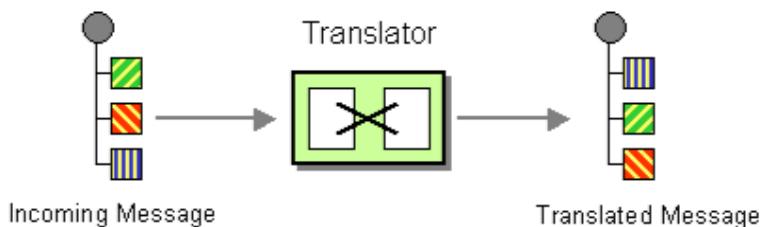
### Problema

“Como é possível realizar a comunicação usando mensageria, entre sistemas que usam formatos de dados diferentes?”

### Solução

“Use um filtro especial, um *Tradutor de Mensagens (Message Translator)*, entre outros filtros ou aplicações para traduzir de um formato de dados para outro.”

### Diagrama



### Descrição

Dados embutidos nas mensagens podem ter formatos diferentes no remetente e destinatário. Para traduzi-los, uma mensagem pode ser enviada através de um *Tradutor de Mensagens (Message Translator)* que irá convertê-la no formato esperado.

Um tradutor é um *Adaptador (Adapter, Padrão [GoF])*, que permite a comunicação entre sistemas que possuem interfaces incompatíveis. Também é um *Decorador (Decorator, Padrão GoF)*, pois tem interface consistente e pode ser conectado em série para realizar transformações cumulativas.

A transformação de uma mensagem pode ocorrer em vários níveis e camadas. Um tradutor que atua na camada de transporte pode mover dados entre protocolos (ex: extraír dados de mensagens SOAP, construir pacotes de dados UDP, etc.). A transformação entre diferentes representações de dados pode realizar conversões entre formatos binários *big endian* e *little endian*, *encoding*, criptografia. Tipos de dados (formatos padrão ou tipos definidos no domínio da aplicação) podem ser transformados usando expressões regulares, ferramentas de busca e substituição, transformações XSL, etc. Estruturas de dados podem ser reordenadas.

Embora classificado dentre os padrões relacionados a canais, um *Adaptador de Canal (Channel Adapter)* também pode ser descrito como um terminal tradutor, que adapta dados de uma aplicação externa ao sistema de mensageria, acoplado a um canal, que recebe dados compatíveis com o sistema, ou seja, consiste de um (10) *Terminal de Mensageria (Messaging Endpoint)* que é um (9) *Tradutor de Mensagens (Message Translator)* conectado a um (5) *Canal de Mensagens (Message Channel)*.

Oa padrões relacionados à tradução de dados incluem (43) *Filtro de Conteúdo (Content Filter)*, que filtra dados de uma mensagem, (42) *Enriquecedor de Conteúdo (Content Enricher)* que adiciona dados, (45) *Normalizador (Normalizer)*, para traduzir mensagens com mesmos dados organizados em formatos diferentes para um formato comum, (46) *Modelo de Dados Canônico (Canonical Data Model)*, para determinar um formato comum usado por todas as aplicações, (44) *Recibo de Bagagem (Claim Check)*, para despachar os dados através de um mecanismo de storage e guardar apenas um identificador para recuperá-lo, e (41) *Envelope (Envelope Wrapper)*, que descreve como empacotar uma mensagem no corpo de outra mensagem.

## Aplicações

Tradutores são usados quando é necessário realizar transformações no conteúdo de uma mensagem, para que possa ser processada pelo destinatário ou enviada a canais que requerem um formato diferente.

### Tradutor de Mensagens em Java (JMS)

Java oferece vários recursos para transformar dados. Exemplos anteriores mostraram como fazer uma transformação simples do conteúdo de texto de uma mensagem usando Java e JMS. O código abaixo demonstra a implementação de um componente de transformação que procura hashtags, usuários e URLs no corpo da mensagem (que é um Tweet), extrai o remetente de um cabeçalho e constrói um bloco `<div>` contendo as informações decoradas com tags HTML:

```

01.  public class TranslateTweetToHtml {
02.      public Message translate(String sender, String message) {
03.          String[] words = message.split(" ");
04.          StringBuilder buffer = new StringBuilder();
05.          for(String word: words) {
06.              if(word.startsWith("#")) {
07.                  buffer.append("<span class='hashtag'>")
08.                      .append(word).append("</span>");
09.              } else if (word.startsWith("@")) {
10.                  buffer.append("<span class='user'>")
11.                      .append(word)
12.                      .append("</span>");
13.              } else if (word.startsWith("http://")) {
14.                  buffer.append("<a href='")
15.                      .append(word)
16.                      .append("'>")
17.                      .append(word)
18.                      .append("</a>");
19.              } else {
20.                  buffer.append(word);
21.              }
22.          }
23.          return new TextMessage(buffer.toString());
24.      }
25.  
```

```

13.         buffer.append(word);
14.     }
15.     buffer.append(" ");
16. }
17.
18.     String newText = buffer.toString().substring(0,buffer.length()-1);
19.     String result = "<div class='tweet'><span class='sender'>" +
                     + sender
                     + "</span>" +
                     + newText + "</div>";
20.     return message; // texto transformado!
21. }
22. }
```

A transformação pode ser feita em qualquer componente que interceptar a mensagem:

```

01. TextMessage msg = (TextMessage) receiver.receive();
02. String sender = (String)msg.getStringProperty("sender");
03. String payload = msg.getText();
04. String newPayload = new TranslateTweetToHtml().translate(sender, payload);
05. msg.setText(newPayload);
06. sender.send(msg);
```

Este processador pode então ser chamado por um componente que está participando de uma arquitetura de *Dutos e Filtros*, interceptar as mensagens do canal de entrada, passar pelo tradutor, e depositar a mensagem traduzida no canal de saída.

## Tradutor de Mensagens em Apache Camel

Camel fornece uma infraestrutura que facilita a criação e o reuso de componentes de transformação em vários níveis. A transformação no nível de transporte é feita implicitamente pelos componentes que interagem com o mundo externo (Endpoints). A transformação em nível de dados é feito pela classe *TypeConverter* e pela interface *DataFormat* (um plug-in para fazer marshalling e unmarshalling) e pode ser incorporada em Componentes ou usada em outras partes da aplicação.

A transformação dos dados e estrutura de uma mensagem podem ser feitos implementando processadores (*Processors*) que chamam classes (como a que foi descrita no exemplo JMS) escritas em Java, ou expressões compactas. Os exemplos abaixo mostram duas maneiras de realizar a transformação mostrada no exemplo em Java/JMS:

```

01. from("jms:queue:inbox").process(new Processor() {
02.     @Override
03.     public void process(Exchange exchange) {
04.         Message in = exchange.getIn();
05.         String sender = in.getHeader("sender");
06.         String payload = in.getBody(String.class);
07.         String newBody =
                     new TranslateTweetToHtml().translate(sender, payload);
08.         in.setBody(newBody);
```

```

09.      }
10.  }).to("jms:queue:outbox");

```

## Tradutor de Mensagens em Spring Integration

É possível realizar a transformação mostrada acima em Spring Integration implementando um Transformer:

```

01.  public class HtmlDecoratorTransformer implements Transformer {
02.      public Message<?> transform(Message<?> message) {
03.          String[] words = message.getPayload().toString().split(" ");
04.          StringBuilder buffer = new StringBuilder();
05.          for(String word: words) {
06.              if(word.startsWith("#")) {
07.                  buffer.append("<span class='hashtag'>")
08.                      .append(word).append("</span>");
09.              } else if (word.startsWith("@")) {
10.                  buffer.append("<span class='user'>")
11.                      .append(word).append("</span>");
12.              } else if (word.startsWith("http")) {
13.                  buffer.append("<a href='")
14.                      .append(word).append("'>")
15.                      .append(word).append("</a>");
16.              }
17.          }
18.          String sender = (String)message.getHeaders().get("sender");
19.          String newPayload = buffer.toString().substring(0,buffer.length()-1);
20.          String result = "<div class='tweet'><span class='sender'>"
21.                          + sender + "</span>" + newPayload + "</div>";
22.          Message<String> decoratedMessage =
23.              MessageBuilder.withPayload(result)
24.                  .copyHeaders(message.getHeaders())
25.                  .build();
26.          return decoratedMessage;
27.      }
28.  }

```

E depois configurando a rota usando o tag `<transformer>`:

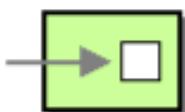
```

01.  <int:transformer input-channel="raw-tweets"
02.                  output-channel="decorated-tweets">
03.      <bean class="br.com.argonavis.si.examples.TweetTransformer" />
04.  </int:transformer>

```

## (10) Terminal de Mensageria (Messaging Endpoint)

### Ícone



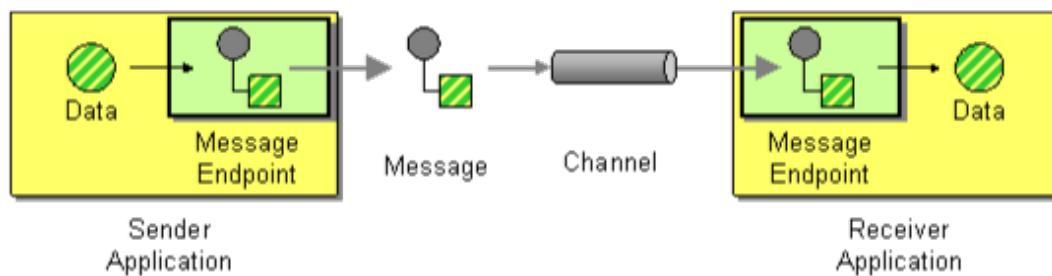
### Problema

“Como pode uma aplicação conectar-se a um canal de mensageria para enviar e receber mensagens?”

### Solução

“Conecte uma aplicação a um canal de mensageria usando um *Terminal de Mensagens (Message Endpoint)*, um cliente do sistema de mensageria que a aplicação pode usar para enviar ou receber mensagens.”

### Diagrama



### Descrição

Um *Terminal de Mensageria (Messaging Endpoint)* é a parte de uma aplicação que realiza a comunicação necessária à integração, que envia e recebe mensagens. É um cliente do sistema de mensageria. Pode ser um remetente ou destinatário. O catálogo EIP classifica como *Terminais* apenas os componentes de uma solução de integração que criam ou consomem mensagens (não inclui os roteadores e transformadores que alteram ou re-endereçam mensagens existentes). Em JMS os componentes *Producer (Sender e Publisher)* e *Consumer (Receiver e Subscriber)* podem ser considerados terminais.

Terminais são geralmente a porta de entrada e saída de uma aplicação em um sistema de mensageria. Geralmente é implementado como um *Adaptador (Adapter)* que produz ou consome mensagens, fornecendo a interface necessária para unir o sistema externo ao sistema de mensageria.

Vários padrões estão relacionados a terminais de mensageria.

(47) *Gateway de Mensageria (Messaging Gateway)* é um adaptador bidirecional que permite que uma aplicação participe de um sistema de mensageria, fornecendo uma interface que encapsula o código relacionado a mensageria. É análogo a um DAO com relação ao acesso a banco de dados. Um gateway pode ser usado por um (48) *Mapeador de Mensageria (Messaging Mapper)*, que mapeia objetos do

domínio da aplicação a mensagens abstraindo o gateway completamente. Um (57) *Ativador de Serviço (Service Activator)* permite que um serviço externo seja chamado de dentro do sistema de mensageria.

Um terminal receptor pode ser um (51) *Consumidor de Sondagem (Polling Consumer)*, que sonda periodicamente um canal esperando a chegada de mensagens, ou um (50) *Consumidor Ativado por Evento (Event-Driven Consumer)*, que é notificado quando uma mensagem é recebida por um canal. Pode haver vários consumidores disputando as mensagens enviadas para um canal. Essa estratégia é descrita no padrão (52) *Consumidores Concorrentes (Competing Consumers)*.

O consumo de mensagens pode ser organizado por um (53) *Despachante de Mensagens (Message Dispatcher)* que consome todas as mensagens de um canal e as distribui para processadores especializados. Pode-se também usar um (54) *Consumidor Seletivo (Selective Consumer)*, que utiliza-se de um filtro para determinar quais mensagens de um canal irá consumir.

Mensagens duplicadas podem ser indesejáveis e causar efeitos colaterais. Um (56) *Receptor Idempotente (Idempotent Receiver)* pode lidar com isto, garantindo que mensagens recebidas duas vezes não causem efeitos diferentes.

Finalmente, para consumidores que podem não estar ativos durante a transmissão de mensagens enviadas em canais de difusão, o uso do padrão (55) *Assinante Durável (Durable Subscriber)* garante que o sistema retenha as mensagens durante a inatividade e as envie novamente quando o consumidor estiver ativo.

## Aplicações

*Terminais de Mensageria* são os *Produtores* e *Consumidores* de mensagens. São a interface de entrada e saída de um sistema de mensageria com as aplicações que serão integradas.

### Terminal de Mensageria em Java (JMS)

Os terminais de mensageria em JMS são representados pelos componentes que produzem e consomem mensagens: *MessageConsumer* e *MessageProducer*. Embora eles apareçam em qualquer componente que recebe e re-envia mensagens, um *Terminal*, rigorosamente seria apenas o componente que *inicia* a rota fornecendo as mensagens, e o que consome a mensagem no final. Um componente intermediário (ex: um *Tradutor de Mensagens*) embora utilize-se um par *Produtor/Consumidor*, não seria considerado aqui um *Terminal* (os frameworks não concordam sobre esses conceitos – veja abaixo).

Anteriormente foram mostrados neste capítulo (seção *Dutos e Filtros*) exemplos da implementação de uma rota em JMS que passa por um consumidor, um transformador, e um produtor (dois *Terminais de Mensageria* e um *Tradutor de Mensagens*). Neste outro exemplo uma pasta é monitorada periodicamente a espera de um arquivo. Quando o arquivo é depositado na pasta, a aplicação abre-o, extrai o texto encontrado na primeira linha, cria uma mensagem e envia para uma fila chamada “*contagem*”:

```
01.  public class EndpointExample {
02.      private File dataFile = null;
03.      private File directory = new File("/tmp/inbox");
```

```

04.
05.    public void start() {
06.        System.out.print("Waiting for file:");
07.        while (dataFile == null) {
08.            dataFile = this.loadFile();
09.            try {Thread.sleep(5000); } catch (InterruptedException e) {}}
10.            System.out.print(".");
11.        }
12.        System.out.println("\nLoaded file. Will process and send message.");
13.        BufferedReader reader = null;
14.        try {
15.            reader = new BufferedReader(new FileReader(dataFile));
16.            String data = reader.readLine();
17.            dataFile.delete();
18.            sendMessage(data);
19.        } catch (Exception e) {...}
20.    }
21.
22.    private File loadFile() {
23.        String[] files = directory.list(new FilenameFilter() {
24.            public boolean accept(File dir, String name) {
25.                return name.endsWith(".txt");
26.            }
27.        });
28.        if (files.length > 0) {
29.            return new File(directory, files[0]);
30.        }
31.        return null;
32.    }
33.
34.    private void sendMessage(String data) throws Exception {
35.        Context ctx = new InitialContext();
36.        ConnectionFactory factory = ...
37.        Destination destination = (Queue) ctx.lookup("contagem");
38.        Connection con = factory.createConnection();
39.        Session session = con.createSession(...);
40.        MessageProducer producer = session.createProducer(destination);
41.        TextMessage message = session.createTextMessage(data);
42.        producer.send(message);
43.    }
44. }

```

Este *Terminal* é um componente que *adapta* o sistema de arquivos ao domínio do JMS, portanto é um *Adaptador de Canal (Channel Adapter)*.

## Terminal de Mensageria em Apache Camel

*Terminais de Mensageria em Camel* têm duas partes:

- *Endpoint*, que funciona como um canal para onde podem ser enviadas mensagens, ou que produz mensagens, e
- *Component*, que é um adaptador que converte uma fonte ou destino externa para que possa participar da mensageria através do Camel.

Portanto, o Terminal de Mensageria (*Messaging Endpoint*) em Camel é representado pelo conjunto *Endpoint + Component*.

O exemplo abaixo mostra uma solução similar à mostrada em JMS usando Camel e um servidor *ActiveMQ*. O serviço espera por 20 segundos que um arquivo seja colocado na pasta. Quando ele chega, ele é empacotado em uma mensagem e enviado para uma fila JMS:

```

01. public class MoveFileToJmsService {
02.     public static void main(String[] args) throws Exception {
03.         CamelContext context = new DefaultCamelContext();
04.         context.addRoutes(new RouteBuilder {
05.             @Override
06.             public void configure() throws Exception {
07.                 from("file:///tmp/inbox")
08.                     .to("jms:queue:contagem");
09.             }
10.         });
11.         context.start();
12.
13.         System.out.println("O servidor está no ar por 20 segundos.");
14.         Thread.sleep(20000);
15.         context.stop();
16.     }
17. }
```

Para testar, coloque um arquivo na pasta */tmp/inbox*. A URI *file:* automaticamente instancia um *Endpoint* e *Componente* que conecta-se ao sistema de arquivos. Por default, o conteúdo do arquivo será copiado ao corpo da mensagem gerada.

## Terminal de Mensageria em Spring Integration

O conceito de *Terminal*, ou *Endpoint*, é diferente em EIP, JMS, Camel e Spring Integration. De acordo com a documentação do Spring Integration, qualquer componente que produz ou consome mensagens é um *Endpoint* (isto inclui tradutores e roteadores). Se considerarmos como endpoints apenas os componentes de entrada ou de saída que aparecem nas pontas de uma rota, o melhor representante para *Terminal de Mensageria em Spring* é o que o *Spring Integration* chama de *Channel Adapter*, que consiste de um componente que adapta uma fonte ou destino de mensagens externa e um canal, e que é similar ao Componente + Endpoint do Camel.

Assim, podemos implementar a solução apresentada acima em *Spring Integration* usando *Channel Adapters*. Usamos um Adaptador de Canal de entrada para ler o arquivo:

```
<int-file:inbound-channel-adapter
    id="si-contagem"
    directory="file:///tmp/inbox"
    filename-pattern="*.txt" />
```

A solução acima está apenas transferindo os arquivos para um canal nativo do Spring. Para enviar as mensagens para uma fila JMS, seria necessário conectar um Adaptador de Canal de saída ao canal “si-contagem”:

```
<int-jms:outbound-channel-adapter
    destination-name="contagem"
    channel="si-contagem"/>
```

Spring também contém vários componentes que representam tipos específicos de clientes, de sondagem e ativados por eventos, ativadores de serviços, pontes e gateways. Eles são configuráveis através do XML do Spring e serão exemplificados nos próximos capítulos.

## Padrões de gerenciamento do sistema

Vários padrões do catálogo EIP não são associados a componentes específicos da arquitetura de mensageria, mas referem-se ao gerenciamento do sistema como um todo. Esses padrões descrevem mecanismos de logging, testes, monitoração e controle do sistema.

(58) *Barramento de Controle (Control Bus)* descreve uma arquitetura que permite a monitoração completa do sistema, de forma paralela, sem interferir no fluxo de mensagens. (59) *Desvio (Detour)* permite rotear mensagens através de canais paralelos para validação, testes e debugging. Uma (60) *Escuta (Wire Tap)* pode ser usada para inspecionar mensagens que transitam em um canal *Ponto-a-Ponto* sem consumi-las. Um log do caminho por onde passou uma mensagem pode ser implementado com um (61) *Histórico da Mensagem (Message History)*. (62) *Repositório de Mensagens (Message Store)* permite manter essas mensagens armazenadas mesmo depois que elas forem consumidas. Um (63) *Proxy Inteligente (Smart Proxy)* pode ser usado para coletar métricas, realizar testes de roteamento, enviando uma (64) *Mensagem de Teste (Test Message)* através das interfaces do sistema. Para garantir que dados residuais não interfiram nos testes, um (65) *Purificador de Canal (Channel Purger)* pode ser empregado para limpar o canal.

Exemplos desses componentes serão apresentados no capítulo 9.

## Frameworks

Depois da publicação do catálogo EIP, surgiram vários frameworks que implementam grande parte dos padrões. Os mais populares (em 2015) e que implementam a maior quantidade de padrões de integração são *Apache Camel* e *Spring Integration*. Embora adotem estratégias, arquiteturas e até conceitos diferentes, todos oferecem APIs e DSLs que quase sempre usam a mesma nomenclatura adotada no catálogo EIP, com pequenas variações. Nenhum implementa todos os padrões, e todos oferecem componentes adicionais e mecanismos que não são descritos nos catálogos de padrões.

Implementar uma solução projetada com padrões EIP usando Camel ou Spring Integration é bem mais simples que construir tudo “do zero” usando uma API de baixo-nível como JMS (embora a construção de alguns padrões mais elaborados possa ser igualmente complexa). Uma vez configurado o ambiente, rotas podem ser descritas usando instruções de uma API (Java DSL, Scala DSL) ou tags XML (Spring). Vários padrões são implementados com uma simples instrução e parâmetro, anotações, URIs, implementação de um método, ou configuração de um tag e atributos XML. Outros, principalmente os que não têm uma implementação nativa pelo framework, requerem mais trabalho e podem ser construídos combinando mecanismos do framework ou adaptando implementações externas.

A seguir uma breve descrição de dois dos principais frameworks que implementam padrões do catálogo [EIP]: *Apache Camel* e *Spring Integration*. Este curso está considerando as últimas versões de cada framework disponibilizadas em julho e agosto de 2015, que são: Apache Camel 2.15 e Spring Integration 4.1.

## Apache Camel

Apache Camel é um serviço de roteamento e integração que fornece um framework que pode ser usado para integrar diferentes aplicações usando EIP. Camel não oferece um serviço próprio de mensageria; ele apenas cuida do roteamento de mensagens e da integração. É possível usar camel para integrar, por exemplo, mensagens disponíveis em uma fila JMS com um sistema de arquivos, um banco de dados, um Web Service, e-mail, Twitter, etc.

A principal tarefa que um usuário do framework precisa realizar é a configuração de rotas. Existem duas maneiras:

- Usando uma Domain-Specific Language (DSL) “Fluent API”, que pode ser Java ou Scala, e encadeando métodos que representam componentes, ou
- Usando Spring, e configurando rotas através de tags e atributos XML.

A arquitetura do Camel consiste de:

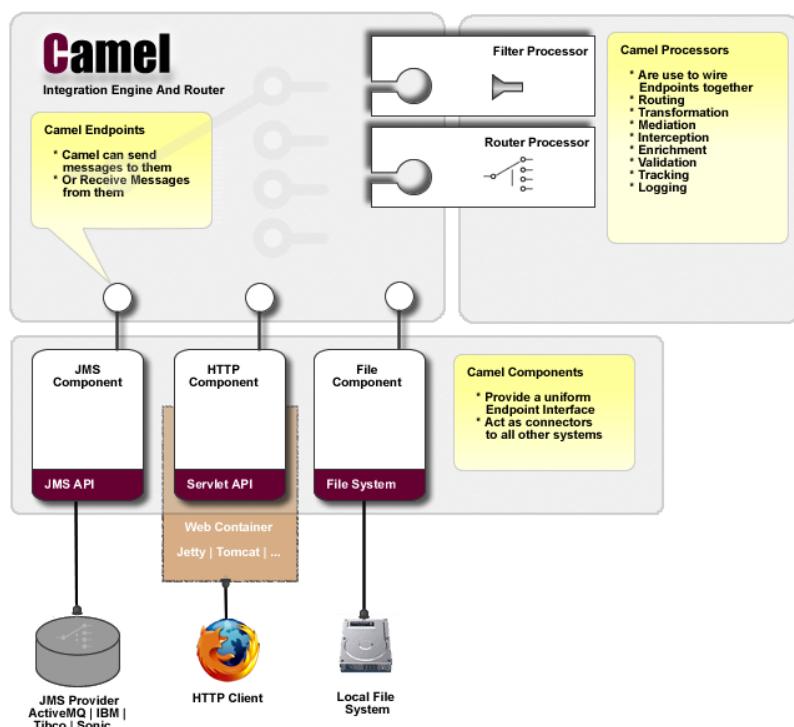
- Endpoints + Componentes (que incluem canais)
- Processadores
- Mensagens
- Rotas e Contexto

Para cada serviço externo que fornece canais ou mensagens, o Camel fornece um Componente, que é um *adaptador*. Cada Componente está associado a um ou mais canais especiais chamados de Endpoints, que são capazes de receber ou enviar mensagens de e para o Camel. Todos os outros componentes intermediários (filtros, roteadores, transformadores, loggers, etc.) são *Processadores*. Eles interligam dois Endpoints formando uma Rota.

O Apache Camel suporta uma lista enorme de Componentes. É possível também construir novos Componentes. Cada Componente age como uma fábrica de Endpoints. O Endpoint é necessário para

que o Componente possa participar da mensageria com o Camel. Alguns Componentes/Endpoints comuns são JMS, File, Direct, etc. Um Endpoint é identificado por uma URI. Quando essa URI é usada numa rota, uma instância desse Endpoint é localizada (ou criada, se não existir) e seu Componente é instanciado automaticamente. Componentes/Endpoints devem ser previamente configurados e instalados no ambiente (devem ser acessíveis pelo Classpath) para que seu uso seja possível. Em um ambiente de desenvolvimento usando Camel, geralmente os componentes que serão utilizados são incluídos como dependências (ex: Maven, Ivy) do projeto.

O diagrama abaixo, da documentação oficial do Camel, ilustra o relacionamento entre componentes, endpoints e processadores:



Exemplo de configuração explícita de um Endpoint no Camel:

```

01. JMSEndpoint endpoint =
        (JMSEndpoint) camelContext.getEndpoint("jms:queue:fila");
02. endpoint.setAlwaysCopyMessage(true);
03. endpoint.setAllowNullBody(false);
04. endpoint.setConcurrentCustomers(10);
05.
06. routeBuilder.from("jms:queue:fila").to("jms:queue:debug");
    
```

Um *Endpoint Camel* pode criar *Consumers* e *Producers*. Um *Producer* pode criar um *Exchange*. *Exchange* é um objeto que contém mensagens (*Message*). Um *Exchange* pode conter até três mensagens: uma mensagem de entrada, uma mensagem de saída e uma mensagem de erro. As mensagens implementam a interface *Message*. O conteúdo de uma mensagem pode ser obtido através do método *getBody()*.

O trecho de código abaixo ilustra o uso de vários métodos de *Exchange* para alterar uma mensagem (fonte: documentação do Camel):

```

01. public void process(Exchange exchange) throws Exception {
02.     String body = exchange.getIn().getBody(String.class);
03.     // change the message to say Hello
04.     exchange.getOut().setBody("Hello " + body);
05.     // copy headers from IN to OUT to propagate them
06.     exchange.getOut().setHeaders(exchange.getIn().getHeaders());
07.     // copy attachments from IN to OUT to propagate them
08.     exchange.getOut().setAttachments(exchange.getIn().getAttachments());
09. }
```

Os componentes Camel que implementam roteadores e transformadores são *Processadores* (interface *Processor*). Eles podem ser implementados localmente ou pré-instanciados e obtidos de um registro. *Processors* são *Consumidores Ativados por Evento* (*Event-Driven Consumers*). São os filtros da arquitetura *Pipes and Filters* e handlers de eventos que recebem uma mensagem (como o JMS *MessageListener*). Quando um *Endpoint* cria um *Consumer*, ele deve receber um *Processor*:

```

01. Endpoint e = ...
02. Processor p = ...
03. Consumer c = e.createConsumer(p);
```

A exceção é se for usado um *Consumidor de Sondagem* (*Polling Consumer*), que recebe mensagens de forma síncrona. Neste caso o método para criar um *Consumer* é outro:

```
04. e.createPollingConsumer()
```

Usando Java DSL, rotas são construídas encadeando métodos que retornam processadores. Uma rota é construída configurando um *RouteBuilder* e adicionando o resultado a um contexto: o *CamelContext*, que é o container. Rotas em *Camel* também utilizam e reutilizam expressões e predicados, passadas como parâmetros para determinados processadores.

O exemplo abaixo ilustra a configuração de rotas usando Camel com Java DSL:

```

01. CamelContext context = new DefaultCamelContext();
02. context.addRoutes(new RouteBuilder() {
03.     @Override
04.     public void configure() throws Exception {
05.         from("file:inbox").to("jms:objetos");
06.
07.         from("jms:objetos")
08.             .choice()
09.                 .when(header("CamelFileName"))
10.                     .regex("^.*(jpg|png|gif|jpeg)$")
11.                     .to("jms:imagens")
12.                 .when(header("CamelFileName"))
13.                     .endsWith(".xml"))
14.                     .to("jms:docsXML")
```

```

15.         .otherwise()
16.             .to("jms:invalidos")
17.                 .stop() // estes objetos não seguem para próxima etapa
18.             .end() // fim do choice
19.         .to("jms:nextStep"); // proxima etapa (todos q n chamam stop())
20.
21.     from("jms:imagens")
22.         .process(new Processor() {
23.             public void process(Exchange exchange) throws Exception {
24.                 System.out.println("Imagen recebida: " +
25.                     exchange.getIn().getHeader("CamelFileName"));
26.             }
27.         })
28.         .multicast()
29.             .to("jms:backup", "jms:imagensParaProcessar")
30.                 .parallelProcessing(); // processa requisições em paralelo
31.
32.     from("jms:docsXML")
33.         .choice()
34.             .when(xpath("/Movies")) // expressão
35.                 .process(new Processor() {
36.                     public void process(Exchange exchange) throws Exception {
37.                         System.out.println("XML Movie recebido: "
38.                             + exchange.getIn().getHeader("CamelFileName"));
39.                     }
40.                 })
41.             .otherwise() // expressão
42.                 .process(new Processor() {
43.                     public void process(Exchange exchange) throws Exception {
44.                         System.out.println("XML inválido: "
45.                             + exchange.getIn().getHeader("CamelFileName"));
46.                     }
47.                 }).to("jms:invalidos")
48.             .end();
49.
50.     from("jms:invalidos")
51.         .process(new Processor() {
52.             public void process(Exchange exchange) throws Exception {
53.                 System.out.println("Arquivo inválido: "
54.                     + exchange.getIn().getHeader("CamelFileName"));
55.             }
56.         });
57.
58.     from("jms:nextStep")
59.         .process(new Processor() {
60.             public void process(Exchange exchange) throws Exception {
61.                 System.out.println("Arquivo copiado p próxima etapa: "
62.                     + exchange.getIn().getHeader("CamelFileName"));
63.             }
64.         })

```

```

59.         } );
60.
61.         from("jms:backup")
62.             .process(new Processor() {
63.                 public void process(Exchange exchange) throws Exception {
64.                     System.out.println("Imagen de backup: "
65.                         + exchange.getIn().getHeader("CamelFileName"));
66.                 }
67.             });
68.         from("jms:imagensParaProcessar")
69.             .process(new Processor() {
70.                 public void process(Exchange exchange) throws Exception {
71.                     System.out.println("Imagen para processamento: "
72.                         + exchange.getIn().getHeader("CamelFileName"));
73.                 }
74.             });
75.     });
76.     context.start();

```

Veja mais sobre o Camel no Apêndice A.

## Spring Integration

*Spring Integration* é um componente do microcontainer e framework *Spring*, que fornece um ambiente onde objetos e serviços podem ser registrados e injetados como dependências em outros objetos e serviços. *Spring Integration* estende o modelo de programação do *Spring* para o domínio da mensageria, fornecendo uma série de componentes de roteamento e transformação, além de abstrações de canais e mensagens. Embora adote uma arquitetura e um processo completamente diferente, pode-se alcançar os mesmos resultados de integração com *Spring Integration* ou *Apache Camel*.

Assim como Camel, Spring também implementa grande parte dos padrões do catálogo EIP. Os três componentes fundamentais da arquitetura do Spring Integration são:

- Mensagem
- Canal
- Endpoint

Um *Endpoint* em Spring tem um significado bem diferente de um *Endpoint* em Camel. As arquiteturas possuem várias diferenças. *Endpoint* é um termo usado em Spring para representar qualquer componente que processa mensagens da arquitetura *Dutos e Filtros*. É o *filtro* da arquitetura Dutos e Filtros. Seria equivalente a um *Processador*, em Camel.

O Canal é o duto da arquitetura Dutos e Filtros. Desacopla componentes de mensageria. Diferentemente de Camel, Spring provê uma implementação própria de Canais, embora também possa

interagir com canais externos. Um Canal em Spring pode ser um *Consumidor de Sondagem* (implementado com *PollableChannel*) ou de *Ativado por Eventos* (*SubscribableChannel*). Há implementações para canais ponto-a-ponto e de difusão.

O equivalente a um *Endpoint + Componente* do *Camel* em *Spring Integration* é o *Channel Adapter*, que consiste de um adaptador que permite a integração com serviços externos, como filas JMS, sistema de arquivos, e-mail, Twitter, etc. O *Channel Adapter* contém um Canal que pode ser usado na construção de rotas em *Spring Integration*. Existem vários *Channel Adapters* disponíveis, e pode-se também escrever outros se necessário. Assim como o *Camel*, o suporte para os adaptadores depende de código que deve ser instalado. Tipicamente em um ambiente de desenvolvimento, as dependências são baixadas usando Maven ou Ivy, e disponibilizadas no Classpath.

Outro componente fundamental do *Spring Integration* é o *Poller*, que implementa um mecanismo de sondagem e pode ser usado para construir canais e consumidores.

Uma mensagem em *Spring Integration* tem duas partes: um *cabeçalho* e um corpo, o *payload*.

Os processadores do *Spring Integration*, todos chamados de *Endpoints*, incluem componentes para roteamento (*Message Routers*), tradução (*Message Transformers*) e terminais (*Messaging Endpoints*).

A configuração pode ser feita via *Spring XML*, que é o mais comum, ou usando anotações e uma *DSL Java 8* usando lambdas. Para a configuração em XML, a utilização de *Channel Adapters* também requer o registro dos schemas (XSD) referentes aos serviços incluídos.

O exemplo abaixo ilustra a construção de uma rota usando *Spring Integration*.

```

01.  <?xml version="1.0" encoding="UTF-8"?>
02.  <beans xmlns="http://www.springframework.org/schema/beans"
03.      xmlns:xsi="..." xmlns:int="..." xmlns:int-feed="..."
04.      xmlns:int-file="..." xmlns:int-twitter="..." xmlns:context="..."
05.      xsi:schemaLocation="...">
06.
07.      <context:property-placeholder location="classpath:oauth.properties"
08.      />
09.      <bean id="twitterTemplate"
10.          class="org...TwitterTemplate">
11.          <constructor-arg value="${twitter.oauth.consumerKey}" />
12.          <constructor-arg value="${twitter.oauth.consumerSecret}" />
13.          <constructor-arg value="${twitter.oauth.accessToken}" />
14.          <constructor-arg value="${twitter.oauth.accessTokenSecret}" />
15.      </bean>
16.      <bean id="techTweetSelector" class="br.com...LinkTweetSelector" />
17.      <bean id="htmlDecoratorTransformer"
18.          class="br.com...HtmlDecoratorTransformer" />
19.      <bean id="tweetTagger" class="br.com...TweetSubjectTagger" />
20.      <int:channel id="tweets" />
21.      <int:channel id="tech-tweets" />
```

```
22.      <int:channel id="tagged-tweets" />
23.      <int:channel id="decorated-tweets" />
24.      <int:channel id="java-tweets" />
25.      <int:channel id="other-tweets" />
26.      <int:channel id="discarded-tweets" />
27.      <int:channel id="discarded-tweets-file" />
28.
29.      <int-twitter:search-inbound-channel-adapter
30.          twitter-template="twitterTemplate"
31.          query="#thedevconf OR #tdc2015"
32.          id="raw-tweets">
33.          <int:poller fixed-rate="60000" max-messages-per-poll="10" />
34.      </int-twitter:search-inbound-channel-adapter>
35.
36.      <int:transformer input-channel="raw-tweets"
37.          output-channel="tweets">
38.          <bean class="br.com...TweetTransformer" />
39.      </int:transformer>
40.
41.      <int:filter input-channel="tweets" output-channel="tech-tweets"
42.          discard-channel="discarded-tweets" ref="techTweetSelector" />
43.
44.      <int:header-enricher input-channel="tech-tweets"
45.          output-channel="tagged-tweets">
46.          <int:header name="subject"
47.              method="setSubjectHeader" ref="tweetTagger" />
48.      </int:header-enricher>
49.
50.      <int:transformer input-channel="tagged-tweets"
51.          output-channel="decorated-tweets"
52.          ref="htmlDecoratorTransformer" />
53.
54.      <int:header-value-router input-channel="decorated-tweets"
55.          header-name="subject">
56.          <int:mapping value="java" channel="java-tweets" />
57.          <int:mapping value="other" channel="other-tweets" />
58.      </int:header-value-router>
59.
60.      <int-file:outbound-channel-adapter
61.          channel="other-tweets" charset="UTF-8" mode="APPEND"
62.          directory="web-document-root/tweets"
63.          filename-generator-expression="'OtherTweets.html'" />
64.      <int-file:outbound-channel-adapter
65.          channel="java-tweets" charset="UTF-8" mode="APPEND"
66.          directory="web-document-root/tweets"
67.          filename-generator-expression="'JavaTweets.html'" />
68.
69.      <int:transformer
```

```
65.           input-channel="discarded-tweets"
66.           expression="headers.get('sender') + ':' + payload
67.                           + '#{systemProperties['line.separator']}''"
68.           output-channel="discarded-tweets-file"/>
69. 
70. <int-file:outbound-channel-adapter
71.   channel="discarded-tweets-file" charset="UTF-8" mode="APPEND"
72.   directory="web-document-root/tweets"
73.   filename-generator-expression="'DiscardedTweets.txt'" />
74. 
75. </beans>
```

## Revisão

Os padrões de integração de sistemas relacionados a mensageria são:

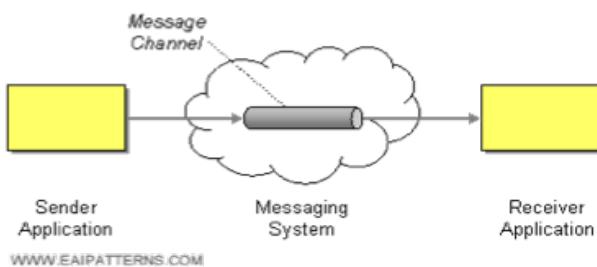
- *Canal (Message Channel)*: um local ou destino usado para compartilhar mensagens; mensagens podem ser colocadas em um canal, onde também podem ser consumidas.
- *Mensagem (Message)*: uma abstração que contém a informação enviada para canais; tipicamente uma mensagem contém um corpo, com seu conteúdo, e um cabeçalho com metadados e informações necessárias ao endereçamento.
- *Dutos e filtros (Pipes and Filters)*: uma arquitetura de componentes conectados por canais, usada para construir soluções usando padrões do catálogo [EIP].
- *Roteador de mensagens (Message Router)*: qualquer componente que desvia a rota de uma mensagem para um canal diferente sem alterar seu conteúdo.
- *Tradutor de mensagens (Message Translator)*: qualquer componente que transforma o conteúdo ou meta-dados de uma mensagem.
- *Terminal de mensageria (Messaging Endpoint)*: um componente que produz ou consome as mensagens enviadas através de canais.
- *Padrões de gerenciamento*: uma coleção de componentes e soluções usadas para testar e depurar sistemas de integração.

Apache Camel e Spring Integration são frameworks que implementam vários padrões do catálogo [EIP]. Eles possuem uma arquitetura diferente, têm conceitos diferentes para alguns padrões, mas são equivalentes quanto ao nível de suporte aos padrões e recursos disponíveis para implementar soluções de integração.

# Capítulo 4

# Canais

Um canal de mensagem é o duto através do qual uma aplicação pode trocar informações com outra. Para que possam se comunicar, um *remetente* e um *destinatário* deverão usar o mesmo canal. Canais viabilizam a comunicação entre partes que não se conhecem. Uma aplicação produtora envia dados para um canal onde uma ou mais aplicações consumidoras podem consumi-los. O produtor não precisa saber quem é o destinatário, e o consumidor não precisa saber quem é o produtor. O canal, nessa arquitetura, tem o papel de *mediador*.



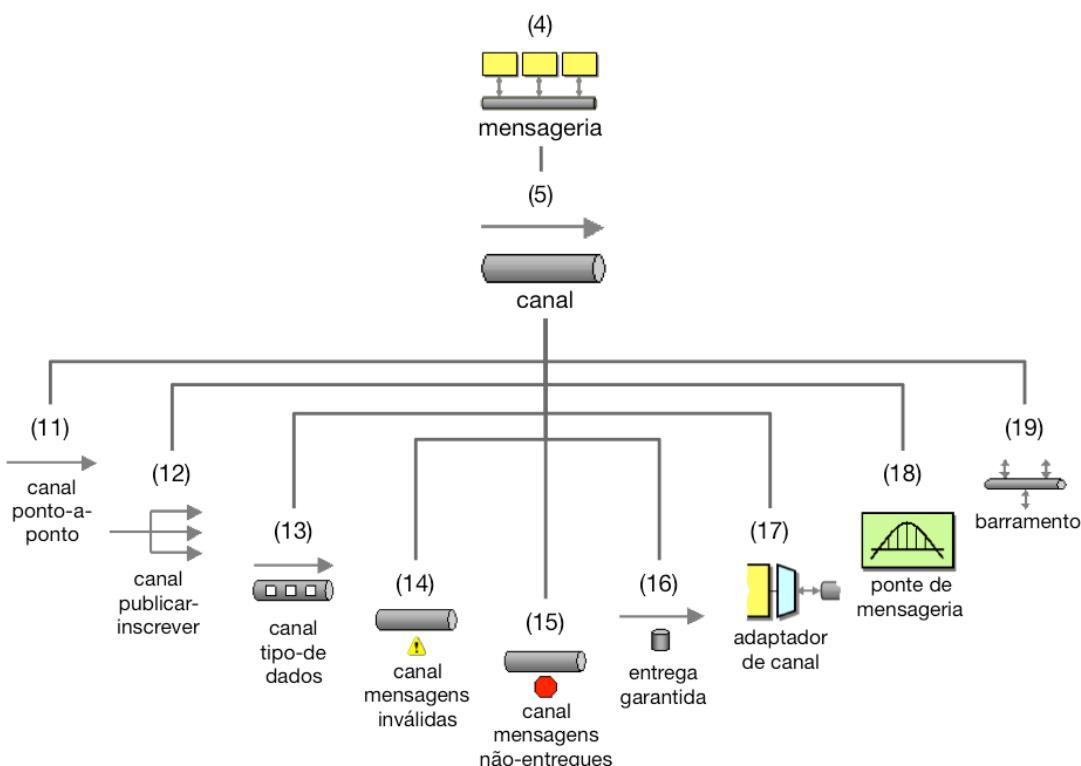
O duto pode não ser a melhor analogia para um canal. Apesar do nome, o canal na verdade é um *local* ou *destino* onde mensagens são colocadas e retiradas. Um canal não possui uma direção. Não faz sentido falar em canais unidirecionais ou bidirecionais. A direção do fluxo de mensagens em um canal é, na verdade, estabelecido pelos componentes que interagem com ele (se produzem ou consomem mensagens), ou seja, faz sentido falar em *comunicação* bidirecional ou unidirecional em um canal. Geralmente um canal é usado para comunicação unidirecional. A comunicação bidirecional normalmente é implementada usando dois canais.

Tipicamente, compartilham o mesmo canal pelo menos um componente que produz, e outro que consome mensagens, estabelecendo assim um fluxo de mensagens. Se um mesmo componente produz e também consome do mesmo canal, ele poderá consumir as suas próprias mensagens.

Canais são geralmente definidos estaticamente, antes da aplicação iniciar, por serem pontos de comunicação que devem ser conhecidos pelos componentes que irão interagir. Mas nada impede que uma aplicação crie um novo canal dinamicamente, havendo suporte para isto pela infraestrutura de mensageria, e informe da sua existência a outras aplicações que poderão posteriormente usá-lo na comunicação.

Os padrões ajudam a determinar como usar canais ao projetar um sistema de integração. Durante o projeto, os canais necessários são descobertos ao determinar as rotas que as mensagens deverão seguir. Uma solução pode empregar canais dedicados a tarefas e tipos de mensagens específicos; outras soluções poderão reusar canais filtrando as mensagens. Essa escolha terá impacto na qualidade da solução quanto ao acoplamento, e poderá ter impacto na performance. Canais podem controlar o que acontece com as mensagens que são depositadas neles: se guardam em meio persistente, se guardam apenas certo tipo de mensagem, se distribuem cópias da mensagem para vários destinatários, ou se entregam a mensagem apenas uma vez, a apenas a um destinatário. Existem também canais especiais que servem para comunicação com sistemas externos.

Os padrões EIP relacionados a canais incluem tipos diferentes de canais, atributos de canais e microarquiteturas que incluem canais. São nove padrões relacionados abaixo:



Os padrões podem ser classificados em:

- Canais*, de acordo com o *número de destinatários*: um *Canal Ponto-a-Ponto (Point-to-Point Channel)* permite apenas um destinatário por mensagem enviada; um *Canal de Difusão (Publish-Subscribe Channel)* permite distribuir mensagens a vários destinatários;
- Canais*, de acordo com o *tipo de mensagem*: um *Canal de Tipo de Dados (Datatype Channel)* retém apenas mensagens compatíveis com o tipo estabelecido pelo canal; um *Canal de Mensagens Inválidas (Invalid Message Channel)* guarda mensagens que não são válidas dentro de critérios estabelecidos pela aplicação; um *Canal de Mensagens Não-entregues (Dead-Letter Channel)* guarda mensagens que não chegaram a seus destinatários;

- c) *Canais persistentes: canais de entrega garantida* (Guaranteed Delivery) mantém cópia persistente da mensagem para entrega quando o destinatário estiver no ar e disponível para recebê-la;
- d) *Interfaces de comunicação com canais; adaptadores* (Channel Adapter), *pontes* (Messaging Bridge) e *barramentos* (Message Bus) permitem integrar canais a sistemas externos.

## (11) Canal Ponto-a-Ponto (Point-to-Point Channel)

### Ícone



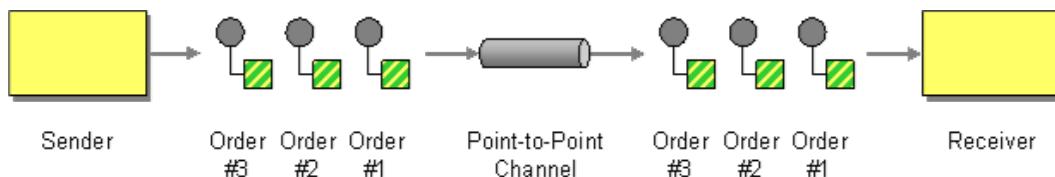
### Problema

“Como o remetente pode ter certeza que apenas um destinatário irá receber a mensagem ou executar um comando?”

### Solução

“Envie a mensagem através de um *Canal Ponto-a-Ponto* (Point-to-Point Channel), que garante que apenas um destinatário irá receber a mensagem.”

### Diagrama



### Descrição

Mensagem é enviada para um canal. Muitos destinatários podem ter acesso ao canal, mas apenas um deles irá consumir a mensagem. Depois que a mensagem for consumida ela não estará mais disponível no canal.

Se nenhum destinatário estiver disponível no momento em que a mensagem for enviada, ela poderá permanecer disponível até que o primeiro destinatário a receba e consuma.

### Aplicações

Situações em que apenas um destinatário pode receber uma mensagem. Por exemplo, uma mensagem que contém um comando que será executado pelo destinatário, ou uma mensagem que contenha um documento que é enviado como resposta a um comando.

A estratégia de recebimento para determinar qual destinatário irá consumir uma mensagem pode ser estabelecida nos endpoints (Ex: padrão Consumidores Concorrentes / Competing Consumers).

A mensagem deverá ser necessariamente consumida por um destinatário, mas pode perder-se caso sua validade expire (Message Expiration), ou não seja persistente e o sistema saia do ar. Um canal pode implementar o padrão Entrega Garantida (Guaranteed Delivery) para garantir a entrega nessas situações.

## Canal Ponto-a-Ponto em Java (JMS)

Em JMS a interface Queue representa um canal ponto-a-ponto, que deve ser previamente configurado em um sistema de mensageria (ex: ActiveMQ). Em JMS 1.1 recomenda-se usar a interface Destination (que é estendida por Queue e Topic). O canal é obtido através de lookup JNDI ou injeção de dependências.

O exemplo abaixo ilustra o uso de canais ponto-a-ponto em JMS. O Canal Ponto-a-Ponto foi previamente configurado em um servidor de Mensageria. Neste exemplo, usamos o ActiveMQ (veja o apêndice com instruções de como configurar o ActiveMQ para os exemplos). No ActiveMQ criamos um Canal Ponto-a-Ponto configurando uma Queue chamada de test-queue. Depois, disponibilizamos o acesso a essa fila via JNDI através do nome simple-p2p-channel.

A classe MessageSender abaixo obtém via JNDI acesso a uma conexão JMS e à fila simple-p2p-channel. Inicia a conexão, cria uma sessão e cria um Produtor. Em seguida constrói dez mensagens e envia para o canal simple-p2p-channel.

```

01.  public class MessageSender {
02.      private ConnectionFactory factory;
03.      private Queue queue;
04.
05.      public void init() throws NamingException {
06.          Context ctx = new InitialContext();
07.          this.factory = (ConnectionFactory)ctx.lookup("ConnectionFactory");
08.          this.queue = (Queue)ctx.lookup("simple-p2p-channel");
09.      }
10.
11.      public void sendJms11() {
12.          try (Connection con = factory.createConnection()) {
13.              con.start();
14.              Session session =
15.                  con.createSession(false, Session.AUTO_ACKNOWLEDGE);
16.              MessageProducer producer = session.createProducer(queue);
17.
18.              System.out.println("Provider: "
19.                  + con.getMetaData().getJMSPublisherName() + " "
20.                  + con.getMetaData().getProviderVersion());
21.
22.              for(int i = 0; i < 10; i++) {
23.                  System.out.println("Sending message " + (i+1));
24.                  TextMessage message =
25.                      session.createTextMessage("Message number " + (i+1)
26.                                              + " sent " + new Date());
27.              }
28.          }
29.      }
30.  }

```

```

22.         producer.send(message);
23.     }
24.     System.out.println("All messages sent!");
25. } catch(JMSException e){
26.     System.err.println("JMS Exception: " + e);
27. }
28. }
29.

30. public static void main(String[] args) throws NamingException {
31.     MessageSender sender = new MessageSender();
32.     sender.init();
33.     sender.sendJms11();
34. }
35. }
```

A saída do programa deve ser:

```

Provider: ActiveMQ 5.11.1
Sending message 1
Sending message 2
Sending message 3
Sending message 4
Sending message 5
Sending message 6
Sending message 7
Sending message 8
Sending message 9
Sending message 10
All messages sent!
```

Se o servidor estiver no ar, as mensagens devem ser recebidas pela fila test-queue no ActiveMQ. Isto pode ser verificado pelo console em localhost:8161 que informa 10 mensagens que foram enfileiradas (enqueued) e pendentes (ainda não consumidas):

Name	Number Of Pending Messages	Number Of Consumers	Messages Enqueued	Messages Dequeued	Views	Operations
test-queue	10	0	10	0	Browse Active Consumers Active Producers <a href="#">atom</a> <a href="#">rss</a>	Send To Producers Purge Delete

Assim como MessageSender, classe MessageReceiver abaixo obtém via JNDI proxies para os mesmos objetos, uma conexão JMS e o canal simple-p2p-channel / test-queue, mas em vez de um Produtor, cria um Consumidor que irá para a fila consumir quantas mensagens estiverem disponíveis.

```

01.  public class MessageReceiver {
02.      private ConnectionFactory factory;
03.      private Queue queue;
04.      private String name;
05.      private long delay;
06.
07.      public MessageReceiver(String name, long delay) {
08.          this.name = name;
09.          this.delay = delay;
10.      }
11.
12.      public void init() throws NamingException {
13.          Context ctx = new InitialContext();
14.          this.factory = (ConnectionFactory)ctx.lookup("ConnectionFactory");
15.          this.queue = (Queue)ctx.lookup("simple-p2p-channel");
16.      }
17.
18.      public void receiveJms11() {
19.          try (Connection con = factory.createConnection()) {
20.              con.start();
21.              Session session =
22.                  con.createSession(false, Session.AUTO_ACKNOWLEDGE);
23.              MessageConsumer consumer = session.createConsumer(queue);
24.
25.              while(true) {
26.                  TextMessage message = (TextMessage)consumer.receive();
27.                  System.out.println(name + " received: " + message.getText());
28.                  Thread.sleep(delay);
29.              }
30.          } catch(JMSEException e){...}
31.
32.      public void run(Executor thread) {
33.          thread.execute(new Runnable() {
34.              public void run() {
35.                  receiveJms11();
36.              }
37.          });
38.      }
39.  }

```

Cada mensagem só pode ser consumida uma única vez. Para demonstrar isto executamos dois Consumidores que irão extraírem mensagens do canal ao mesmo tempo, interrompendo cada thread em tempos diferentes para evitar que o primeiro leve todas as mensagens.

```

01.  public static void main(String[] args) throws NamingException {
02.      Executor thread = Executors.newFixedThreadPool(2);

```

```

03.
04.     System.out.println("Waiting for messages... (^C to cancel)");
05.
06.     MessageReceiver receiver1 = new MessageReceiver("Receiver 1", 500);
07.     receiver1.init();
08.     receiver1.run(thread);
09.
10.    MessageReceiver receiver2 = new MessageReceiver("Receiver 2", 1000);
11.    receiver2.init();
12.    receiver2.run(thread);
13. }
14. ...

```

Executando a aplicação, podemos ver que cada uma das 10 mensagens foram consumidas uma única vez, distribuídas de forma desigual para cada consumidor:

```

Waiting for messages... (^C to cancel)
Receiver 1 received: Message number 2 sent Thu Sep 24 10:18:47 BRT 2015
Receiver 2 received: Message number 1 sent Thu Sep 24 10:18:47 BRT 2015
Receiver 1 received: Message number 4 sent Thu Sep 24 10:18:47 BRT 2015
Receiver 2 received: Message number 3 sent Thu Sep 24 10:18:47 BRT 2015
Receiver 1 received: Message number 6 sent Thu Sep 24 10:18:47 BRT 2015
Receiver 1 received: Message number 8 sent Thu Sep 24 10:18:47 BRT 2015
Receiver 2 received: Message number 5 sent Thu Sep 24 10:18:47 BRT 2015
Receiver 1 received: Message number 10 sent Thu Sep 24 10:18:47 BRT 2015
Receiver 2 received: Message number 7 sent Thu Sep 24 10:18:47 BRT 2015
Receiver 2 received: Message number 9 sent Thu Sep 24 10:18:47 BRT 2015

```

No console do ActiveMQ agora podemos ver que 10 mensagens foram removidas da fila (dequeued) e que não há mais mensagens pendentes. O painel também informa que há dois consumidores ativos. O serviço continua no ar.

Name	Number Of Pending Messages	Number Of Consumers	Messages Enqueued	Messages Dequeued	Views	Operations
test-queue	0	2	10	10		<a href="#">Send To</a> <a href="#">Purge</a> <a href="#">Delete</a>

Se novas mensagens chegarem, elas serão consumidas. Executando o MessageSender novamente, as mensagens serão disputadas por Receiver 1 e Receiver 2 à medida em que chegarem até que a fila seja esvaziada mais uma vez. Receiver 1 e Receiver 2 são um exemplo de Consumidores Concorrentes (Competing Consumers) e Consumidor de Sondagem (Polling Consumer).

## Canal Ponto-a-Ponto em Camel

Camel não possui uma implementação explícita deste padrão. O comportamento ponto-a-ponto depende da escolha do Endpoint usado. Para construir uma rota com canais ponto-a-ponto usa-se normalmente um endpoint JMS ou um canal nativo do Camel (componente Direct, SEDA ou VM). O comportamento ponto-a-ponto é obtido conectando uma origem (from) a um único destino (to). No exemplo abaixo, a mesma origem tem três possíveis destinos, e as mensagens enviadas para test-queue serão distribuídas entre eles:

```

01.          from("jms:queue:test-queue")
02.              .to("direct:a");
03.
04.          from("jms:queue:test-queue")
05.              .to("seda:b");
06.
07.          from("jms:queue:test-queue")
08.              .to("vm:c");

```

Através da integração Camel-JMS um canal Ponto-a-Ponto pode ser obtido através de um Endpoint (URI) que faça referência a um JMS Queue previamente configurado. Por exemplo: "jms:queue:nome-da-fila" ou "jms:nome-da-fila". Por exemplo, para disputar o canal com os receptores JMS do exemplo anterior, poderíamos usar a classe abaixo:

```

01.  public class ChannelReceiver {
02.      public static void main(String[] args) throws Exception {
03.          CamelContext context = new DefaultCamelContext();
04.          ConnectionFactory cf =
05.              new ActiveMQConnectionFactory(Configuration.ACTIVEMQ_URL);
06.          context.addComponent("jms", JmsComponent.jmsComponentAutoAcknowledge(cf));
07.
08.          context.addRoutes(new RouteBuilder() {
09.              public void configure() throws Exception {
10.                  from("jms:test-queue")
11.                      .delay(1000)
12.                      .process(new Processor() {
13.                          public void process(Exchange ex) {
14.                              System.out.println("Camel received: "
15.                                  + ex.getIn().getBody(String.class));
16.                          }
17.                      });
18.              context.start();
19.          });

```

```

20.         System.out.println("O servidor está no ar por 60 segundos.");
21.         Thread.sleep(60000);
22.         context.stop();
23.     }
24. }
```

Enviando 10 mensagens para a fina test-queue e executando os três receptores simultaneamente, obtivemos três mensagens recebidas pelo receptor Camel acima:

```

Camel received: Message number 3 sent Thu Sep 24 11:19:00 BRT 2015
Camel received: Message number 6 sent Thu Sep 24 11:19:00 BRT 2015
Camel received: Message number 9 sent Thu Sep 24 11:19:00 BRT 2015
```

As restantes foram consumidas pelos reeceptores JMS:

```

Receiver 1 received: Message number 1 sent Thu Sep 24 11:19:00 BRT 2015
Receiver 1 received: Message number 4 sent Thu Sep 24 11:19:00 BRT 2015
Receiver 1 received: Message number 7 sent Thu Sep 24 11:19:00 BRT 2015
Receiver 1 received: Message number 10 sent Thu Sep 24 11:19:00 BRT 2015
Receiver 2 received: Message number 5 sent Thu Sep 24 11:19:00 BRT 2015
Receiver 2 received: Message number 2 sent Thu Sep 24 11:19:00 BRT 2015
Receiver 2 received: Message number 8 sent Thu Sep 24 11:19:00 BRT 2015
```

## Canal Ponto-a-Ponto em Spring Integration

O canal default em Spring Integration, representado em XML por <channel> funciona com comportamento ponto-a-ponto e recebimento assíncrono (subscribable). É implementado pela classe DirectChannel. Configurando <channel> como atributos e sub-elementos, é possível selecionar uma implementação diferente. Por exemplo, usando <queue/> dentro da declaração de <channel> transforma-o em um QueueChannel, que também tem semântica ponto-a-ponto mas funciona como uma pilha (tem uma ordem para inserção e retirada de mensagens, cujo algoritmo também pode ser configurado para funcionar com base em prioridade ou fifo).

Filas JMS requerem o uso de implementação própria ou adaptadores. O exemplo abaixo contém um adaptador JMS que sincroniza com a fila JMS do tipo Queue test-queue (o canal no Spring Integration é identificado como entrada). Cada mensagem recebida será consumida apenas uma vez. Como há três outros components registrados para consumir da fila entrada, eles disputarão as mensagens que serão distribuídas entre eles. O componente <service-activator> executa o único método do bean referenciado que recebe o payload da mensagem. O <stdout-channel-adapter> imprime a mensagem transformada em string na saída padrão.

```

<bean id="worker1"
      class="br.com.argonavis.eipcourse.channel.pubsub.spring.Worker">
    <constructor-arg value="Worker-1"/>
</bean>

<bean id="worker2"
      class="br.com.argonavis.eipcourse.channel.pubsub.spring.Worker">
    <constructor-arg value="Worker-2"/>
```

```

</bean>

<int-jms:channel id="entrada" queue-name="test-queue" message-driven="true"/>

<int:service-activator ref="worker1" input-channel="entrada" />
<int:service-activator ref="worker2" input-channel="entrada" />

<int-stream:stdout-channel-adapter channel="entrada" append-newline="true" />

```

## (12) Canal de Difusão (Publish-Subscribe Channel)

### Ícone



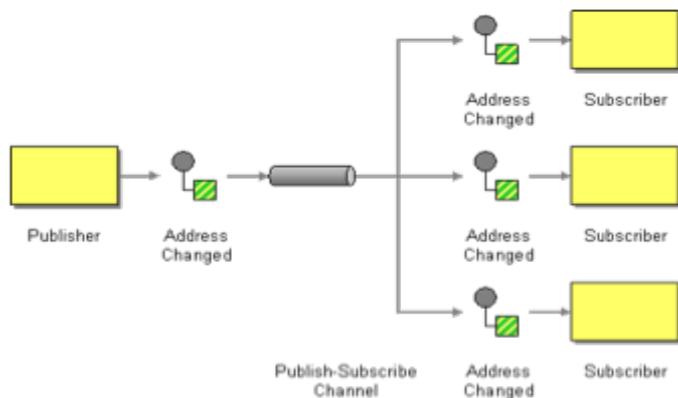
### Problema

“Como pode um remetente transmitir um evento a todos os destinatários interessados?”

### Solução

“Envie o evento para um Canal de Difusão (Publish-Subscribe Channel), que entrega uma cópia do evento a cada destinatário.”

### Diagrama



### Descrição

Diferentemente do canal ponto-a-ponto, o canal de difusão envia a mensagem para zero ou mais destinatários. Os destinatários precisam estar previamente associados ao canal como assinantes, ou bloqueando o thread para receber a mensagem de forma síncrona.

É mais comum que os assinantes registrem-se para serem notificados quando as mensagens estiverem disponíveis (padrão Observer). Se um canal tem 10 assinantes, 10 cópias da mensagem serão enviadas,

uma para cada assinante. Se não houver assinantes, nenhuma mensagem enviada ao canal será preservada. A mensagem é consumida por cada assinante apenas uma vez.

## Aplicações

Situações onde mensagens precisam ser enviadas para vários destinatários, por exemplo, a difusão de notificações para informar os assinantes sobre a ocorrência de determinado evento. É também útil para debugging, pois o debugger pode simplesmente registrar-se como um novo assinante.

Pode-se usar canais de difusão em situações onde um canal ponto-a-ponto seria suficiente, mas não necessário, e ter a flexibilidade de plugar assinantes extras para monitoração, debugging, extensão, etc.

### Canal de Difusão em Java (JMS)

Em JMS a interface Topic representa um canal publicar-inscrever, que deve ser previamente configurado em um sistema de mensageria (ex: ActiveMQ). Em JMS 1.1 recomenda-se usar a interface Destination (que é estendida por Queue e Topic). O canal é obtido através de lookup JNDI ou injeção de dependências.

O exemplo abaixo ilustra o uso de canais de difusão em JMS. O Canal foi previamente configurado em um servidor de Mensageria. Neste exemplo, usamos o ActiveMQ (veja o apêndice com instruções de como configurar o ActiveMQ para os exemplos). No ActiveMQ criamos um Canal de Difusão configurando uma Topic chamada de test-topic. Depois, disponibilizamos o acesso a essa fila via JNDI através do nome simple-ps-channel.

A classe MessagePublisher abaixo obtém via JNDI acesso a uma conexão JMS e à fila simple-ps-channel. Inicia a conexão, cria uma sessão e cria um Produtor. Em seguida constrói dez mensagens e envia para o canal simple-ps-channel.

```
01.  public class MessagePublisher {  
02.      private ConnectionFactory factory;  
03.      private Topic topic;  
04.  
05.      public void init() throws NamingException {  
06.          Context ctx = new InitialContext();  
07.          this.factory = (ConnectionFactory) ctx.lookup("ConnectionFactory");  
08.          this.topic = (Topic) ctx.lookup("simple-ps-channel");  
09.      }  
10.  
11.      public void sendJms11() {  
12.          try (Connection con = factory.createConnection()) {  
13.              con.start();  
14.              Session session =  
15.                  con.createSession(false, Session.AUTO_ACKNOWLEDGE);  
16.              MessageProducer producer = session.createProducer(topic);  
17.              System.out.println("Provider: "  
18.                  + con.getMetaData().getJMSPublisherName() + " ")  
19.          }  
20.      }  
21.  }
```

```

19.             + con.getMetaData().getProviderVersion());
20.
21.         for (int i = 0; i < 10; i++) {
22.             System.out.println("Sending message " + (i + 1));
23.             TextMessage message =
24.                 session.createTextMessage("Message number " + (i + 1)
25.                                         + " sent " + new Date());
26.             producer.send(message);
27.         }
28.     }
29.     System.out.println("All messages sent!");
30. } catch (JMSEException e) {
31.     System.err.println("JMS Exception: " + e);
32. }
33. public static void main(String[] args) throws NamingException {
34.     MessagePublisher publisher = new MessagePublisher();
35.     publisher.init();
36.     publisher.sendJms11();
37. }
38. }
```

Se rodarmos o programa, 10 mensagens serão enviadas mas como não há nenhum assinante cadastrado para receber, elas serão todas perdidas. Então precisamos primeiro rodar um programa que crie e registre assinantes ao canal. Uma maneira de registrar um assinante é simplesmente executar receive() e esperar as mensagens chegarem.

```

01. public class MessageSubscriber {
02.
03.     private ConnectionFactory factory;
04.     private Topic topic;
05.     private String name;
06.     private long delay;
07.
08.     public MessageSubscriber(String name, long delay) {
09.         this.name = name;
10.         this.delay = delay;
11.     }
12.
13.     public void init() throws NamingException {
14.         Context ctx = new InitialContext();
15.         this.factory =
16.             (ConnectionFactory)ctx.lookup("ConnectionFactory");
17.         this.topic = (Topic)ctx.lookup("simple-ps-channel");
18.     }
19.     public void receiveJms11() {
20.         try (Connection con = factory.createConnection()) {
```

```

21.         con.start();
22.         Session session =
23.             con.createSession(false, Session.AUTO_ACKNOWLEDGE);
24.         MessageConsumer consumer = session.createConsumer(topic);
25.
26.         while(true) {
27.             TextMessage message = (TextMessage)consumer.receive();
28.             System.out.println(name + " received: " + message.getText());
29.             Thread.sleep(delay);
30.         } catch(JMSEException e){
31.             System.err.println("JMS Exception: " + e);
32.         } catch(InterruptedException e) {
33.             e.printStackTrace();
34.         }
35.     }
36.
37.     public void run(Executor thread) {
38.         thread.execute(new Runnable() {
39.             public void run() {
40.                 receiveJms11();
41.             }
42.         });
43.     }
44.
45.     public static void main(String[] args) throws NamingException {
46.         Executor thread = Executors.newFixedThreadPool(2);
47.         System.out.println("Waiting for messages... (^C to cancel)");
48.         MessageSubscriber subscriber1 =
49.             new MessageSubscriber("Subscriber 1", 500);
50.         subscriber1.init();
51.         subscriber1.run(thread);
52.         MessageSubscriber subscriber2 =
53.             new MessageSubscriber("Subscriber 2", 1000);
54.         subscriber2.init();
55.         subscriber2.run(thread);
56.     }

```

O programa irá esperar o envio das mensagens:

Waiting for messages... (^C to cancel)

Na console do ActiveMQ podemos verificar que há dois consumidores registrados:

## Topics

Name	Number Of Consumers	Messages Enqueued	Messages Dequeued	Operations
test-topic	2	0	0	Send To Active Subscribers Active Producers Delete

Executando agora o MessagePublisher, dez mensagens serão enviadas. A saída do MessageSubscriber revela que cada assinante recebeu uma cópia das 10 mensagens enviadas:

```
Subscriber 1 received: Message number 1 sent Thu Sep 24 11:41:16 BRT 2015
Subscriber 2 received: Message number 1 sent Thu Sep 24 11:41:16 BRT 2015
Subscriber 1 received: Message number 2 sent Thu Sep 24 11:41:16 BRT 2015
Subscriber 2 received: Message number 2 sent Thu Sep 24 11:41:16 BRT 2015
Subscriber 1 received: Message number 3 sent Thu Sep 24 11:41:16 BRT 2015
Subscriber 1 received: Message number 4 sent Thu Sep 24 11:41:16 BRT 2015
Subscriber 2 received: Message number 3 sent Thu Sep 24 11:41:16 BRT 2015
Subscriber 1 received: Message number 5 sent Thu Sep 24 11:41:16 BRT 2015
Subscriber 1 received: Message number 6 sent Thu Sep 24 11:41:16 BRT 2015
Subscriber 2 received: Message number 4 sent Thu Sep 24 11:41:16 BRT 2015
Subscriber 1 received: Message number 7 sent Thu Sep 24 11:41:16 BRT 2015
Subscriber 1 received: Message number 8 sent Thu Sep 24 11:41:16 BRT 2015
Subscriber 2 received: Message number 5 sent Thu Sep 24 11:41:16 BRT 2015
Subscriber 1 received: Message number 9 sent Thu Sep 24 11:41:16 BRT 2015
Subscriber 1 received: Message number 10 sent Thu Sep 24 11:41:16 BRT 2015
Subscriber 2 received: Message number 6 sent Thu Sep 24 11:41:16 BRT 2015
Subscriber 2 received: Message number 7 sent Thu Sep 24 11:41:16 BRT 2015
Subscriber 2 received: Message number 8 sent Thu Sep 24 11:41:16 BRT 2015
Subscriber 2 received: Message number 9 sent Thu Sep 24 11:41:16 BRT 2015
Subscriber 2 received: Message number 10 sent Thu Sep 24 11:41:16 BRT 2015
```

E na console do ActiveMQ vemos que 10 mensagens foram enfileiradas, e 20 enviadas:

Name	Number Of Consumers	Messages Enqueued	Messages Dequeued	Operations
test-topic	2	10	20	Send To Active Subscribers Active Producers Delete

- Queue Views
  - Graph
  - XML
- Topic Views
  - XML
- Subscribers Views

## Canal de Difusão em Apache Camel

Camel não implementa canais explicitamente. Eles são componentes referenciados através de URIs. Mas uma arquitetura de difusão pode ser implementada com quaisquer tipos de canal ou endpoint (Direct, SEDA, VM, JMS) usando Multicast. O exemplo abaixo envia cópias das mensagens que chegam no JMS em test-queue para direct:x, direct:y e direct:z:

```
from("jms:queue:test-queue")
    .multicast()
    .to("direct:x", "direct:y", "direct:z");
```

Mas pode-se integrar Camel com JMS e obter um canal de Difusão verdadeiro através do Endpoint JMS e uma URI da forma “jms:topic:nome-do-canal”.

Como exemplo, podemos usar o mesmo código que foi usado no exemplo do Canal Ponto-a-Ponto para receber mensagens de um canal de Difusão. A única diferença será a URI que deve apontar para um Topic. Portanto, no nosso exemplo basta modificar a linha

```
from("jms:queue:test-queue")
```

e trocar por

```
from("jms:topic:test-topic")
```

para que a classe agora passe a receber mensagens de um Topic. Executando os assinantes JMS mais o assinante Camel, a console do ActiveMQ mostra que há 3 consumidores. A execução irá distribuir 30 mensagens (3 cópias de cada mensagem) de forma que cada um dos 3 assinantes receberão uma cópia.

## Canal de Difusão em Spring Integration

A implementação PublishSubscribeChannel em Spring Integration permite que consumidores interessados se inscrevam para receberem mensagens como notificação quando elas chegarem no canal. No exemplo abaixo, mensagens recebidas por uma fila JMS foram redirecionadas para <publish-subscribe-channel>. A notificação enviará notificações para dois components <service-activator> (que estão inscritos na fila) e para um component <stdout-channel-adapter>. Cada um deles receberá uma cópia de cada mensagem recebida pelo canal notificacoes:

```
<bean id="worker1"
      class="br.com.argonavis.eipcourse.channel.pubsub.spring.Worker">
      <constructor-arg value="Worker-1"/>
</bean>

<bean id="worker2"
      class="br.com.argonavis.eipcourse.channel.pubsub.spring.Worker">
      <constructor-arg value="Worker-2"/>
</bean>

<int-jms:publish-subscribe-channel id="notifications" topic-name="test-topic" />
```

```

<int:service-activator ref="worker1" input-channel="notifications" />
<int:service-activator ref="worker2" input-channel="notifications" />

<int-stream:stdout-channel-adapter channel="notifications"
                                         append-newline="true" />

```

## (13) Canal de Tipo de Dados (Datatype Channel)

### Ícone



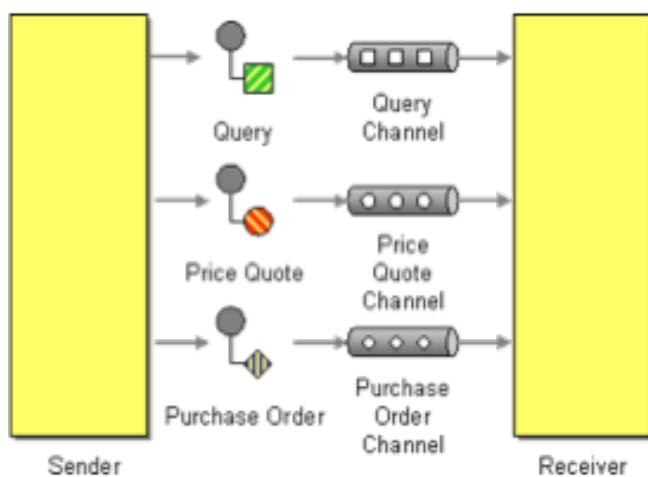
### Problema

“Como pode uma aplicação enviar um ítem de dados de tal maneira que o destinatário seja capaz de processá-lo?”

### Solução

“Use um Canal de Tipo de Dados (Datatype Channel) separado para cada tipo, para que todas as mensagens enviadas nesse canal contenham o mesmo tipo”

### Diagrama



### Descrição

Um Datatype Channel é um canal que contém apenas mensagens de determinado tipo. O “tipo de dados” de uma mensagem é um conceito amplo. Pode representar um formato padrão para dados (XML, imagem, texto, etc.) ou uma definição de tipo de dados específica ao domínio da aplicação (é um Pedido, Consulta, Cancelamento, Resposta, Comando, etc.). O tipo pode ser conhecido analisando o conteúdo da mensagem ou, mais eficientemente, através de meta-informação presente no cabeçalho. Pode ser uma definição formal (ex: um XML Schema), e pode depender de dados adicionais (ex: versão).

O padrão não estabelece *como* um canal garante conter apenas mensagens do tipo especificado. Normalmente isso é determinado pelos remetentes. Um Datatype Channel para mensagens que contém dados do tipo “Pedido” pode ser simplesmente um canal chamado “pedidos”. Deve-se implementar um mecanismo que impeça que mensagens de outro tipo sejam enviados ao canal. Normalmente usa-se um Roteador Baseado em Conteúdo para distribuir mensagens recebidas em um canal genérico para Datatype Channels correspondentes ao seu tipo. Se as mensagens tiverem algum tipo de Indicador de Formato - um ou mais cabeçalhos contendo informações sobre o conteúdo da mensagem, estes podem ser usados pelo roteador para detectar o tipo e redirecionar as mensagens ao canal adequado.

## Aplicações

Um Datatype Channel simplifica a aplicação ao agrupar mensagens de mesmo tipo em um único canal.

### Canal de Tipo de Dados em Java (JMS)

Um Canal de Tipo de Dados é um canal conceitual. Pode ser tanto um Canal Ponto-a-Ponto como um Canal Publica-Inscreve. O que o distingue é apenas o tipo de dados das mensagens que contém. O critério usado para distinguir tipos diferentes de mensagens é arbitrário e faz parte do domínio da aplicação. Uma aplicação ou parte da aplicação pode precisar distinguir documentos JSON, XML e CSV, outra pode tratar diferentes esquemas de XML como tipos diferentes, por exemplo um XML que representa um pedido, e outro que representa um cancelamento. JMS não oferece nenhum mecanismo para restringir tipos de dados em canais, mas é possível filtrar mensagens através da análise de seus cabeçalhos e conteúdos e impedir que mensagens que não sejam de determinado tipo sejam recebidas em um canal.

A implementação desse padrão depende dos componentes que enviam mensagens ao canal. Um filtro ou um roteador de mensagens pode ser usado para essa finalidade. No exemplo abaixo, um Roteador Baseado em Conteúdo (Content-Based Router) é usado para separar tipos diferentes de mensagens em canais distintos. Esses canais são, conceitualmente, Canais de Tipo de Dados. No nosso exemplo, o tipo de mensagem será determinado por uma propriedade de cabeçalho “data-type” que pode conter os valores “type-1” ou “type-2”. Se nenhum outro código tiver acesso aos canais, pode-se garantir que os canais conterão apenas mensagens do mesmo tipo.

O trecho de código abaixo (veja a listagem completa na classe SimpleHeaderTypeRouter.java) ilustra como Canais de Tipo de Dados (Datatype Channels) podem ser implementados em Java/JMS:

```
01.  public class SimpleHeaderTypeRouter implements MessageListener {  
02.      Destination datatypeChannel1;  
03.      Destination datatypeChannel2;  
04.      ...  
05.  
06.      @Override  
07.      public void onMessage(Message message) {  
08.          String type = message.getStringProperty("data-type");  
09.          Destination destination;  
10.          if (type != null && type.equals("type-1")) {  
11.              destination = datatypeChannel1;
```

```

12.         } else if (type != null && type.equals("type-2")) {
13.             destination = datatypeChannel2;
14.         }
15.         ...
16.         MessageProducer producer = session.createProducer(destination);
17.         producer.send(message);
18.     }
19. }
```

Por default, nada impede que outra aplicação use os mesmos canais e envie mensagens que não são do tipo permitido para o canal. Para garantir a implementação do Canal de Tipo de Dados, deve-se garantir que nenhuma aplicação consiga enviar mensagens do tipo incorreto para um canal.

## Canal de Tipo de Dados em Apache Camel

Por ser um padrão conceitual, O Canal de Tipo de Dados também não é implementado em Camel explicitamente. Se um Canal de Tipo de Dados é indicado em um diagrama de integração, a implementação Camel deve garantir que esse canal só receba dados do tipo especificado através de outros componentes, como roteadores e tradutores que depositam mensagens no canal, assim como foi mostrado em JMS.

Na rota Camel exemplo abaixo, um Roteador Baseado em Conteúdo (Content-Based Router) é usado para distribuir diferentes tipos de mensagens em canais exclusivos para cada tipo. Neste exemplo uma propriedade de cabeçalho chamada “filename” foi usada, e ela contém o nome de um arquivo. Mensagens que têm nomes de arquivo que terminam em jpg, png, gif ou jpeg são colocadas em um Canal de Tipo de Dados que aceita imagens, e as mensagens que tem arquivos terminados em xml vão para uma fila de documentos XML. Mensagens de outros tipos são descartadas

```

01.   from("jms:objetos")
02.     .choice()
03.       .when(header("filename").regex("^.*(jpg|png|gif|jpeg)$"))
04.         .to("jms:imagens") // Datatype Channel 1
05.       .when(header("filename").endsWith(".xml"))
06.         .to("jms:docsXML") // Datatype Channel 2
07.     .stop()
```

## Canal de Tipo de Dados em Spring Integration

Spring Integration difere de Camel e JMS porque permite que um Canal declare o tipo de dados que pode receber. A granularidade, porém, é de uma classe Java. Se o payload for um XML (java.lang.String) não há como distinguir um XML de outro a menos que se use um mecanismo como mapeamento Java-XML (ex: JAXB) que permite representar um esquema XML como uma classe java.

O tipo de dados é informado com o atributo datatype em qualquer tipo de <channel>. Esse atributo recebe como argumento o nome qualificado de uma ou mais classes. Por exemplo, se um canal aceita mensagens do tipo com.example.Suggestion e com.example.Complaint ele pode ser configurado como:

```
<int:channel datatype=" com.example.Suggestion, com.example.Complaint" />
```

e provocará uma exceção caso receba uma mensagem com um payload de tipo diferente.

## (14) Canal de Mensagens Inválidas (Invalid Message Channel)

### Ícone



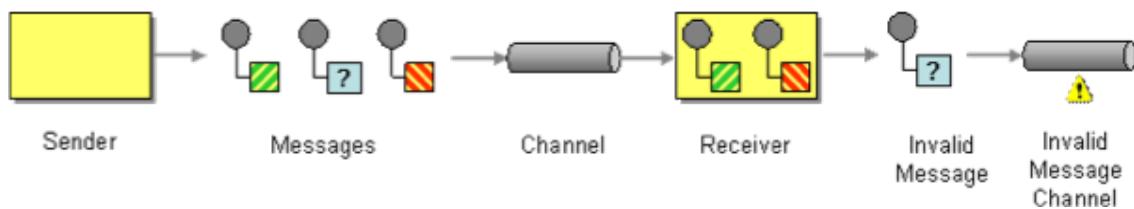
### Problema

“Como pode um destinatário de mensageria lidar com o recebimento de uma mensagem que não faz sentido?”

### Solução

“O receptor deve mover a mensagem imprópria para um Invalid Message Channel, um canal especial para mensagens que não puderam ser processadas por seus receptores.”

### Diagrama



### Descrição

Este pode ser considerado um caso especial de Datatype Channel. Um canal que recebe uma mensagem que é rejeitado por todos os outros canais, por ser inválida. Pode ser uma mensagem que não possui um tipo suportado, ou que esteja inconsistente de alguma forma (cabecalhos faltando, conteúdo corrompido, etc.), ou que não passe em uma validação (ex: XML Schema). Assim como Datatype Channel, a definição do que é uma mensagem inválida faz parte do domínio da aplicação.

A validade da mensagem pode estar relacionada com o canal ao qual ela se destina. Uma mensagem pode ser válida para um canal e não ser válida para outro. Um Invalid Message Channel oferece a oportunidade de reaproveitar mensagens em canais onde elas sejam válidas.

A validade pode ocorrer em vários níveis. Por exemplo, um documento é XML bem-formado e tem sua validade testada contra um XML Schema, e passa. É válido. Mas se esses dados válidos posteriormente causarem um erro devido a uma restrição do domínio da aplicação, é um erro de aplicação, e a mensagem pode ser considerada inválida.

Normalmente Invalid-Message Channel é usado para mensagens que contém *dados inválidos*, seja em cabeçalhos ou no corpo. Há cabeçalhos que não se referem aos dados contidos na mensagem, mas são usados pelo sistema de mensageria para rotear mensagens. Esses cabeçalhos podem impedir a entrega

da mensagem. Por exemplo: o prazo de validade de uma mensagem (Message Expiration), se ultrapassado, torna uma mensagem inválida e impossível de ser entregue. Nesse caso a mensagem seria redirecionada (normalmente pelo sistema) para um Canal de Mensagens Não-Entregues (Dead-Letter Channel).

## Aplicações

Log de erros e exceções. Pode-se usar mais de um canal de mensagens inválidas em uma aplicação, para lidar com diferentes tipos de erros e exceções.

### Canal de Mensagens Inválidas em Java (JMS)

Assim como o Canal de Tipo de Dados, este é um padrão conceitual. Pode ser qualquer canal escolhido para receber as mensagens que não podem ser processadas. A forma de implementar depende de como uma mensagem é definida como “inválida” no domínio de uma aplicação., por exemplo, se pode ser válida em outro contexto, se pode ser consertada ou se é erro ou exceção.

O exemplo abaixo é uma adaptação do exemplo mostrado para Canal de Tipo de Dados, onde o Roteador Baseado em Conteúdo em vez de descartar as mensagens que não são “tipo-1” ou “tipo-2”, as redireciona para um canal especialmente configurado para receber mensagens inválidas:

```

01.  public class SimpleHeaderTypeRouter implements MessageListener {
02.      Destination datatypeChannel1;
03.      Destination datatypeChannel2;
04.      Destination invalidMessageChannel;
05.      ...
06.
07.      @Override
08.      public void onMessage(Message message) {
09.          String type = message.getStringProperty("data-type");
10.          Destination destination;
11.          if (type != null && type.equals("type-1")) {
12.              destination = datatypeChannel1;
13.          } else if (type != null && type.equals("type-2")) {
14.              destination = datatypeChannel2;
15.          } else {
16.              destination = invalidMessageChannel;
17.          }
18.          ...
19.          MessageProducer producer = session.createProducer(destination);
20.          producer.send(message);
21.      }
22.  }
```

Se as mensagens inválidas forem mensagens cujo processamento provoque erros ou exceções, o Canal de Mensagens Inválidas provavelmente será usado dentro de um bloco catch:

```

01.  public void onMessage(Message message) {
02.      try {
```

```

03.         process(message);
04.     } catch (JMSEException e) {
05.         ...
06.         MessageProducer producer =
07.             session.createProducer(invalidMessageChannel);
08.         producer.send(message);
09.     }

```

## Canal de Mensagens Inválidas em Apache Camel

Não há uma implementação de Canal de Mensagens Inválidas em Camel, já que é um canal conceitual como em JMS. Pode ser implementado como um canal qualquer e cabe ao restante da aplicação garantir que ele receba apenas mensagens inválidas.

Camel, porém, possui a interface ErrorHandler que pode ser configurada para processar mensagens inválidas, tentar corrigi-las e enviá-las de novo. Em caso de exceção, o Default Error Handler interrompe a transmissão e devolve a exceção ao cliente. Pode-se, como em Java, interceptar a exceção e tentar tratá-la.

No exemplo abaixo, se o XML não validar na etapa to("bean:validateXML") uma exceção irá ocorrer. Ela será marcada como *handled* (mesmo que usar um bloco catch) e transformada em um XML válido que indica o problema. Assim o fluxo não será interrompido e será concluído.

```

01.    onException(InvalidXMLException.class)
02.        .handled(true)
03.        .transform(body(constant("<message>Bad XML</message>")));
04.
05.    from("jms:queue:incoming")
06.        .to("bean:validateXML") // exceção ocorre aqui!
07.        .to("jms:queue:processing");

```

Mas isto apenas delega o problema para a fila de processamento, em vez de enviar para um Canal de Mensagens Inválidas. Uma solução melhor seria usar onException() em um RouteBuilder e redirecionar a outros canal se houver exceção. Nesse caso, pode-se criar receptores dedicados que consomem mensagens desses canais:

```

01.    onException(InvalidXMLException.class).
02.        to("jms:queue:badxml");
03.
04.    onException(OrderException.class).
05.        to("jms:queue:badorder");
06.
07.    from("jms:queue:incoming").to("jms:queue:processing");

```

## Canal de Mensagens Inválidas em Spring Integration

Spring Integration também não possui uma implementação explícita deste padrão, mas assim como em Camel, há componentes para capturar e tratar exceções, como o <exception-type-router>, que pode ser configurado para processar e rotear mensagens inválidas:

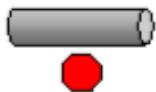
```

01. <int:exception-type-router input-channel="invalidas"
02.      default-output-channel="nao-tratadas">
03.      <int:mapping exception-type="PedidoExpiradoException"
04.          channel="expiradas" />
05.      <int:mapping exception-type="PedidoIncompativelException"
06.          channel="canceladas" />
07.  </int:exception-type-router>

```

## (15) Canal de Mensagens Não-Entregues (Dead Letter Channel)

### Ícone



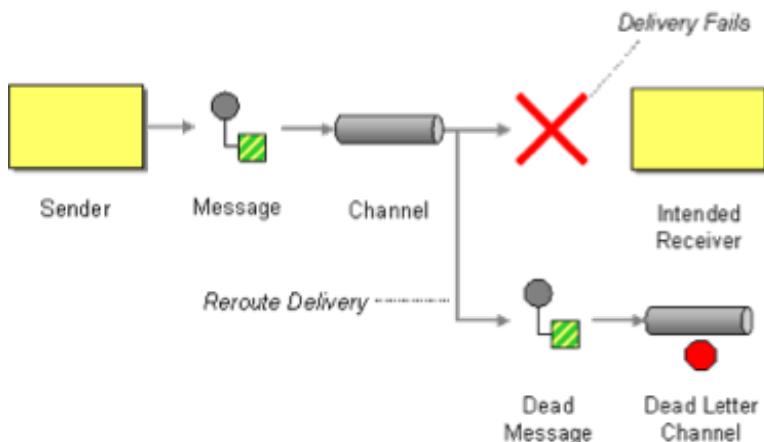
### Problema

“O que deve o sistema fazer com uma mensagem que ele não pode entregar”

### Solução

“Quando um sistema determina que não pode ou não deve entregar uma mensagem, poderá decidir mover a mensagem para um Dead Letter Channel”

### Diagrama



### Descrição

A entrega de mensagens é responsabilidade do sistema de mensageria, portanto o conteúdo da mensagem ou a validade dos seus dados não é relevante. O sistema de mensageria sabe quais as mensagens que não foram entregues (geralmente pode-se descobrir isto através de meta-informação no cabeçalho). Essas mensagens podem ser enviadas para um Dead Letter Channel que pode ser usado como log ou repositório temporário para tentativas futuras de envio.

Alguns motivos que podem levar uma mensagem a não ser entregue:

- O canal não foi configurado corretamente
- O canal pode não existir ou ter sido removido
- A mensagem pode expirar antes da entrega (Message Expiration)
- A mensagem foi ignorada por todos os destinatários em uma implementação de Consumidor Seletivo (Selective Consumer)
- A mensagem tem problemas no cabeçalho que impedem a sua entrega

## Aplicações

Logs de erros e exceções. Neste caso o canal irá guardar mensagens que não foram entregues, tenham elas dados válidos ou não.

Caixa de saída de mensagens. As mensagens recebidas em um Dead-Letter Channel podem ser monitoradas e consumidas por clientes capazes de re-empacotar seus dados e realizar novas tentativas de entrega.

## Canal de Mensagens Não-Entregues em Java/JMS

O Dead Letter Channel é dependente de implementação. É diferente no WebSphere/MQ, no Glassfish/OpenMQ, WildFly/HornetQ e ActiveMQ. Alguns possuem DLQs separados para mensagens vencidas e mensagens que não puderam ser entregues por outra razão, ou permitem configurar DLQs adicionais.

No ActiveMQ há uma fila padrão chamada ActiveMQ.DLQ para onde são enviadas todas as mensagens que não puderam ser entregues. É possível configurar quais mensagens serão armazenadas, por exemplo, não incluir as mensagens vencidas, determinar o tempo de permanência das mensagens na fila ou guardar mensagens não-persistentes. Pode-se também configurar DLQs extras.

Para demonstrar o uso do DLQ em ActiveMQ, criaremos uma mensagem persistente em JMS com prazo de validade curto para que vença antes que possa ser entregue. Isto causará o envio da mensagem ao DLQ. Em seguida, listaremos as mensagens que estão no DLQ. Abaixo um trecho da classe ExpiredMessageSender que tenta enviar 10 mensagens vencidas:

```
01. Session session = con.createSession(false, Session.AUTO_ACKNOWLEDGE);
02. MessageProducer producer = session.createProducer(queue);
03. producer.setDeliveryMode(javax.jms.DeliveryMode.PERSISTENT);
04. producer.setTimeToLive(500); // message lives 1/2 second
05.
06. for(int i = 0; i < 10; i++) {
07.     System.out.println("Sending message " + (i+1));
```

```

08.     TextMessage message =
        session.createTextMessage("Message " + (i+1)
                                + " sent " + new Date());
09.     Thread.sleep(1001);
10.     producer.send(message);
11. }
```

Na console do ActiveMQ elas aparecem na fila ActiveMQ.DLQ que é criada automaticamente:

Queues						
Name ↑	Number Of Pending Messages	Number Of Consumers	Messages Enqueued	Messages Dequeued	Views	Operations
ActiveMQ.DLQ	10	0	10	0	<a href="#">Browse Active Consumers</a> <a href="#">Active Producers</a>  	<a href="#">Send To</a> <a href="#">Purge</a> <a href="#">Delete</a>
test-queue	0	0	80	80	<a href="#">Browse Active Consumers</a> <a href="#">Active Producers</a>  	<a href="#">Send To</a> <a href="#">Purge</a> <a href="#">Delete</a>

O programa abaixo (ExpiredMessageReceiver) lê as mensagens da fila DLQ (a fila ActiveMQ.DLQ foi registrada em JNDI como “dead-queue”):

```

01. Destination queue = (Queue)ctx.lookup("dead-queue"); //ActiveMQ.DLQ
02. Connection con = ...;
03. Session session = ...;
04. MessageConsumer consumer = session.createConsumer(queue);
05.
06. while(true) {
07.     TextMessage message = (TextMessage)consumer.receive();
08.     System.out.println("DLQ: " + name + " received: " + message.getText());
09. }
```

## Canal de Mensagens Não-Entregues em Apache Camel e Spring

Apache Camel suporta Canal de Mensagens Não-Entregues através do processador org.apache.camel.processor.DefaultErrorHandler que é um ErrorHandler. O DefaultErrorHandler apenas realiza a propagação de exceções. O DeadLetterChannel permite tentar reenviar a mensagem e redirecionar.

Dentro de uma rota, a instrução a seguir configura uma fila como DLQ:

```
errorHandler(deadLetterChannel("jms:queue:dlq"))
```

Pode-se configurar o DLQ com número de tentativas de entrega e intervalo entre as tentativas. Neste caso a mensagem só será movida para a fila após esgotadas as tentativas.

```
errorHandler(deadLetterChannel("jms:queue:dlq"))
    .maximumRedeliveries(5)
    .redeliveryDelay(10000);
```

Pode-se tentar modificar a mensagem antes de re-enviar usando onRedelivery():

```

01. errorHandler(deadLetterChannel("jms:queue:dlq"))
02. .maximumRedeliveries(5)
03. .redeliveryDelay(10000)
04. .onRedelivery(new Processor() {
05.     public void process(Exchange e) {
06.         // alterar algo
07.     }
08. });

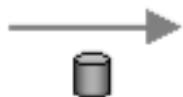
```

Por default a mensagem enviada para a DLQ é a mensagem que não pôde ser entregue. Esta mensagem pode conter algo que tenha impedido sua entrega. Adicionando `useOriginalMessage()` o sistema guardará no DLQ não a última mensagem, mas a mensagem que foi enviada para o início da rota:

```
errorHandler(deadLetterChannel("jms:queue:dlq")).useOriginalMessage();
```

## (16) Entrega Garantida (Guaranteed Delivery)

### Ícone



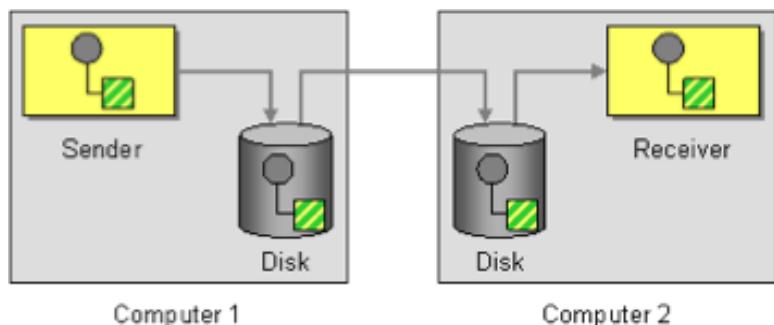
### Problema

“Como pode o remetente ter certeza que uma mensagem será entregue, mesmo que o sistema de mensageria falhe”

### Solução

“Use Entrega Garantida (Guaranteed Delivery) para fazer as mensagens persistentes para que elas não se percam, mesmo que o sistema saia do ar”

### Diagrama



### Descrição

Em canais ponto-a-ponto o sistema guarda a mensagem no canal até que seja consumida por um destinatário. Em canais publicar-inscrever as mensagens são guardadas até sejam consumidas por

todos os assinantes ativos ou duráveis (Durable Consumer). Mas se o sistema sair do ar, as mensagens são geralmente perdidas, a menos que a persistência de mensagens seja implementada por default.

Guaranteed Delivery é usado para mensagens que precisam ser entregues de qualquer maneira. Se o destinatário não estiver disponível, o sistema precisa guardar a mensagem e tentar entregar em outro momento. Para garantir esse funcionamento mesmo que o sistema saia do ar temporariamente, é preciso que cada computador participante do sistema de mensageria guarde uma cópia persistente da mensagem. O remetente grava uma cópia local da mensagem e a envia. Somente quando a mensagem é recebida com sucesso no outro repositório, sua cópia local será removida.

Guaranteed Delivery pode ser limitado por timeout (Message Expiration), limite de tentativas ou outros fatores. Nesse caso pode-se ainda preservar a mensagem enviando-a para outro canal (ex: Dead-Letter Channel).

## Aplicações

Quando for necessário garantir a comunicação entre sistemas que podem não estar disponíveis ao mesmo tempo.

Guaranteed Delivery tem um impacto na performance e aumenta o consumo de espaço em disco. Deve ser usado apenas quando for necessário, para mensagens que realmente precisam ser entregues (não deve ser habilitado como default para todas as mensagens, respostas, comandos, eventos, etc).

## Entrega Garantida em Java (JMS)

Em JMS pode-se configurar o produtor para que todas as mensagens enviadas por ele sejam armazenadas em meio persistente usando o método `setDeliveryMode()` com a opção `DeliveryMode.PERSISTENT`:

```
Session session = ...;
MessageProducer producer = session.createProducer(queue);
producer.setDeliveryMode(javax.jms.DeliveryMode.PERSISTENT);
```

Se, por algum motivo, apenas algumas mensagens devam ser persistentes, pode-se também determinar essa propriedade por mensagem, como parâmetro do método `send()`. Se o modo default estiver em `PERSISTENT`, pode-se desligar a persistência para um envio com:

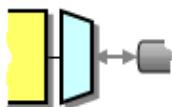
```
producer.send(message,
    javax.jms.DeliveryMode.NON_PERSISTENT,
    javax.jms.Message.DEFAULT_PRIORITY,
    javax.jms.Message.DEFAULT_TIME_TO_LIVE);
```

## Entrega Garantida em Apache Camel e Spring Integration

Entrega garantida em Camel e Spring Integration depende da integração com um sistema externo que garanta persistência. Componentes JPA e File existentes em Camel e Spring garantem persistência pela sua própria natureza, mas componentes de integração com JMS precisam ser configurados para enviar mensagens persistentes para garantir a entrega, como foi mostrado no exemplo JMS.

## (17) Adaptador de Canal (Channel Adapter)

### Ícone



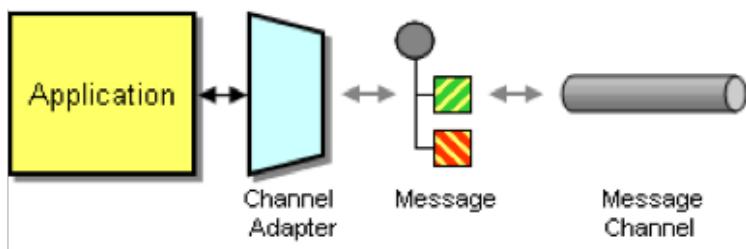
### Problema

“Como conectar uma aplicação ao sistema de mensageria para que ela possa enviar e receber mensagens?”

### Solução

“Use um Adaptador de Canal (Channel Adapter) que pode acessar a API da aplicação ou seus dados, e publique mensagens em um canal baseado nesses dados, e que possa receber mensagens e chamar funcionalidades dentro da aplicação”

### Diagrama



### Descrição

Adaptadores de Canal são unidirecionais. Existem dois tipos: *inbound*, ou de entrada, quando alimentam um canal com mensagens contendo dados obtidos de uma aplicação externa, e *outbound*, ou de saída, quando as mensagens de um canal de mensageria são usadas para fornecer dados para uma aplicação externa. Um Channel Adapter permite que uma aplicação externa participe de um sistema de mensageria, viabilizando a integração entre os sistemas.

Um Channel Adapter é construído a partir de um ponto de acesso. Embora a maior parte das aplicações não tenham sido construídas pensando na conectividade com uma infraestrutura de mensageria, elas possuem pontos de acesso como sockets, arquivos, registros em bancos de dados ou APIs que podem ser adaptadas para permitir a comunicação com o sistema de mensageria.

A adaptação de uma interface pode ser uma tarefa complexa. A função do Channel Adapter é encapsular essa complexidade. Se a aplicação a ser adaptada fornecer uma API, o cabe ao adaptador usar essa API extrair ou fornecer dados usados na construção ou leitura de mensagens enviadas ou recebidas do canal. Quando não existe uma API pode-se usar outros mecanismos, como notificações e eventos do sistema operacional, triggers de bancos de dados, técnicas como “screen scraping”, monitoração de rede e sistema de arquivos.

Muitas vezes as mensagens obtidas de um Channel Adapter requerem transformações adicionais para que possam ser usados pela aplicação. Um Message Translator pode ser usado para converter os dados em um formato compatível com um Canonical Data Model.

Um par de Channel Adapters (inbound e outbound) operando de forma coordenada é uma implementação do padrão Messaging Gateway.

## Aplicações

O Channel Adapter é a principal forma de comunicação de um sistema de mensageria com o mundo externo. Frameworks como Spring Integration, Mule e Camel oferecem vários adaptadores prontos que podem ser usados para comunicação com serviços como sistema de arquivos, sockets, bancos de dados, email, Twitter, etc.

### Adaptador de Canal em Java (JMS)

O adaptador de canal abaixo é um programa que monitora uma pasta periodicamente. Sempre que um arquivo com extensão .xml é colocado na pasta, ele abre o arquivo, extrai o texto e encapsula em uma mensagem e envia para o canal “inbound-channel”:

```
01. public class FileAdapter {
02.     private File directory;
03.     public FileAdapter(File directory) throws JMSEException {
04.         this.directory = directory;
05.     }
```

O método send() envia uma lista de mensagens através de um produtor.

```
06.     public void send(List<Message> messages, MessageProducer producer) {
07.         try {
08.             for(Message message: messages) {
09.                 System.out.println("Sending message");
10.                 producer.send(message);
11.             }
12.             System.out.println(messages.size() + " messages sent!");
13.         } catch(JMSEException e){
14.             System.err.println("JMS Exception: " + e);
15.         }
16.     }
```

Cada mensagem é criada a partir de uma lista de arquivos. Depois que uma mensagem é criada, o arquivo é apagado.

```
17.     public List<Message> createMessages(Session session, List<File> files)
18.                                         throws JMSEException {
19.         List<Message> messages = new ArrayList<>();
20.         TextMessage message = session.createTextMessage();
21.         for(File file: files) {
22.             String xmlDocument = readContents(file);
23.             if(xmlDocument != null) {
```

```

23.         message.setStringProperty("Length",(""+file.length()));
24.         message.setText(xmlDocument);
25.         messages.add(message);
26.         file.delete();
27.     }
28. }
29. return messages;
30. }
```

Os arquivos são documentos XML depositados em uma pasta específica (directory). Eles serão lidos se tiverem extensão “.xml”.

```

31.     public List<File> loadFiles() {
32.         List<File> files = new ArrayList<>();
33.         String[] fileNames = directory.list(new FilenameFilter() {
34.             @Override
35.             public boolean accept(File dir, String name) {
36.                 return name.endsWith(".xml");
37.             }
38.         });
39.         if(fileNames != null && fileNames.length > 0) {
40.             for(String fileName : fileNames) {
41.                 files.add(new File(directory, fileName));
42.             }
43.         }
44.         return files;
45.     }
```

O método readContents extrai o texto de cada arquivo:

```

46.     public String readContents(File file) {
47.         Reader reader = null;
48.         Writer writer = null;
49.         try {
50.             reader = new FileReader(file);
51.             writer = new StringWriter();
52.             char[] buffer = new char[4096];
53.             int len = reader.read(buffer);
54.             while(len > 0) {
55.                 writer.write(buffer, 0, len);
56.                 len = reader.read(buffer);
57.             }
58.             writer.flush();
59.             return writer.toString();
60.         } catch (IOException e) {
61.             e.printStackTrace();
62.             return null;
63.         } finally {
64.             try {
65.                 reader.close();
```

```

66.         writer.close();
67.     } catch (IOException e) { }
68.   }
69. }
```

Finalmente, o método main() abaixo checa 12 vezes a pasta /tmp/inbox/jms. Quaisquer arquivos depositados na pasta serão abertos, terão o conteúdo extraído e usado para formar uma mensagem enviada para a fila. Depois os arquivos serão apagados.

```

70. public static void main(String[] args) throws Exception {
71.     FileAdapter adapter = new FileAdapter(new File("/tmp/jms/inbox"));
72.     Context ctx = new InitialContext();
73.     ConnectionFactory factory =
74.         (ConnectionFactory)ctx.lookup("ConnectionFactory");
75.     Destination queue = (Destination)ctx.lookup("inbound-channel");
76.     Connection con = factory.createConnection();
77.     con.start();
78.     int polls = 12;
79.     while(polls > 0) {
80.         System.out.println("Checking for files... "
81.             + polls + " polls left.");
82.         Session session =
83.             con.createSession(false, Session.AUTO_ACKNOWLEDGE);
84.         MessageProducer producer = session.createProducer(queue);
85.         List<File> files = adapter.loadFiles();
86.         if(!files.isEmpty()) {
87.             List<Message> messages = adapter.createMessages(session, files);
88.             adapter.send(messages, producer);
89.         }
90.     }
91. }
```

Executamos o adaptador, e depositamos primeiro três arquivos, depois outros dois:

```

Checking for files... 12 polls left.
Checking for files... 11 polls left.
Checking for files... 10 polls left.
Checking for files... 9 polls left.
Sending message
Sending message
Sending message
3 messages sent!
Checking for files... 8 polls left.
Checking for files... 7 polls left.
Sending message
Sending message
2 messages sent!
```

```
Checking for files... 6 polls left.
Checking for files... 5 polls left.
Checking for files... 4 polls left.
Checking for files... 3 polls left.
Checking for files... 2 polls left.
Checking for files... 1 polls left.
```

Ao final, a console do ActiveMQ revela 5 mensagens esperando no canal inbound-queue:

Queues						
Name ↑	Number Of Pending Messages	Number Of Consumers	Messages Enqueued	Messages Dequeued	Views	Operations
inbound-queue	5	0	5	0	<a href="#">Browse</a> <a href="#">Active Consumers</a> <a href="#">Active Producers</a> <a href="#">Send To Purge</a>  	<a href="#">Delete</a>

## Adaptador de Canal em Apache Camel

Apache Camel não tem uma implementação explícita desse padrão, mas realiza a mesma função através de componentes e endpoints, que são a principal forma de realizar a conexão do sistema de mensageria com recursos externos. A rota do exemplo mostrado em JMS pode ser implementada em Camel usando um Componente e Endpoint File:

```
01.   context.addRoutes(new RouteBuilder() {
02.     @Override
03.     public void configure() throws Exception {
04.       from("file:/tmp/jms/inbox")
05.         .to("jms:queue:inbound-queue");
06.     }
07.   });
```

## Adaptador de Canal em Spring Integration

Spring Integration possui várias implementações deste padrão. É a principal forma de comunicação com aplicações externas. O Channel Adapter do Spring não precisa conectar a um canal (ele já vem com um: o id do adaptador é o id do canal). Spring Integration 4.13 contém adaptadores inbound para as seguintes aplicações: Spring AMQP, Events, Feed, File, FTP(S), Gemfire, HTTP, JDBC, JMS, JMX, JPA, Mail, MongoDB, MQTT, Redis, Resource, SFTP, Stream, Syslog, TCP, Twitter, UDP, Web Sockets, XMPP, e outbound para todos exceto Feed, Resource e Syslog. Para usar um adaptador é necessário incluir o JAR no Classpath (tipicamente através da configuração de dependência do Maven/Ivy) e referenciar o namespace do componente no arquivo XML de contexto.

O exemplo abaixo usa dois adaptadores de canal. A entrada ocorre através de um adaptador inbound para arquivos, que sonda a cada 5 segundos uma pasta (/tmp/jms/inbox) para verificar se há arquivos. Encontrado o arquivo, ele é transformado em mensagem e enviado para um adaptador JMS, que transfere a mensagem para uma fila JMS.

```
01. <int-file:inbound-channel-adapter directory="/tmp/jms/inbox"
                                         channel="inbound">
```

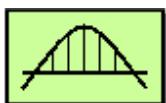
```

02.           <int:poller fixed-rate="5000" />
03.     </int-file:inbound-channel-adapter>
04.
05.     <int-jms:outbound-channel-adapter id="inbound"
06.       destination-name="inbound-queue"/>
06.     <int-stream:stdout-channel-adapter channel="inbound"
07.       append-newline="true" />

```

## (18) Ponte de Mensageria (Messaging Bridge)

### Ícone



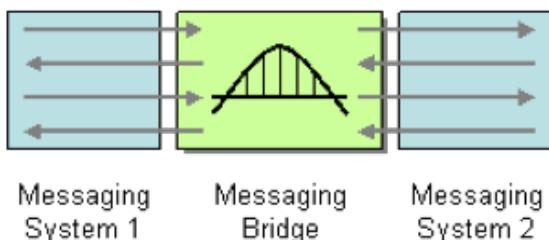
### Problema

“Como diferentes sistemas de mensageria podem se conectar para que mensagens disponíveis em um também sejam disponibilizados em outros?”

### Solução

“Use uma Ponte de Mensageria (Messaging Bridge), uma coleção entre diferentes sistemas de mensageria, para replicar mensagens entre sistemas”

### Diagrama



### Descrição

JMS pode ser usado para ter uma interface comum para um sistema de mensageria, mas se houver uma aplicação usando ActiveMQ, e outra usando IBM MQ, elas não irão se comunicar automaticamente porque os formatos de mensagens e canais são diferentes. Nesse caso, uma Messaging Bridge, um Endpoint que interage com um conjunto de Channel Adapters, pode ser usado para fazer as conversões necessárias.

Messaging Bridge normalmente é fornecido por um framework que deseja oferecer a possibilidade de integração com um sistema concorrente. Pode ser tão simples quanto um par de adaptadores de canal, ou algo complexo com vários adaptadores e transformadores trabalhando de forma coordenada.

## Aplicações

Quando for necessário realizar a integração entre dois sistemas de mensageria de fabricantes diferentes.

### Ponte de Mensageria em Java (JMS)

Um par de Adaptadores de Canal podem ser considerados uma ponte. O exemplo do FileAdapter mostrado em Adaptador de Canal lê arquivos de uma pasta e transfere para o canal. Se implementarmos o outro lado onde uma mensagem JMS é transmitida para o sistema de arquivos teremos uma ponte.

### Ponte de Mensageria em Apache Camel e Spring Integration

Uma ponte é uma coleção de adaptadores de canal. Em Apache Camel toda rota da forma `from("origem").to("destino")` pode ser considerada uma ponte se interliga sistemas de mensageria diferentes (ex: sistema de arquivos a JMS).

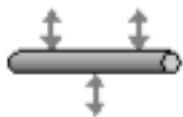
Há também uma ponte especial que permite integrar JMS, Camel e Spring Integration, que é o contexto do Spring. Através do Spring todos esses componentes podem ser configurados como beans, e integrados. Tanto Camel como Spring Integration possuem componentes de integração com JMS e a comunicação Spring Integration e Camel pode ser feita através de beans intermediários, ou diretamente referenciando componentes `<channel>` do Spring Integration pelo seu ID dentro de uma rota no `<camelContext>`.

Por exemplo, para ler de um canal “spring-output” no Spring Integration e enviar as mensagens recebidas para o canal “camel-input” pode-se usar a seguinte configuração Spring:

```
01. <beans:beans xmlns ="http://www.springframework.org/schema/beans"
02.      xmlns:int="http://www.springframework.org/schema/integration"
03.      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
04.      xmlns:camel="http://camel.apache.org/schema/spring"
05.      xsi:schemaLocation="...">
06.
07.      <int:channel id="spring-output"/>
08.
09.      <camel:camelContext >
10.          <camel:route>
11.              <camel:from uri="spring-output"/>
12.              <camel:to uri="camel-input"/>
13.          </camel:route>
14.      </camel:camelContext>
15....
```

## (19) Barramento de Mensagens (Message Bus)

### Ícone



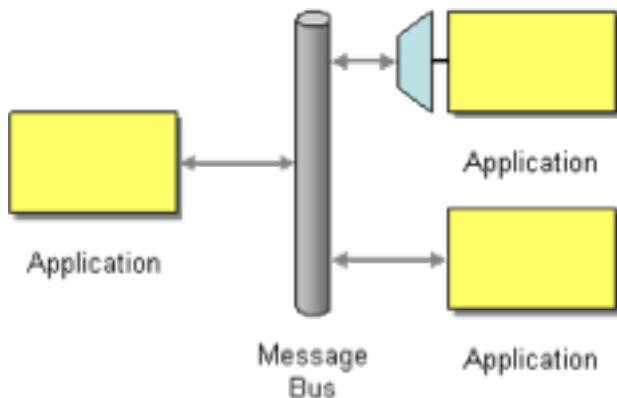
### Problema

“Qual é a arquitetura que permite que aplicações separadas trabalhem juntas, mas de uma forma desacoplada de tal maneira que aplicações possam ser facilmente adicionadas ou removidas sem afetar as outras?”

### Solução

“Estruture o middleware de conexão entre essas aplicações como um barramento de mensagens que os habilite a trabalhar juntos usando Mensageria”

### Diagrama



### Descrição

O Barramento de Mensagens (Message Bus) é uma arquitetura que representa todo um sistema de mensageria implementado com um menu de serviços. O barramento é um sistema capaz de identificar mensagens e rotear automaticamente para canais onde estão conectados tradutores, roteadores, destinatários, etc. Também representa um SOA, onde cada serviço tem um canal para requisições e respostas. Os canais formam um diretório de serviços disponíveis.

Pode-se considerar o Camel, Spring Integration ou Mule como um Message Bus de *propósito geral*. Eles fornecem uma infraestrutura de comunicação comum (multi-plataforma e multi-linguagem), coleções de adaptadores para diversos sistemas, e uma infraestrutura comum de comandos.

Pode-se implementar um Barramento de Mensagens simples, determinando um vocabulário mínimo e comum de comandos, um modelo de dados canônico em um mecanismo que permita que Endpoints registrem-se e possam ter acesso a serviços comuns.

## Revisão

Padrões de integração de sistemas relacionados a canais de mensageria:

- (11) Canal ponto-a-ponto (Point-to-Point Channel): mensagens enviadas a este canal são consumidas apenas por um receptor.
- (12) Canal publicar-inscrever (Publish-Subscribe Channel): mensagens enviadas a este canal são consumidas por todos os assinantes do canal.
- (13) Canal tipo-de-dados (Datatype Channel): um canal que recebe mensagens de apenas um tipo.
- (14) Canal de mensagens inválidas (Invalid-Message Channel): canal para onde são enviadas que não são válidas.
- (15) Canal de mensagens não-entregues (Dead-Letter Channel): canal para onde o sistema envia mensagens que não conseguiu entregar.
- (16) Entrega Garantida (Guaranteed Delivery): canal que guarda a mensagem até que consiga entregar ao destinatário.
- (17) Adaptador de canal (Channel Adapter): componente que adapta um Terminal de Mensageria a um canal permitindo que sistemas externos participem da comunicação.
- (18) Ponte de mensageria (Messaging Bridge): um conjunto de canais que interligam sistemas de mensageria incompatíveis, permitindo que eles se comuniquem.
- (19) Barramento (Messaging Bus): um barramento universal que permite que aplicações se conectem usando mensageria, um formato comum de comandos e dados, permitindo que compartilhem recursos e serviços disponibilizados no barramento.

# Capítulo 5

# Mensagens

Sistemas de mensageria trocam mensagens. Mensagens são como cartas endereçadas: o sistema de correios tem todas as informações necessárias para entregar a carta, sem precisar saber o que tem dentro do envelope. A mensagem em um sistema de mensageria funciona de forma similar. O sistema usa algumas informações presentes no cabeçalho da mensagem, que deve estar em um formato que ele consiga entender. O formato de uma mensagem é específico ao sistema de mensageria que está sendo usado. Para que um canal possa transmitir quaisquer tipo de dados, eles precisam ser encapsulados em uma mensagem.

Embora cada sistema de mensageria possa definir seu próprio formato de mensagem, uma mensagem é basicamente composta de um cabeçalho, onde estão dados usados pelo sistema de mensageria e meta-informação sobre o conteúdo da mensagem, e um corpo, onde são armazenados os dados.



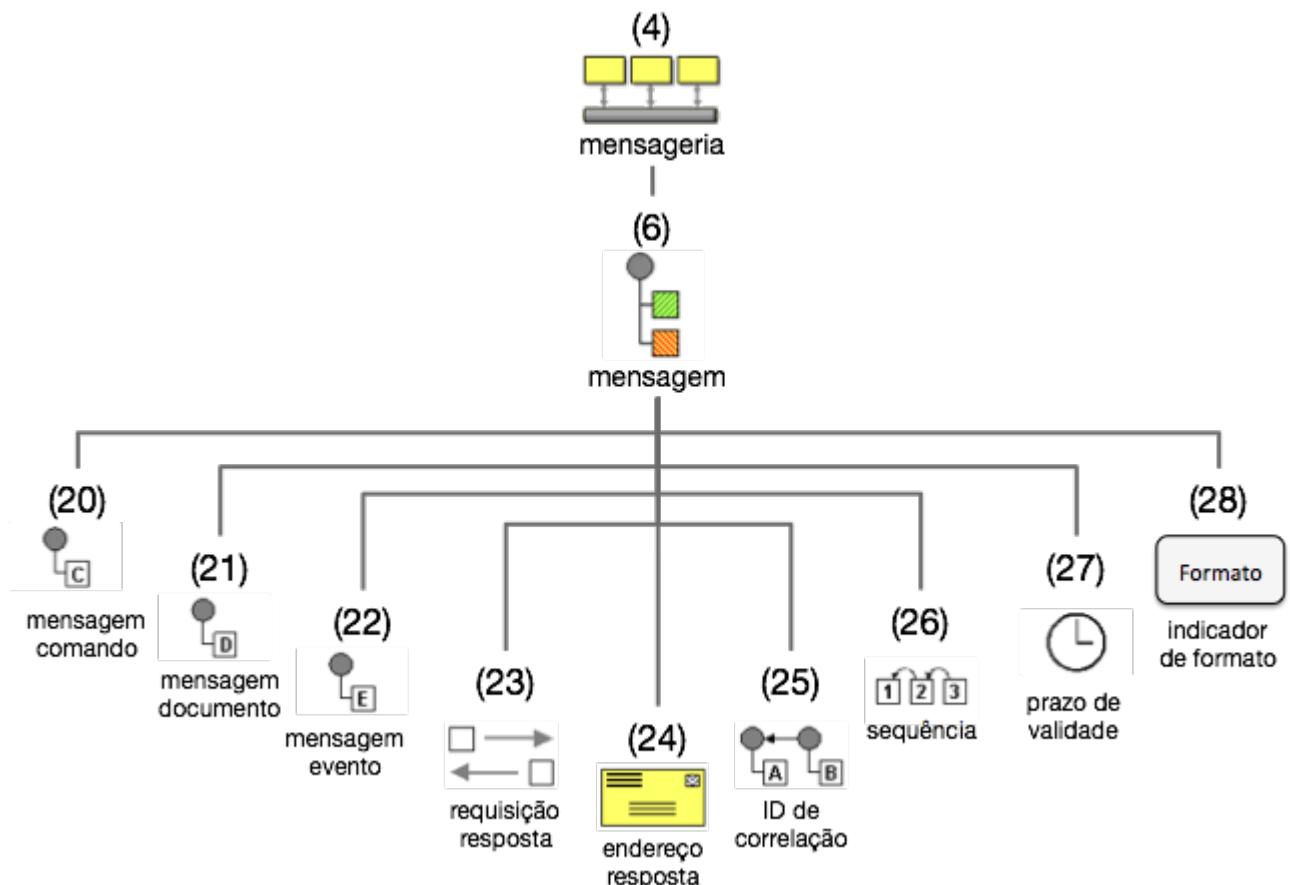
Os padrões de integração de sistemas relacionados com a construção de mensagens distinguem tipos de mensagem, mecanismos para controlar conjuntos de mensagens, associando pares de mensagens e mantendo seqüências de mensagens em ordem, e indicadores de formato e prazo de validade.

Os padrões podem ser classificados em:

- Finalidade da mensagem: uma *Mensagem-comando* (Command Message) é usada para encapsular a execução de um comando remoto. O envio de dados é feito através de uma *Mensagem-documento* (Document Message). *Mensagens-evento* (Event Message) podem ser usadas para enviar notificações de eventos.

- b) Conjuntos de mensagens: *Requisição-Resposta* (Request-Reply) representa um par de mensagens usadas em comunicação síncrona. Um *Endereço de Retorno* (Return Address) às vezes é usado caso a resposta seja enviada para um lugar diferente. Um *Identificação de Correlação* (Correlation Identifier) pode ser usado para associar uma determinada resposta a uma requisição. Quando blocos grandes de dados são enviados, é mais eficiente enviá-los em mensagens menores, que precisam ser reconstruídas posteriormente. Para isto implementa-se o padrão *Seqüência de Mensagens* (Message Sequence) que permite que o destinatário verifique o recebimento de todas as partes antes de reconstruir os dados.
- c) Versionamento e validade: o tipo de uma mensagem, sua versão ou outras informações sobre o formato dos dados contidos na mensagem pode ser especificado através de um *Indicador de Formato* (Format Indicator), que contém informação para descrever e validar o conteúdo de uma mensagem. Um *Prazo de Validade* (Message Expiration) pode também ser usado para estipular um limite de tempo de envio da mensagem.

O diagrama abaixo mostra a hierarquia dos padrões relacionados à construção de mensagens:



## (20) Mensagem-comando (Command Message)

### Ícone



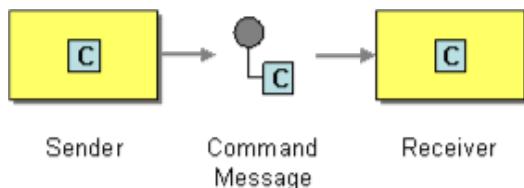
### Problema

“Como usar Mensageria para chamar um comando em outra aplicação”

### Solução

“Use uma Mensagem-comando (Command Message) para chamar um procedimento em outra aplicação de forma confiável”

### Diagrama



**C** = getLastTradePrice("DIS");

### Descrição

Uma Mensagem-comando é uma forma de executar um comando remoto usando mensageria. A mensagem deve conter todos os dados necessários para executar o comando remoto. Pode ser um identificador ou código para selecionar um comando pré-configurado, pode ser o nome de um método ou função, contendo ou não parâmetros. Os parâmetros podem ser dados anexados no corpo da mensagem. Normalmente a mensagem deve também conter informações sobre tipo de dados dos parâmetros ou alguma interface de execução (IDL, WSDL).

Mensagens-comando geralmente acontecem como parte de uma Requisição-Resposta, a menos que o cliente não precise saber o resultado de um comando, por exemplo, para ligar ou desligar algum serviço remoto. A mensagem apenas encapsula as informações necessárias para que o comando seja executado remotamente. É responsabilidade da aplicação decodificar os dados enviados na mensagem e executar o comando. Por exemplo, uma mensagem pode enviar uma instrução que será executada por um shell do sistema operacional (ex: shutdown now), junto com credenciais de um usuário habilitado para o serviço. A aplicação receptora precisa extraír esses dados da mensagem e ser capaz de executar o comando.

É útil para comandos que podem ser chamados de forma assíncrona, mas também pode ser usado para executar comandos em estilo RPC, neste caso implementando também o padrão Requisição-Resposta (Request-Reply), onde uma Mensagem-comando é enviada, e uma Mensagem-documento é retornada.

Geralmente mensagens-comando são enviadas a canais Ponto-a-Ponto para garantir que a mensagem seja consumida apenas uma vez, garantindo uma única execução do comando pelo cliente.

## Aplicações

Tunneling de comandos através de mensagens (ex: Web Services). Uma aplicação típica desse padrão são mensagens SOAP.

### Mensagem-comando usando SOAP

Web Services SOAP usam Mensagens-comando para enviar requisições. Muitas vezes essas mensagens são parte de uma implementação SOAP-RPC configurada em WSDL com um par de mensagens, que representa o padrão Requisição-Resposta. Uma requisição SOAP, que é uma Mensagem-comando, e uma resposta SOAP que é uma Mensagem-documento. O documento abaixo ilustra uma Mensagem-comando SOAP enviada via HTTP POST:

```
POST /xmlrpc-bookstore/bookpoint/BookstoreIF HTTP/1.0
Content-Type: text/xml; charset="utf-8"
Content-Length: 585
SOAPAction: ""

<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope
    xmlns:env="http://www.w3.org/2001/12/soap-envelope"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:enc="http://www.w3.org/2001/12/soap-encoding/"
    env:encodingStyle="http://www.w3.org/2001/12/soap-encoding/">
<env:Body>
    <ans1:getPrice xmlns:ans1="http://mybooks.org/wsdl">
        <String_1 xsi:type="xsd:string">2877142566</String_1>
    </ans1:getPrice>
</env:Body>
</env:Envelope>
```

### Mensagem-comando usando Java/JMS

Mensagens comando podem encapsular métodos remotos a ser executado em um serviço síncrono EJB em um Ativador de Serviço (Service Activator). A mensagem precisa conter todas as informações necessárias para executar o comando. Isto inclui além do nome do método, os valores e tipos dos parâmetros (se houver). Informações sobre tipos de dados podem estar nos cabeçalhos da mensagem, já que é meta-informação, mas também podem estar no corpo, dependendo de como foi projetada a aplicação. No exemplo a seguir, uma Mensagem-comando é usada para executar o método void print(String text, int copies) em um sistema remoto. Uma das maneiras de enviar os dados na mensagem é usar apenas propriedades do cabeçalho:

```
Message command = session.createMessage();
command.setStringProperty("Text-to-print", "This is the text to be printed");
command.setIntProperty("Number-of-copies", 3);
```

```
command.setStringProperty("Method-name", "print");
```

Outra é incluir tudo no corpo usando alguma estrutura como CSV, JSON ou XML:

```
String xml = "<command>
    <method>
        <name>print</name>
        <parameters>
            <String> This is the text to be printed </String>
            <Integer>3</Integer>
        </parameters>
    </method>
</command>";
TextMessage command = session.createTextMessage();
command.setText(xml);
```

No primeiro caso, a aplicação poderá recuperar as informações usando métodos getProperty(). O exemplo abaixo extrai os parâmetros do cabeçalho e usa Java Reflection para executar o método pelo nome:

```
01. public void onMessage(Message message) {
02.     try {
03.         String data = message.getStringProperty("Text-to-print");
04.         int copies = message.getIntProperty("Number-of-copies");
05.         String methodName = message.getStringProperty("Method-name");
06.
07.         Method method = PrintingService.class.getMethod(methodName,
08.                                               String.class,
09.                                               Integer.class);
10.         method.invoke(data, copies);
11.     } catch (Exception e) {...}
12. }
```

Neste segundo exemplo, a aplicação receptora precisará processar o XML e extrair as informações desejadas. Usamos também um mecanismo de reflection mais genérico:

```
01. public void onMessage(Message message) {
02.     try {
03.         String xmlText = ((TextMessage)message).getText();
04.         DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
05.         DocumentBuilder db = dbf.newDocumentBuilder();
06.         Document doc = db.parse(xmlText);
07.         XPath xPath = XPathFactory.newInstance().newXPath();
08.
09.         String methodName = xPath.compile("/command/method/name")
10.             .evaluate(doc);
11.         NodeList nodes =
12.             (NodeList) xPath.compile("/command/method/parameters/*")
13.                 .evaluate(doc, XPathConstants.NODESET);
```

```

11.
12.     List<Class> parameterTypesList = new ArrayList<>();
13.     List<Object> parameterValuesList = new ArrayList<>();
14.     for(int i = 0; i < nodes.getLength(); i++) {
15.         Node node = (Node)nodes.item(i);
16.         Class<T> parameterType =
17.             (Class<T>)Class.forName(node.getLocalName());
18.         String parameterStringValue = node.getTextContent();
19.         T parameterValue = parameterType
20.             .getConstructor(new Class[] {String.class})
21.             .newInstance(parameterStringValue);
22.     }
23.     Class[] parameterTypes =
24.         parameterTypesList.toArray(new
Class[parameterTypesList.size()]);
25.     Class[] parameterValues =
26.         parameterValuesList.toArray(new
Class[parameterTypesList.size()]);
27.     Method method =
28.         PrintingService.class.getMethod(methodName, parameterTypes);
29.     method.invoke(parameterValues);
30. } catch (Exception e) {
31.     e.printStackTrace();
32. }

```

## Mensagem-comando usando Apache Camel ou Spring Integration

Por ser um padrão conceitual, não há um componente especial para esse padrão, já que é o conteúdo que vai determinar o tipo da mensagem.

### (21) Mensagem-documento (Document Message)

#### Ícone



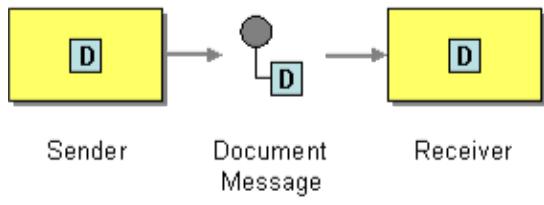
#### Problema

“Como Messaging pode ser usado para transferir dados entre aplicações?”

## Solução

“Use uma Mensagem-documento (Document Message) para transferir uma estrutura de dados entre aplicações de forma confiável”

## Diagrama



**D** = aPurchaseOrder

## Descrição

Uma Mensagem-documento é simplesmente o encapsulamento de dados em uma mensagem. O conteúdo é a parte mais importante da mensagem (diferente da mensagem-evento, que dá prioridade à notificação) mas o conteúdo não tem nenhuma finalidade especificada (diferente da mensagem-comando, onde o conteúdo representa um comando a ser executado). O objetivo é simplesmente a transferência de informações.

Mensagens-documento podem ser enviadas a canais Ponto-a-Ponto quando apenas um destinatário deve receber os dados, ou para canais Publicar-Inscrever, caso as informações sejam destinadas a mais de um destinatário. É comum usar Guaranteed Delivery em canais que entregam Mensagens-Documento.

## Aplicações

Sempre que for necessário transferir informações genéricas através de um canal.

## Mensagem-documento em Java/JMS

Vários dos exemplos mostrados neste livro até agora ilustram o envio e recebimento de Mensagens-documento contendo textos simples. Em JMS a classe Message pode ser usada para Mensagens-documento contendo apenas propriedades (sem corpo). Para mensagens com corpo, há cinco subclasses:

- TextMessage: contém texto (ex: XML).
- ObjectMessage: contém um objeto serializado.
- MapMessage: contém um Map com pares nome/valor que podem ser lidos randomicamente.
- StreamMessage: contém um stream de tipos primitivos que devem ser lidos sequencialmente.
- BytesMessage: contém dados não interpretados

Todas são criadas através de métodos de Producer: `createMessage()`, `createTextMessage()`, `createObjectMessage()`, etc. Cada classe possui métodos próprios para extrair e substituir os dados do corpo da mensagem (ex: `getText()`, `setText()`, `writeBytes()`, etc.)

### Mensagem-documento usando Apache Camel ou Spring Integration

Por ser um padrão conceitual, não há um componente especial para esse padrão, já que é o conteúdo que vai determinar o tipo da mensagem.

## (22) Mensagem-evento (Event Message)

### Ícone



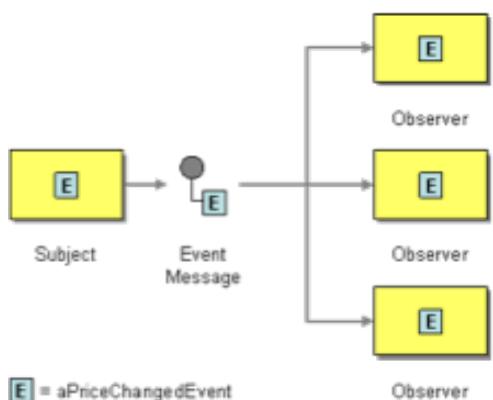
### Problema

“Como mensageria pode ser usado para transmitir eventos de uma aplicação para outra”

### Solução

“Use uma Mensagem-evento para notificação assíncrona e confiável entre aplicações”

### Diagrama



### Descrição

Uma Mensagem-evento pode ser qualquer tipo de mensagem em um sistema de mensageria. O importante é o papel que ela exerce como agente de notificação. O seu conteúdo são dados relacionados à notificação, mas são menos importantes que o momento que a mensagem é enviada e recebida. Pode haver um canal, por exemplo, para receber notificações que simplesmente registre a chegada de uma mensagem-evento para disparar alguma tarefa. Neste caso ela sequer precisaria consultar o conteúdo da mensagem.

Uma notificação pode anexar os dados que o assinante espera como anexo no corpo da mensagem (modelo push). Se os dados forem grandes, ou se houver muitos destinatários, pode-se usar o padrão Recibo de Bagagem (Claim-Check) e enviar com a notificação apenas uma referência ou link para que o destinatário possa baixar os dados (modelo pull).

Eventos tipicamente interessam a mais de um destinatário, portanto geralmente são enviados para canais Publicar-Inscrever. Eventos são descartáveis e perdem a importância se não são recebidos dentro de um prazo (em contraste com Mensagem-documento). Podem ser implementados com um Prazo de Validade curto e transmitidos em canais não-persistentes.

## Aplicações

Uma mensagem-evento é usada sempre que um evento ocorrer em uma aplicação e outra precisar ser notificada. Por exemplo, o estoque para um produto está vazio e a aplicação que repõe o estoque precisa ser notificada.

### Mensagem-evento usando Java/JMS

Mensagens-evento são tipicamente enviadas para Canais Publicar-Inscrever (JMS Topics). Podemos implementar um mecanismo de notificação genérico (modelo push ou pull) em JMS usando esses dois padrões.

Uma Mensagem-evento pode ou não conter corpo. No modelo push, em que a notificação contém os dados esperados pelo observador, ela será implementada como uma subclasse de Message (TextMessage, ObjectMessage, etc), para que esse corpo possa ser inserido. No modelo pull, no qual o observador apenas é notificado que os dados estão disponíveis (e precisa depois fazer uma requisição para buscá-los) pode ser uma Message simples.

O exemplo abaixo foi adaptado do catálogo [EIP] e ilustra a implementação de uma solução de Observer (event handler) usando Mensagens-evento e um canal Publicar-Inscrever. O modelo é o mais simples, usando Push. No capítulo sobre Endpoints mostraremos um exemplo usando Pull com o Consumidor Ativado por Eventos. Como estamos implementando Push, a Mensagem-evento enviada precisa incluir os dados no corpo da mensagem. No modelo Pull, poderia ser uma mensagem vazia, pois a informação relevante é a notificação, já que a responsabilidade de buscar os dados passa a ser do Observador.

O Observador é um MessageListener que guarda um estado (String state) e a Mensagem-evento recebida é a notificação que contém um novo estado que deve substituir o anterior.

```
01.  public class ObserverGateway implements MessageListener {  
02.  
03.      private String state;  
04.  
05.      public void init(String initialState, Connection con,  
06.                          Destination notificationsTopic) throws JMSEException {  
07.          this.state = initialState;
```

```
08.         Session session = con.createSession(false,
Session.AUTO_ACKNOWLEDGE);
09.         MessageConsumer updateConsumer = session
10.             .createConsumer(notificationsTopic);
11.             updateConsumer.setMessageListener(this);
12.             con.start();
13.     }
14.
15.     @Override
16.     public void onMessage(Message message) {
17.         try {
18.             System.out.println("Notification has arrived.");
19.             TextMessage eventMessage = (TextMessage) message;
20.             String newState = eventMessage.getText();
21.             this.update(newState);
22.         } catch (JMSEException e) {
23.             e.printStackTrace();
24.         }
25.     }
26.
27.     private void update(String newState) {
28.         this.state = newState;
29.         System.out.println("Current state is now: " + newState);
30.     }
31.
32.     public String getState() {
33.         return state;
34.     }
35.
36.     public static void main(String[] args) throws Exception {
37.         Context ctx = new InitialContext();
38.         Destination notificationsTopic = (Destination) ctx
39.             .lookup("notifications");
40.
41.         ConnectionFactory factory = (ConnectionFactory) ctx
42.             .lookup("ConnectionFactory");
43.         Connection con = factory.createConnection();
44.
45.         ObserverGateway observer = new ObserverGateway();
46.         observer.init("The door is closed!", con, notificationsTopic);
47.         System.out.println("Initial state: " + observer.getState());
48.
49.         System.out.println("Waiting for notifications.");
50.
51.         Thread.sleep(60000); // 1 minute to receive notification
52.         con.close();
53.     }
54. }
```

A execução da classe Observador irá registrar um observador do canal de “notifications” por 60 segundos.

```
Initial state: The door is closed!
Waiting for notifications.
```

Qualquer mensagem de texto enviada para a fila “notifications” será considerada uma notificação, e o texto contido na mensagem será o novo estado. O SubjectGateway abaixo implementa o componente que causa o evento:

```
01.  public class SubjectGateway {
02.
03.      MessageProducer updateProducer;
04.      Session session;
05.
06.      public void init(Connection con, Destination notificationsTopic)
07.                      throws
08.          JMSEException {
09.              session = con.createSession(false, Session.AUTO_ACKNOWLEDGE);
10.              updateProducer = session.createProducer(notificationsTopic);
11.              con.start();
12.
13.      public void notify(String state) throws JMSEException {
14.          TextMessage message = session.createTextMessage(state);
15.          updateProducer.send(message);
16.
17.
18.      public static void main(String[] args) throws Exception {
19.          Context ctx = new InitialContext();
20.          Destination notificationsTopic =
21.              (Destination) ctx.lookup("notifications");
22.          ConnectionFactory factory =
23.
24.              (ConnectionFactory) ctx.lookup("ConnectionFactory");
25.              Connection con = factory.createConnection();
26.
27.              SubjectGateway subject = new SubjectGateway();
28.              subject.init(con, notificationsTopic);
29.
30.              String state = "The door is now open! Come in!";
31.              System.out.println("Will send a notification now!");
32.              subject.notify(state);
33.      }
34.  }
```

A execução da classe SubjectGateway irá enviar uma Mensagem-evento para a fila Publicar-Inscrever “notifications”. Todos os observadores que estiverem registrados como consumidores da fila irão receber as mensagens enviadas. Se o ObjectGateway ainda estiver no ar, ele receberá a notificação e modificará o seu estado:

Notification has arrived.

Current state is now: The door is now open! Come in!

## Mensagem-evento usando Apache Camel ou Spring Integration

Por ser um padrão conceitual (como Mensagem-comando e Mensagem-documento), não há um componente de framework especial para esse padrão, já que é o conteúdo da mensagem e a forma como é usada que vai determinar o tipo da mensagem.

### (23) Requisição-resposta (Request-Reply)

#### Ícone



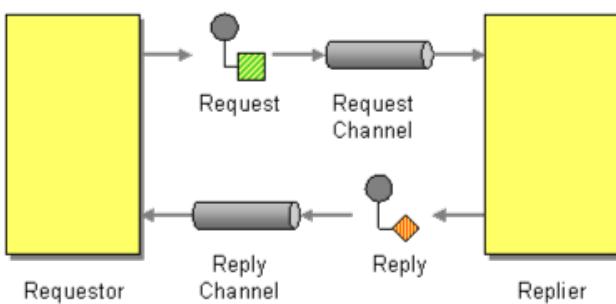
#### Problema

“Quando uma aplicação envia uma mensagem, como ela pode obter uma resposta do servidor?”

#### Solução

“Envie um par de mensagens Request-Reply, cada uma no seu próprio canal”

#### Diagrama



#### Descrição

A comunicação em um sistema de mensageria é unidirecional e assíncrona. Mensagens seguem de um remetente para um destinatário sem que o remetente espere uma resposta. A comunicação estilo RPC é bi-direcional e pode ser implementada usando Mensageria através de um par de mensagens relacionadas. Essa solução é representada pelo padrão Requisição-Resposta (Request-Reply).

Uma Requisição-Resposta tem dois participantes: o componente que faz a requisição, e o componente que responde à requisição. O canal de requisição pode ser Ponto-a Ponto ou Publicar-Inscrever, dependendo se uma requisição deve ser enviada a todos os interessados ou apenas um consumidor. O canal de resposta geralmente é Ponto-a-Ponto.

Existem duas formas de receber a resposta: de forma assíncrona, através da observação de um evento, ou de forma síncrona, bloqueando a continuidade do processamento até que a mensagem de resposta tenha chegado.

Tipicamente uma implementação de Requisição-Resposta também vai conter um Identificador de Correlação (Correlation Identifier) e/ou um Endereço de Resposta (Return Address).

## Aplicações

Quando for necessário implementar uma comunicação em estilo RPC usando mensageria. Alguns cenários comuns de Requisição-Resposta são:

- Arquitetura RPC, onde a requisição é uma Mensagem-comando contendo os dados para a execução de um procedimento no destinatário, e a resposta é uma Mensagem-documento contendo os valores de retorno da execução ou uma exceção.
- Pesquisa, onde a requisição é uma Mensagem-comando contendo uma pesquisa, e a resposta pode ser uma Message Sequence contendo os resultados.
- Notificação, onde a requisição é uma Mensagem-evento (notificação) e a resposta uma Mensagem-documento reconhecendo a notificação, ou uma Mensagem-comando enviada a outro Endereço de Retorno (Return Address) para delegar a execução de alguma tarefa.

## Requisição-Resposta usando Java/JMS

Este exemplo é baseado em um exemplo simples apresentado no catálogo [EIP]. Consiste de duas classes. RequestingClient é um Terminal que faz solicitações enviando Mensagens-comando para o canal “request-queue” e espera respostas no canal “reply-queue”. RespondingListener, é o Terminal que é notificado quando uma Mensagem-comando chega na “request-queue”, processa a mensagem e envia uma Mensagem-documento contendo a resposta para o canal “reply-queue”, que será consumida pelo RequestingClient.

Os serviços que podem ser executados pelo processador são operações definidas em classes Java. Por exemplo, o cliente pode desejar executar uma das operações da classe Operation:

```
public class Operation {
    public Double add(Double arg1, Double arg2) {
        return arg1 + arg2;
    }

    public Double subtract(Double arg1, Double arg2) {
        return arg1 - arg2;
    }
}
```

```

public Double multiply(Double arg1, Double arg2) {
    return arg1 * arg2;
}

public Double divide(Double arg1, Double arg2) {
    return arg1 / arg2;
}
}

```

Para executar uma dessas operações, por exemplo multiply(3.0, 7.0), RequestingClient precisa enviar uma Mensagem-comando especificando o nome da classe, o nome do método, os tipos e valores dos parâmetros num documento XML como este:

```

01.   <command>
02.     <method class='br.com.argonavis.eipcourse.msg.reqres.Operation'
03.       name='multiply'>
04.         <params>
05.           <java.lang.Double>3.0</java.lang.Double>""
06.           <java.lang.Double>7.0</java.lang.Double>""
07.         </params>
08.       </method>
09.   <command>

```

O código do RequestingClient está mostrado abaixo. Observe que, no método send(), ele guarda a referência para a fila de resposta “reply-queue” via setJMSReplyTo(). Isto é necessário para que o RespondingListener saiba para onde enviar a resposta (a aplicação poderia ter diferentes clientes esperando respostas em filas diferentes).

```

public class RequestingClient {
    private Session session;
    private Destination replyQueue, requestQueue;
    private MessageProducer requestProducer;
    private MessageConsumer replyConsumer;

    protected void init(Connection con,
                        Destination requestQueue, Destination replyQueue)
                        throws NamingException, JMSEException {
        session = con.createSession(false, Session.AUTO_ACKNOWLEDGE);
        this.replyQueue = replyQueue;
        this.requestQueue = requestQueue;
        requestProducer = session.createProducer(requestQueue);
        replyConsumer = session.createConsumer(replyQueue);
        con.start();
    }

    public void send() throws JMSEException {
        TextMessage requestMessage = session.createTextMessage();
        requestMessage.setText("<command>""

```

```

+ "    <method class='br...Operation' name='multiply'>"
+ "      <params>
+ "        <java.lang.Double>3.0</java.lang.Double>"
+ "        <java.lang.Double>7.0</java.lang.Double>"
+ "      </params>
+ "    </method>
+ "</command>");
requestMessage.setJMSReplyTo(replyQueue);
requestProducer.send(requestMessage);

System.out.println("Request was sent.");
Utils.printMessage(requestMessage);
}

public void receive() throws JMSEException {
    System.out.print("Waiting for reply... ");
    TextMessage replyMessage = (TextMessage) replyConsumer.receive();

    System.out.println("Received reply.");
    Utils.printMessage(replyMessage);
}

public static void main(String[] args) throws Exception {
    Context ctx = new InitialContext();
    Destination requestQueue = (Destination) ctx.lookup("request-queue");
    Destination replyQueue = (Destination) ctx.lookup("reply-queue");

    ConnectionFactory factory =
(ConnectionFactory) ctx.lookup("ConnectionFactory");
    Connection con = factory.createConnection();

    RequestingClient requestor = new RequestingClient();
    requestor.init(con, requestQueue, replyQueue);

    requestor.send();
    requestor.receive();

    con.close();
}
}

```

Utils.printMessage() imprime o conteúdo da mensagem e cabeçalhos JMSReplyTo, JMSMessageID e JMSCorrelationID.

Executando o cliente, o método send() é executado, enviando uma Mensagem-comando. Em seguida o método receive() bloqueia o thread até que a resposta chegue.

Request was sent.

JMSMessageID: ID:Helders-MacBook-Pro.local-49523-1443276141553-1:1:1:1:1

```

JMSCorrelationID: null
JMSReplyTo: queue://reply-queue
Contents: <command>
<method class='br.com.argonavis.eipcourse.msg.reqres.Operation'
name='multiply'>
<params>
<java.lang.Double>3.0</java.lang.Double>
<java.lang.Double>7.0</java.lang.Double>
</params>
</method>
</command>
Waiting for reply...

```

Abaixo está o código do RespondingListener. Observe que ele recupera o Endereço de Resposta usando getJMSReplyTo() para obter a fila para onde a resposta deve ser enviada. Ele também guarda o ID da mensagem de requisição em um Identificador de Correlação da mensagem de resposta, para que o Requestor possa saber, quando receber a resposta, a qual requisição que ela está associada. No nosso exemplo isto não seria necessário, pois há um único cliente, uma única fila de resposta e uma única mensagem, mas em um cenário com várias mensagens, o Identificador de Correlação será necessário para que se possa saber que a resposta <result>21.0</result> corresponde à requisição que mandou executar multiply(7.0, 3.0) e não à requisição que pediu add(2.0, 4.5).

```

01.  public class RespondingListener implements MessageListener {
02.
03.  private Session session;
04.  private MessageProducer replyProducer;
05.  private MessageConsumer requestConsumer;
06.
07.  private ExampleProcessor processor;
08.
09.  protected void init(Connection con, Destination requestQueue)
09.                      throws NamingException, JMSEException {
10.      session = con.createSession(false, Session.AUTO_ACKNOWLEDGE);
11.      requestConsumer = session.createConsumer(requestQueue);
12.      requestConsumer.setMessageListener(this);
13.      processor = new ExampleProcessor();
14.  }
15.
16.  public void onMessage(Message message) {
17.      try {
18.          if((message instanceof TextMessage)
18.              && (message.getJMSReplyTo() != null)) {
19.              TextMessage requestMessage = (TextMessage) message;
20.              System.out.println("Received request.");
21.              Utils.printMessage(requestMessage);
22.
23.              String result = processor.process(requestMessage);
24.
25.              // prepare response

```

```

26.           Destination replyDestination =
requestMessage.getJMSReplyTo();
27.           replyProducer = session.createProducer(replyDestination);
28.
29.           TextMessage replyMessage = session.createTextMessage();
30.           replyMessage.setText(result);
31.
32.           replyMessage.setJMSCorrelationID(requestMessage.getJMSMessageID());
33.           replyProducer.send(replyMessage);
34.
35.           System.out.println("Reply was sent.");
36.           Utils.printMessage(replyMessage);
37.       } else {
38.           System.out.println("Bad message.");
39.       } catch (JMSEException e) {
40.           e.printStackTrace();
41.       }
42.   }
43.
44.   public static void main(String[] args) throws Exception {
45.       Context ctx = new InitialContext();
46.       Destination requestQueue = (Destination) ctx.lookup("request-
queue");
47.
48.       ConnectionFactory factory =
49.           (ConnectionFactory)
ctx.lookup("ConnectionFactory");
50.       Connection con = factory.createConnection();
51.
52.       RespondingListener replier = new RespondingListener();
53.       replier.init(con, requestQueue);
54.
55.       con.start();
56.       System.out.println("Replier started.");
57.       Thread.sleep(30000); // 30 seconds to receive a request
58.       System.out.println("Done.");
59.       con.close();
60.   }

```

O serviço fica no ar por 30 segundos. Nesse intervalo, o RequestingClient precisa enviar sua requisição. Se ele já foi executado, a requisição está esperando na fila “request-queue” e será recebida logo que o servidor for iniciado:

```

Replier started.
Received request.
JMSMessageID: ID:Helders-MacBook-Pro.local-49523-1443276141553-1:1:1:1:1
JMSCorrelationID: null

```

```
JMSReplyTo: queue://reply-queue
Contents: <command...</command>
Reply was sent.
```

Assim que a mensagem é recebida ela é repassada para um processador que irá extrair os dados do XML e executar a operação (ExampleProcessor.java – veja código-fonte no repositório). Ao final, o processador devolverá um XML contendo a resposta:

```
<result><java.lang.Double>21.0</java.lang.Double></result>
```

Esse texto é incluído no corpo da Mensagem-documento que é enviada para a fila “reply-queue”. O RequestingClient estava esperando por ela. Recebida a mensagem ele pode extrair o texto e processar o XML para obter o valor. Se ele enviou mais de uma requisição para a fila, ele pode descobrir a quais dentre elas se refere a resposta recebida comparando o Identificador de Correlação.

```
Waiting for reply... Received reply.
```

```
JMSMessageID: ID:Helders-MacBook-Pro.local-49526-1443276147840-1:1:1:1:1
JMSCorrelationID: ID:Helders-MacBook-Pro.local-49523-1443276141553-1:1:1:1:1
JMSReplyTo: null
Contents: <result><java.lang.Double>21.0</java.lang.Double></result>
```

## Requisição-Resposta usando Apache Camel

Este padrão é implementado pela interface Exchange do Camel, que abstrai a troca de mensagens Requisição-Resposta. Para isto o Exchange precisa ser criada com o padrão InOut:

```
Endpoint endpoint = ...;
Exchange e = endpoint.createExchange(ExchangePatternInOut);
```

É mais comum configurar o Endpoint para que ele só crie Exchanges no padrão InOut, por exemplo:

```
from("jms:MyQueue?exchangePattern=InOut")
```

O Exchange terá então métodos getIn() e getOut() para recuperar as mensagens de cada interação.

O fragmento de código abaixo ilustra como uma Requisição-Resposta pode ser processada em Camel:

```
Endpoint ep = context.getEndpoint("request-channel");
Producer producer = ep.createProducer();
Exchange exchange = ep.createExchange(ExchangePatternInOut);
Message request = myExchange.getIn();
request.setBody("<comando>...</comando>");
producer.process(exchange); // envia a requisição
Message response = exchange.getOut(); // obtém a resposta
```

A requisição é enviada para o “request-channel”. Camel automaticamente cria um canal temporário para receber as respostas que é guardado no cabeçalho JMSReplyTo da mensagem de requisição (request) e define um JMSCorrelationID na mensagem de requisição. Ao consumir a resposta Camel associa a mensagem recebida com a requisição usando o JMSCorrelationID.

## (24) Endereço de Resposta (Return Address)

### Ícone



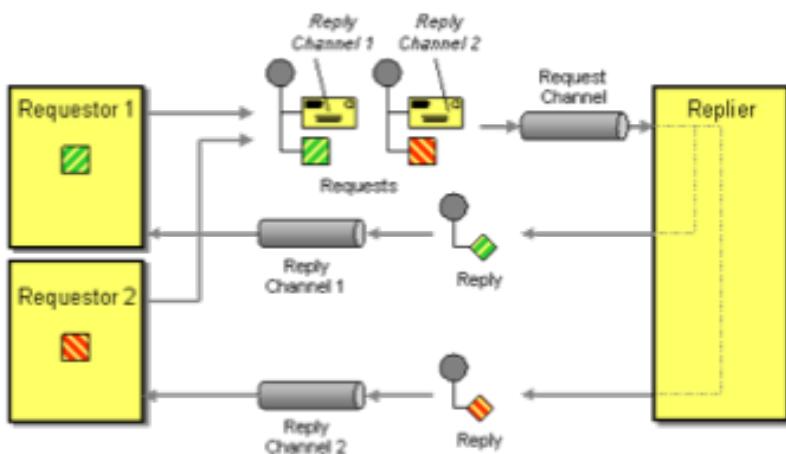
### Problema

“Como é que o componente que envia a resposta sabe para onde enviá-la?”

### Solução

“A mensagem de requisição deve contar um Endereço de Resposta (Return Address) que indica para onde a resposta deve ser enviada”

### Diagrama



### Descrição

Em um sistema de mensageria, onde canais são usados para comunicação unidirecional, a implementação do padrão Requisição-Resposta precisa incluir no cabeçalho da mensagem de requisição, o endereço para o qual a resposta deve ser enviada (que pode ser o mesmo endereço do remetente da requisição, ou não). Esse atributo é semelhante ao campo “Reply-To” de um email.

### Aplicações

Em comunicação na qual existe uma mensagem de resposta, quando for necessário informar o endereço (canal) para onde a resposta deve ser enviada.

### Endereço de Resposta em Java/JMS

Mensagens JMS utilizam o cabeçalho padrão JMSReplyTo para Endereço de Resposta, que deve conter um objeto Destination. O valor pode ser lido ou definido com os métodos get/setJMSReplyTo().

No exemplo abaixo [EIP], um remetente especifica um canal como Endereço de Resposta:

```
Destination requestChannel = ...;
Destination replyChannel = ...;
MessageProducer requestSender = session.createProducer(requestChannel);
Message request = ...;
request.setJMSReplyTo(replyChannel);
requestSender.send(request);
```

O receptor pode, então responder usando getJMSReplyTo() para descobrir o destino:

```
Destination requestChannel = ...;
MessageConsumer requestReceiver = session.createConsumer(requestChannel);
Message request = requestReceiver.receive();
Destination replyChannel = request.getJMSReplyTo();

MessageProducer replySender = session.createProducer(replyChannel);
Message response = ...;
remetente.send(response);
```

## Endereço de Resposta em Apache Camel

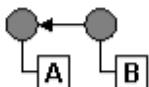
Camel usa o cabeçalho JMSReplyTo como Endereço de Resposta, que é implicitamente implementado em cada Producer. Em mensagens enviadas com InOut, o Producer inclui um endereço de retorno no cabeçalho da mensagem para retornar a resposta através da interface Exchange. Por default, Camel usa filas temporárias para ReplyTo com padrão InOut.

Pode-se também definir a opção replyTo explicitamente. Isto é útil para redirecionar a resposta para um canal específico e não o canal temporário da resposta (pode ser necessário usar também replyToType pois o default é Temporary, que pode não atender se canais forem compartilhados):

```
jms:queue:fila?replyTo=jms:queue:other&replyToType=Shared
```

## (25) Identificador de Correlação (Correlation Identifier)

### Ícone



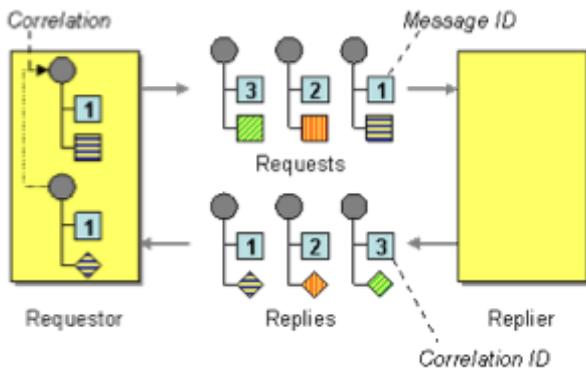
### Problema

“Em uma aplicação usando Request-Reply, como o solicitante que recebeu uma resposta sabe qual das suas requisições gerou a resposta que ele está recebendo?”

### Solução

“Cada mensagem de resposta deve conter um Identificador de Correlação - um identificador único que indica para qual mensagem de requisição corresponde esta resposta”

## Diagrama



## Descrição

Em uma comunicação Requisição-Resposta realizada através de mensageria, havendo várias requisições, existe a possibilidade de uma resposta recebida em um canal não corresponder à requisição enviada. Um Identificador de Correlação (Correlation Identifier) soluciona este problema incluindo informações que indicam qual sua requisição correspondente.

Para implementar o padrão Identificador de Correlação a aplicação que recebe a requisição precisa obter um identificador da requisição e incluí-lo na resposta como um identificador da correlação. A aplicação que enviou a requisição pode então comparar esse identificador com o da requisição, e decidir se a resposta deve ser consumida.

## Aplicações

Implementações de comunicação Requisição-Resposta que enviam múltiplas requisições e precisam identificar qual requisição gerou a resposta recebida.

## Exemplos

Uma Mensagem na API JMS possui um cabeçalho padrão JMSCorrelationID que pode ser lido e definido através de métodos get/set. Normalmente JMSCorrelationID é usada para guardar o JMSMessageID da mensagem com a qual está correlacionada. JMSMessageID é gerado automaticamente pelo provedor de mensageria quando a mensagem é enviada. Já o JMSCorrelationID pode ser lido e redefinido pela aplicação.

Por exemplo, em uma solução Requisição-Resposta as duas mensagens podem ser vinculadas da forma ilustrada no código abaixo [EIP]:

```
Message requisição = ...;
Message resposta = ... ;
String requisicaoID = requisição.getJMSMessageID();
resposta.setJMSCorrelationID(requisição ID);
```

Veja mais exemplos usando Identificador de Correlação nas seções sobre Requisição-Resposta e Sequência de Mensagens.

## Identificador de Correlação usando ApacheCamel

Como Camel representa a mensagem como um Exchange, um componente que já relaciona um par de mensagens (usando ExchangePatternInOut), o Identificador de Correlação pode ser atribuído automaticamente na implementação de uma solução Requisição-Resposta.

Nos padrões Splitter, Multicast, Recipient List e Wire Tap, Camel adiciona automaticamente um Identificador de Correlação no Exchange como propriedade com a chave Exchange.CORRELATION\_ID.

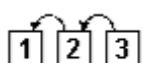
Pode-se também usar o JMSMessageID como Identificador de Correlação quando uma solução Requisição-Resposta é usada:

```
from("jms:queue:in")
    .to(ExchangePatternInOut, "jms:queue:out?useMessageIDAsCorrelationID=true")
    .to("jms:topic:results");
```

Veja mais exemplos usando Identificador de Correlação na seção sobre Requisição-Resposta e Sequência de Mensagens, que precisam relacionar mensagens entre si.

## (26) Sequência de Mensagens (Message Sequence)

### Ícone



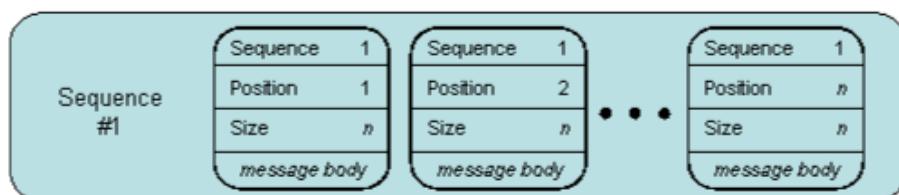
### Problema

“Como uma aplicação pode transportar uma grande quantidade de dados?”

### Solução

“Sempre que um grande conjunto de dados precisar ser partido em pedaços do tamanho das mensagens disponíveis, envie os dados como uma Message Sequence e marque cada mensagem com campos de identificação da sequência.”

### Diagrama



### Descrição

Uma mensagem grande pode ser quebrada em partes menores que são enviadas pela rede, possivelmente por rotas diferentes, até o destino. O Destinatário precisa verificar se todas as partes

chegaram com sucesso antes de tentar reconstruir a mensagem original. Isto é possível se cada mensagem for rotulada de forma a identificar a parte que contém. Isto é similar a um transporte de mercadorias que distribui mercadorias em caixas, numerando cada uma da forma “caixa 1 de 3”, “caixa 2 de 3”, etc.

Os campos de identificação de uma Seqüência de Mensagens são:

1. Identificador da sequencia
2. Identificador da posição
3. Identificador do fim “esta é a última mensagem” (true ou false) ou do tamanho da sequencia (neste caso o receptor precisa contar quantas mensagens já chegaram).

Se um receptor espera uma Seqüência de Mensagens, então cada mensagem enviada para ele deve ser parte de uma sequencia, mesmo que seja uma sequencia de um. Se um receptor receber algumas das mensagens em uma sequencia mas nunca receber todas, deve redirecionar as mensagens recebidas para o Invalid Message Channel.

Pode-se usar um Transactional Client para adiar a entrega das mensagens até o momento em que todas as partes tenham sido enviadas. O receptor também pode usar uma transação para receber as mensagens de forma que só comece a consumi-las quando todas as partes tiverem sido recebidas.

Uma Seqüência de Mensagens normalmente é enviada através de um canal Ponto-a-Ponto com um único consumidor.

## Aplicações

Uma mensagem grande pode ser dividida em várias mensagens menores por vários motivos. Pode ser uma restrição imposta por limitações de infraestrutura, mas pode também ser mais eficiente em termos de performance, principalmente quando for possível enviar as partes em rotas paralelas. A implementação deste padrão rotula as partes da mensagem de forma que elas possam ser verificadas, identificadas e validadas pelo destinatário que irá reconstruir a mensagem original.

Uma alternativa é o Recibo de Bagagem (Claim Check). Em vez de transmitir um documento grande entre duas aplicações, se ambas têm acesso a uma base comum ou sistema de arquivos, grave o documento lá e simplesmente transmita o recibo em uma mensagem simples (similar a compartilhar um link de um disco virtual, como DropBox).

## Seqüência de Mensagens em Java/JMS

Para este exemplo usaremos um Terminal “MessageSequenceSender” que recebe uma mensagem e divide seu conteúdo em partes menores (usando um processador especial “SplitterProcessor”). Antes de enviar para o canal, a o corpo da mensagem é dividido em uma ou mais partes, e cada parte é enviada para o canal em uma mensagem diferente.

Para identificar a sequência usamos o JMSCorrelationID. Adicionamos mais duas propriedades: "Position", que conterá um número indicando a posição da mensagem na sequência, e "Size", que guardará o tamanho da sequência.

Como exemplo, usaremos dois métodos para dividir o corpo das mensagens de texto. Um dividirá por linhas, o outro dividirá em blocos de caracteres:

```

01.  public class SplitterUtil {
02.      public static String[] splitLines(String text) {
03.          return text.split("\n");
04.      }
05.
06.      public static String[] split(String text, int size) throws IOException
07.      {
08.          BufferedReader reader = new BufferedReader(new StringReader(text));
09.          char[] cbuf = new char[size];
10.          int len = reader.read(cbuf);
11.          List<String> blocks = new ArrayList<>();
12.          while(len != -1) {
13.              blocks.add(new String(cbuf).trim());
14.              len = reader.read(cbuf);
15.          }
16.      }
17.  }

```

Enviando a mensagem a seguir, que contém um soneto de Shakespeare, o resultado serão 14 mensagens se divididos em linhas, ou 4 mensagens se divididos em blocos de 200 caracteres:

```

My mistress' eyes are nothing like the sun;
Coral is far more red than her lips' red;
If snow be white, why then her breasts are dun;
If hairs be wires, black wires grow on her head.
I have seen roses damask'd, red and white,
But no such roses see I in her cheeks;
And in some perfumes is there more delight
Than in the breath that from my mistress reeks.
I love to hear her speak, yet well I know
That music hath a far more pleasing sound;
I grant I never saw a goddess go;
My mistress, when she walks, treads on the ground:
    And yet, by heaven, I think my love as rare
        As any she belied with false compare.

```

O MessageSequenceSender, listado abaixo, fará a divisão da mensagem em partes menores e construirá as mensagens individuais, enviando-as para o canal "sequence-queue".

```

01.  public class MessageSequenceSender {
02.

```

```
03.     MessageProducer producer;
04.     Session session;
05.
06.     public void init(Connection con, Destination queue) throws
JMSEException {
07.         session = con.createSession(false, Session.AUTO_ACKNOWLEDGE);
08.         producer = session.createProducer(queue);
09.     }
10.
11.    public void sendLines(String text) throws JMSEException {
12.        String[] payloads = SplitterUtil.splitLines(text);
13.        Message[] messages = createSequence(payloads);
14.        send(messages);
15.    }
16.
17.    public void sendBlocks(String text, int size)
18.                           throws JMSEException,
19.                           IOException {
20.        String[] payloads = SplitterUtil.split(text, size);
21.        Message[] messages = createSequence(payloads);
22.        send(messages);
23.    }
24.
25.    private void send(Message[] messages) throws JMSEException {
26.        for(Message message : messages) {
27.            System.out.println("Sending message "
28.                               + message.getIntProperty("Position") + " "
29.                               + message.getIntProperty("Size"));
30.            producer.send(message);
31.        }
32.    }
33.
34.    private Message[] createSequence(String[] payloads) throws
JMSEException {
35.        Message[] messages = new Message[payloads.length];
36.        String sequenceID =
37.            payloads[0].length() > 25 ?
38.                payloads[0].substring(0,25) : payloads[0];
39.        sequenceID = "["+sequenceID.toUpperCase() + ":" +
40.                     System.nanoTime() +"]";
41.    }
```

```

42.         return messages;
43.     }

```

Na execução usaremos duas sequências com o mesmo texto, um dividindo por linhas, o outro dividindo por blocos de 200 caracteres:

```

44.     public static void main(String[] args) throws Exception {
45.         Context ctx = new InitialContext();
46.         Destination sequenceQueue = (Destination) ctx
47.             .lookup("sequence-queue");
48.         ConnectionFactory factory = (ConnectionFactory) ctx
49.             .lookup("ConnectionFactory");
50.         Connection con = factory.createConnection();
51.
52.         MessageSequenceSender sender = new MessageSequenceSender();
53.         sender.init(con, sequenceQueue);
54.
55.         String payload = "My mistress' eyes are nothing like the
sun;\n"
56.                     + "Coral is far more red than her lips' red;\n"
57.                     + "If snow be white, why then her breasts are
dun;\n"
58.                     + "If hairs be wires, black wires grow on her
head.\n"
59.                     + "I have seen roses damask'd, red and white,\n"
60.                     + "But no such roses see I in her cheeks;\n"
61.                     + "And in some perfumes is there more delight\n"
62.                     + "Than in the breath that from my mistress
reeks.\n"
63.                     + "I love to hear her speak, yet well I know\n"
64.                     + "That music hath a far more pleasing sound;\n"
65.                     + "I grant I never saw a goddess go;\n"
66.                     + "My mistress, when she walks, treads on the
ground:\n"
67.                     + "    And yet, by heaven, I think my love as
rare\n"
68.                     + "    As any she belied with false compare.\n";
69.
70.         System.out.println("Sending seq 1 (lines)");
71.         sender.sendLines(payload);
72.
73.         System.out.println("Sending seq 2 (blocks)");
74.         sender.sendBlocks(payload, 200);
75.
76.         con.close();
77.     }
78. }

```

Executando a classe acima, 18 mensagens serão enviadas para a fila “sequence-queue”.

MessageSequenceReceiver é um MessageListener que receberá todas as mensagens enviadas para “sequence-queue” e aguardará 10 segundos pela chegada de todas as mensagens. As mensagens serão agrupadas em um Map indexado pelo ID da sequência à medida em que forem chegando. Elas não serão ordenadas. Se todas as mensagens chegarem a tempo elas serão mostradas, caso contrário uma mensagem irá informar que não foi possível recuperar a sequência (idealmente deveríamos enviar as mensagens para um Invalid Message Queue).

```

01.  public class MessageSequenceReceiver implements MessageListener {
02.
03.      // sequenceID, List of blocks
04.      private Map<String, Set<Message>> sequences;
05.
06.      public void init(Connection con, Destination queue) throws
JMSEException {
07.          sequences = new HashMap<>();
08.
09.          Session session = con.createSession(false,
Session.AUTO_ACKNOWLEDGE);
10.          MessageConsumer sequenceConsumer =
session.createConsumer(queue);
11.          sequenceConsumer.setMessageListener(this);
12.          con.start();
13.      }
14.
15.      public void onMessage(Message message) {
16.          try {
17.              String sequenceID = message.getJMSCorrelationID();
18.              if (sequences.containsKey(sequenceID)) {
19.                  Set<Message> messages = sequences.get(sequenceID);
20.                  messages.add(message);
21.              } else {
22.                  Set<Message> messages = new HashSet<>();
23.                  messages.add(message);
24.                  sequences.put(sequenceID, messages);
25.              }
26.          } catch (JMSEException e) {
27.              e.printStackTrace();
28.          }
29.      }
30.
31.      public void verifyAndPrint() throws JMSEException {
32.          for (Set<Message> messages : sequences.values()) {
33.              Message testMsg = messages.iterator().next();
34.              int size = testMsg.getIntProperty("Size");
35.              String seqID = testMsg.getJMSCorrelationID();
36.
37.              if (size == messages.size()) { // all arrived (if no
dups)
38.                  System.out.println("\nSequence: " + seqID);

```

```

39.             for (Message message : messages) {
40.                 int position =
41.                     message.getIntProperty("Position");
42.                 String text =
43.                     ((TextMessage)message).getText();
44.                 System.out.println("[ "+position+" ] >" +
45.                     + text + "<"); // send all messages to invalid message queue
46.                 System.out.println("Incomplete sequence: " +
47.                     + seqID + ", size = " +
48.                     messages.size());
49.             }
50.         }
51.     public static void main(String[] args) throws Exception {
52.         Context ctx = new InitialContext();
53.         Destination sequenceQueue =
54.             (Destination) ctx.lookup("sequence-queue");
55.         ConnectionFactory factory = (ConnectionFactory) ctx
56.             .lookup("ConnectionFactory");
57.         Connection con = factory.createConnection();
58.         MessageSequenceReceiver receiver = new
59.             MessageSequenceReceiver();
60.         receiver.init(con, sequenceQueue);
61.         System.out.println("Wait 10 seconds before verifying
62.             messages..."); Thread.sleep(10000); // 10 seconds to receive sequence
63.         System.out.println("Verifying sequences..."); receiver.verifyAndPrint();
64.         con.close();
65.     }
66. }
67. }
68. }
69. }

```

As mensagens serão impressas na tela na ordem em que chegaram. Esta é uma possível saída do MessageSequenceReceiver:

```

Wait 10 seconds before verifying messages...
Verifying sequences...

```

```

Sequence: [MY MISTRESS' EYES ARE NOT:1443314488197845000]
[5]: I have seen roses damask'd, red and white,
[4]: If hairs be wires, black wires grow on her head.

```

[7]: And in some perfumes is there more delight  
 [6]: But no such roses see I in her cheeks;  
 [1]: My mistress' eyes are nothing like the sun;  
 [3]: If snow be white, why then her breasts are dun;  
 [2]: Coral is far more red than her lips' red;  
 [13]: And yet, by heaven, I think my love as rare  
 [12]: My mistress, when she walks, treads on the ground:  
 [14]: As any she belied with false compare.  
 [9]: I love to hear her speak, yet well I know  
 [8]: Than in the breath that from my mistress reeks.  
 [11]: I grant I never saw a goddess go;  
 [10]: That music hath a far more pleasing sound;

Sequence: [MY MISTRESS' EYES ARE NOT:1443314488228058000]

[3]: at music hath a far more pleasing sound;  
 I grant I never saw a goddess go;  
 My mistress, when she walks, treads on the ground:  
     And yet, by heaven, I think my love as rare  
     As any she belied with f  
 [2]: damask'd, red and white,  
 But no such roses see I in her cheeks;  
 And in some perfumes is there more delight  
 Than in the breath that from my mistress reeks.  
 I love to hear her speak, yet well I know  
 Th  
 [4]: alse compare.  
 a far more pleasing sound;  
 I grant I never saw a goddess go;  
 My mistress, when she walks, treads on the ground:  
     And yet, by heaven, I think my love as rare  
     As any she belied with f  
 [1]: My mistress' eyes are nothing like the sun;  
 Coral is far more red than her lips' red;  
 If snow be white, why then her breasts are dun;  
 If hairs be wires, black wires grow on her head.  
 I have seen roses

Esta aplicação não está completa. Seria preciso reordenar as mensagens na ordem correta (de acordo com a propriedade de posição) e reconstruí-las. Uma forma eficiente e desacoplada de fazer isto é incluir na rota um Resequenciador (Resequencer). Isto será mostrado no capítulo seguinte.

## Sequência de Mensagens em Apache Camel e Spring Integration

Sequência de Mensagens é um padrão conceitual e não é implementado explicitamente em Camel e Spring Integration, mas sequências de mensagens são usadas em vários outros padrões, como Divisor (Splitter), que divide uma mensagem em várias partes, Agregador (Aggregator), que reagrupa mensagens e Resequenciador (Resequencer), que reordena. Vários desses padrões usam Sequências de Mensagens (com as três propriedades usadas para agrupar e reordenar).

## (27) Prazo de Validade (Message Expiration)

### Ícone



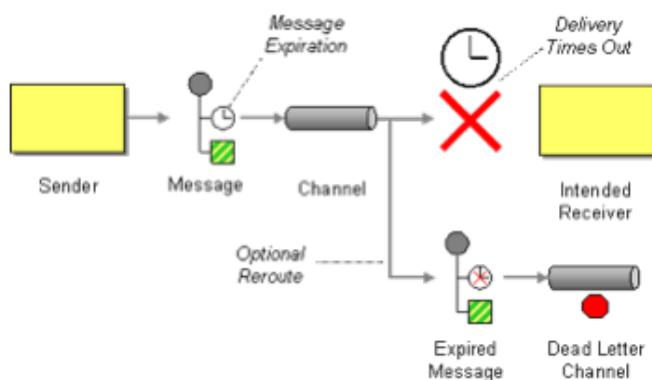
### Problema

“Como um remetente indica quando uma mensagem deve ser considerada obsoleta, não devendo mais ser processada?”

### Solução

“Defina um Prazo de Validade (Message Expiration) para especificar um limite de tempo no qual a mensagem é viável”

### Diagrama



### Descrição

Existem mensagens que precisam ser entregues em um prazo determinado, depois do qual tornam-se inúteis ou inválidas. Um exemplo típico são mensagens-evento usadas para notificações.

Geralmente um sistema de mensageria é configurado para não entregar mensagens vencidas (que podem ser redirecionadas para um Dead-Letter Channel). Se uma mensagem vencida for recebida, ela pode ser redirecionada para um Canal de Mensagens Inválidas.

Às vezes não é a mensagem que deve ter um timeout, mas o cliente. Se uma mensagem precisa ser entregue de qualquer maneira (ex: a resposta de uma requisição) ela não deve ter um Prazo de Validade, mas o cliente destinatário pode estipular um timeout, e decidir por quanto tempo irá esperar a resposta.

A implementação deste padrão geralmente consiste em um cabeçalho contendo a data de vencimento da mensagem, ou uma diferença de tempo relativa à hora do envio.

## Aplicações

Mensagens que só têm validade dentro de um determinado prazo.

### Prazo de Validade em Java / JMS

JMS possui três maneiras de definir o Prazo de Validade de uma mensagem:

- `Message#setJMSExpiration(long)`
- `MessageProducer#setTimeToLive(long)`
- `MessageProducer#send(Message, int, int, long)`

Apenas os dois métodos de MessageProducer devem ser usados. O primeiro (ou qualquer outra tentativa de definir o valor do cabeçalho JMSExpiration) não funciona. Ele é sobreposto pelo provedor assim que a mensagem é enviada. É um erro comum e um problema de design do JMS. A documentação também não é muito clara sobre isto.

Portanto, para definir um Prazo de Validade é preciso especificar um valor em milissegundos usando métodos de MessageProducer. O primeiro estabelece o Prazo de Validade de todas as mensagens enviadas pelo produtor:

```
MessageProducer produtor = session.createProducer(queue);
produtor.setTimeToLive(60000); // mensagens têm um minuto para chegar ao destino
```

Para sobrepor esse valor para uma mensagem em particular, um novo valor pode ser especificado no método `send()` durante o envio:

```
produtor.send(DeliveryMode.PERSISTENT, Message.DEFAULT_PRIORITY, 10000);
```

### Prazo de Validade em Apache Camel

Este padrão não é explicitamente implementado em Camel. O prazo de validade de uma mensagem pode ser definido usando opções do sistema de mensageria usado. No caso de JMS, pode-se alterar o tempo de vida das mensagens em um canal usando `setTimeToLive()`, como foi mostrado no exemplo anterior. As opções podem ser passadas para o Camel através da URI do Endpoint:

```
to("jms:queue:destino?timeToLive=5000");
```

## (28) Indicador de Formato (Format Indicator)

### Problema

“Como o formato de dados de uma mensagem pode ser projetado de forma a permitir mudanças futuras?”

### Solução

“Formato de dados devem incluir um Format Indicator para que a mensagem possa especificar qual formato está usando”

## Descrição

O indicador de formato pode ser simplesmente um número de versão incluído no cabeçalho de uma mensagem. Pode também ser um identificador ou localizador que associe a mensagem com um documento de formato que pode ser usado para validar os dados (ex: um XML Schema para validar conteúdo XML). Pode ser determinado por uma coleção de cabeçalhos que contenham dados sobre o formato dos dados incluídos na mensagem.

Alguns exemplos de indicadores de formato usados em cabeçalhos:

```
Encoding: UTF-8
Type: application/xml
Schema: http://www.example.com/xsd/example\_2\_3.xsd
Version: 2.3
Tipo: pedido
Tipo: imagem
Formato: JPEG
```

## Aplicações

O Indicador de Formato pode ser simplesmente um cabeçalho informando o tipo de dados do conteúdo da mensagem. Isto facilitará o trabalho dos componentes que irão processar a mensagem, como um Roteador Baseado em Conteúdo que poderá analisar o cabeçalho e redirecioná-lo a um Canal de Tipo de Dados específico.

O Indicador de Formato também pode conter outros cabeçalhos e dados que descrevam melhor o formato dos dados, como encodings, versões ou esquemas que podem ser usados para validar os dados. Esse tipo de informação deve ser incluída quando o conteúdo das mensagens estiver sujeito a alterações futuras que podem afetar o seu processamento.

## Exemplos em JMS, Camel ou Spring Integration

Indicador de formato é uma propriedade da mensagem e não aparece formalmente implementado em nenhum framework ou API. Pode ser enviado no cabeçalho da mensagem, mas também pode estar no corpo. O catálogo [EIP] descreve três formas de implementar um Indicador de Formato: *versão*, *chave* ou *documento*, e apresenta como exemplo um documento XML, que têm as três. A versão, no prólogo:

```
<?xml version="1.0"?>
```

A *chave* (no caso a URI) para indicar a localização de um esquema (DTD) que valida o arquivo:

```
<!DOCTYPE exemplo SYSTEM "exemplo.dtd">
<exemplo>teste</exemplo>
```

E o esquema de validação como *documento*, embutido na própria mensagem:

```
<!DOCTYPE exemplo [
    <!ELEMENT exemplo (#PCDATA)>
]>
```

```
<exemplo>teste</exemplo>
```

Se incluído no cabeçalho, o Indicador de Formato pode ser definido pelo produtor da mesma forma como se define cabeçalhos nas mensagens em JMS, Camel ou Spring Integration.

## Revisão

Padrões de integração de sistemas relacionados a mensagens:

- (20) Mensagem-comando (Command Message): mensagem usada para transmitir um comando.
- (21) Mensagem-documento (Document Message): mensagem usada para transmitir dados em geral.
- (22) Mensagem-evento (Event Message): mensagem usada para notificação.
- (23) Requisição-resposta (Request-Reply): par de mensagens sincronizadas usada para permitir execução de operações remotas que retornam dados.
- (24) Endereço de Resposta (Return Address): propriedade de uma mensagem que contém o canal para onde uma resposta deve ser enviada.
- (25) Identificador de Correlação (Correlation Identifier): propriedade de uma mensagem que a associa a uma outra mensagem ou a um grupo de mensagens.
- (26) Sequência de Mensagens (Message Sequence): um conjunto de mensagens que representa um pedaço único de informação.
- (27) Prazo de Validade (Message Expiration): propriedade temporal de uma mensagem que indica ao provedor de mensageria se ela ainda pode ser enviada para um canal.
- (28) Indicador de Formato (Format Indicator): propriedade de uma mensagem que indica o formato de uma mensagem de forma que possa ser validada ou filtrada.

# Capítulo 6

# Roteamento

Um Roteador de Mensagens (Message Router) é um componente capaz de re-endereçar uma mensagem. Além de Roteador de Mensagens, os padrões EIP relacionados a roteamento são especializações ou composições de Roteador de Mensagens (Message Router).

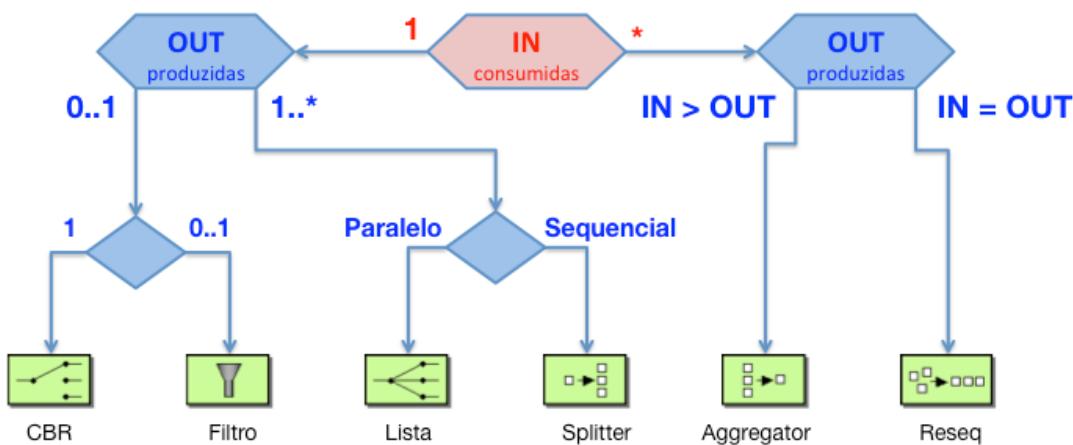
- Roteadores simples (variantes de Message Router)
- Roteadores compostos (combinam vários roteadores simples para criar fluxos mais complexos)
- Padrões de arquitetura (descrevem estilos arquitetônicos baseados em Message Routers)

Os roteadores simples são:

- *Roteador Baseado em Conteúdo (Content-Based Router)* (CBR): analisa o conteúdo de cada mensagem recebida no canal de entrada e re-endereça para um dentre vários canais de saída, de acordo com o seu conteúdo. Consome uma mensagem e produz uma mensagem.
- *Filtro de Mensagens (Message Filter)*: similar ao Roteador Baseado em Conteúdo, mas tem apenas uma saída; o filtro é usado para determinar quais mensagens serão *descartadas*. Consome uma mensagem e produz zero ou uma mensagem.
- *Lista de Receptores (Recipient List)*: é um Roteador de Mensagens que recebe mensagens em um canal de entrada, e redireciona para zero ou mais canais, de acordo com uma lista de destinos. Consome uma mensagem e produz zero, uma ou várias mensagens.
- *Roteador Dinâmico (Dynamic Router)*: recebe mensagens em um canal de entrada e usa uma tabela de roteamento dinâmica para determinar qual, dentre vários canais de saída, receberá a mensagem. Pode ser qualquer um dos três roteadores anteriores.
- *Divisor (Splitter)*: recebe uma mensagem e divide-a em mensagens menores, que são enviadas ao seu canal de saída. Recebe uma mensagem e produz uma ou mais.

- *Agregador (Aggregator)*: faz o oposto do Divisor, agregando mensagens recebidas em seu canal de entrada para uma mensagem maior enviada para o canal de saída. Recebe múltiplas mensagens e produz uma. É necessário manter estado.
- *Re-sequenciador (Resequencer)*: recebe mensagens fora de ordem em seu canal de entrada, e repassa ao canal de saída na ordem correta. Recebe múltiplas mensagens e produz múltiplas mensagens. É necessário manter estado.

Qual roteador escolher?

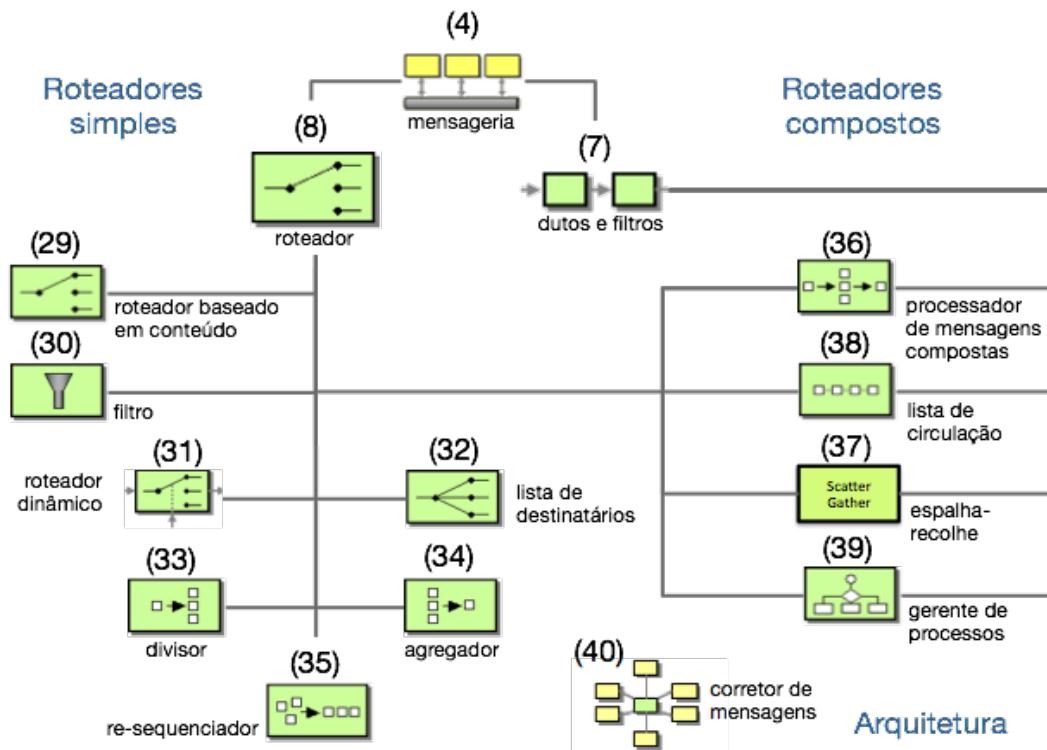


Os roteadores compostos utilizam-se da arquitetura *Dutos e Filtros (Pipes and Filters)* para compor múltiplos roteadores simples em uma solução maior.

- *Processador de Mensagens Compostas (Composed Message Processor) (CMP)*: um componente capaz de processar mensagens compostas (mensagens com anexos ou divididas em seções que precisam ser processadas em separado).
- *Espalha-Recolhe (Scatter-Gather)*: uma composição de roteadores em paralelo que recolhem mensagens espalhadas por diferentes canais.
- *Lista de Circulação (Routing Slip)*: uma composição de roteadores em série.
- *Gerente de Processos (Process Manager)*: um barramento de roteadores desacoplados.
- *Corretor de Mensagens (Message Broker)*: é mais que um roteador composto. Descreve uma solução de arquitetura completa onde vários roteadores desacoplados são mediados por um componente central.

A escolha dentre os roteadores compostos consiste na decisão de ter processamento paralelo (CMP e Espalha-Recolhe) com divisão (CMP) ou difusão (Espalha-Recolhe) de mensagens; ou processamento sequencial (Gerente de Processos e Lista de Circulação), em caminho pre-determinado (Lista) ou arbitrário (Gerente de Processos).

O diagrama abaixo ilustra a hierarquia dos padrões relacionados a roteadores.



## (29) Roteador baseado em conteúdo (Content-Based Router) (CBR)

### Ícone



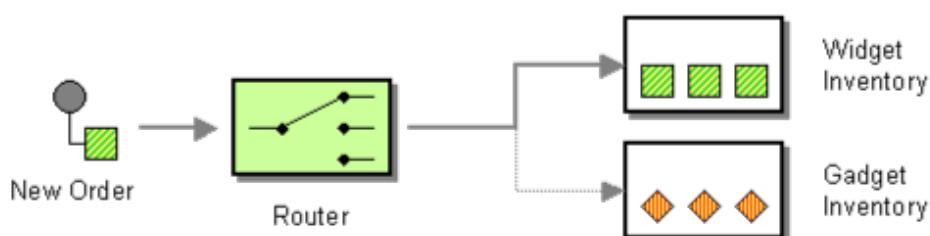
### Problema

“Como lidar com a situação onde a implementação de uma única função lógica está espalhada por múltiplos sistemas físicos?”

### Solução

“Use um *Roteador Baseado em Conteúdo (Content-Based Router) (CBR)* para rotear cada mensagem para o receptor correto baseado no conteúdo da mensagem”

### Diagrama



## Descrição

Um Roteador Baseado em Conteúdo analisa o conteúdo de uma mensagem recebida no seu canal de entrada, e, redireciona para um dentre vários canais de saída, de acordo com esse conteúdo. Os dados analisados podem ser propriedades disponíveis no cabeçalho da mensagem, ou informações que precisam ser extraídas do corpo da mensagem. Essas informações são testadas, e, com base no resultado dos testes, o roteador decide para qual canal a mensagem deve ser redirecionada.

Por exemplo, a análise de um cabeçalho de encoding ou tipo de dados pode ser usado para separar mensagens em formato XML para um canal e mensagens em JSON para outro. Uma expressão XPath e expressões regulares podem ser usadas para separar mensagens XML em canais diferentes de acordo com tags, atributos e valores presentes no seu conteúdo. Geralmente os canais de saída de um Roteador Baseado em Conteúdo são Canais de Tipo-de-Dados (Datatype Channels).

Roteadores Baseados em Conteúdo precisam conhecer detalhes do domínio da aplicação usada e às vezes detalhes dos dados transmitidos na aplicação (nomes de cabeçalhos, estruturas internas de documentos, etc.) Isto aumenta o acoplamento deste componente com o restante do sistema. Se novos receptores precisarem ser adicionados ou alterados, a lógica de seleção do Roteador precisa ser atualizada. Em situações onde a lógica de seleção de conteúdo não pode ser estática, pode-se realizar o roteamento usando outros tipos de roteadores, como uma coleção de Filtros de Mensagem (Message Filter), Lista de Circulação (Routing Slip) para transferir a responsabilidade de seleção para uma corrente de roteadores, ou um Roteador Dinâmico (Dynamic Router), cujas regras de roteamento são descobertas em tempo de execução.

## Aplicações

Aplicações que estão integradas a canais que recebem dados de tipos diferentes precisam de um Roteador Baseado em Conteúdo para analisar o conteúdo de cada mensagem e despachá-la para um canal capaz de processá-la. Os canais devem ser Canais de Tipo-de-Dados.

### CBR em Java/JMS

Um CBR pode ser implementado em Java utilizando algum tipo de mecanismo de controle de fluxo procedural (switch, if-else) após a análise do conteúdo da mensagem “Conteúdo” refere-se a qualquer parte da mensagem, corpo ou cabeçalho.

Um exemplo de CBR já foi mostrado no capítulo 4, na seção sobre Canal de Tipo-de-Dados. O exemplo abaixo é similar, mas roteia mensagens que contém arquivos.

Um Adaptador de Canal (classe FileAdapter) monitora uma pasta (/tmp/jms/inbox) aguardando que arquivos sejam depositadas nela.

```
70.  public class FileAdapter {  
71.      private File directory;  
72.      public FileAdapter(File directory) throws JMSEException {...}  
73.      public void send(List<Message> messages, MessageProducer producer)  
{...}
```

```

74.     public List<Message> createMessages(Session session, List<File> files)
{...}
75.     public List<File> loadFiles() {...}
76.     public byte[] readBytes(File file) {...}
77.     public String readChars(File file) {...}
78.
79.     public static void main(String[] args) throws Exception {
80.
81.         Context ctx = new InitialContext();
82.         ConnectionFactory factory =
83.             (ConnectionFactory)ctx.lookup("ConnectionFactory");
84.         Destination queue = (Destination)ctx.lookup("inbound-channel");
85.         Connection con = factory.createConnection();
86.         con.start();
87.
88.         FileAdapter adapter = new FileAdapter(new File("/tmp/jms/inbox"));
89.
90.         int polls = 12;
91.         while(polls > 0) {
92.             System.out.println("Checking for files... " + polls + " polls
left.");
93.             Session session = con.createSession(false,
Session.AUTO_ACKNOWLEDGE);
94.             MessageProducer producer = session.createProducer(queue);
95.             List<File> files = adapter.loadFiles();
96.             if(!files.isEmpty()) {
97.                 List<Message> messages = adapter.createMessages(session,
files);
98.                 adapter.send(messages, producer);
99.             }
100.            Thread.sleep(5000); // wait 5 seconds
101.            --polls;
102.        }
103.    }
104. }
```

O método `loadFiles()` seleciona os arquivos que são “png”, “xml” ou “txt” (escolhe pela extensão do arquivo) e põe numa lista:

```

105.    public List<File> loadFiles() {
106.        List<File> files = new ArrayList<>();
107.        String[] fileNames = directory.list(new FilenameFilter() {
108.            @Override
109.            public boolean accept(File dir, String name) {
110.                return name.endsWith(".xml")
111.                    || name.endsWith(".png")
112.                    || name.endsWith(".txt");
113.            }
114.        });
115.        for (String fileName : fileNames) {
116.            File file = new File(fileName);
117.            files.add(file);
118.        }
119.    }
120.
```

```

114.         });
115.         if(fileNames != null && fileNames.length > 0) {
116.             for(String fileName : fileNames) {
117.                 files.add(new File(directory, fileName));
118.             }
119.         }
120.     return files;
121. }

```

Em seguida a lista é processada pelo método `createMessages` e cada arquivo é encapsulado em uma mensagem com algumas propriedades de cabeçalho, dentre elas o “Tipo” que será usado para rotear a mensagem posteriormente.

```

122. public List<Message> createMessages(Session session, List<File> files)
                                         throws JMSException {
123.     List<Message> messages = new ArrayList<>();
124.     for(File file: files) {
125.         Message message = null;
126.         boolean valid = true;
127.         String type = file.getName().substring(file.getName()
                                         .lastIndexOf('.')+1).toLowerCase();
128.         if(type.equals("xml") || type.equals("txt")) {
129.             String data = readChars(file);
130.             message = session.createTextMessage(data);
131.         } else if (type.equals("png")) {
132.             byte[] data = readBytes(file);
133.             message = session.createBytesMessage();
134.             ((BytesMessage)message).writeBytes(data);
135.         } else {
136.             valid = false;
137.         }
138.
139.         if(valid) {
140.             message.setLongProperty("Length", file.length());
141.             message.setStringProperty("Name", file.getName());
142.             message.setStringProperty("Type", type);
143.             messages.add(message);
144.             file.delete();
145.         }
146.     }
147.     return messages;
148. }

```

Se a mensagem for criada, o arquivo é removido da pasta.

Por último, o método `send()` processa a lista de mensagens e envia cada mensagem para o “inbound-channel”:

```

149. public void send(List<Message> messages, MessageProducer producer) {
150.     try {

```

```

151.         for(Message message: messages) {
152.             System.out.println("Sending message");
153.             producer.send(message);
154.         }
155.         System.out.println(messages.size() + " messages sent!");
156.     } catch(JMSEException e){...}
157. }
```

Executando o adaptador, ele começará a monitorar a pasta a cada 5 segundos. Abaixo colocamos 4 arquivos na pasta, depois outros 2:

```

Checking for files... 12 polls left.
Checking for files... 11 polls left.
Checking for files... 10 polls left.
Checking for files... 9 polls left.
Sending message
Sending message
Sending message
Sending message
4 messages sent!
Checking for files... 8 polls left.
Checking for files... 7 polls left.
Checking for files... 6 polls left.
Checking for files... 5 polls left.
Sending message
Sending message
2 messages sent!
Checking for files... 4 polls left.
Checking for files... 3 polls left.
Checking for files... 2 polls left.
Checking for files... 1 polls left.
```

A console do ActiveMQ mostra que todos foram para o “inbound-queue”:

Queues						
Name	Number Of Pending Messages	Number Of Consumers	Messages Enqueued	Messages Dequeued	Views	Operations
inbound-queue	6	0	6	0	<a href="#">Browse Active Consumers</a> <a href="#">Active Producers</a>  	<a href="#">Send To Purge</a> <a href="#">Delete</a>

O CBR monitora o “inbound-channel” como um MessageListener, portanto assim que uma mensagem chegar no canal, ela será recebida pelo método `onMessage()`, onde será escolhido o canal para onde a mensagem será roteada. Se por algum motivo aparecer no canal uma mensagem que não tenha a propriedade “Tipo” ou se ela não for “png”, “txt”, ou “xml”, a mensagem será redirecionada a um Canal de Mensagens Inválidas.

```

158. public class ContentBasedRouter implements MessageListener {
159.
160.     private ConnectionFactory factory;
```

```
161.     Destination imageChannel;
162.     Destination textChannel;
163.     Destination xmlChannel;
164.     Destination invalidMessageChannel;
165.     Destination inboundChannel;
166.
167.     public Connection init() throws NamingException, JMSException {
168.         Context ctx = new InitialContext();
169.         this.factory = (ConnectionFactory)
ctx.lookup("ConnectionFactory");
170.         this.imageChannel = (Destination) ctx.lookup("image-channel");
171.         this.textChannel = (Destination) ctx.lookup("text-channel");
172.         this.xmlChannel = (Destination) ctx.lookup("xml-channel");
173.         this.invalidMessageChannel = (Destination) ctx
174.             .lookup("invalid-message-channel");
175.         this.inboundChannel = (Destination) ctx.lookup("inbound-
channel");
176.
177.         Connection con = factory.createConnection();
178.         Session session = con.createSession(false,
Session.AUTO_ACKNOWLEDGE);
179.         MessageConsumer consumer =
session.createConsumer(inboundChannel);
180.         consumer.setMessageListener(this);
181.         con.start();
182.         return con;
183.     }
184.
185.     @Override
186.     public void onMessage(Message message) {
187.         try {
188.             String type = message.getStringProperty("Type");
189.             System.out.println("Inbound channel: Received message: " +
type);
190.             Destination destination;
191.
192.             if (type != null && type.equals("png")) {
193.                 destination = imageChannel;
194.             } else if (type != null && type.equals("txt")) {
195.                 destination = textChannel;
196.             } else if (type != null && type.equals("xml")) {
197.                 destination = xmlChannel;
198.             } else {
199.                 destination = invalidMessageChannel;
200.             }
201.             routeMessage(destination, message);
202.
203.         } catch (JMSException e) {
204.             e.printStackTrace();
```

```

205.         }
206.     }
207.
208.     public void routeMessage(Destination destination, Message message)
209.             throws JMSEException {
210.         try (Connection con = factory.createConnection()) {
211.             con.start();
212.             Session session =
213.                 con.createSession(false,
Session.AUTO_ACKNOWLEDGE);
214.             MessageProducer producer =
215.                 session.createProducer(destination);
216.             producer.send(message);
217.         }
218.     public static void main(String[] args) throws Exception {
219.         ContentBasedRouter router = new ContentBasedRouter();
220.         Connection con = router.init();
221.
222.         Thread.sleep(60000); // Will wait one minute for files
223.         con.close();
224.     }
225. }
```

Executando o CBR, ele irá consumir imediatamente as seis mensagens que estão no “inbound-queue”:

```

Inbound channel: Received message: xml
Inbound channel: Received message: xml
Inbound channel: Received message: xml
Inbound channel: Received message: png
Inbound channel: Received message: png
Inbound channel: Received message: txt
```

A console do ActiveMQ agora mostra que as mensagens foram redirecionadas para outras filas (dt-queue-1 = image-channel, dt-queue-2 = text-channel e dt-queue-3 = xml-channel):

Queues						
Name ↑	Number Of Pending Messages	Number Of Consumers	Messages Enqueued	Messages Dequeued	Views	Operations
dt-queue-1	2	0	2	0	Browse Active Consumers Active Producers atom rss	Send To Purge Delete
dt-queue-2	1	0	1	0	Browse Active Consumers Active Producers atom rss	Send To Purge Delete
dt-queue-3	3	0	3	0	Browse Active Consumers Active Producers atom rss	Send To Purge Delete
inbound-queue	0	1	6	6	Browse Active Consumers Active Producers atom rss	Send To Purge Delete

## CBR em Apache Camel

Um CBR é implementado em Camel usando o ChoiceProcessor, que funciona como um controle de fluxo procedural if-else em Java. O ChoiceProcessor pode ser inserido diretamente na configuração de uma rota usando choice(). O FileAdapter e o ContentBasedRouter implementados na seção anterior em JMS podem ter as rotas configuradas no mesmo lugar em Apache Camel. O exemplo abaixo ilustra a configuração das rotas:

```

226. context.addRoutes(new RouteBuilder() {
227.     @Override
228.     public void configure() throws Exception {
229.
230.         // Rota 1 - Do sistema de arquivos para a fila inbound-queue
231.         from("file:/tmp/jms/inbox")
232.             .setHeader("Name", header("CamelFileNameOnly"))
233.             .setHeader("Length", header("CamelFileLength"))
234.             .process(new Processor() {
235.                 public void process(Exchange exchange) throws Exception {
236.                     String name = exchange.getIn().getHeader("Name",
237.                         String.class);
238.                     String type = name.substring(name.lastIndexOf('.')+1)
239.                         .toLowerCase();
240.                     exchange.getIn().setHeader("Type", type);
241.                 }
242.             })
243.             .to("jms:queue:inbound-queue");
244.
245.         // Rota 2 - Da fila inbound-queue para as dt-queues passando pelo
246.         // CBR
247.         from("jms:queue:inbound-queue")
248.             .process(new Processor() {
249.                 public void process(Exchange exchange) throws Exception {
250.                     System.out.println("Received message: "
251.                         + exchange.getIn().getHeader("Type")));
252.             })
253.             .choice()
254.                 .when(header("Type").isEqualTo("text"))
255.                     .to("jms:queue:dt-queue-1");
256.                 .when(header("Type").isEqualTo("xml"))
257.                     .to("jms:queue:dt-queue-2");
258.                 .when(header("Type").isEqualTo("json"))
259.                     .to("jms:queue:dt-queue-3");
260.             .otherwise()
261.                 .to("jms:queue:dt-queue-1");
262.         }
263.     }
264. }
```

```

248.         }
249.     })
250.
251.     .choice()
252.         .when(header("Name").endsWith(".png"))
253.             .to("jms:queue:dt-queue-1")
254.         .when(header("Name").endsWith(".xml"))
255.             .to("jms:queue:dt-queue-2")
256.         .when(header("Name").endsWith(".txt"))
257.             .to("jms:queue:dt-queue-3")
258.         .otherwise()
259.             .to("jms:queue:invalid-queue")
260.         .stop() // boa prática - garante que inválidos não prossigam
261.     .end(); // fim do choice
262. ...
263. }
264. );

```

Executando o código acima, com os mesmos seis arquivos usados no exemplo com JMS, teremos o mesmo resultado. Os cabeçalhos “CamelFileName\*” guardam informações sobre o arquivo, como o nome do arquivo lido, tamanho, etc. (veja mais cabeçalhos em <http://camel.apache.org/file2.html>).

## CBR em Spring Integration

Spring Integration tem dois tipos de CBR: Payload Type Router, que é usado para rotear mensagens analisando o conteúdo do corpo da mensagem, e Header Value Router que faz o roteamento analisando valores de propriedades do cabeçalho.

O Payload Type Router pré-configurado no Spring permite mapear mensagens a canais com base no tipo de dados representado por uma classe Java [Spring]:

```

<int:payload-type-router input-channel="routingChannel">
    <int:mapping type="java.lang.String" channel="stringChannel" />
    <int:mapping type="java.lang.Integer" channel="integerChannel" />
</int:payload-type-router>

```

Já o Header Value Router verifica valores de propriedades do cabeçalho:

```

<int:header-value-router input-channel="incoming" header-name="Type">
    <int:mapping value="png" channel="dt-queue-1" />
    <int:mapping value="txt" channel="dt-queue-2" />
</int:header-value-router>

```

Para implementar o exemplo demonstrado em Spring Integration, podemos usar a seguinte configuração de rotas (sem usar JMS):

```

<int-file:inbound-channel-adapter id="files-channel"
    directory="file:/tmp/jms/inbox"
    filename-regex="^.*\.(png|txt|xml)$" />

```

```

<int:header-enricher input-channel="files-channel" output-channel="incoming-channel">
    <int:header name="Name" expression="payload?.name()"/>
    <int:header name="Length" expression="payload?.length()"/>
    <int:header name="Type" expression="payload?.name().substring(payload?.name().lastIndexOf('.')+1).toLowerCase()"/>
</int:header-enricher>

<int:channel id="png-channel" />
<int:channel id="txt-channel" />
<int:channel id="xml-channel" />

<int:header-value-router input-channel="incoming-channel" header-name="Type">
    <int:mapping value="png" channel="png-channel" />
    <int:mapping value="txt" channel="txt-channel" />
    <int:mapping value="xml" channel="xml-channel" />
</int:header-value-router>

```

## (30) Filtro de mensagens (Message Filter)

### Ícone



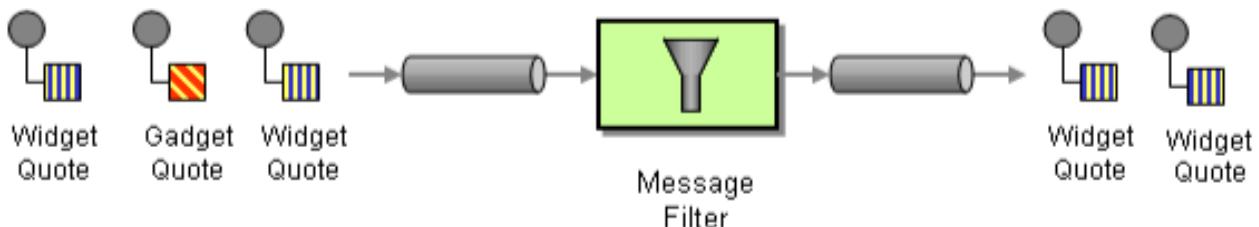
### Problema

“Como um componente pode evitar o recebimento de mensagens que não interessam?”

### Solução

“Use um tipo especial de roteador, um Filtro de Mensagens (Message Filter), para eliminar mensagens indesejadas de um canal com base em um conjunto de critérios.”

### Diagrama



### Descrição

Diferentemente do Roteador Baseado em Conteúdo, o Filtro de Mensagens (Message Filter) possui um único canal de saída. Se a mensagem passa pelos critérios de filtragem ela segue adiante para o canal

de saída. Se não, ela é descartada (pode também ser redirecionada para um Canal de Mensagens Inválidas).

Uma solução equivalente ao Roteador Baseado em Conteúdo poderia ser implementada conectando um canal Publicar-Inscrever às entradas de vários Filtros de Mensagens.

Um Filtro de Mensagens é útil quando um consumidor deseja consumir apenas algumas mensagens de um canal. Por exemplo, em vez de ler diretamente de um Canal Publica-Inscreve, o consumidor poderia consumir as mensagens a partir da saída de um Filtro de Mensagens que contenha apenas as mensagens de interesse. Essa solução é similar ao uso de um Consumidor Seletivo (Selective Consumer), onde a responsabilidade de filtrar as mensagens é transferida para o consumidor.

Um Filtro de Mensagens geralmente é implementado como um componente stateless, mas algumas tarefas que se baseiam em outras mensagens (ex: eliminação de mensagens duplicadas) requerem a manutenção de estado.

A escolha entre usar um CBR e uma coleção de Filtros de Mensagens consumindo mensagens de um Canal Publicar-Inscrever geralmente é decidido com base em quem mantém o controle sobre as decisões de roteamento. O CBR centraliza essas decisões enquanto que no Filtro de Mensagens a filtragem é distribuída. Se houver um novo tipo de mensagem a ser roteada, na solução com CBR seria necessário alterar sua lógica de roteamento, mas na solução com Filtros seria apenas adicionar mais um Filtro de Mensagens como assinante do canal. Também pode ocorrer que em uma solução com vários Filtros de Mensagens, mais de um deles consuma uma mensagem. No CBR apenas um dos potenciais consumidores receberá a mensagem roteada.

Um Roteador Baseado em Conteúdo é geralmente mais eficiente com canais baseados em filas (queues), e é tipicamente usado quando a mensagem segue um fluxo sequencial e definido (ex: transações). Message Filter funciona melhor com Canais Publicar-Inscrever e é frequentemente usado para notificações.

## Aplicações

Um Filtro de Mensagens é usado para filtrar as mensagens indesejadas de um canal, transferindo ao canal de saída apenas mensagens selecionadas.

### Filtro de Mensagens em Java / JMS

O Filtro de Mensagens abaixo (MessageTypeFilter) é similar ao CBR mostrado no exemplo anterior, mas ele está conectado a um canal Publicar-Inscrever (Topic) para onde são enviadas mensagens contendo arquivos, e repassa para a fila de saída apenas mensagens que sejam do tipo selecionado.

Para enviar os arquivos para o Topic usamos uma classe igual à FileAdapter do exemplo da seção anterior, mudando apenas o canal onde as mensagens são publicadas.

No método main() abaixo (de MessageTypeFilter) criamos dois filtros, ambos assinando o mesmo Topic, e consequentemente recebendo cópias das mesmas mensagens. O primeiro vai filtrar as

mensagens que forem do tipo “png” e o segundo apenas as mensagens “txt”. Cada um irá depositar as mensagens sobreviventes em uma fila própria:

```

265. ...
266.     public static void main(String[] args) throws Exception {
267.         Context ctx = new InitialContext();
268.         ConnectionFactory factory = ...;
269.         Destination inTopic = (Destination) ctx.lookup("files-topic");
270.         Connection con = factory.createConnection();
271.
272.         Destination imageChannel = (Destination) ctx.lookup("image-
channel");
273.         Destination textChannel = (Destination) ctx.lookup("text-
channel");
274.
275.         MessageTypeFilter imageFilter = new MessageTypeFilter("png");
276.         imageFilter.init(con, inTopic, imageChannel);
277.
278.         MessageTypeFilter textFilter = new MessageTypeFilter("txt");
279.         textFilter.init(con, inTopic, textChannel);
280.
281.         System.out.println("Waiting 60 seconds for messages...");
282.
283.         Thread.sleep(60000); // Will wait one minute for files
284.         con.close();
285.     }
286. }
```

Iremos colocar os mesmos seis arquivos na pasta /tmp/jms/inbox (3 XML, 2 PNG e 1 TXT), que serão consumidos e enviados para o Topic “inbound-topic” (“files-topic”).

Este é o código do MessageTypeFilter. Observe que só há uma saída e a mensagem só será redirecionada para ela se passar no filtro:

```

287.     public class MessageTypeFilter implements MessageListener {
288.         MessageProducer producer;
289.         String messageType;
290.
291.         public MessageTypeFilter(String messageType) {
292.             this.messageType = messageType;
293.         }
294.
295.         public void init(Connection con, Destination inTopic, Destination
outQueue) `

                throws NamingException, JMSEException {
296.             Session session = con.createSession(false,
Session.AUTO_ACKNOWLEDGE);
297.             MessageConsumer consumer = session.createConsumer(inTopic);
298.             producer = session.createProducer(outQueue);
```

```

299.         consumer.setMessageListener(this);
300.         con.start();
301.     }
302.
303.     @Override
304.     public void onMessage(Message message) {
305.         try {
306.             String typeProperty      = message.getStringProperty("Type");
307.             String filenameProperty = message.getStringProperty("Name");
308.             System.out.println(messageType.toUpperCase()
309.                                 + " Filter received: " + filenameProperty);
310.             if (typeProperty != null && typeProperty.equals(messageType) )
311.             {
312.                 producer.send(message);
313.                 System.out.println(messageType.toUpperCase()
314.                                     + " Filter selected : " + filenameProperty);
315.             }
316.         } catch (JMSEException e) {...}
317.     }
318.     public static void main(...) {...}
319. }
```

Executando a classe acima, ela ficará aguardando mensagens. O FileAdapter é executado em seguida enviando seis mensagens para o Topic de entrada. Quando isto acontecer, os dois filtros que assinam o Topic irão receber notificações com as mensagens. Cada filtro irá selecionar algumas mensagens. As que não forem selecionadas serão perdidas, pois não há outros assinantes.

```

Waiting 60 seconds for messages...
PNG Filter received: Africal1898.png
PNG Filter selected : Africal1898.png
PNG Filter received: asteroides.xml
PNG Filter discarded: asteroides.xml
PNG Filter received: canto_20.txt
PNG Filter discarded: canto_20.txt
PNG Filter received: jupiter.xml
PNG Filter discarded: jupiter.xml
PNG Filter received: movies.xml
PNG Filter discarded: movies.xml
PNG Filter received: TheFish.png
PNG Filter selected : TheFish.png

TXT Filter received: Africal1898.png
TXT Filter discarded: Africal1898.png
TXT Filter received: asteroides.xml
TXT Filter discarded: asteroides.xml
```

```

TXT Filter received: canto_20.txt
TXT Filter selected : canto_20.txt
TXT Filter received: jupiter.xml
TXT Filter discarded: jupiter.xml
TXT Filter received: movies.xml
TXT Filter discarded: movies.xml
TXT Filter received: TheFish.png
TXT Filter discarded: TheFish.png

```

## Filtro de Mensagens em Apache Camel

Filtros de Mensagens são processados usando em Camel usando a interface FilterProcessor. A configuração em Java DSL usa filter(), que recebe como parâmetro uma expressão Camel condicional (predicado). Pode-se também usar outras linguagens de expressão (XPath, JavaScript, etc.) ou implementar a interface Expression.

A configuração do filtro mostrado em JMS pode ser feito usando as seguintes rotas (veja classe FilterExample):

```

320.    context.addRoutes(new RouteBuilder() {
321.        @Override
322.        public void configure() throws Exception {
323.
324.            // Rota 1 - Do sistema de arquivos para a fila inbound-topic
325.            from("file:/tmp/jms/inbox")
326.                ...
327.                .to("jms:topic:inbound-topic");
328.
329.            // Rotas de filtros
330.            from("jms:topic:inbound-topic")
331.                .process(new Processor() {
332.                    public void process(Exchange exchange) throws Exception {
333.                        System.out.println("PNG filter received: "
334.                            +
335.                            exchange.getIn().getHeader("Name"));
336.                    }
337.                    .filter(header("Type").isEqualTo("png"))
338.                    .process(new Processor() {
339.                        public void process(Exchange exchange) throws Exception {
340.                            System.out.println("PNG filter SELECTED: "
341.                                +
342.                                exchange.getIn().getHeader("Name"));
343.                            }
344.                            .to("jms:queue:dt-queue-1");
345.
346.                            from("jms:topic:inbound-topic")
347.                                .process(new Processor() {

```

```

348.         public void process(Exchange exchange) throws Exception {
349.             System.out.println("TXT filter received: "
350.                                 +
351.             exchange.getIn().getHeader("Name") );
352.         }
353.         .filter(header("Type").isEqualTo("txt"))
354.         .process(new Processor() {
355.             public void process(Exchange exchange) throws Exception {
356.                 System.out.println("TXT filter SELECTED: "
357.                                     +
358.             exchange.getIn().getHeader("Name") );
359.         }
360.         .to("jms:queue:dt-queue-2");
361.     }
362. });

```

O exemplo acima pode ser mais compacto. Colocamos os filtros em rotas separadas para refletir melhor a arquitetura Dutos e Filtros, mas poderíamos tê-los definido em uma única rota.

Executando a aplicação, e pondo os mesmos arquivos na pasta /tmp/jms/inbox, teremos:

```

O servidor está no ar por 60 segundos.
PNG filter received: Africal1898.png
TXT filter received: Africal1898.png
PNG filter SELECTED: Africal1898.png
TXT filter received: asteroides.xml
TXT filter received: canto_20.txt
TXT filter SELECTED: canto_20.txt
PNG filter received: asteroides.xml
TXT filter received: jupiter.xml
PNG filter received: canto_20.txt
TXT filter received: movies.xml
PNG filter received: jupiter.xml
TXT filter received: TheFish.png
PNG filter received: movies.xml
PNG filter received: TheFish.png
PNG filter SELECTED: TheFish.png

```

## Filtro de Mensagens em Spring Integration

Em Spring Integration um Filtro de Mensagens é configurado via XML, mas a lógica de filtragem ou é implementada em uma expressão SpEL (Spring Expression Language), se for simples, ou em um bean.

Como nosso exemplo é simples poderíamos usar:

```
<int:filter input-channel="files-channel" output-channel="image-channel"
            expression="headers['Type'] == 'png'" />
```

Se a expressão for mais complexa, o ideal é construí-la em uma classe:

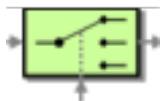
```
public class ImageTypeSelector implements MessageSelector {
    public boolean accept(Message<?> message) {
        String type = (String)message.getHeader("Type");
        if (type.equals("png")) {
            return true;
        }
        return false;
    }
}
```

Referenciar no arquivo de configuração como bean e associá-lo ao filtro. É possível também especificar um canal (Invalid Message Channel) para onde serão enviadas mensagens que não forem aproveitadas pelo filtro:

```
<int:filter input-channel="files-channel" output-channel="image-channel"
            discard-channel="invalid-channel ">
    <bean id="imageSelector"
          class="br.com.argonavis.si.examples.router.ImageTypeSelector" />
</int:filter>
```

## (31) Roteador dinâmico (Dynamic Router)

### Ícone



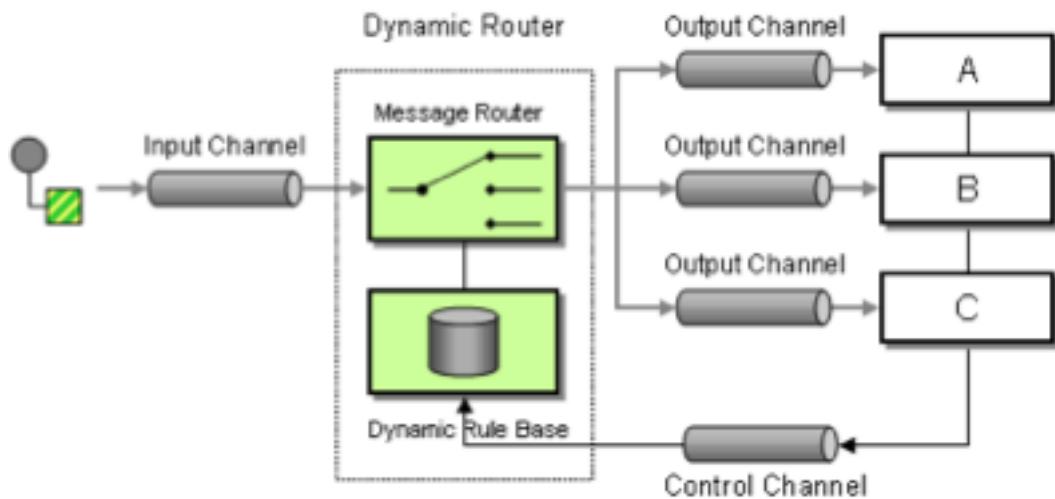
### Problema

“Como evitar a dependência do roteador em todos os destinos possíveis enquanto mantém a sua eficiência?”

### Solução

“Use um Roteador Dinâmico (Dynamic Router), que é capaz de se auto-configurar com base em mensagens de configuração enviadas pelos destinos participantes”

## Diagrama



## Descrição

Um Roteador Dinâmico (Dynamic Router) possui um canal de entrada de mensagens a serem roteadas, e vários canais de saída, como outros Message Router. A diferença é que possui um canal adicional de entrada onde recebe mensagens contendo as regras de roteamento. Esse canal é alimentado pelos receptores. Com os dados extraídos desse canal, um roteador dinâmico constrói uma tabela dinâmica de roteamento que usa para rotear as mensagens na saída.

Como as regras de roteamento são informadas por agentes externos (geralmente são os receptores que informam as regras), existe a possibilidade de haver conflitos entre elas. O Roteador Dinâmico precisa resolver esses conflitos antes de formar a tabela.

É preciso medir o custo-benefício desta implementação, já que tem potenciais impactos na performance, complexidade e dificuldade de depuração, embora possa ser mais eficiente (ao não precisar analisar conteúdo ou usar filtros). Uma implementação pode buscar um meio-termo entre um roteador estático e dinâmico decidindo quando e como atualizar a tabela de roteamento (ex: uma determinada aplicação pode gerar as tabelas de roteamento apenas no início do serviço; outra pode gerar periodicamente sempre que receber notificação de mudança de regras, e manter um cache de regras). É importante também logar as tabelas de roteamento, para facilitar a depuração.

É comum usar Roteadores Dinâmicos para a descoberta transparente de serviços em implementações de SOA: se uma aplicação pretende acessar um determinado serviço, ela envia uma mensagem a um Roteador Dinâmico contendo o nome do serviço, para que o Roteador possa redirecionar para o serviço correto. Uma generalização do conceito de Roteador Dinâmico, como um controlador central que processa regras complexas pode ser usado para implementar um padrão como Gerente de Processos.

## Aplicações

Um Roteador Dinâmico é usado quando o destino das mensagens não pode ser estabelecido previamente e pode mudar em tempo de execução.

### Roteador Dinâmico em Java / JMS

No exemplo abaixo implementamos um Roteador Dinâmico que redireciona mensagens que contém arquivos (5 arquivos enviados pelo FileAdapter dos exemplos anteriores) para canais diferentes, dependendo de quantos bytes esses canais já processaram. A classe DynamicRouter mantém um Map de regras contendo os canais para os quais envia mensagens e a quantidade de bytes que eles já processaram. Logo que uma mensagem é recebida no canal “inbound-channel”, ela é enviada para o Canal corrente (currentDestination) que é calculado a cada envio. Inicialmente esse canal será o “a-channel”, mas logo que a mensagem é recebida, o componente (classe MessageReceiver) envia de volta uma mensagem de controle informando quantos bytes já foram processados por ele até o momento, e o DynamicRouter usa essa informação para atualizar o Map de regras. A próxima mensagem será enviada para o canal que tiver recebido a menor quantidade de dados.

A classe MessageReceiver é usada para construir dois consumidores. Um para o “a-channel”, e outro para o “b-channel”:

```

363. public class MessageReceiver implements MessageListener {
364. ...
365.     public static void main(String[] args) throws Exception {
366.         ...
367.         MessageReceiver receiver1 = new MessageReceiver("a-channel");
368.         receiver1.init(con);
369.
370.         MessageReceiver receiver2 = new MessageReceiver("b-channel");
371.         receiver2.init(con);
372.
373.         System.out.println("Waiting 60 seconds for messages...");
374.         Thread.sleep(60000); // Will wait one minute for files
375.         con.close();
376.     }
377. }
```

Sempre que recebe uma Mensagem-documento (contendo um arquivo), a classe MessageReceiver envia uma Mensagem-evento para o canal “ctrl-channel” contendo informações de controle (uma mensagem contendo duas propriedades, informando o nome do canal “ChannelName” e a quantidade de dados que ele já processou “TotalSize”):

```

378. public class MessageReceiver implements MessageListener {
379.
380.     private String channelName;
381.     private MessageProducer control;
382.     private Session session;
383.     private Queue dataChannel;
```

```

384.     private Topic controlChannel;
385.
386.     private long size = 0; // accumulates the size of messages received
387.
388.     public MessageReceiver(String channelName) {
389.         this.channelName = channelName;
390.     }
391.
392.     public void init(Connection con) throws NamingException,
JMSEException {
393.         Context ctx = new InitialContext();
394.         this.dataChannel = (Queue) ctx.lookup(channelName);
395.         this.controlChannel = (Topic) ctx.lookup("ctrl-channel");
396.
397.         session = con.createSession(false, Session.AUTO_ACKNOWLEDGE);
398.         MessageConsumer consumer =
session.createConsumer(dataChannel);
399.         control = session.createProducer(controlChannel);
400.         consumer.setMessageListener(this);
401.         con.start();
402.     }
403.
404.     public void onMessage(Message msg) {
405.         try {
406.             Long fileSize = msg.getLongProperty("Length");
407.             String fileName = msg.getStringProperty("Name");
408.             System.out.println(channelName.toUpperCase()
+ " received message with file " + fileName
+ "and " + fileSize + " bytes.");
409.
410.             size += fileSize;
411.             System.out.println(channelName.toUpperCase()
+ ": Total size is now " + size);
412.
413.             Message controlMessage = session.createMessage();
414.             controlMessage.setLongProperty("TotalSize", size);
415.             controlMessage.setStringProperty("ChannelName",
channelName);
416.             control.send(controlMessage);
417.         } catch (JMSEException e) {
418.             e.printStackTrace();
419.         }
420.     }
421. }

```

A classe DynamicRouter inicializa o Map de regras com entradas contendo o nome de cada canal associado aos dados que já processou (inicialmente zero). Cada mensagem recebida precisa saber o seu destino, que é calculado em calculateDestination(). Em seguida a mensagem é enviada e o roteador espera até 10 segundos por uma mensagem de controle que poderá alterar a tabela de regras.

```
422. public class DynamicRouter implements MessageListener {  
423.  
424.     private Session messageSession;  
425.     private Destination controlChannel;  
426.     private Destination inboundChannel;  
427.     private Destination currentDestination;  
428.     private MessageConsumer control;  
429.  
430.     Map<String, Long> rules = new HashMap<>();  
431.  
432.     public void init(Connection con) throws NamingException,  
JMSEException {  
433.         Context ctx = new InitialContext();  
434.         this.controlChannel = (Destination) ctx.lookup("ctrl-  
channel");  
435.         this.inboundChannel = (Destination) ctx.lookup("inbound-  
channel");  
436.  
437.         rules.put("a-channel", 0L);  
438.         rules.put("b-channel", 0L);  
439.  
440.         messageSession = con.createSession(false,  
Session.AUTO_ACKNOWLEDGE);  
441.  
442.         MessageConsumer consumer =  
                messageSession.createConsumer(inboundChannel);  
443.         consumer.setMessageListener(this);  
444.  
445.         Session controlSession =  
                con.createSession(false, Session.AUTO_ACKNOWLEDGE);  
446.         control = controlSession.createConsumer(controlChannel);  
447.         con.start();  
448.     }  
449.  
450.     @Override  
451.     public void onMessage(Message message) {  
452.         try {  
453.             String filename = message.getStringProperty("Name");  
454.             System.out.println("Received message: " + filename);  
455.  
456.             calculateDestination(); // dynamic routing!  
457.  
458.             MessageProducer producer = messageSession  
                    .createProducer(currentDestination);  
459.             producer.send(message);  
460.             System.out.println("Message with " + filename + " sent  
to "  
462.                             + currentDestination);  
463.
```

```

464.             System.out.println("Wait for routing info (timeout
10s)");
465.             Message controlMessage = control.receive(10000);
466.             Long totalSize =
controlMessage.getLongProperty("TotalSize");
467.             String channelName = controlMessage
468.                     .getStringProperty("ChannelName");
469.             rules.put(channelName, totalSize);
470.         } catch (Exception e) {
471.             e.printStackTrace();
472.         }
473.     }
474.
475.     /**
476.      * Returns channel that received the least amount of data
477.     */
478.     private void calculateDestination() throws NamingException {
479.         Context ctx = new InitialContext();
480.         Long min = Long.MAX_VALUE;
481.         String channelName = "a-channel"; // initially use this
channel
482.         for (Map.Entry<String, Long> entry : rules.entrySet()) {
483.             System.out.println(entry.getKey() + ":" +
entry.getValue());
484.             if (entry.getValue() < min) {
485.                 min = entry.getValue();
486.                 channelName = entry.getKey();
487.             }
488.         }
489.         currentDestination = (Destination) ctx.lookup(channelName);
490.     }
491.
492.     public static void main(String[] args) throws Exception {
493.         Context ctx = new InitialContext();
494.         ConnectionFactory factory = (ConnectionFactory) ctx
495.                         .lookup("ConnectionFactory");
496.         Connection con = factory.createConnection();
497.
498.         DynamicRouter router = new DynamicRouter();
499.         router.init(con);
500.
501.         System.out.println("Waiting 60 seconds for messages...");
502.         Thread.sleep(60000); // Will wait one minute for files
503.         con.close();
504.     }
505.
506. }

```

Na execução de testes foram enviados cinco arquivos para “inbound-queue” nesta ordem:

```
asteroides.xml: 925386 bytes
canto_20.txt: 4113 bytes
jupiter.xml: 11955 bytes
movies.xml: 475 bytes
TheFish.png: 85393 bytes
```

A primeira será enviada para o “b-channel” (por causa do algoritmo usado para calcular o destino e ambos inicialmente são iguais a zero), mas logo que o roteador for calcular a rota novamente, ele mudará para o “a-channel” porque este será menor, e assim por diante. Como o controle e as mensagens rodam em threads separados, pode acontecer das regras não chegarem a tempo e às vezes o canal que já processou mais receber uma mensagem.

## Roteador Dinâmico em Apache Camel

Em Camel, o método `dynamicRouter()` recebe um bean que deve retornar um Endpoint para onde a rota deve seguir. O `DynamicRouter` será chamado cada vez que a rota for usada. É preciso retornar null para que ele pare.

```
public class MyRouter {
    private List<String> payloads = ...
    private int chamadas = 0;
    public String tabela(String payload) {
        payloads.add(payload);
        chamadas++;

        if (chamadas == 1) {
            return "mock:a";
        } else if (chamadas == 2) {
            return "mock:b, mock:c";
        } else if (chamadas == 3) {
            return "direct:foo";
        } else if (chamadas == 4) {
            return "mock:result";
        }
        return null;
    }
}
```

Para configurar a rota:

```
from("jms:queue:inicio").dynamicRouter(method(MyRouter.class, "tabela"));
```

## (32) Lista de receptores (Recipient List)

### Ícone



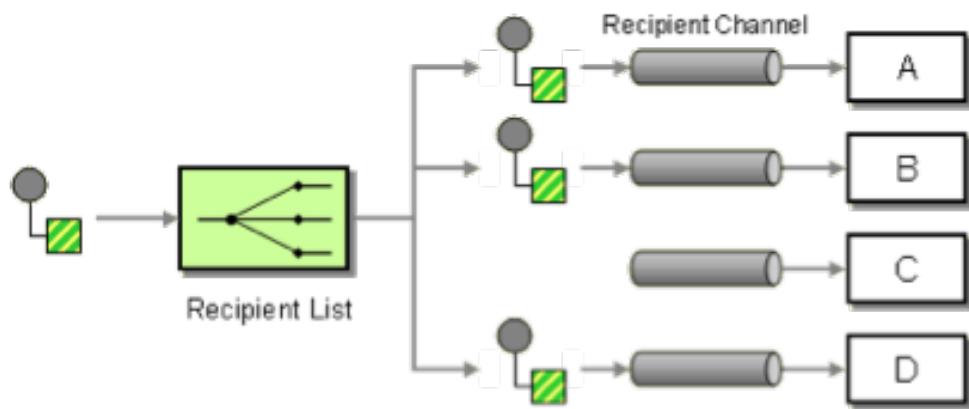
## Problema

“Como rotear uma mensagem para uma lista de receptores especificados dinamicamente?”

## Solução

“Defina um canal para cada receptor. Depois use um Lista de Receptores (Recipient List) para inspecionar uma mensagem entrante, determine a lista de receptores desejados, e repasse a mensagem para todos os canais associados com os receptores da lista.”

## Diagrama



## Descrição

Uma implementação de Lista de Receptores (Recipient List) pode ser usada para rotear mensagens a endereços específicos, conhecidos previamente (ou no momento em que a imagem será reendereçada). Primeiro, o componente que lê a imagem de um canal precisa obter a lista de endereços para o qual ela será enviada. Isto pode estar na própria mensagem. Com essa informação, o roteador envia a mensagem a todos os canais da lista.

A lista de endereços não precisa estar na mensagem. Pode ser uma lista estática ou pode ser obtida dinamicamente (ex: os receptores interessados podem registrar-se). Listas estáticas tornam o roteador fortemente acoplado aos canais de saída.

A solução de integração proporcionada por uma Lista de Receptores pode, às vezes, ser implementada mais facilmente usando um Canal Publica-Inscreve. A diferença é que todas as mensagens enviadas a um Canal Publica-Inscreve são destinadas aos mesmos destinatários (a responsabilidade de filtrar as mensagens indesejadas é deles), enquanto que em uma Lista de Receptores, a lista pode ser computada a cada mensagem. Em geral, quanto mais distantes estiverem os receptores, mais eficiente é manter a responsabilidade de seleção das mensagens com o remetente, ou seja, usar uma Lista de Receptores.

A Lista de Receptores deve ser executada em uma única transação, ou seja, só deve considerar que as mensagens forem enviadas depois que todos os canais envolvidos tenham recebido suas mensagens. Pode ser implementada de forma stateful, para que possa lembrar quais mensagens já foram enviadas

e poder recuperar-se depois de uma falha (ou re-enviar as mensagens se os receptores forem Receptores Idempotentes).

## Aplicações

Quando os canais para onde uma mensagem deve ser enviada já forem conhecidos, pode-se usar uma Lista de Receptores (Recipient List) que providenciará o envio direto aos canais desejados, já que não será necessário analisar a imagem nem computar rotas. A lista pode vir de qualquer lugar. Pode ser estática e embutida no próprio roteador, pode ser dinâmica e calculada com base em fatores externos, e pode ser incluída em cada mensagem pelo remetente (neste caso a análise do conteúdo - geralmente um ou mais cabeçalhos - será necessário).

### Lista de Receptores em Java / JMS

O exemplo abaixo utiliza o mesmo Adaptador de Canal usado na seção sobre CBR que envia mensagens contendo arquivos para “inbound-queue” (“inbound-channel”). Desta vez enviaremos cada mensagem para uma Lista de Receptores que será calculada com base no tipo do arquivo contido na mensagem. Como documentos XML também são documentos de texto, as mensagens XML serão enviadas tanto para a fila “xml-queue” quanto para a fila “text-queue”. Todas as mensagens serão enviadas também para a fila “all-queue”.

```
507. public class RecipientList implements MessageListener {  
508.     private Session session;  
509.     private Destination imageChannel;  
510.     private Destination textChannel;  
511.     private Destination xmlChannel;  
512.     private Destination allChannel;  
513.     private Destination inboundChannel;  
514.  
515.     public void init(Connection con) throws NamingException,  
JMSEException {  
516.         Context ctx = new InitialContext();  
517.         this.imageChannel = (Destination) ctx.lookup("image-channel");  
518.         this.textChannel = (Destination) ctx.lookup("text-channel");  
519.         this.xmlChannel = (Destination) ctx.lookup("xml-channel");  
520.         this.allChannel = (Destination) ctx.lookup("all-channel");  
521.         this.inboundChannel = (Destination) ctx.lookup("inbound-  
channel");  
522.  
523.         session = con.createSession(false, Session.AUTO_ACKNOWLEDGE);  
524.         MessageConsumer consumer =  
session.createConsumer(inboundChannel);  
525.         consumer.setMessageListener(this);  
526.         con.start();  
527.     }  
528.  
529.     @Override  
530.     public void onMessage(Message message) {
```

```
531.         try {
532.             String filename = message.getStringProperty("Name");
533.             String type = message.getStringProperty("Type");
534.             System.out.println("Received message: " + filename);
535.
536.             if (type != null && type.equals("png")) {
537.                 routeMessage(imageChannel, message);
538.                 System.out.println("Routing " + filename + " to image-
queue");
539.             }
540.             if (type != null && type.equals("txt") || type.equals("xml")) {
541.                 routeMessage(textChannel, message);
542.                 System.out.println("Routing " + filename + " to text-
queue");
543.             }
544.             if (type != null && type.equals("xml")) {
545.                 routeMessage(xmlChannel, message);
546.                 System.out.println("Routing " + filename + " to xml-
queue");
547.             }
548.             routeMessage(allChannel, message);
549.             System.out.println("Routing " + filename + " to all-
queue");
550.
551.         } catch (JMSEException e) {
552.             e.printStackTrace();
553.         }
554.     }
555.
556.     public void routeMessage(Destination destination, Message message)
557.         throws JMSEException {
558.         MessageProducer producer =
559.             session.createProducer(destination);
560.             producer.send(message);
561.     }
562.
563.     public static void main(String[] args) throws Exception {
564.         Context ctx = new InitialContext();
565.         ConnectionFactory factory =
566.             (ConnectionFactory) ctx.lookup("ConnectionFactory");
567.         Connection con = factory.createConnection();
568.
569.         RecipientList router = new RecipientList();
570.         router.init(con);
571.
572.         System.out.println("Waiting 60 seconds for messages...");
573.         Thread.sleep(60000); // Will wait one minute for files
574.         con.close();
575.     }
576. }
```

```
572.      }
573. }
```

Executando o FileAdapter, enviando os mesmos seis arquivos enviados nos exemplos anteriores, e depois rodando a classe acima, as mensagens serão consumidas do “inbound-queue” e repassadas para um ou mais endereços, de acordo com seu tipo. Esta é a saída do programa:

```
Waiting 60 seconds for messages...
Received message: Africal898.png
Routing Africal898.png to dt-queue-1
Routing Africal898.png to all-queue
Received message: asteroides.xml
Routing asteroides.xml to dt-queue-2
Routing asteroides.xml to dt-queue-3
Routing asteroides.xml to all-queue
Received message: canto_20.txt
Routing canto_20.txt to dt-queue-2
Routing canto_20.txt to all-queue
Received message: jupiter.xml
Routing jupiter.xml to dt-queue-2
Routing jupiter.xml to dt-queue-3
Routing jupiter.xml to all-queue
Received message: movies.xml
Routing movies.xml to dt-queue-2
Routing movies.xml to dt-queue-3
Routing movies.xml to all-queue
Received message: TheFish.png
Routing TheFish.png to dt-queue-1
Routing TheFish.png to all-queue
```

E esta a console do ActiveMQ. Observe que a fila “all-queue” possui uma cópia de todas as mensagens que estavam na “inbound-queue”:

Queues						
Name ↑	Number Of Pending Messages	Number Of Consumers	Messages Enqueued	Messages Dequeued	Views	Operations
all-queue	6	0	6	0	Browse Active Consumers Active Producers atom rss	Send To Purge Delete
dt-queue-1	2	0	2	0	Browse Active Consumers Active Producers atom rss	Send To Purge Delete
dt-queue-2	4	0	4	0	Browse Active Consumers Active Producers atom rss	Send To Purge Delete
dt-queue-3	3	0	3	0	Browse Active Consumers Active Producers atom rss	Send To Purge Delete
inbound-queue	0	1	6	6	Browse Active Consumers Active Producers atom rss	Send To Purge Delete

## Lista de Receptores em Apache Camel

A documentação do Apache Camel descreve duas maneiras de configurar uma Lista de Receptores em Java DSL: usando `multicast().to(lista)`, onde lista é uma sequência de destinos estáticos, ou usando `.recipientList(expressao)`, onde a expressão retorna uma `java.util.List` de destinos.

No exemplo abaixo, no momento em que os arquivos são empacotados nas mensagens enviadas para a fila “inbound-queue”, é calculada uma lista de destinos, com base no tipo do arquivo. Essa lista de destinos guardada em uma propriedade de cabeçalho “List”. O componente seguinte, que consome mensagens da “inbound-queue” usa a lista de endereços do cabeçalho “List” para endereçar cada mensagem:

```

574. context.addRoutes(new RouteBuilder() {
575.     @Override
576.     public void configure() throws Exception {
577.
578.         // Rota 1 - Do sistema de arquivos para a fila inbound-topic
579.         from("file:/tmp/jms/inbox")
580.             .setHeader("Name", header("CamelFileNameOnly"))
581.             .setHeader("Length", header("CamelFileNameOnly"))
582.             .process(new Processor() {
583.                 public void process(Exchange exchange) throws Exception {
584.                     String name = exchange.getIn().getHeader("Name",
585. String.class);
586.                     String type =
587.                         name.substring(name.lastIndexOf('.') +
588. 1).toLowerCase();
589.                     exchange.getIn().setHeader("Type", type);
590.                     StringBuilder recipients = new StringBuilder();
591.
592.                     switch(type) {
593.                         case "png":
594.                             recipients.append("jms:queue:dt-queue-1,");
595.                             break;
596.                         case "xml":
597.                             recipients.append("jms:queue:dt-queue-3,");
598.                             break;
599.                         case "txt":
600.                             recipients.append("jms:queue:dt-queue-2,");
601.                             break;
602.                     }
603.                     recipients.append("jms:queue:all-queue");
604.                     exchange.getIn().setHeader("List",
605. recipients.toString());
606.                 }
607.             })
608.             .to("jms:queue:inbound-queue");
609.
610.         // Rota 2
611.         from("jms:queue:inbound-queue")
612. 
```

```

607.         .process(new Processor() {
608.             public void process(Exchange exchange) throws Exception {
609.                 Message msg = exchange.getIn();
610.                 String list = msg.getHeader("List", String.class);
611.                 String filename = msg.getHeader("Name", String.class);
612.                 System.out.println("File " + filename
613.                         + " redirected to: " + list);
614.             }
615.         })
616.     }
617. });

```

O resultado é o mesmo obtido pelo exemplo em JMS:

```

O servidor está no ar por 60 segundos.
File Africal898.png redirected to:
    jms:queue:dt-queue-1,jms:queue:all-queue
File asteroides.xml redirected to:
    jms:queue:dt-queue-3,jms:queue:dt-queue-2,jms:queue:all-queue
File canto_20.txt redirected to:
    jms:queue:dt-queue-2,jms:queue:all-queue
File jupiter.xml redirected to:
    jms:queue:dt-queue-3,jms:queue:dt-queue-2,jms:queue:all-queue
File movies.xml redirected to:
    jms:queue:dt-queue-3,jms:queue:dt-queue-2,jms:queue:all-queue
File TheFish.png redirected to:
    jms:queue:dt-queue-1,jms:queue:all-queue

```

## [Lista de Receptores em Spring Integration](#)

Spring Integration implementa Lista de Receptores através do RecipientListRouter que pode ser configurado usando <recipiente-list-router> da forma abaixo, para fazer o mesmo roteamento realizado nos exemplos mostrados.

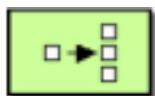
```

<int:recipient-list-router input-channel="routingChannel">
    <int:recipient channel="image-channel"
        selector-expression="headers['Type'].equals('png')"/>
    <int:recipient channel="text-channel"
        selector-expression="headers['Type'].equals('txt')
            || headers['Type'].equals('xml')"/>
    <int:recipient channel="xml-channel"
        selector-expression="headers['Type'].equals('xml')"/>
    <int:recipient channel="text-channel" />
</int:recipient-list-router>

```

## (33) Divisor (Splitter)

### Ícone



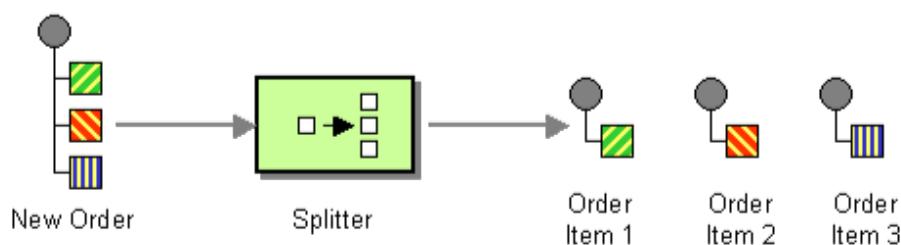
### Problema

“Como processar uma mensagem se ela contém múltiplos elementos, onde cada um pode ter que ser processado de maneira diferente?”

### Solução

“Use um Divisor (Splitter) para partir a mensagem composta em uma série de mensagens individuais, cada uma contendo dados relacionados a um item.”

### Diagrama



### Descrição

Um Divisor (Splitter) é um componente que divide o conteúdo de uma mensagem e re-envia no seu canal de saída como duas ou mais mensagens. Um Divisor também pode enviar cada tipo de mensagem produzida a canais diferentes, combinando sua funcionalidade com aquela de um Roteador Baseado em Conteúdo.

Pode haver vários motivos para dividir uma mensagem. A divisão pode ser estática (ex: grandes blocos de dados divididos em partes iguais para facilitar a transmissão), ou pode ser baseada em algum critério computado por um algoritmo (fragmentos de XML, componentes de uma estrutura composta, etc.) Uma mensagem pode ser dividida visando a sua re-agregação posterior ou não. Um Divisor combinado com um Filtro de Conteúdo pode ser usado para receber uma mensagem maior, extrair as partes de interesse e enviá-los em mensagens separadas aos canais de saída.

A divisão de uma mensagem em partes geralmente requer a adição de meta-informação adicional nos cabeçalhos e repetição de cabeçalhos que existiam na mensagem original. Às vezes também é necessário incluir dados no conteúdo que foi partido, por exemplo, incluir um ID para identificar a mensagem como um fragmento e permitir a sua agregação posterior, ou fornecer um elemento raiz para que um fragmento de XML possa ser validado.

## Aplicações

Um Divisor (Splitter) deve ser usado sempre que houver necessidade de partir uma mensagem maior em partes menores. Aplicações incluem tornar mais eficiente a transmissão dos dados (enviar pacotes em tamanhos específicos) e simplificar o processamento (distribuir o processamento de partes de uma mensagem separadamente em componentes especializados).

### Divisor em Java/JMS

No exemplo abaixo iremos enviar um documento XML que possui partes que teriam que ser processadas separadamente (em uma arquitetura CMP ou Espalha-Recolhe). Para isto precisa ser dividido. Iremos arrancar o elemento raiz e enviar cada um dos elementos-filho do XML como uma mensagem separada.

```
<sistemaEstelar>
    <centro>
        <imagem href="sun.gif" />
        <estrela nome="Sol" diametrokm="1390000" cor="amarela" />
    </centro>
    <orbita raioMedUA="0.387">
        <planeta id="p1" nome="Mercúrio" diametrokm="4879">
            <imagem href="mercury.jpg" />
        </planeta>
    </orbita>
    <orbita raioMedUA="0.723">...</orbita>
    <orbita raioMedUA="1">...</orbita>
    <orbita raioMedUA="1.52">...</orbita>
    .
</ sistemaEstelar >
```

O Divisor precisará dividir o XML em múltiplas partes antes de enviar as mensagens. Usaremos o processador abaixo, que localiza os elementos-filho usando XPath e devolve uma List contendo um fragmento do documento original.

```
618. public class SolarSystemSplitterProcessor {
619.     public List<String> split(String xmlText) {
620.         try {
621.             List<String> documents = new ArrayList<>();
622.
623.             DocumentBuilderFactory dbf =
624.                 DocumentBuilderFactory.newInstance();
625.             DocumentBuilder db = dbf.newDocumentBuilder();
626.             Document doc =
627.                 db.parse(new ByteArrayInputStream(xmlText.getBytes("UTF-
8")));
628.             XPath xpath = XPathFactory.newInstance().newXPath();
```

```

628.         Node header =
629.             (Node) xpath.evaluate("//centro", doc,
XPathConstants.NODE);
630.         NodeList nodes =
631.             (NodeList)xpath.evaluate("//orbita", doc,
XPathConstants.NODESET);
632.         documents.add(XMLUtils.nodeToString(header));
633.         for (int i = 0; i < nodes.getLength(); i++) {
634.             documents.add(XMLUtils.nodeToString(nodes.item(i)));
635.         }
636.         return documents;
637.     } catch (Exception e) {
638.         e.printStackTrace();
639.         return null;
640.     }
641. }
642. }
```

O Divisor usará o processador listado acima para construir cada mensagem. Além do novo payload em XML, serão adicionadas três propriedades na mensagem: “SequenceSize” informando o tamanho da coleção, “SequencePosition”, informando que posição o fragmento de XML ocupava no documento anterior, e um JMSCorrelationID que está sendo usado para identificar a sequência. O JMSMessageID da primeira mensagem será o ID da sequência. Ele também adicionará um cabeçalho “Type” para que outro componente possa rapidamente identificar essa série e saber se é capaz de processá-la.

```

643. public class SolarSystemMessageSplitter implements MessageListener {
644.     private Session session;
645.     private MessageProducer producer;
646.     private MessageConsumer consumer;
647.
648.     Map<String, Destination> routes;
649.
650.     public SolarSystemMessageSplitter(Connection con, Destination in,
                                         Destination out)
                                         throws JMSEException {
651.         session = con.createSession(false, Session.AUTO_ACKNOWLEDGE);
652.         consumer = session.createConsumer(in);
653.         producer = session.createProducer(out);
654.         consumer.setMessageListener(this);
655.         con.start();
656.     }
657.
658.     @Override
659.     public void onMessage(Message message) {
660.         try {
661.             String messageID = message.getJMSMessageID();
662.             String type = message.getStringProperty("Type");

```

```
663.  
664.        if(type != null && type.equals("Solar System")) {  
665.            System.out.println("Processing one file.");  
666.            TextMessage tm = (TextMessage)message;  
667.            String payload = tm.getText();  
668.            List<String> splitPayload = new  
669.                SolarSystemSplitterProcessor().split(payload);  
670.  
671.            for(int i = 0; i < splitPayload.size(); i++) {  
672.                TextMessage newMessage =  
673.                    session.createTextMessage(splitPayload.get(i));  
674.                newMessage.setJMSCorrelationID(messageID); //  
SequenceID  
675.                newMessage.setIntProperty("SequencePosition", i+1);  
676.                newMessage.setIntProperty("SequenceSize",  
677.                    splitPayload.size());  
678.                newMessage.setStringProperty("Type", "Solar System  
Fragment");  
679.                producer.send(newMessage);  
680.            }  
681.        } else {  
682.            producer.send(message);  
683.        }  
684.    }  
685.  
686.    public static void main(String[] args) {  
687.        Connection con = null;  
688.        try {  
689.            Context ctx = new InitialContext();  
690.            ConnectionFactory factory =  
691.                (ConnectionFactory) ctx.lookup("ConnectionFactory");  
692.            con = factory.createConnection();  
693.            Destination from = (Destination) ctx.lookup("a-channel");  
694.            Destination to = (Destination) ctx.lookup("b-channel");  
695.  
696.            new SolarSystemMessageSplitter(con, from, to);  
697.  
698.            System.out.println("Receiving messages for 60 seconds...");  
699.            Thread.sleep(60000);  
700.            System.out.println("Done.");  
701.            con.close();  
702.  
703.        } catch (Exception e) {...} finally {...}  
704.    }  
705.}
```

Enviaremos primeiro o arquivo para um canal de entrada: “a-channel” usando um produtor simples e rotulando a mensagem com um cabeçalho de tipo:

```
706. public void send() throws JMSEException, IOException {
707.     String data = XMLUtils.loadFile("sol.xml");
708.     TextMessage message = session.createTextMessage(data);
709.     message.setStringProperty("Type", "Solar System");
710.     producer.send(message);
711. }
```

A console do ActiveMQ mostra que o “a-channel” recebeu uma mensagem.

Queues						
Name	Number Of Pending Messages	Number Of Consumers	Messages Enqueued	Messages Dequeued	Views	Operations
a-channel	1	0	1	0	<a href="#">Browse Active Consumers</a> <a href="#">Active Producers</a>  atom  rss	<a href="#">Send To</a> <a href="#">Purge</a> <a href="#">Delete</a>

Em seguida executamos o Divisor, que irá consumir a mensagem e depositar o resultado na “b-channel”:

Queues						
Name ↑	Number Of Pending Messages	Number Of Consumers	Messages Enqueued	Messages Dequeued	Views	Operations
a-channel	0	1	1	1	<a href="#">Browse Active Consumers</a> <a href="#">Active Producers</a>  atom  rss	<a href="#">Send To</a> <a href="#">Purge</a> <a href="#">Delete</a>
b-channel	11	0	11	0	<a href="#">Browse Active Consumers</a> <a href="#">Active Producers</a>  atom  rss	<a href="#">Send To</a> <a href="#">Purge</a> <a href="#">Delete</a>

A ordem das mensagens não está garantida. Dependendo de como forem consumidas poderão chegar fora de ordem. Normalmente elas também devem ter um Prazo de Validade, pois se elas não chegarem todas no destino, poderá ser impossível construir a mensagem novamente, e os fragmentos devem ser enviados a um Invalid-Message Channel.

Na próxima seção mostraremos como reconstruir a mensagem usando um Agregador.

## Divisor em Apache Camel

Em Camel, o método `Split()` do `RouteBuilder` permite dividir o payload de uma mensagem de acordo com uma expressão simples ou um método em um processador dedicado. A expressão ou método deve retornar uma lista de partes de mensagens.

```
from("jms:a-channel").split().method(SolarSystemSplitterProcessor.class,
"split")
.to("jms:b-channel");
```

Se for uma expressão simples, pode-se usar um delimitador:

```
from("jms:a-channel").split(body.tokenize("\n"))
.to("jms:b-channel");
```

Splitter copia todos os cabeçalhos da mensagem recebida e repete nas mensagens divididas. Também adiciona cabeçalho `SPLIT_SIZE` E `SPLIT_COUNTER` (quantidade de partes e número sequencial) que irá facilitar o trabalho do Agregador e Re-sequenciador.

## Divisor em Spring Integration

Em Spring Integration existe o elemento `<splitter>` que recebe uma mensagem em um canal e deposita as partes em outro usando um bean e método que receba uma `Message` ou `payload` e devolva uma coleção ou array de `Message` ou `payload`.

```
<splitter id="testSplitter" input-channel="inChannel" method="split"
          output-channel="outChannel">
    <beans:bean class="br.com...SolarSystemSplitterProcessor "/>
</splitter>
```

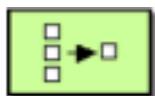
O bean pode ser qualquer POJO. Se o método não for explicitado no XML, pode ser indicado com uma anotação:

```
public class SolarSystemSplitterProcessor {
    @Splitter
    List<String> split(String xmlText) {
        ...
        return documents;
    }
}
```

O Splitter define os cabeçalhos `CORRELATION_ID`, `SEQUENCE_SIZE`, e `SEQUENCE_NUMBER` que irá facilitar o trabalho do Agregador ou Re-sequenciador mais adiante.

## (34) Agregador (Aggregator)

### Ícone



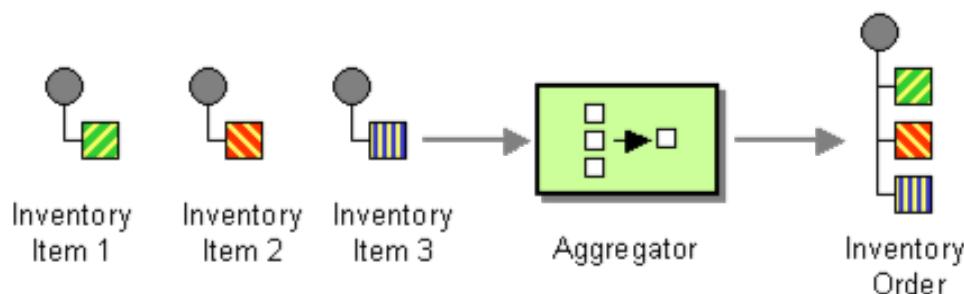
### Problema

“Como combinar os resultados de mensagens individuais, mas relacionadas, para que possam ser processadas como uma só?”

### Solução

“Use um filtro stateful, um Agregador (Aggregator), para coletar e armazenar mensagens individuais até que um conjunto completo de mensagens relacionadas tenha sido recebido. Depois, o Agregador publica uma única mensagem obtida a partir das mensagens individuais.”

### Diagrama



### Descrição

Um Agregador (Aggregator) recebe um conjunto de mensagens na entrada e devolve uma única mensagem na saída. É um componente *stateful*, pois precisa analisar cada mensagem e guardar as informações que serão usadas para determinar se elas serão combinadas. Quando recebe um conjunto completo de mensagens, seleciona dados de cada uma delas e organiza uma mensagem agregada, que é repassada a um canal de saída. Um Agregador faz o inverso do Divisor (Splitter). É comum que sejam usados em conjunto.

As mensagens que serão agregadas precisam estar co-relacionadas. Pode ser um cabeçalho com um ID em comum ou algum critério computado dinamicamente. Essa informação deve ser suficiente para que o agregador saiba como agrupar mensagens co-relacionadas (a ordem, que partes devem ser extraídas ou adicionadas na mensagem final) e quando o grupo de mensagens a serem combinadas estiver completo (como saber se todas já foram recebidas). Se a mensagem foi dividida previamente por um Divisor, os dois componentes podem ser coordenados de forma a padronizar as informações usadas na divisão e agregação.

Um agregador pode fazer mais que simplesmente colar os dados de mensagens diferentes. Pode interpretar os dados e gerar uma mensagem resultante mais condensada, eliminar informações redundantes e utilizar informações das mensagens agregadas para gerar mensagem de saída completamente diferente, com base em informações obtidas da combinação das mensagens de entrada.

Por exemplo, um agregador pode receber várias mensagens contendo pedidos de compra e combiná-los em um pedido novo, contendo a soma dos totais dos pedidos anteriores (omitindo o valor individual que consistia em cada um deles). Outra possibilidade é ler várias mensagens, construir uma agregação de todas elas, e selecionar apenas uma (ex: o menor preço).

## Aplicações

Um Agregador (Aggregator) deve ser usado sempre que for necessário combinar informações presentes em várias mensagens recebidas em uma só mensagem de saída.

### Agregador em Java / JMS

Um Agregador que reconstrói a mensagem dividida no Splitter está listado abaixo. A estratégia de agregação deste Agregador está no método `reassemble()` abaixo e consiste simplesmente de combinar as partes da mensagem que foram coletadas e envolver o resultado em um elemento raiz “`<joined>`”.

```

712. public class SolarSystemMessageAggregator implements MessageListener {
713.     ...
714.
715.     Map<String, Set<TextMessage>> messageSets = new HashMap<>();
716.
717.     public SolarSystemMessageAggregator(Connection con, Destination in,
718.                                         Destination out)
719.                                         throws JMSEException {...}
720.
721.     @Override
722.     public void onMessage(Message message) {
723.         try {
724.             String sequenceID = message.getJMSCorrelationID();
725.             int size = message.getIntProperty("SequenceSize");
726.             String type = message.getStringProperty("Type");
727.
728.             if(type != null && type.equals("Solar System Fragment")) {
729.                 Set<TextMessage> messageSet = null;
730.
731.                 if (messageSets.containsKey(sequenceID)) {
732.                     messageSet = messageSets.get(sequenceID);
733.                     messageSet.add((TextMessage)message);
734.                 } else {
735.                     messageSet = new HashSet<>();
736.                     messageSet.add((TextMessage)message);
737.                     messageSets.put(sequenceID, messageSet);
738.                 }
739.             }
740.         }
741.     }
742. }
```

```
737.  
738.          if (messageSet.size() == size  
739.                      &&  
740.                         messageSets.containsKey(sequenceID)) {  
741.                             TextMessage newMessage = reassemble(messageSet);  
742.                             messageSets.remove(sequenceID);  
743.                             newMessage.setStringProperty("Type", "SolarSystem");  
744.                             producer.send(newMessage);  
745.                         } else {  
746.                             producer.send(message);  
747.                         } catch (JMSEException e) {...}  
748.         }  
749.  
750.     public TextMessage reassemble(Set<TextMessage> messages) throws  
JMSEException {  
751.         String[] parts = new String[messages.size()];  
752.         for(TextMessage message : messages) {  
753.             String fragment = message.getText();  
754.             int index = message.getIntProperty("SequencePosition") - 1;  
755.             parts[index] = fragment;  
756.         }  
757.         String newPayload = "<joined>" + String.join("\n", parts) +  
"</joined>";  
758.         return session.createTextMessage(newPayload);  
759.     }  
760.  
761.     public static void main(String[] args) {  
762.         Connection con = null;  
763.         try {  
764.             ...  
765.  
766.             Destination from = (Destination) ctx.lookup("b-channel");  
767.             Destination to = (Destination) ctx.lookup("c-channel");  
768.  
769.             new SolarSystemMessageAggregator(con, from, to);  
770.  
771.             System.out.println("Receiving messages for 60 seconds...");  
772.             Thread.sleep(60000);  
773.             System.out.println("Done.");  
774.             con.close();  
775.  
776.         } catch (Exception e) {...} finally {...}  
777.     }  
778. }
```

O programa acima processará as mensagens que foram depositadas no “b-channel” pelo Divisor mostrado na seção anterior e depositará uma única mensagem resultante no “c-channel”. O resultado pode ser visto na console do ActiveMQ:

Queues						
Name ↑	Number Of Pending Messages	Number Of Consumers	Messages Enqueued	Messages Dequeued	Views	Operations
a-channel	0	0	1	1	Browse Active Consumers Active Producers <a href="#">atom</a> <a href="#">rss</a>	<a href="#">Send To</a> <a href="#">Purge</a> <a href="#">Delete</a>
b-channel	0	1	11	11	Browse Active Consumers Active Producers <a href="#">atom</a> <a href="#">rss</a>	<a href="#">Send To</a> <a href="#">Purge</a> <a href="#">Delete</a>
c-channel	1	0	1	0	Browse Active Consumers Active Producers <a href="#">atom</a> <a href="#">rss</a>	<a href="#">Send To</a> <a href="#">Purge</a> <a href="#">Delete</a>

As mensagens enviadas por um Divisor devem ter um Prazo de Validade (Message Expiration) para que sejam enviadas para o DLQ caso não sejam consumidos em um determinado prazo. Isso pode acontecer se todas as mensagens não forem recebidas pelo Agregador, impossibilitando a reconstrução da mensagem original.

Um Agregador pode ter uma estratégia de agregação mais complexa, pode ignorar partes que não deseja agrregar e até fazer transformações, embora o ideal seja delegar transformações e estratégias de eliminação de partes da mensagem a componentes intermediários como Filtros de Mensagens e Tradutores, garantindo maior reuso e desacoplamento. Se um Filtro elimina alguma mensagem pelo caminho, o Agregador vai sentir falta da mensagem. Nesse caso será necessário que o Agregador tenha outro meio de saber quando a sequência terminou (ex: receber o total em uma mensagem separada através de um canal de controle).

## Agregador em Apache Camel

Um agregador é configurado em Camel usando o nó aggregate(), que vem seguido de uma expressão de correlação que seleciona o grupo de mensagens que devem ser agregadas. A estratégia de agregação default seleciona a mensagem mais recente do grupo.

O exemplo abaixo usa uma estratégia (default) que seleciona a última mensagem dentro do intervalo de 5 segundos de acordo com a expressão XPATH /queote/@symbol:

```
from("jms:topic:quotes")
    .aggregate()
        .xpath("/quote/symbol")
```

```
.completionInterval(5000)
.to("direct:quotes");
```

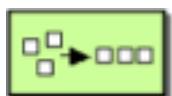
Uma implementação de AggregationStrategy precisa implementar aggregate() para dizer como as mensagens serão recombinaadas. Esse método chama aggregate(Exchange atual, Exchange recebido) a cada mensagem recebida para combinar a mensagem até o momento, com a parte recebida. A agregação vai continuar até que se atinja um limite de tempo ou número de mensagens recebidas.

```
public class SSXMLStrategy implements AggregationStrategy {
    public Exchange aggregate(Exchange oldm, Exchange newm) {
        String accum = oldm.getIn().getBody(String.class)
            + "\n" + newm.getIn().getBody(String.class);;
        return oldm.setBody(accum);
    }
}
```

Veja mais exemplos de aggregate() na documentação do Camel.

## (35) Re-sequenciador (Resequencer)

### Ícone



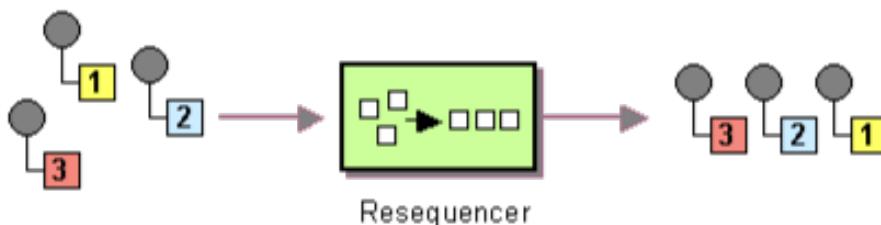
### Problema

“Como podemos fazer com que um fluxo de mensagens relacionadas que estão fora de ordem sejam colocadas de volta na ordem correta?”

### Solução

“Use um filtro estático, um Resequencer, para coletar e reordenar mensagens para que elas possam ser publicadas ao canal de saída em uma ordem especificada.”

### Diagrama



### Descrição

Um Re-sequenciador (Resequencer) é similar ao Agregador no sentido de que procura co-relacionar mensagens de entrada, mas em vez de combiná-las em uma única mensagem, envia as mesmas

mensagens no seu canal de saída em uma ordem determinada previamente. Assim como o Agregador, o Re-sequenciador também precisa manter o estado das mensagens recebidas. Ele guardará mensagens que estão fora de ordem até que uma sequencia seja obtida, e então as mensagens são publicadas na saída na ordem correta.

Para que possam ser re-ordenadas, é necessário que as mensagens tenham um número sequencial. Este número não é o mesmo usado como Message Identifier ou Correlation Identifier. É preciso gerá-lo no momento da transmissão ou criação da sequência de mensagens (isto pode ser um gargalo).

O Re-sequenciador não precisa receber todas as mensagens co-relacionadas antes de enviar mensagens para a saída, mas precisa saber quando iniciar o envio. Ele guarda a mensagem com o número sequencial mais alto até receber todos os números mais baixos, e só então ele libera para a saída essa sequencia. Enquanto isso mais números podem estar chegando e ele precisará guardar e repetir a operação.

## Aplicações

Um Re-sequenciador é necessário para re-ordenar mensagens que precisam chegar em ordem mas podem estar fora de ordem por algum motivo (ex: por terem seguido caminhos paralelos e passados por processadores mais rápidos ou mais lentos.)

### Resequenciador em Java / JMS

Uma forma de implementar o Re-sequenciador é manter um conjunto de mensagens para cada ID de sequência, o total de mensagens da sequência e a posição atual.

```
private Map<String, TreeSet<Message>> messageSetMap = new HashMap<>();
private Map<String, Integer> sizeMap = new HashMap<>();
private Map<String, Integer> positionMap = new HashMap<>();
```

Cada mensagem que chegar para cada sequência é armazenada até que chegue a mensagem da primeira posição. Ela é enviada e depois verifica-se se já foi armazenada a segunda posição, se sim, ela também enviada, a posição atual é incrementada, e a verificação prossegue. Se a próxima da sequência ainda não tiver chegado, então espera-se pela próxima mensagem.

É mais eficiente guardar um conjunto ordenado. Em Java pode-se usar o TreeSet de Mensagens. Será preciso implementar um Comparator para que o TreeMap saiba como ordenar mensagens:

```
779. public class MessageComparator implements Comparator<Message> {
780.     @Override
781.     public int compare(Message o1, Message o2) {
782.         try {
783.             int position1 =
o1.getIntProperty(MessageSequence.POSITION_HEADER);
784.             int position2 =
o2.getIntProperty(MessageSequence.POSITION_HEADER);
785.             return position1 - position2;
786.         } catch (JMSEException e) {
```

```

787.         return 0;
788.     }
789. }
790. }
```

POSITION\_HEADER e outros nomes de cabeçalhos foram mantidos em uma interface para que os nomes pudessem ser reusados e evitar bugs:

```

791. public interface MessageSequence {
792.     static String POSITION_HEADER = "SequencePosition";
793.     static String SEQUENCE_ID_HEADER = "SequenceID";
794.     static String SIZE_HEADER = "SequenceSize";
795. }
```

A seguir, o código do onMessage() do Re-sequenciador.

```

796. @Override
797. public void onMessage(Message message) {
798.     try {
799.         String sequenceID =
800.             message.getStringProperty(MessageSequence.SEQUENCE_ID_HEADER);
801.
802.         if (messageSet == null) {
803.             int sequenceSize =
804.                 message.getIntProperty(MessageSequence.SIZE_HEADER);
805.             messageSet = new TreeSet<>(new MessageComparator());
806.             sizeMap.put(sequenceID, sequenceSize);
807.             messageSetMap.put(sequenceID, messageSet);
808.             positionMap.put(sequenceID, 1); // sequences starting in 1
809.         }
810.
811.         int currentSetSize = sizeMap.get(sequenceID);
812.         int currentPosition = positionMap.get(sequenceID); // always
starts in 1
813.
814.         int seqPosition =
815.             message.getIntProperty(MessageSequence.POSITION_HEADER);
816.         if (seqPosition >= currentPosition) { // ignore duplicates!
817.             messageSet.add(message); // automatically orders
Message lowest = messageSet.first(); // get lowest message in
set
818.
819.             while (lowestPosition == currentPosition) { // never <
currentPosition
820.
821.                 producer.send(lowest);
822. }
```

```

823.             if (currentPosition == currentSetSize) { // done
824.                 messageSet.clear();
825.                 messageSetMap.remove(sequenceID);
826.                 sizeMap.remove(sequenceID);
827.                 positionMap.remove(sequenceID);
828.                 currentPosition = 0;
829.             } else { // increment next lowest message
830.                 positionMap.put(sequenceID, ++currentPosition);
831.                 messageSet.remove(lowest);
832.
833.             // Check next message in set
834.             lowest = messageSet.first();
835.             lowestPosition = lowest
836.
837.             .getIntProperty(MessageSequence.POSITION_HEADER);
838.         }
839.     }
840.
841. } catch (JMSEException e) {
842.     e.printStackTrace();
843. }
844. }
```

Ao receber a mensagem, o Re-sequenciador guarda o ID da sequência e tenta acha-lo no messageSetMap. Se não encontra cria um novo TreeSet com o Comparator de mensagens e guarda no mapa. Guarda também o tamanho da sequencia e a posição atual (inicia em 1) da sequência.

Se a posição da mensagem recebida for menor que a posição da sequência atual, a mensagem já foi processada antes e será ignorada. Caso contrário, será adicionada ao conjunto. Em seguida, a primeira mensagem é lida do conjunto (que será a que tem menor posição, já que o conjunto é ordenado). Se a posição atual for igual à posição dessa mensagem, ela pode ser enviada, pois não há nenhuma outra que deva ser enviada antes dela. Se a posição for igual ao tamanho da sequência, então o processamento da sequência terminou, as filas serão esvaziadas e os valores reiniciados. Caso contrário, a mensagem enviada é removida da fila, a posição é incrementada e a próxima mensagem do conjunto é analisado. Se estiver na vez (sua posição corresponde à nova posição atual), ela será enviada e o processo continua, caso contrário, o programa irá esperar pela próxima mensagem.

Para testar o Re-sequenciador precisamos gerar algumas mensagens em uma sequência fora-de-ordem. O método sendSequence() da classe abaixo recebe uma lista de posições em ordem arbitrária e atribui à propriedade POSITION\_HEADER de um conjunto de mensagens geradas. Depois envia as mensagens para a fila.

```

845. public class MessageSequenceGenerator {
846.
847.     private Session session;
848.     private MessageProducer producer;
849.
```

```
850.     public MessageSequenceGenerator(Connection con, Destination
destination)
                                         throws JMSEException {
851.         session = con.createSession(false, Session.AUTO_ACKNOWLEDGE);
852.         producer = session.createProducer(destination);
853.         con.start();
854.     }
855.
856.     public void sendSequence(int[] positions, String sequenceID)
                                         throws JMSEException {
857.         System.out.println("\nSequence: " + sequenceID);
858.         for(int i = 0; i < positions.length; i++) {
859.             Message message = session.createMessage();
860.             message.setStringProperty(MessageSequence.SEQUENCE_ID_HEADER,
sequenceID);
861.             message.setIntProperty(MessageSequence.SIZE_HEADER,
positions.length);
862.             message.setIntProperty(MessageSequence.POSITION_HEADER,
positions[i]);
863.
864.             System.out.println("    " +
message.getIntProperty(MessageSequence.POSITION_HEADER));
865.
866.             producer.send(message);
867.         }
868.     }
869.
870.     public static void main(String[] args) {
871.         Connection con = null;
872.         try {
873.             Context ctx = new InitialContext();
874.             ConnectionFactory factory = ...;
875.             con = factory.createConnection();
876.
877.             Destination destination = (Destination) ctx.lookup("b-
channel");
878.
879.             MessageSequenceGenerator generator =
new MessageSequenceGenerator(con, destination);
880.             generator.sendSequence(new int[]{8,3,5,1,6,7,2,4}, "Seq A");
881.             generator.sendSequence(new int[]{2,3,1,5,4}, "Seq B");
882.             generator.sendSequence(new int[]{1,2,3}, "Seq C");
883.             generator.sendSequence(new int[]{4,3,2,1}, "Seq D");
884.
885.             System.out.println("All sequences sent.");
886.             con.close();
887.
888.         } catch (Exception e) {...} finally {...}
```

```
889.      }
890. }
```

Depois de rodar o gerador de mensagens, o ActiveMQ mostra que há 20 mensagens na fila “b-queue”. Elas estão em quatro sequências e todas estão desordenadas.

Queues						
Name	Number Of Pending Messages	Number Of Consumers	Messages Enqueued	Messages Dequeued	Views	Operations
b-channel	20	0	20	0	<a href="#">Browse Active Consumers</a> <a href="#">Active Producers</a>  	<a href="#">Send To</a> <a href="#">Purge</a> <a href="#">Delete</a>

Depois de rodar o Re-Sequenciador, elas foram transferidas para a fila “c-queue” devidamente ordenadas.

Queues						
Name	Number Of Pending Messages	Number Of Consumers	Messages Enqueued	Messages Dequeued	Views	Operations
b-channel	0	1	20	20	<a href="#">Browse Active Consumers</a> <a href="#">Active Producers</a>  	<a href="#">Send To</a> <a href="#">Purge</a> <a href="#">Delete</a>
c-channel	20	0	20	0	<a href="#">Browse Active Consumers</a> <a href="#">Active Producers</a>  	<a href="#">Send To</a> <a href="#">Purge</a> <a href="#">Delete</a>

Rode os programas MessageSequenceGenerator e MessageResequencer para ver os resultados.

## (36) Processador de Mensagens Compostas (Composed Message Processor) (CMP)

### Ícone



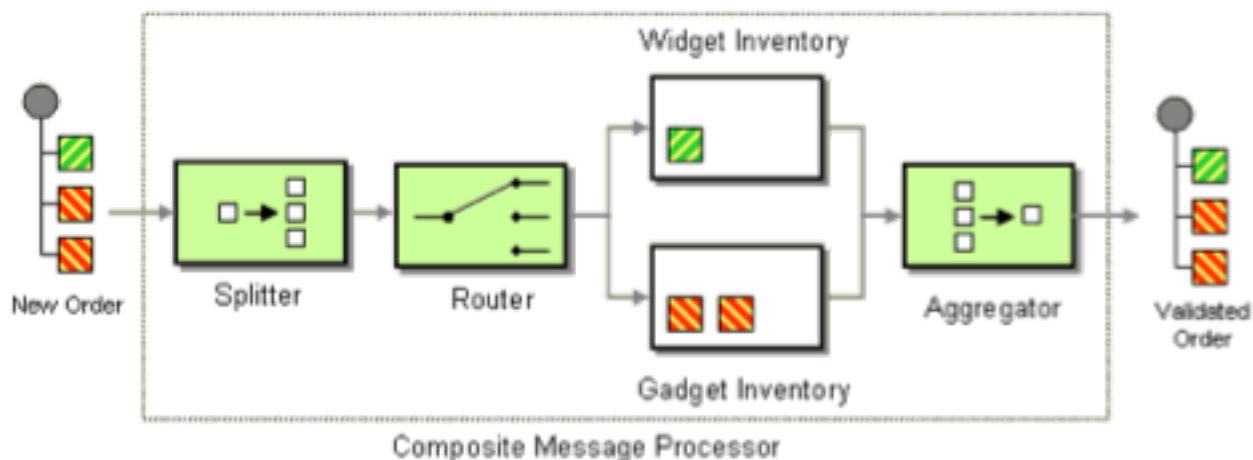
### Problema

“Como manter o fluxo de mensagem enquanto se processa uma mensagem que consiste de múltiplos elementos, cada um requerendo processamento diferenciado.”

## Solução

“Use um Processador de Mensagens Compostas (Composed Message Processor) (CMP) para processar uma mensagem composta. O Processador de Mensagens Compostas divide a mensagem, redireciona as sub-mensagens para os destinos apropriados e reagrega as respostas de volta em uma única mensagem.”

## Diagrama



## Descrição

O CMP é formado por um conjunto de componentes usados para processar em mensagens complexas, estruturadas, e que podem ser divididas em mensagens menores para processamento separado.

Um Divisor (Splitter) divide a mensagem recebida em sub-mensagens. Um Roteador Baseado em Conteúdo (Content-Based Router) redireciona cada sub-mensagem a um canal diferente, de acordo com o seu tipo. Cada canal está ligado a um processador especializado no tipo da sub-mensagem. À medida em que são processadas, as sub-mensagens são enviadas a um canal usado para alimentar a entrada de um Agregador, que reconstrói a mensagem enviada para a saída.

## Aplicações

Um CMP pode ser usado quando for necessário realizar algum tipo de processamento em parte de uma mensagem complexa e estruturada que tenha partes que podem ser processadas separadamente.

Por exemplo, para validar uma mensagem que representa um pedido de compra e verificar se os itens do pedido existem em estoque, é necessário dividir a mensagem em partes, encapsulando cada item em uma nova mensagem enviada para o validador. Os itens validados são enviados para um agregador que irá reconstruir o pedido contendo apenas os itens válidos, recalculando os sub-totais e total do pedido se necessário.

## CMP em Java / JMS, Camel e Spring Integration

Uma implementação usando este padrão será apresentada no estudo de caso (Capítulo 10).

## (37) Espalha-Recolhe (Scatter-Gather)

### Ícone



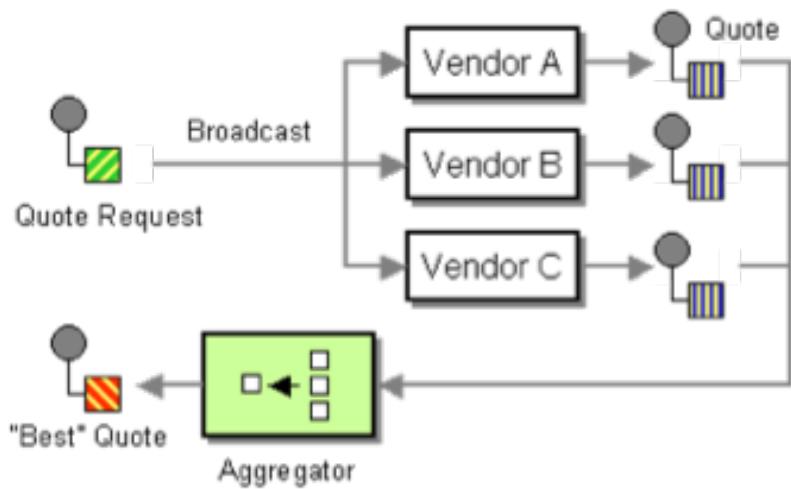
### Problema

“Como manter o fluxo de mensagens quando uma mensagem precisa ser enviada para múltiplos receptores, e cada um deles poderá enviar uma resposta?”

### Solução

“Use um Espalha-Recolhe (Scatter-Gather) que difunde uma mensagem para múltiplos receptores e re-agregá-la em uma única mensagem”

### Diagrama



### Descrição

O Espalha-Recolhe (Scatter-Gather) combina um módulo de difusão (Canal Publica-Inscreve ou Lista de Receptores) com um Agregador. Uma mensagem é difundida ou enviada para diversos destinatários que a processam e produzem respostas redirecionadas a um Agregador (as respostas devem ter como Endereço de Retorno o canal de entrada do Agregador). Depois que o Agregador recebe todas as mensagens, envia para a saída uma agregação das informações recebidas.

A decisão entre usar um Canal Publica-Inscreve ou uma Lista de Receptores está na forma de controlar os receptores. Um Canal Publica-Inscreve envia mensagens a qualquer assinante da lista, enquanto que uma Lista de Receptores tem controle sobre os receptores.

Uso de um ou outro representa diferentes mecanismos de Espalha-Recolhe: distribuição com controle da lista de receptores, ou anúncio para todos os interessados.

## Aplicações

Um Espalha-Recolhe é usado sempre que for necessário extrair informações de um conjunto de mensagens, obtidas como resposta de uma ou mais requisições; quando for necessário enviar uma requisição a vários componentes e escolher apenas uma dentre as respostas, ou uma combinação de parte ou todas as respostas.

Por exemplo, para descobrir dentre uma grande lista de servidores quais deles estão fora do ar, pode-se enviar uma mensagem de difusão para um canal observado por componentes que monitoram esses sites. Cada componente irá retornar uma resposta indicando o seu status. O agregador pode incluir na mensagem de saída uma lista dos servidores disponíveis. Outro cenário seria descobrir qual servidor tem o menor tráfego. Neste caso o Agregador poderia simplesmente repassar a mensagem retornada pelo servidor de menor tráfego, após comparar com as outras mensagens recebidas.

## Exemplos usando Apache Camel e Spring Integration

Exemplos de implementações usando este padrão serão apresentados no Capítulo 10.

### (38) Lista de circulação (Routing Slip)

#### Ícone



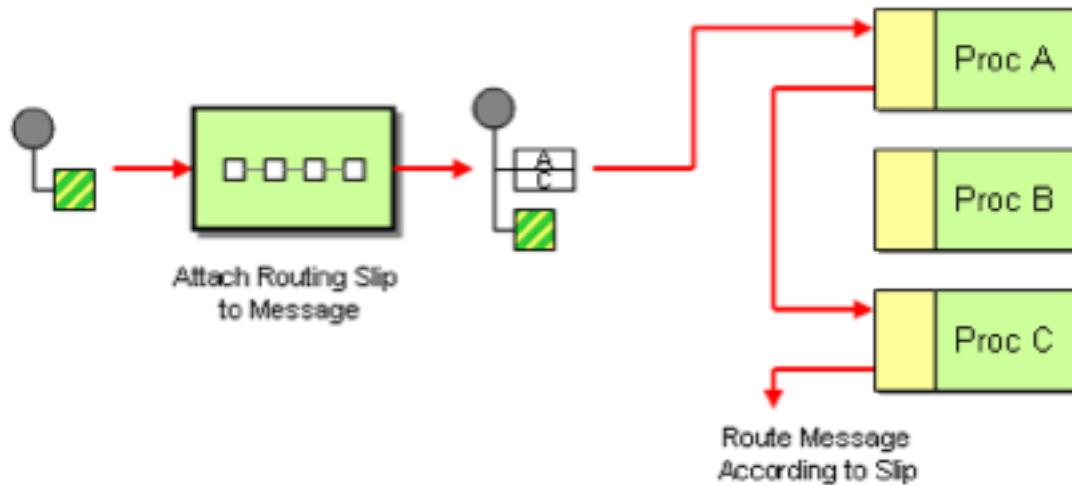
#### Problema

“Como rotear uma mensagem consecutivamente através de uma série de etapas de processamento, quando a sequencia de passos não é conhecida em tempo de design, e pode variar para cada mensagem?”

#### Solução

“Anexe uma Lista de Circulação (Routing Slip) a cada mensagem, especificando a sequencia de passos de processamento. Empacote cada componente com um roteador que leia a Lista de Circulação e redirecione a mensagem para o próximo componente dessa lista”

## Diagrama



## Descrição

Uma Lista de Circulação é anexada a cada mensagem antes do envio a um conjunto de componentes que serão executados em série. Apenas os componentes que estão na Lista de Circulação serão executados. Cada componente deve verificar, ao receber a mensagem, se faz parte da Lista de Circulação. Se estiver, processa a mensagem, se não, repassa para o próximo componente (similar ao padrão de design GoF Chain of Responsibility).

O processo inteiro pode falhar caso uma mensagem se perca pelo caminho, pois não será mais possível saber o que vem depois.

A responsabilidade de rotear para o próximo componente da Lista de Circulação pode ser transferida para um controlador central. Nessa configuração a solução se parece mais com um Gerente de Processos (Process Manager), mas com uma lista sequencial e estática. Neste caso o Endereço de Retorno de cada mensagem é o endereço do controlador central, que precisa manter o controle sobre as etapas já realizadas e enviar a mensagem para o próximo processador da lista.

## Aplicações

Uma Lista de Circulação é usada quando uma mensagem deve ser enviada para vários processadores, em sequência. Esses processadores podem, cada um, realizar uma etapa de processamento na mensagem, alterando-a cumulativamente (padrão Decorador) ou repassar ao próximo da lista até encontrar um que possa processar a mensagem (padrão Chain of Responsibility).

### Lista de Circulação usando Apache Camel

Camel suporta o padrão Lista de Circulação através do método `routingSlip()` que recebe o nome de um cabeçalho que deve ter uma lista de endpoints separados por vírgula. Usando a lista, o Camel irá enviar a mensagem para cada um dos endereços, em sequência:

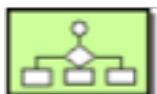
```
from("activemq:SomeQueue").routingSlip("aRoutingSlipHeader");
```

A partir de Camel 2.4 é possível passar uma expressão que deve retornar a lista de endereços. A versão abaixo tem o mesmo efeito que a versão anterior, mas está usando a expressão header:

```
from("activemq:SomeQueue").routingSlip(header("aRoutingSlipHeader"));
```

## (39) Gerente de Processos (Process Manager)

### Ícone



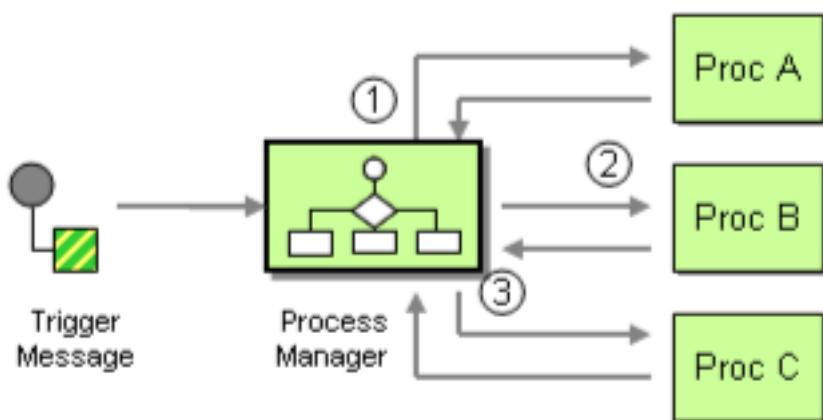
### Problema

“Como rotear uma mensagem através de múltiplos passos de processamento quando os passos necessários podem não ser conhecidos em tempo de projeto e não serem sequenciais?”

### Solução

“Use uma unidade central de processamento, um Gerente de Processos (Process Manager), para manter o estado da sequencia e determinar o próximo passo de processamento com base em resultados intermediários.”

### Diagrama



### Descrição

O Gerente de Processos é mais flexível que a Lista de Circulação porque pode executar qualquer sequência de passos, sequencial ou paralelo. Diferentemente da Lista de Circulação em série, esta solução usa uma arquitetura de hub central (CPU) onde está concentrada a responsabilidade de repassar a mensagem para o próximo componente. Os componentes processadores não se preocupam com roteamento (como acontece em Lista de Circulação). O Gerente de Processos envia a primeira mensagem, que envia uma mensagem de retorno à CPU, que envia mensagem para o segundo processador, e assim por diante. A lista pode ser estática, mas fica sob controle da CPU e pode ser alterada dinamicamente. A CPU precisa manter o estado para saber qual o passo atual da sequência.

Tipicamente um Gerente de Processos precisa controlar a execução de uma sequência de processos e também manter estado entre mensagens. Isso envolve a manutenção do estado de várias informações, como a mensagem que está sendo processada, IDs de correlação de outras mensagens envolvidas no processo, o passo atual que está sendo executado, etc. Geralmente ele terá que lidar com vários processos que serão executados ao mesmo tempo, portanto essas informações todas devem ser armazenadas em instâncias de processos separados rodando em threads paralelos.

## Aplicações

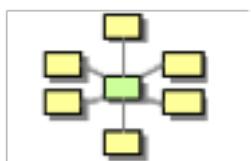
Quando a sequência de etapas de processamento de uma mensagem não forem sequenciais, ou se os componentes processadores não puderem realizar roteamento, ou se as rotas puderem ser alteradas dinamicamente, pode-se implementar uma solução baseada no padrão Gerente de Processos (Process Manager), que consiste de um controlador central (CPU) que processa regras dinâmicas onde é possível controlar todo o roteamento e decidir qual o próximo destino de uma mensagem.

## Exemplos usando Java/JMS, Camel e Spring Integration

Exemplos de implementação deste padrão serão explorados no Capítulo 10.

## (40) Corretor de mensagens (Message Broker)

### Ícone



### Problema

“Como desacoplar o destino de uma mensagem do seu remetente e manter um controle central sobre o fluxo das mensagens?”

### Solução

“Usar um Corretor de Mensagens (Message Broker) central que possa receber mensagens de múltiplos destinos, determinar o destino correto e rotear a mensagem para o canal correto. Implemente a infraestrutura do Corretor de Mensagens com os padrões de design de roteamento”

### Descrição

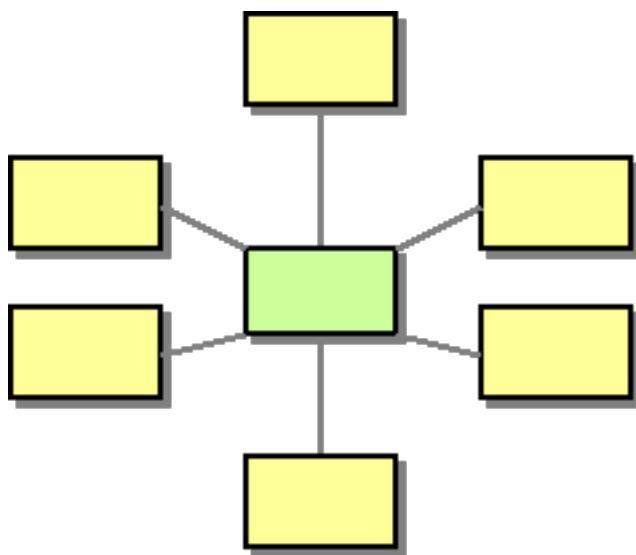
O Corretor de Mensagens (Message Broker) é um padrão de arquitetura que tem uma topologia baseada em Hub & Spikes (Cubo e Raios, como uma roda de bicicleta). Pode-se construir uma arquitetura assim através da orquestração de padrões de roteamento. É comparável ao padrão de arquitetura Dutos e Filtros pois descreve como conectar componentes, mas vai além sugerindo uma topologia específica que representa o padrão Mediador [GoF].

Esta é a arquitetura comum de provedores de mensageria (MOMs), onde os componentes são Terminais (Endpoints) que se conectam ao Cubo central, através dos Raios. Cada Raio corresponde a um ou mais Canais. Por ser um mediador, o Corretor de Mensagens permite a troca de mensagens entre clientes que não se conhecem, através de um canal comum de comunicação. Ao abstrair o controlador central, podemos representar soluções implementadas com MOMs usando a arquitetura Dutos e Filtros, como a maior parte dos padrões de integração.

Normalmente um Corretor de Mensagens usa um Modelo de Dados Canônico, de forma que os dados que circulam pelos canais devem ser transformados pelos seus Endpoints.

O principal gargalo dessa solução é o fato de depender de um componente centralizado. Uma solução típica pode ser construída com vários Message Brokers hierarquicamente organizados em torno de um Message Broker central. Novos processadores podem se registrar automaticamente, usando o Corretor como um mediador.

## Diagrama



## Aplicações

Quando for necessário uma arquitetura capaz de intermediar cada etapa da comunicação, com uma interface genérica e desacoplada dos processadores (que poderão se cadastrar ou se desligar do sistema em tempo real).

Se a comunicação com o controlador central for totalmente transparente e irrelevante para a descrição da solução, o fluxo pode ignorar o controlador central e ser descrito usando uma arquitetura Dutos e Filtros.

## Corretor de Mensagens em Java / JMS

JMS é uma implementação de Corretor de Mensagens. O Cubo é o provedor que fornece a infraestrutura de roteamento e os Raios são Queues e Topics. Cabe ao programador usar as interfaces

de MessageProducer e MessageConsumer para construir Endpoints, e enviar mensagens de acordo com o Modelo de Dados Canônico definido na interface Message.

Qualquer provedor de mensageria usado ou não através de JMS também implementa Corretor de Mensagens. É comum chamar o IBM MQ, ActiveMQ, RabbitMQ, etc. de Message Brokers.

A arquitetura de Corretor de Mensagens pode ser aplicada a qualquer modelo de roteamento. Por exemplo, pode-se implementar um Corretor de Mensagens simples baseado em uma Lista de Receptores criando um componente central que controla um Topic e um conjunto de Queues. Cada Produtor envia mensagens para o Topic contendo o endereço de um ou mais Queues. O componente central então redireciona a mensagem para os Queues, que são lidos por Consumidores.

## Corretor de Mensagens em Camel e Spring Integration

Ferramentas como Camel e Spring Integration simplificam a construção dos componentes usados em uma arquitetura de Corretor de Mensagens. Eles fornecem adaptadores, transformadores e endpoints prontos que já estão preparados para a conversão e transferência de dados. Tanto Camel como Spring Integration vêm com uma grande coleção de Adaptadores de Canal (Channel Adapter) e Tradutores de Mensagens (Message Translator), para os mais diversos meios (sistema de arquivos, banco de dados, JMS, JPA, FTP, Twitter, etc.) e formatos (bytes, texto, XML, JSON, etc.), além de linguagens de script para construir predicados e expressões (EL, SpEL, JavaScript, XPath, etc.) para transformação e roteamento. Ambas também possuem ferramentas visuais que permitem construir soluções graficamente, alterando a topologia e monitorando o fluxo de mensagens.

## Revisão

Padrões de integração de sistemas relacionados a roteadores de mensagens:

### Roteadores simples

*Roteadores de mensagens* (veja Capítulo 3): usam uma regra estática para redirecionar uma mensagem recebida para um dentre vários canais possíveis.

*Roteador baseado em conteúdo*: é um roteador que decide para onde redirecionar a mensagem com base em informações fornecidas pela própria mensagem.

*Filtro de mensagens*: filtra as mensagens recebidas em um canal de entrada, analisando o seu conteúdo e descartando as que não correspondem a uma regra estabelecida; apenas as mensagens que passarem no filtro serão enviadas para o canal de saída.

*Roteador dinâmico*: recebe as regras de roteamento em tempo de execução, através um canal de controle alimentado por componentes em outra parte do sistema.

*Lista de receptores*: recebe uma mensagem e re-envia para um ou mais canais, de acordo com uma lista de receptores que pode ser estática ou calculada para cada mensagem com base em informações contidas na própria mensagem.

*Divisor*: recebe uma mensagem no canal de entrada, divide-a em mensagens menores que são enviadas para seu canal de saída.

*Agregador*: acumula mensagens relacionadas que são recebidas no seu canal de entrada; quando receber todas, combina em uma única mensagem enviada ao canal de saída.

*Re-sequenciador*: recebe mensagens em qualquer ordem no seu canal de entrada, e devolve ao canal de saída da ordem correta.

## Roteadores compostos

*Processador de mensagens compostas*: combinação de Divisores, Agregadores e possivelmente Resequenciadores para desmontar mensagens compostas, processar seus componentes individualmente e depois reconstruir cada mensagem antes de enviar para o canal de saída.

*Lista de circulação*: permite anexar na mensagem uma rota que deverá ser seguida.

*Espalha-recolhe*: envia as mensagens para canais de difusão (Publicar-Inscrever) e recolhe as respostas, possivelmente descartando ou combinando algumas em uma única mensagem.

*Gerente de processos*: é uma unidade central de processamento que controla cada etapa do fluxo de mensagens e determina dinamicamente qual será o próximo passo com base em resultados intermediários. O fluxo de processamento pode ser especificado através de uma linguagem que será interpretada pelo Gerente de processos.

## Arquitetura

*Corretor de mensagens*: é uma arquitetura com controlador central Hub & Spike (Cubo e Raio) frequentemente usada para implementar Roteadores Dinâmicos (Dynamic Router), Gerentes de Processos (Process Manager), Barramentos de Controle (Control Bus) e outros padrões que precisam controlar cada etapa do processamento. É também a arquitetura do JMS e dos provedores de mensageria (ActiveMQ, IBM MQ, etc.)

# Capítulo 7

# Transformação

A transformação de dados é essencial na integração de sistemas, já que é raro que dois sistemas que precisem de integração compartilhem o mesmo formato de dados. Por exemplo, um ponto de vendas guarda dados em um banco de dados e exporta em XML e precisa se comunicar com um sistema que usa CSV. Outro sistema usa também XML mas tem um formato com estrutura incompatível. Pode ainda ser necessário converter encodings, comprimir e descomprimir, cifrar e decifrar, dentre outras transformações.

Enquanto canais e roteadores podem eliminar várias dependências entre aplicações, a incompatibilidade de formatos de dados ainda é um gargalo. A solução de integração precisa lidar com as diferenças entre formatos de dados das diferentes aplicações. O padrão Tradutor de Mensagens (Message Translator) descreve um componente que tem a finalidade de eliminar mais esta dependência.

A transformação é uma tarefa que não está limitada às soluções de mensageria. Os padrões podem ser adaptados a soluções de transferência de arquivos, RPC e banco de dados, já que as entradas e saídas não estão dependentes de canais ou estrutura de mensagens.

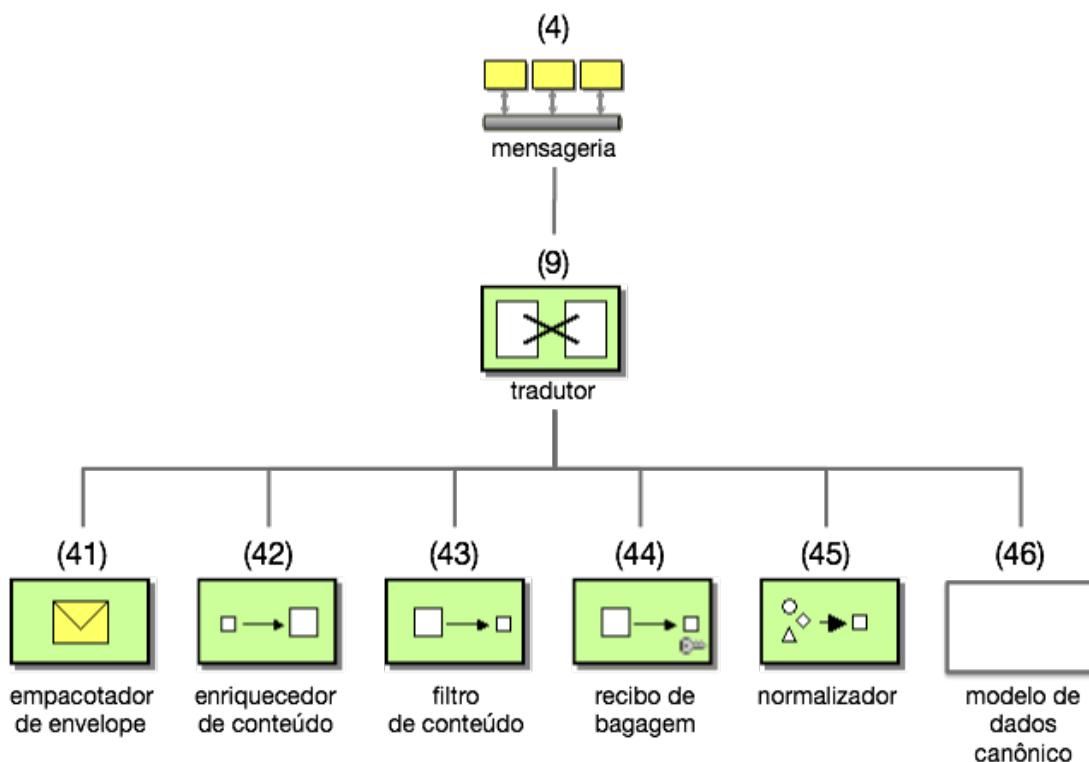
Para que uma solução de transformação seja reutilizável, ela precisa trabalhar também com os metadados que descrevem os tipos transformados. O algoritmo de transformação baseia-se nos metadados para transformar os dados encapsulados em cada mensagem. Exemplos de metadados são um esquema de banco de dados, um XSD, uma classe, um protótipo, um modelo estrutural, uma matriz, uma expressão regular. Por exemplo, um componente que transforma um XML em um objeto Java pode mapear um XML Schema em uma classe Java, no nível de metadados, e durante o processamento de mensagens, usar o mapa para transferir dados de um documento XML para preencher atributos de um objeto Java.

Existem várias maneiras de implementar e usar um Tradutor de Mensagens, com impactos de performance, acoplamento, complexidade, nível de reuso, etc. O catálogo EIP descreve outros seis padrões, relacionados a *Message Translator*, que mostram como solucionar determinados problemas de integração usando componentes de transformação.

Um *Envelope (Envelope Wrapper)* permite empacotar uma mensagem em uma estrutura que seja compatível com o sistema de mensageria usado. Mensagens podem ser empacotadas em um formato compatível com os canais usados, e depois, desempacotadas quando forem consumidas.

Um *Normalizador (Normalizer)* pode ser usado para traduzir mensagens que têm o mesmo significado mas têm formatos diferentes a um formato comum. Um *Modelo de Dados Canônico (Canonical Data Model)* descreve a criação de um modelo de dados neutro, independente de qualquer sistema, para o qual podem ser traduzidas as mensagens dos sistemas integrados.

Um *Enriquecedor de Conteúdo (Content Enricher)* é um transformador que acrescenta dados adicionais a uma mensagem. Um *Filtro de Conteúdo (Content Filter)* faz o oposto, removendo dados indesejados ou desnecessários de uma mensagem. Se os dados removidos da mensagem precisarem ser recuperados em outro ponto da rota de integração, ele pode ser despachado a um repositório em troca de um *Recibo de Bagagem (Claim Check)* incluído na mensagem. Com o Recibo, os dados poderão ser recuperados posteriormente.



## (41) Envelope (Envelope Wrapper)

### Ícone



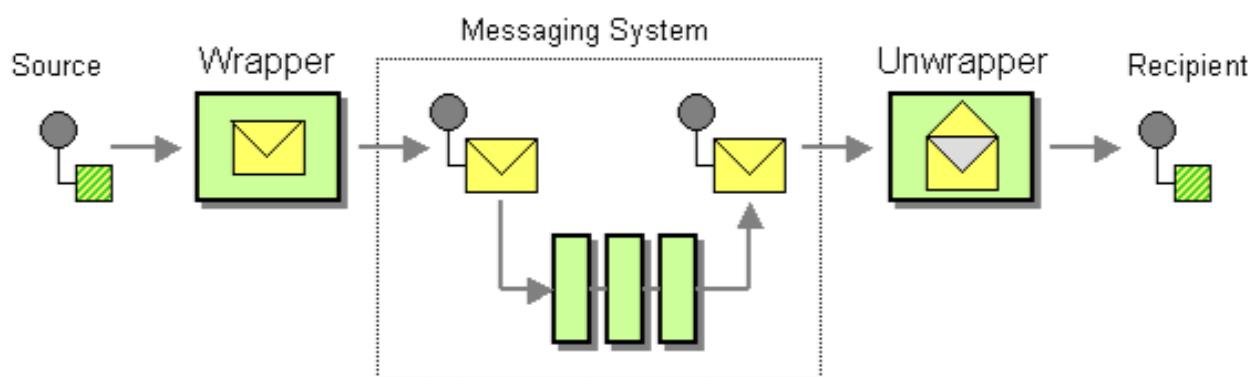
## Problema

“Como podem sistemas existentes participarem em uma troca de mensageria que impõe requerimentos específicos no formato das mensagens, como determinados campos de cabeçalho e encriptação”

## Solução

“Use um Envelope (Envelope Wrapper) para embrulhar os dados da aplicação dentro de um envelope que seja compatível com a infraestrutura de mensageria usada. Desembrulhe a mensagem quando ela chegar no seu destino”

## Diagrama



## Descrição

Pode-se usar um Envelope para transmitir uma mensagem de e-mail em um sistema JMS. A mensagem de e-mail, assim como a mensagem JMS, possui cabeçalho e corpo, mas é diferente e tem formato incompatível com a mensagem JMS. Não pode ser simplesmente enviada para um canal. A solução é criar uma mensagem JMS, incluir a mensagem de e-mail no corpo, e talvez copiar alguns cabeçalhos do e-mail para o cabeçalho da mensagem JMS, para que possa ser usado em roteamento. A mensagem JMS neste caso está funcionando como um envelope.

Envelopes podem ser usados em cascata em uma arquitetura similar ao padrão Decorator (GoF). Esse tipo de solução é comum em Pontes de Mensageria, que integram sistemas de mensageria incompatíveis entre si, permitindo que mensagens de um sistema sejam transmitidas nos canais do outro sistema (ex: Ponte Camel - Spring Integration).

O processo de empacotar uma mensagem no envelope, e depois desempacotar, pode ser facilitado utilizando meta-dados para construir uma API que poderia ser usada para empacotar a mensagem criada pela aplicação, definir e copiar cabeçalhos, validar, criptografar, comprimir, etc. Depois que o sistema consumir a mensagem, ele provavelmente enviará para um processador que precisará desempacotar a mensagem, descomprimir, decriptografar e extraí-la da mensagem encapsulada.

Um exemplo no mundo real é o sistema de correios. Uma encomenda precisa ser colocada em uma caixa com um formato padronizado (um Modelo de Dados Canônico) que inclui determinadas

dimensões, e determinados campos de metadados (endereço, CEP, país, etc.) além de selos e controles usados pelo serviço. Em alguns casos, uma encomenda pode ganhar um novo envelope ao atravessar a fronteira de algum país. O envelope pode ser apenas um código extra ou é possível que os conteúdos seja extraído do pacote, analisado e depois re-empacotado.

Mensagens SOAP em Web Services possuem um envelope dentro do qual podem vir comandos, dados, anexos, mensagens de erro, etc, de forma que a mensagem é transmitida sem que o sistema de mensageria usado precise saber dos detalhes no seu interior.

## Aplicações

Um Envelope é usado quando o formato de mensagem que precisa ser transmitido é incompatível com o formato de mensagem usado pelo sistema de integração. Para que possa ser transmitida, a mensagem precisa ser empacotada dentro de uma outra mensagem compatível com o sistema, que funciona como um Envelope. Pode ser necessário copiar cabeçalhos necessários da mensagem empacotada para o cabeçalho do Envelope.

### Envelope em Java / JMS, Camel e Spring Integration

Para transmitir um arquivo pelos canais de mensageria via JMS um Adaptador de Canal precisa empacotar o arquivo em um Envelope, que é a Mensagem (`javax.jms.Message`). O ActiveMQ possui seu próprio formato de mensagem. Se o ActiveMQ é configurado como provedor JMS, é responsabilidade do `MessageProducer` empacotar a mensagem ActiveMQ dentro da Mensagem JMS, e prover mapeamentos para cabeçalhos e conteúdo. Do outro lado, a implementação ActiveMQ do `MessageConsumer` irá extrair a mensagem para que possa guardá-la no canal.

O Apache Camel empacota mensagens JMS dentro de Mensagens Camel (`org.apache.camel.Message`) e Mensagens Camel são empacotadas dentro de Exchanges (`org.apache.camel.Exchange`). O Envelope é construído pelo componente de endpoint e tem várias camadas. O componente `JmsMessage` encapsula a mensagem JMS e através de um mapeamento (`JmsBinding`) é possível usar a Message do Camel para ter acesso ao conteúdo e cabeçalhos da mensagem JMS sem precisar desempacotá-la. O empacotamento e desempacotamento são realizados pelos componentes e endpoints JMS do Camel. Da mesma maneira um `FileMessage` empacota um arquivo que pode ser empacotado. Portanto `Message` e `Exchange` são Envelopes padrão usados em Camel para qualquer tipo de conteúdo fornecido pelos componentes/endpoints. Usando padrão Envelope, Camel pode transmitir quaisquer tipos de dados pelos seus canais.

Spring Integration também possui um formato próprio para mensagens, e empacota mensagens JMS e de outros sistemas de mensageria. A responsabilidade para empacotar e desempacotar é dos Adaptadores de Canal.

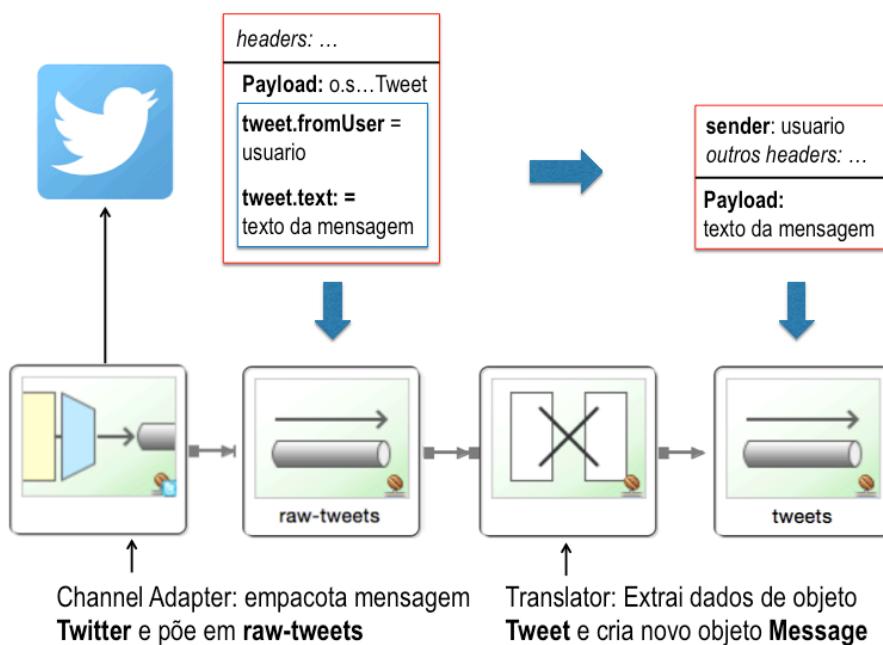
A classe `TweetTransformer` abaixo é um Tradutor de Mensagens (Message Translator). Ela recebe uma `org.springframework.messaging.Message`. Dentro dessa mensagem há outra mensagem, que é uma `o.s.social.twitter.api.Tweet`. O Tweet é a forma como o pacote Spring Social empacota mensagens do Twitter, que tem um formato próprio. Ou seja, há um Envelope (Tweet) encapsulando a mensagem do Twitter, e outro Envelope (Message) encapsulando a mensagem Spring, para que ela possa circular

pelos canais do Spring Integration. A classe abaixo é um Filtro de Conteúdo (Content Filter) que abre os dois Envelopes, extraíndo dados selecionados e empacotando-os diretamente no Envelope externo.

```

01. import org.springframework.integration.support.MessageBuilder;
02. import org.springframework.integration.transformer.Transformer;
03. import org.springframework.messaging.Message;
04. import org.springframework.social.twitter.api.Tweet;
05.
06. public class TweetTransformer implements Transformer {
07.
08.     @Override
09.     public Message<?> transform(Message<?> tweetMessage) {
10.         Tweet tweet = (Tweet)tweetMessage.getPayload();
11.         String sender = tweet.getFromUser();
12.         String text   = tweet.getText();
13.
14.         Message<String> message = MessageBuilder.withPayload(text)
15.                         .setHeader("sender", sender)
16.                         .build();
17.
18.         return message;
19.     }
19. }
```

Para poder extraír o texto e o remetente da mensagem Twitter, é preciso obter o payload da Mensagem Spring, que é outro envelope (Tweet). Dele foi extraída uma propriedade de cabeçalho “getFromUser()”, copiada para o header “sender” da Mensagem Spring, e o corpo “getText()”, substituindo o payload da mensagem. o conteúdo. O diagrama abaixo ilustra como o envelope do Twitter foi extraído.



Dentro do domínio de uma aplicação, o Envelope geralmente é usado para empacotar o Payload, já que a Mensagem é o transporte. Similar ao exemplo acima seria uma mensagem que contém uma imagem como payload, e essa imagem passará por canais que esperam um XML. A imagem pode ser convertida em um formato (ex: Byte64) e encapsulada no XML, enquanto que meta-informação (formato de dados, tamanho, nome, etc.) sejam copiados para o cabeçalho. Depois que não precisar mais do empacotamento XML, a imagem seria desempacotada, convertida de volta e re-empacotada em uma Mensagem JMS para continuar seu fluxo.

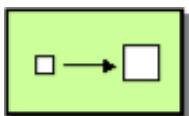
O exemplo abaixo mostra um par de processadores responsáveis por empacotar e desempacotar uma mensagem em um elemento XML. O empacotamento, neste caso, requer a codificação da imagem (que não seria necessário se fosse texto):

```
01. import java.util.Base64;
02.
03. public class XMLDataWrapper {
04.     public String wrap(byte[] data) {
05.         String base64String = Base64.getEncoder().encodeToString(data);
06.         String xml = "<data>" + base64String + "</data>";
07.         return xml;
08.     }
09. }
10.
11. import java.util.Base64;
12.
13. public class XMLDataUnwrapper {
14.     public static String BEGIN_TAG = "<data>";
15.     public static String END_TAG    = "</data>";
16.
17.     public byte[] unwrap(String xmlPayload) {
18.         System.out.println("String to process: " + xmlPayload);
19.
20.         int begin = xmlPayload.indexOf(BEGIN_TAG) + BEGIN_TAG.length();
21.         int end   = xmlPayload.indexOf(END_TAG);
22.
23.         String base64String = xmlPayload.substring(begin, end);
24.         byte[] data = Base64.getDecoder().decode(base64String);
25.         return data;
26.     }
27. }
```

Um processador de payload (no JMS, Camel ou Spring Integration) poderia chamar esses métodos para empacotar/desempacotar a imagem. O processador deve também alterar o tipo (Indicador de Formato) da mensagem para refletir o novo estado (XML) e guardar o tipo da mensagem empacotada para que ela possa ser reconstruída depois.

## (42) Enriquecedor de conteúdo (Content Enricher)

### Ícone



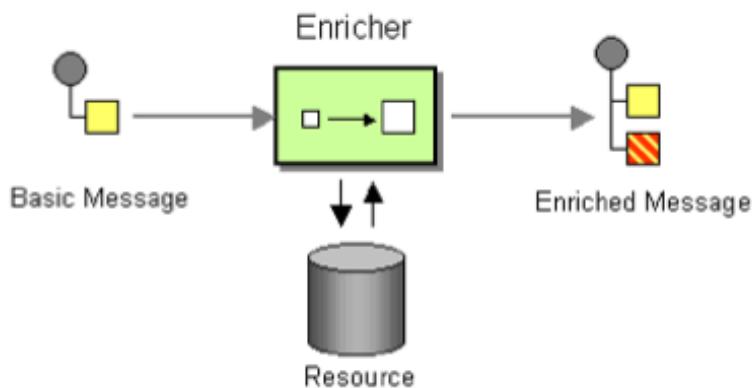
### Problema

“Como podemos nos comunicar com outro sistema se o criador da mensagem não possui todos os dados necessários?”

### Solução

“Use um transformador especializado, um Enriquecedor de Conteúdo (Content Enricher), para acessar uma fonte de dados externa de maneira a adicionar à mensagem informação que lhe falta”

### Diagrama



### Descrição

Um Enriquecedor de Conteúdo (Content Enricher) é um componente de transformação que acrescenta dados à mensagem. Os dados podem ser obtidos de uma fonte externa, podem ser computados a partir de informações presentes na própria mensagem ou no ambiente.

Por exemplo, uma mensagem pode incluir informações de data e hora, usuário logado, configurações do sistema, e outras informações obtidas do ambiente em que está executando. Essas informações podem ser adicionadas em cabeçalhos ou em estruturas do corpo da mensagem. É possível que os dados para enriquecer o conteúdo estejam todos na própria mensagem. Por exemplo, uma mensagem que possui dados comprimidos ou criptografados, pode descomprimir ou decifrar seus dados e usá-los para definir cabeçalhos e estruturas no corpo da mensagem. A mensagem também pode conter uma URL, link, credenciais, token ou outro identificador de recurso que permita acessar um recurso externo e obter dados que irão ser adicionados a uma mensagem. Por exemplo, uma mensagem pode conter várias URLs de imagem que o Enriquecedor de Conteúdo poderá usar para baixar os dados dessas

imagens e anexar na mensagem enviada para a saída. Em uma outra situação a mensagem pode conter um objeto inicialmente vazio que obtém valores para suas propriedades de algum lugar.

Um Enriquecedor de Conteúdo pode ser usado em uma rota que é precedida por um Filtro de Conteúdo (Content Filter) com Recibo de Bagagem (Claim Check). O Recibo pode ser uma URL usada para recuperar dados que foram removidos da mensagem em uma etapa anterior.

## Aplicações

Um Enriquecedor de Conteúdo (Content Enricher) é usado para transformar os dados de uma mensagem de forma a incluir informações adicionais, que podem ser obtidas da própria mensagem ou do ambiente externo.

### Enriquecedor de Conteúdo em Java / JMS

No exemplo abaixo temos um banco de dados que contém uma coleção de produtos com ID (chave primária), código e preço. O produto é representado pela classe abaixo:

```
public class Produto implements Serializable {
    private long id;
    private double preco;
    private String codigo;
    ...
}
```

Um produtor de mensagens cria uma instância vazia (apenas com valores default) mas incluindo uma chave primária, e envia para uma fila:

```
01. Destination to = (Destination) ctx.lookup("produtos");
02. producer = session.createProducer(to);
03. Produto p = new Produto(3);
04. ObjectMessage message = session.createObjectMessage(p);
05. message.setStringProperty("Tipo", "Produto");
06. System.out.println("Enviando " + p);
07. producer.send(message);
```

No caminho, o Content Enricher abre a mensagem, e usa a chave primária para obter os dados sobre o produto do banco. Ele agora atualiza a mensagem com o objeto completo:

```
01. Destination from = (Destination) ctx.lookup("produtos");
02. Destination to = (Destination) ctx.lookup("saida");
03.
04. new JMSChannelBridge(con, from, to, new PayloadProcessor() {
05.     public Object process(Object payload) {
06.         long id = ((Produto)payload).getId();
07.         Produto p = ProductDatabase.getProduto(id);
08.         System.out.println("Produto obtido: " + p);
09.         return p;
10.     }
11.});
```

Resultado:

```
Produto obtido: (3) W008 $79.95
```

## Enriquecedor de Conteúdo em Apache Camel

Um Enriquecedor de Conteúdo pode ser implementado em Camel adicionando qualquer Processor ou Tradutor que obtenha dados de algum lugar e adicione à mensagem. Neste exemplo da documentação do Camel, informação extra é acrescentada ao payload:

```
from("direct:start").setBody(body().append("World!")).to("mock:result");
```

Isto também pode ser feito com um processador.

```
from("direct:start").process(new Processor() {
    public void process(Exchange exchange) {
        Message in = exchange.getIn();
        in.setBody(in.getBody(String.class) + " World!");
    }
}).to("mock:result");
```

Camel também possui uma instrução `enrich()` que pode ser usada em DSL. Ela permite enriquecer o conteúdo de uma mensagem usando dados obtidos de outra mensagem enviada para uma fila (a mensagem obtida da fila substitui a original). Se o objetivo é acrescentar dados da mensagem obtida à mensagem original deve-se criar uma estratégia de agregação (implementação da interface `AggregationStrategy`). Por exemplo, a seguinte estratégia:

```
public class MergeXMLStrategy implements AggregationStrategy {
    public Exchange aggregate(Exchange exchange, Exchange extras) {
        Object payload = exchange.getIn().getBody();
        Object payloadExtra = extras.getIn().getBody();
        return "<novo>" + payload + "\n" + payloadExtra + "</novo>";
    }
}
```

pode ser usada por um Enriquecedor de Conteúdo posicionado no meio de uma rota, da forma:

```
AggregationStrategy strategy = new MergeXMLStrategy();
from("direct:start")
    .enrich("direct:resource", strategy)
    .to("direct:result");
```

## Enriquecedor de Conteúdo em Spring Integration

Em Spring Integration existem Enriquecedores de Conteúdo para payloads e cabeçalhos. Os de cabeçalhos são mais simples, e aceitam dados ou referências para beans que calculam o valor.

```
<int:header-enricher input-channel="in" output-channel="out">
    <int:header name="foo" value="123"/>
    <int:header name="bar" ref="someBean"/>
```

```

<int:header name="foo" method="computeValue" ref="myBean"/>
</int:header-enricher>
<bean id="myBean" class="foo.bar.MyBean"/>

```

O bean acima possui um método que gera o valor do cabeçalho.

```

01. public class MyBean {
02.     public String computeValue(String payload) {
03.         return payload.toUpperCase() + "_US";
04.     }
05. }

```

Enrichers de payload podem ser usados para alterar propriedades de POJOs, como no exemplo mostrado em JMS:

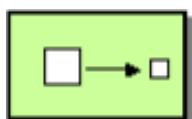
```

<int:enricher input-channel="produtos">
    <int:property name="produto.preco" expression="Math.random() * 2000"/>
</int:enricher>

```

## (43) Filtro de conteúdo (Content Filter)

### Ícone



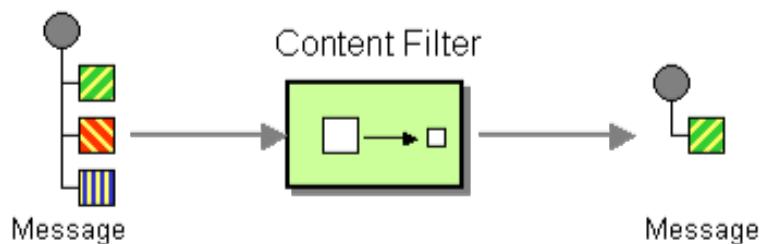
### Problema

“Como simplificar o processo de lidar com uma mensagem grande, quando há interesse em apenas alguns itens de dados?”

### Solução

“Use um Filtro de Conteúdo (Content Filter) para remover partes de uma mensagem que não são necessários, deixando apenas os itens importantes”

### Diagrama



## Descrição

Um Filtro de Conteúdo (Content Filter) é um componente de transformação que remove partes desnecessárias de uma mensagem. Por exemplo, uma aplicação que lista títulos, autores e assuntos de livros talvez receba mensagens contendo informações que não serão necessárias, como ISBN, número de páginas, dimensões, etc. Como essas informações não são usadas pelo destinatário, um Filtro de Conteúdo pode interceptar a mensagem retirando esses dados, deixando-a menor e mais fácil de processar.

Um Filtro de Conteúdo atua no escopo da Mensagem, transformando-a, enquanto que um Filtro de Mensagens atua no escopo do Canal, roteando-a. No contexto das informações que são transmitidas, ambos podem ser usados como soluções para problemas similares, dependendo de como as informações são encapsuladas nas mensagens.

Um filtro sempre descarta informações que não precisa. O Filtro de Conteúdo descarta itens desnecessários de uma mensagem, enquanto um Filtro de Mensagens descarta mensagens desnecessárias em um canal. Em vez de descartar as várias partes, um Filtro de Conteúdo pode ser combinado com um Roteador de Tipo de Conteúdo para enviar as várias partes para canais diferentes, implementando um Divisor (Splitter).

## Aplicações

Um Filtro de Conteúdo (Content Filter) deve ser usado quando for necessário ou desejável remover partes de uma mensagem, mantendo apenas elementos essenciais. Algumas razões que motivam esse tipo de transformação incluem segurança dos dados, permissões, eficiência da transmissão em rede, simplificação das mensagens, etc.

## Filtro de Conteúdo em Java / JMS

Um documento XML enviado através de um canal possui a seguinte estrutura:

```
<document>
    <name>Fish.png</name>
    <size>12290</size>
    <type>PNG</type>
    <owner><user>...</user><address>...</address><legal>...</legal>...</owner>
        <data>P48sn34HSD037hs9017#nd@sk...</data>
</document>
```

Numa determinada aplicação que irá rotear a mensagem, o bloco XML `<owner>` nunca será usado. Podemos otimizar a transmissão de dados se removermos esse bloco da mensagem. Como é XML, podemos filtrá-lo com a expressão XPath: `/document/owner`. Isto é feito no processador abaixo:

```
01.  public class XPathFilter {
02.      DocumentBuilderFactory dbf;
03.      DocumentBuilder db;
04.      XPath xpath;
05. 
```

```

06.     public XPathFilter() throws Exception {
07.         dbf = DocumentBuilderFactory.newInstance();
08.         db = dbf.newDocumentBuilder();
09.         XPath xpath = XPathFactory.newInstance().newXPath();
10.    }
11.
12.    public String removeContents(String expr, String xmlText) throws
Exception {
13.        Document doc =
14.            db.parse(new ByteArrayInputStream(xmlText.getBytes("UTF-
15.            8")));
16.        Node node = (Node) xpath.evaluate(expr, doc, XPathConstants.NODE);
17.        node.setTextContent(""); // empty
18.        return XMLUtils.nodeToString(doc);
19.    }
...

```

O processador é chamado em algum lugar da rota e transforma o payload, que agora prossegue sem esse bloco.

```

01. Destination from = (Destination) ctx.lookup("documents");
02. Destination to   = (Destination) ctx.lookup("filtered-documents");
03.
04. new JMSChannelBridge(con, from, to, new PayloadProcessor() {
05.     public Object process(Object payload) {
06.         String filteredPayload =
07.             new XPathFilter().removeContents("/document/owner",
payload);
08.         System.out.println("Filtered payload: " + filteredPayload);
09.         return filteredPayload;
10.     }
11. });

```

## Filtro de Conteúdo em Camel

Este padrão não é implementado explicitamente mas pode ser implementado com qualquer processador que remova dados ou cabeçalhos da mensagem. O processador usado no exemplo JMS pode ser empregado para selecionar o conteúdo a remover.

## Filtro de Conteúdo em Spring Integration

Em Spring Integration pode-se eliminar cabeçalhos indesejados usando um <header-filter>:

```

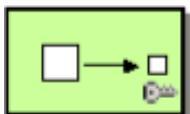
<int:header-filter input-channel="inputChannel"
                   output-channel="outputChannel" header-names="lastName,
state"/>

```

A filtragem de conteúdo pode ser feita usando os mesmos processadores usados nos exemplos JMS e Camel.

## (44) Recibo de bagagem (Claim Check)

### Ícone



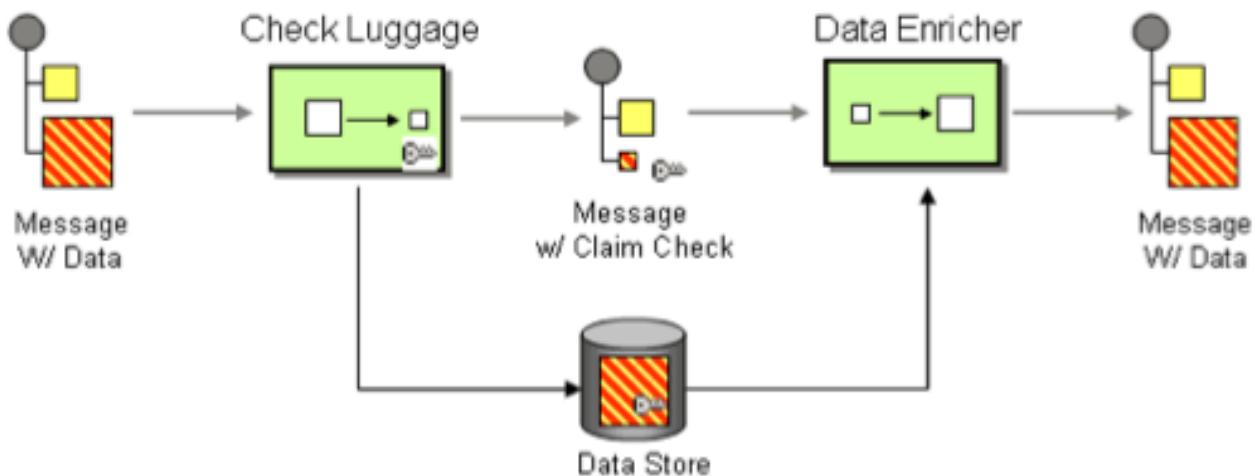
### Problema

“Como podemos reduzir o volume de dados de uma mensagem enviada através do sistema sem sacrificar o conteúdo da informação?”

### Solução

“Guarde os dados da mensagem em um repositório persistente e passe um Recibo de Bagagem (Claim Check) para os componentes seguintes. Esses componentes poderão usar o Recibo para recuperar a informação armazenada”

### Diagrama



### Descrição

Um Recibo de Bagagem serve para diminuir a carga transportada por uma mensagem, despachando os dados em um repositório em troca de um recibo (uma URI, por exemplo). Ao consumir a mensagem o destinatário pode recuperar os dados despachados.

A solução é similar ao Filtro de Conteúdo, com a diferença que os dados removidos da mensagem não são perdidos, mas substituídos por um componente que pode ser usado para recuperá-los posteriormente. É uma analogia com o despacho de bagagens em um aeroporto, e similar ao mecanismo usado em serviços de storage na nuvem, como DropBox, Google Drive, etc. onde em vez de anexar uma informação por email, o link ou a chave para o recurso é enviado.

Os dados podem ser despachados por vários motivos. Um deles é a redução de tráfego na rede, reduzindo a quantidade de dados transportados, e o marshalling e unmarshalling desnecessários que seriam realizados nas etapas intermediárias. Outro motivo pode ser a simplificação da mensagem, que irá facilitar o trabalho dos componentes que precisarem manipular os dados, ou ainda a segurança, ao evitar transferir dados sensíveis. Os dados não precisam esperar até o destino final para serem recuperados. Qualquer componente intermediário poderá usar o Recibo de Bagagem para reaver os dados.

O Recibo pode ser implementado com propriedades no cabeçalho da mensagem. Pode ser um link, uma URI, um ID, ou credenciais necessárias para reaver os dados. A implementação também deve decidir o que fazer com os dados armazenados se o cliente não for buscá-los (se os dados devem ter um prazo de validade antes de expirar, se devem ser removidos, se será enviado para um canal de achados e perdidos, etc.)

## Aplicações

Um Recibo de Bagagem (Claim Check) pode ser usado quando uma mensagem que precisa ser roteada por vários processadores contém dados que não serão alterados no processo e só serão usados pelo destinatário. É uma solução que torna a comunicação mais eficiente, pois evita que dados que não serão processados passem por componentes sem necessidade.

Outras aplicações são segurança (dados que não devem circular pela rede), ou simplificação (facilitar o processamento pelos componentes intermediários).

## Recibo de Bagagem em Java / JMS

No exemplo anterior usamos um Filtro de Conteúdo para remover um bloco da mensagem XML que não era usado durante o roteamento. O bloco <data> que contém uma imagem codificada em Base64 também não é usado durante o processamento, mas será necessária no final, quando a mensagem for reconstruída. Pode-se usar o padrão Recibo de Bagagem em JMS da mesma forma que o Filtro de Conteúdo mostrado. Desta vez filtramos o bloco <data>, mas guardamos esse fragmento de XML em um arquivo ou banco de dados para recuperar depois. Incluímos no lugar ou no cabeçalho da mensagem o caminho ou chave para que possamos recuperar o arquivo depois.

```
01.  public void onMessage(Message incomingMessage) {  
02.      try {  
03.          String payload = ((TextMessage) incomingMessage).getText();  
04.  
05.          String data = new XPathFilter().extractNode("/document/data",  
payload);  
06.          String key = new XPathFilter().extractText("/document/filename",  
payload);  
07.          DataStore.save(key, data);  
08.  
09.          String filteredPayload =  
               new XPathFilter().removeContents("/document/data",  
payload);
```

```

10.     System.out.println("Filtered payload: " + filteredPayload);
11.     TextMessage filtered = session.createTextMessage(filteredPayload);
12.
13.     filtered.setStringProperty("ClaimCheck", key);
14.
15.     producer.send(filtered);
16.
17. } catch (JMSEException e) {
18.     e.printStackTrace();
19. }
20. }
```

## Recibo de Bagagem em Apache Camel

O Recibo de Bagagem pode ser implementado em Camel usando um componente (bean) para guardar os dados em um data source, retendo um ID, e outro para recuperar do data source (um Content Enricher). Pode ser feito com Processors ou com pipeline. A rota abaixo usa um pipeline e dois beans para realizar a operação:

```

from("direct:start")
    .bean(new CheckInFilter(), "checkIn")
    .to("direct:checkpoint")
    .bean(new CheckOutEnricher(), "checkOut");
    .to("mock:result");
```

O primeiro extrai a parte da mensagem que precisa ser guardada, armazena no banco, guarda um cabeçalho com a chave e depois troca o corpo da mensagem pela versão que não contém a parte armazenada.

```

01. public class CheckInFilter {
02.     public void checkIn(Exchange exchange,
03.                         @XPath("/document/data") String data,
04.                         @XPath("/document/name") String filename) {
05.
06.         DataStore.save(filename, data);
07.         exchange.getIn().setHeader("chave", filename);
08.         // remove o conteúdo de <data/>
09.         Object newBody = new
10.             XpathFilter().removeContents("/document/data",
11.
12.             exchange.getIn().getBody());
13.             exchange.getIn().setBody(newBody);
14. }
```

O segundo é um Enriquecedor de Conteúdo, que consulta o banco, obtém os dados e põe de volta na mensagem dentro no bloco <data>:

```
01. public static class CheckOutEnricher {
```

```

02.     public void addDataBackIn(Exchange exchange,
                               @Header("chave") String claimCheck) {
03.
04.         String data = DataStore.get(claimCheck));
05.         Object body = new XPathEnricher().insert("/document/data",
06.                                         data,
07.                                         exchange.getIn().getBody());
08.         exchange.getIn().setBody(body);
09.         DataStore.remove(claimCheck);
10.        exchange.getIn().removeHeader("chave");
11.    }

```

## Recibo de Bagagem em Spring Integration

Spring Integration fornece o elemento <claim-check-in> e <claim-check-out> para armazenar dados temporariamente em um Message Store e recuperar depois. Para armazenar dados deve-se extraí-los e depois transferir apenas os dados que se deseja despachar para um canal que é a entrada do <claim-check-in> (input-channel). O processo envia uma mensagem para o output-channel contendo uma mensagem com payload que é o ID gerado (o Recibo). Esse ID pode ser usado para recuperar os dados

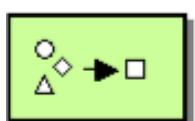
```
<int:claim-check-in input-channel="checkinChannel"
                     message-store="referencia-para-message-store"
                     output-channel="outputClaimCheckChannel"/>
```

Para recuperar os dados, envie a mensagem contendo o Recibo para o canal de entrada de <claim-check-out> e os dados serão disponibilizados em output-channel:

```
<int:claim-check-out input-channel="checkoutChannel"
                     message-store=" referencia-para-message-store "
                     output-channel="output"/>
```

## (45) Normalizador (Normalizer)

### Ícone



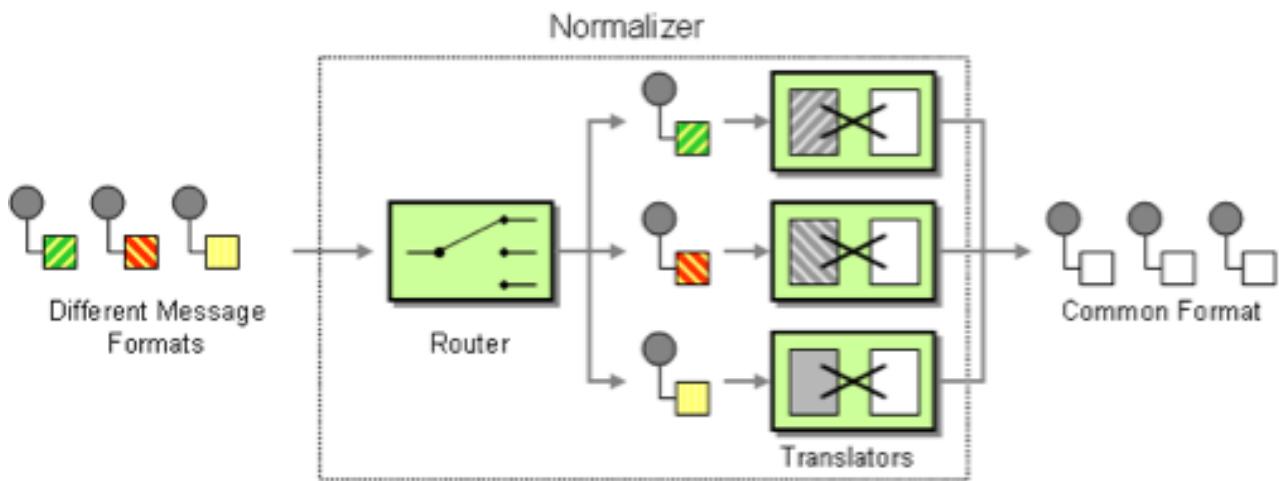
### Problema

“Como processar mensagens que são semanticamente equivalentes mas que chegam em um formato diferente?”

## Solução

“Use um Normalizador (Normalizer) para rotear cada tipo de mensagem através de um Message Translator específico, para que as mensagens resultantes tenham um formato comum”

## Diagrama



## Descrição

O Normalizador (Normalizer) é um componente composto que recebe mensagens de mesmo significado mas em diferentes formatos na entrada, e envia para a saída as mensagens convertidas para um formato comum. Pode ser implementada usando um Roteador Baseado em Conteúdo que envie as mensagens para canais diferentes, de acordo com o seu formato. Cada canal está conectado a um Tradutor de Mensagens que converte as mensagens em um formato único.

## Aplicações

Um Normalizador deve ser usado sempre que houver necessidade de trabalhar com dados semânticamente equivalentes, mas que estão em formatos diferentes.

Por exemplo, uma companhia aérea publica número, data, horário, origem, destino, duração de voo e preço em XML. Outra publica as mesmas informações em JSON. Uma terceira em CSV. Usando um normalizador, pode-se escolher um formato comum para todas as mensagens, e permitir que outro componente compare preços e duração de voos.

Um outro exemplo: uma aplicação define uma *Pessoa* contendo *nome*, *sobrenome* e *userid*. Outra define *Usuario* com *nome* e *login*, onde *nome* contém o nome e o sobrenome separado por um espaço. Um Normalizador poderia ser usado para escolher um dos formatos (ex: *nome* e *login*) ou uma combinação deles (ex: *nome* e *userid*, ou *nome*, *sobrenome* e *login*) de forma que as mensagens resultantes possam ser comparadas.

## Normalizador em Apache Camel

Um Normalizador pode ser implementado com processadores. O CBR abaixo usa uma série de processadores que recebem mensagens em formatos diferentes, converte para um formato comum e deposita as mensagens em um único canal.

```
from("jms:queue"inbound")
    .choice()
        .when(mensagemTipo1)
            .process(Tipo1toCommonProcessor).to("direct:common")
        .when(mensagemTipo2)
            .process(Tipo2toCommonProcessor).to("direct:common")
        .when(mensagemTipo3)
            .process(Tipo3toCommonProcessor).to("direct:common")
        .when(mensagemTipo4)
            .process(Tipo4toCommonProcessor).to("direct:common")
    .otherwise()
        .to("jms:queue:invalidos");
```

## (46) Modelo de dados canônico (Canonical Data Model)

### Ícone



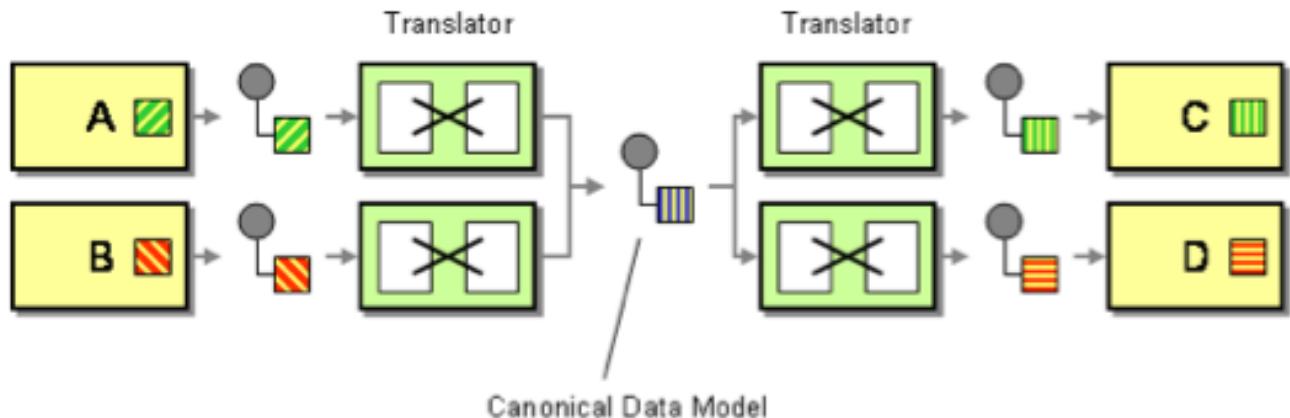
### Problema

“Como minimizar dependências ao integrar aplicações que usam diferentes formatos de dados?”

### Solução

“Crie um Model de Dados Canônico (Canonical Data Model) que seja independente de qualquer aplicação específica. Cada aplicação deve produzir e consumir mensagens neste formato comum”

## Diagrama



## Descrição

Para que haja comunicação entre dois sistemas que possuem mensagens em formatos diferentes, é preciso que haja um par de tradutores entre eles, adicionando-se mais um sistema, serão seis, e assim por diante. O número de tradutores necessários cresce exponencialmente.

Em vez de criar um par de tradutores para viabilizar a comunicação entre dois sistemas, cada sistema pode ter um tradutor que converta suas mensagens em um Modelo de Dados Canônico (Canonical Data Model) - um formato comum que pode ser usado em toda a aplicação. Desta forma, cada componente terá um tradutor apenas, de forma que o número de tradutores será igual ao número de componentes, ou duas vezes a quantidade de aplicações que serão integradas.

Um Modelo de Dados Canônico pode evoluir durante o tempo de vida de uma aplicação e deve ser versionado.

## Aplicações

Quando existe a necessidade de integrar mais de duas aplicações que precisam trocar mensagens em formatos incompatíveis entre si, um Modelo de Dados Canônico pode ser usado para diminuir a quantidade de tradutores que seriam necessários.

## Revisão

Padrões de integração de sistemas relacionados a transformação:

- (41) *Envelope (Envelope Wrapper)*: empacotamento usado para que informações de formatos incompatíveis possam trafegar por um canal.
- (42) *Enriquecedor de conteúdo (Content Enricher)*: tradutor que aumenta a quantidade de dados em uma mensagem.
- (43) *Filtro de conteúdo (Content Filter)*: tradutor que reduz a quantidade de dados em uma mensagem.

- (44) *Recibo de bagagem (Claim Check)*: descreve como substituir um payload por um recibo (link, chave, etc.) que pode ser usado para recuperar o payload posteriormente.
- (45) *Normalizador (Normalizer)*: um componente que traduz mensagens em formatos diferentes a um formato comum.
- (46) *Modelo de Dados Canônico (Canonical Data Model)*: um modelo de dados comum usado em toda uma aplicação, para facilitar o processamento e evitar uma multiplicação de tradutores.

# Capítulo 8

# Endpoints

Endpoints são terminais que conectam uma aplicação a um sistema de mensageria. São clientes de mensageria, produtores e consumidores de mensagens, remetentes e destinatários. Usar uma API como JMS é codificar endpoints: é preciso criar um Producer para enviar mensagens e um Consumer para consumi-las. Frameworks como Camel ou Spring Integration oferecem terminais que conectam a vários tipos de serviços externos. Padrões como Adaptador de Canal (Channel Adapter) incluem um terminal já acoplado a um canal.

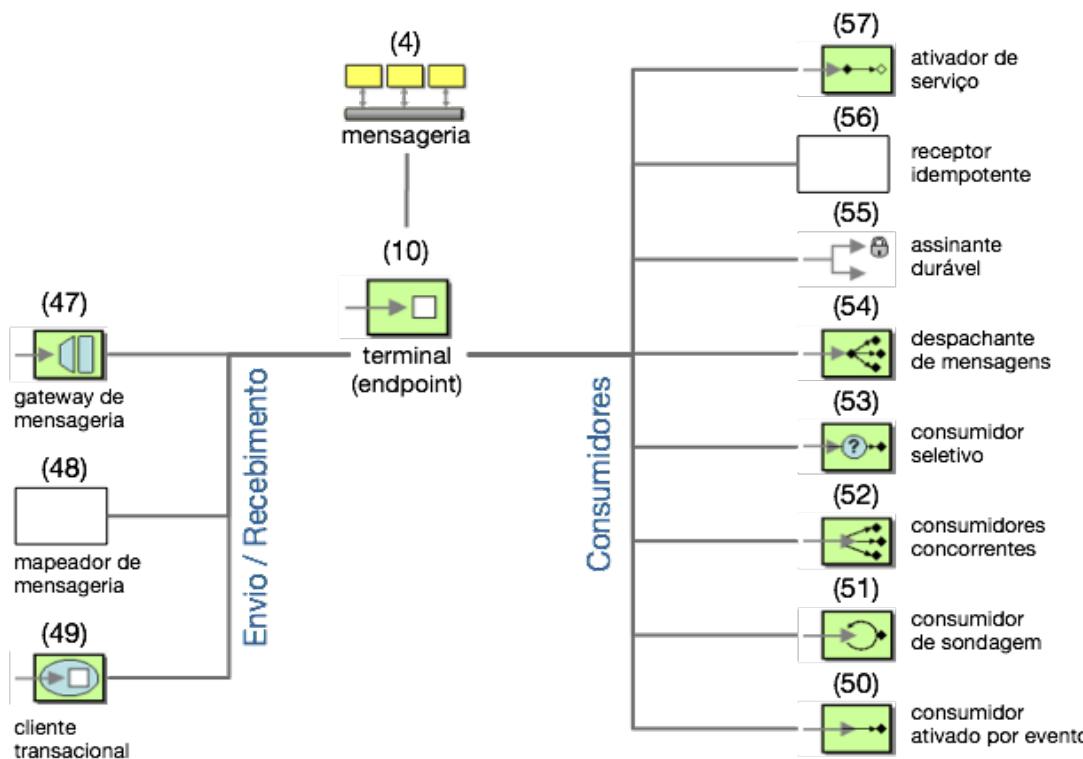
O padrão Terminal de Mensageria (Messaging Endpoint) descreve um produtor ou consumidor genérico em um sistema de mensageria. O catálogo EIP descreve outros onze padrões relacionados com Messaging Endpoint. Três padrões são relacionados a terminais de mensageria em geral (consumidores e produtores):

- *Gateway de Mensageria (Messaging Gateway)* é um padrão que desacopla o sistema de mensageria do restante da aplicação. É análogo a usar um DAO para isolar um banco de dados. Um Gateway de Mensageria fornece uma API genérica para que a aplicação possa enviar e receber mensagens sem precisar saber que está usando um serviço de mensageria.
- Um *Mapeador de Mensageria (Messaging Mapper)* permite um desacoplamento maior que o Gateway de Mensageria ao permitir que objetos de negócio usem mensageria sem explicitamente chamarem operações de envio e recebimento de mensagens. É análogo a usar JPA para permitir que objetos de negócio usem persistência transparentemente.
- Às vezes é necessário enviar ou receber mensagens dentro de um contexto transacional. O padrão *Cliente Transacional (Transactional Client)* descreve a solução onde produtores enviam mensagens e e/ou consumidores consomem mensagens dentro de uma transação.

Os oito padrões restantes descrevem consumidores, e podem ser classificados de acordo com sua finalidade ou estratégia de consumo. A escolha pode ser influenciada pela possibilidade do servidor regular a taxa de processamento das mensagens, que tem a ver com a maneira como as mensagens são consumidas (se consomem imediatamente, se colocam em fila, se guardam em meio persistente, se controlam quem consome, se precisam consumir de forma síncrona, etc.):

- Os padrões *Consumidor de Sondagem (Polling Customer)* e *Consumidor Ativado por Evento (Event-Driven Customer)* descrevem estratégias de consumir uma mensagem de forma síncrona (bloquear o thread até que a mensagem seja recebida) ou assíncrona (escrever um handler de eventos que será notificado quando a mensagem chegar).
- *Consumidores Concorrentes (Competing Customers)* e *Despachante de Mensagens (Message Dispatcher)* são estratégias que determinam se um consumidor disputa as mensagens em um canal com outros consumidores, ou se o sistema mantém controle sobre quais consumidores recebem mensagens.
- Um *Consumidor Seletivo (Selective Consumer)* usa um filtro para decidir quais, dentre as mensagens enviadas para um canal, irá consumir.
- Um *Assinante Durável (Durable Subscriber)* é um consumidor de mensagens enviadas para um canal Publica-Inscreve que terá suas mensagens guardadas mesmo que esteja inativo no momento em que elas forem enviadas.
- O *Receptor Idempotente (Idempotent Receiver)* é um consumidor que pode receber uma mesma mensagem uma vez ou várias vezes, e reagir exatamente da mesma maneira todas as vezes. Mensagens duplicadas não interferem no seu funcionamento.
- Para disponibilizar um serviço síncrono como RPC ou outro através de uma interface assíncrona do sistema de mensageria, o serviço pode ser interceptado por um *Ativador de Serviço (Service Activator.)*

O diagrama abaixo ilustra os padrões relacionados a Terminais de Mensageria:



## (47) Gateway de mensageria (Messaging Gateway)

### Ícone



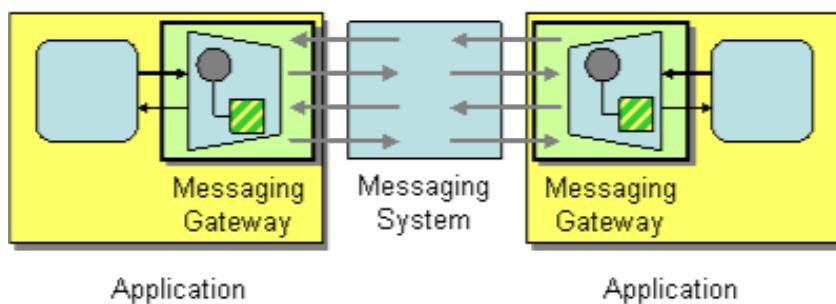
### Problema

“Como isolar o acesso ao sistema de mensageria do restante da aplicação?”

### Solução

“Use um *Gateway de Mensageria (Messaging Gateway)*: uma classe que encapsula chamadas específicas ao sistema de mensageria e expõe uma interface com métodos específicos ao domínio da aplicação”

### Diagrama



### Descrição

Um Gateway de Mensageria (Messaging Gateway) fornece uma API independente do sistema de mensageria usado que encapsula código específico do sistema. Desta forma, a aplicação pode enviar e receber mensagens sem precisar saber nada sobre o sistema de mensageria. Funciona de forma análoga a um DAO (Data Access Object) que encapsula código de acesso a banco de dados, oferecendo uma camada de abstração no domínio da aplicação. O Gateway de Mensageria é um Gateway (Padrão do catálogo PEAA - *Patterns of Enterprise Application Architecture*, de Martin Fowler).

Os métodos de um Gateway de Mensageria podem receber e retornar POJOs que estão mapeadas a mensagens. Exceções específicas do sistema devem ser capturadas e relançadas como exceções do domínio da aplicação. Gateways podem ter interfaces síncronas ou assíncronas. Implementações de Requisição-Resposta podem usar métodos síncronos que bloqueiam o processamento enquanto a resposta não chega, ou listeners que são notificados de forma assíncrona.

Gateways podem ser empilhados em cascata, delegando mensagens entre si. Por exemplo, um gateway genérico oferecendo uma API de acesso a funções genéricas de mensageria pode ser reusado em diferentes aplicações, e encapsulado por outros gateways em camadas mais altas que forneçam APIs no domínio de cada aplicação.

Um gateway é um adaptador e deve conter código para delegar chamadas ao sistema de mensageria, convertendo formatos de dados se necessário. Esse tipo de código muitas vezes é previsível e pode ser gerado por ferramentas. Por exemplo, no Java Runtime Environment (JRE) pode-se usar a ferramenta *wsdl2java* para gerar gateways em Java capazes de acessar serviços de mensageria com Web Services SOAP.

A aplicação que usa o gateway interage apenas com interfaces e não tem acesso ao código que está do outro lado. Isto permite que essas interfaces sejam implementadas com dados simulados e mock objects, viabilizando a realização de testes unitários da aplicação sem a necessidade de ter um sistema de mensageria disponível.

## Aplicações

Um Gateway de Mensageria é usado para isolar uma aplicação do código usado na API de um produto de mensageria, de forma que a aplicação use uma API do domínio da aplicação que encapsula chamadas ao sistema de mensageria.

### Gateway de Mensageria em Java / JMS

A própria API JMS é um gateway de baixo nível, pois abstrai detalhes da infraestrutura do sistema de mensageria usado (ActiveMQ, HornetMQ, IBM MQ, etc.) que é completamente isolada pelas abstrações Destination, Message, MessageListener, etc.

Mas podemos também criar um Gateway de Mensageria em um nível mais elevado, de forma a isolar o JMS do restante da aplicação. O conjunto de classes e interfaces abaixo podem ser usadas para fornecer um gateway simples (e limitado) a um serviço de mensageria.

O Gateway possui três operações:

```

12.  public interface SimpleMessagingGateway {
13.      void send(SimpleChannel c, SimpleMessage m) throws MessagingException;
14.      SimpleMessage receive(SimpleChannel c) throws MessagingException;
15.      void register(MessagingEventHandler handler, SimpleChannel c)
16.          throws
17.          MessagingException;
16.  }
```

A classe SimpleChannel guarda apenas o nome de um canal (cujo significado dependerá do sistema usado):

```

17.  public class SimpleChannel {
18.      private String name;
19.      public SimpleChannel(String name) {...}
20.      public String getName() {...}
21.      public void setName(String name) {...}
22.  }
```

A classe SimpleMessage representa uma mensagem que contém texto e cabeçalhos que têm nome/valor em formato String:

```

23.  public class SimpleMessage {
24.      private Map<String, String> headers = new HashMap<>();
25.      private String payload;
26.
27.      public SimpleMessage(Map<String, String> headers, String payload) {
28.          this.headers = headers;
29.          this.payload = payload;
30.      }
31.
32.      public SimpleMessage(String payload) {
33.          this.payload = payload;
34.      }
35.
36.      public void setHeader(String key, String value) {
37.          headers.put(key, value);
38.      }
39.      public String getHeader(String key) {
40.          return headers.get(key);
41.      }
42.
43.      public Map<String, String> getHeaders() {...}
44.      public void setHeaders(Map<String, String> headers) {...}
45.      public String getPayload() {...}
46.      public void setPayload(String payload) {...}
47.
48.      @Override
49.      public String toString() {
50.          return "SimpleMessage [headers=" + headers + ", payload=" +
51.                  payload + "]";
52.      }

```

O Gateway também inclui um handler para receber mensagens de forma assíncrona:

```

53.  public interface MessagingEventHandler {
54.      void process(SimpleMessage m);
55.  }

```

e uma classe de Exception:

```

56.  public class MessagingException extends Exception {
57.      public MessagingException() {}
58.      public MessagingException(String message) {
59.          super(message);
60.      }
61.      public MessagingException(Throwable cause) {
62.          super(cause);
63.      }
64.  }

```

Por exemplo, um cliente poderia usar essa interface da seguinte forma:

```

65.  public class ExampleMessagingClient {
66.      public static void main(String[] args) {
67.          try {
68.              SimpleMessagingGateway gateway = new JMSMessagingGateway();
69.              SimpleMessage message =
70.                  new SimpleMessage("<message>Hello World!</message>");
71.              message.setHeader("Type", "xml");
72.              message.setHeader("Length", "12");
73.
74.              // Configured in provider-specific file (jndi.properties for
75.              // JMS)
76.              SimpleChannel fromChannel = new SimpleChannel("in-channel");
77.
78.              gateway.register(new MessagingEventHandler() {
79.                  @Override
80.                  public void process(SimpleMessage m) {
81.                      System.out.println("Received: " + m);
82.                  }
83.              }, fromChannel);
84.
85.              System.out.println("Sending message.");
86.              gateway.send(fromChannel, message);
87.
88.          } catch (MessagingException e) {
89.              e.printStackTrace();
90.          }
91.      }

```

Não há nenhuma pista no código de que ele esteja usando JMS ou qualquer outro sistema de troca de mensagens (a não ser na configuração da implementação usada, que poderia ser feita de forma declarativa, usando Spring por exemplo). Eis uma possível implementação para o gateway em JMS:

```

91.  public class JMSMessagingGateway implements SimpleMessagingGateway {
92.
93.      private Connection con;
94.      private Context jndiContext;
95.      MessageConsumer consumer;
96.
97.      public JMSMessagingGateway() throws MessagingException {
98.          try {
99.              jndiContext = new InitialContext();
100.             ConnectionFactory factory =
101.                 (ConnectionFactory)
jndiContext.lookup("ConnectionFactory");
102.             con = factory.createConnection();
103.         } catch (NamingException | JMSEException e) {

```

```
103.             throw new MessagingException(e);
104.         }
105.     }
106.
107.     public void closeConnection() {
108.         try {
109.             con.close();
110.         } catch (JMSEException e) {
111.             e.printStackTrace();
112.         }
113.     }
114.
115.     private Destination getDestination(String jndiName)
116.                     throws MessagingException {
117.         try {
118.             return (Destination) jndiContext.lookup(jndiName);
119.         } catch (NamingException e) {
120.             throw new MessagingException(e);
121.         }
122.
123.         private Session createSession() throws MessagingException {
124.             try {
125.                 return con.createSession(false, Session.AUTO_ACKNOWLEDGE);
126.             } catch (JMSEException e) {
127.                 throw new MessagingException(e);
128.             }
129.         }
130.
131.         private SimpleMessage createSimpleMessage(TextMessage jmsMessage)
132.                         throws
133. JMSEException {
134.             SimpleMessage m = new SimpleMessage(jmsMessage.getText());
135.             Enumeration<?> properties = jmsMessage.getPropertyNames();
136.             while (properties.hasMoreElements()) {
137.                 String key = (String) properties.nextElement();
138.                 m.setHeader(key, jmsMessage.getStringProperty(key));
139.             }
140.
141.             @Override
142.             public void send(SimpleChannel c, SimpleMessage m)
143.                 throws MessagingException {
144.                 Session session = createSession();
145.                 Destination destination = getDestination(c.getName());
146.                 try {
147.                     MessageProducer producer =
session.createProducer(destination);
```

```
148.             TextMessage jmsMessage =
session.createTextMessage(m.getPayload());
149.             for (Map.Entry<String, String> header :
m.getHeaders().entrySet()) {
150.                 jmsMessage
151.                     .setStringProperty(header.getKey(),
header.getValue());
152.             }
153.             producer.send(jmsMessage);
154.         } catch (JMSEException e) {
155.             throw new MessagingException(e);
156.         }
157.     }
158.
159.     @Override
160.     public SimpleMessage receive(SimpleChannel c) throws
MessagingException {
161.         Session session = createSession();
162.         Destination destination = getDestination(c.getName());
163.         try {
164.             consumer = session.createConsumer(destination);
165.             TextMessage jmsMessage = (TextMessage) consumer.receive();
166.             return createSimpleMessage(jmsMessage);
167.         } catch (JMSEException e) {
168.             throw new MessagingException(e);
169.         }
170.     }
171.
172.     @Override
173.     public void register(MessagingEventHandler handler, SimpleChannel c)
throws MessagingException {
174.         Session session = createSession();
175.         Destination destination = getDestination(c.getName());
176.         try {
177.             consumer = session.createConsumer(destination);
178.             consumer.setMessageListener(new MessageListener() {
179.                 @Override
180.                 public void onMessage(Message jmsMessage) {
181.                     try {
182.                         SimpleMessage m =
183.                             createSimpleMessage((TextMessage)
jmsMessage);
184.                         handler.process(m);
185.                     } catch (JMSEException e) {
186.                         e.printStackTrace();
187.                     }
188.                 }
189.             });
190.             con.start();
```

```

191.         } catch (JMSEException e) {
192.             throw new MessagingException(e);
193.         }
194.     }
195. }
```

## Gateway de Mensageria em Apache Camel

Apache Camel também pode ser considerado um Messaging Gateway de baixo-nível, já que abstrai da infraestrutura e sistemas de mensageria que são isolados pelos Componentes. É possível usar beans ou componentes Apache CXF (SOAP Web Services) para isolar endpoints através de interfaces. O exemplo abaixo (da documentação Camel) mostra uma rota entre duas interfaces:

```
from("cxft:bean:soapMessageEndpoint")
    .to("bean:testBean?method=processSOAP");
```

Outra maneira de implementar um Gateway Produtor em Camel é usar um Proxy, que pode ser configurado em Spring ou em Java e posteriormente usado a partir de código Java isolando completamente o Camel da aplicação. Por exemplo:

```
public interface CamelGateway {
    void enviarMensagem(String texto);
}
```

pode ser configurado em Spring usando:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <proxy id="gateway"
        serviceInterface="br.com.argonavis.cursocamel.CamelGateway"
        serviceUrl="jms:queue:in-channel"/>
    <route>
        <from uri="jms:queue:in-channel"/>
    </route>
</camelContext>
```

O cliente pode agora enviar mensagens sem interagir com o Camel:

```
AbstractApplicationContext spring =
    new ClassPathXmlApplicationContext("beans.xml");
CamelGateway proxy = spring.getBean("gateway", CamelGateway.class);
proxy.enviarMensagem("Hello!");
```

## Gateway de Mensageria em Spring Integration

Em Spring Integration é possível definir interfaces Java para serviços de gateway de entrada (envio ou requisição-resposta). Por exemplo, a interface abaixo:

```
public interface FileInboundAdapter {
    void save(byte[] data, @Header("Filename") String filename);
}
```

pode ser usada para enviar um arquivo para o sistema de mensageria. Para configurar o gateway é preciso informar a interface e os canais (default) que serão usados para requisição e resposta.

```
<gateway service-interface="br.com...FileInboundAdapter"
        default-request-channel="inbound"/>
```

## (48) Mapeador de mensageria (Messaging Mapper)

### Ícone



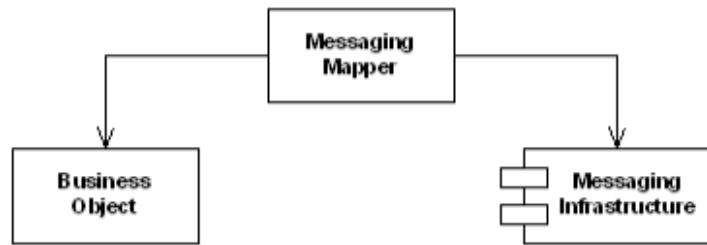
### Problema

“Como mover dados entre objetos de domínio e a infraestrutura de mensageria, e ainda manter os dois independentes um do outro?”

### Solução

“Crie um Mapeador de Mensageria (Messaging Mapper) que contenha a lógica entre a infraestrutura de mensageria e os objetos de domínio. Nem os objetos de domínio, nem a infraestrutura sabem da existência do Messaging Mapper”

### Diagrama



### Descrição

Um Mapeador de Mensageria (Messaging Mapper) permite que uma aplicação utilize serviços de mensageria sem explicitamente se comunicar com o sistema de mensageria, através de um mecanismo que mapeia a infraestrutura de mensageria aos objetos da aplicação.

O Mapeador de Mensageria (Messaging Mapper) é um Mapper (Padrão do catálogo PEAA - *Patterns of Enterprise Application Architecture*, de Martin Fowler). Não é uma camada de abstração como o Gateway. O Gateway esconde a camada de mensageria e usa objetos mapeados a mensagens mas a aplicação tem conhecimento que interage com o Gateway. O Mapeador de Mensageria é transparente à aplicação, que interage com objetos e eventos. Se o Gateway de Mensageria é uma solução análoga ao DAO, o Mapeador de Mensageria é uma solução análoga ao JPA.

O envio e recebimento de mensagens pelo sistema de mensageria ou pela aplicação disparam eventos. O Mapeador de Mensageria é notificado e executa operações na aplicação ou no sistema de mensageria, viabilizando a comunicação transparente entre as duas camadas.

O mapeamento entre objetos de negócio e mensagens faz parte da solução descrita por esse padrão. Em um sistema que utiliza mensagens em XML ou JSON pode-se usar JAXB para mapear objetos a mensagens.

Implementar todo o código de mapeamento pode ser uma tarefa complexa. É mais comum usar um framework que implemente esse padrão e gere automaticamente o código necessário. Exemplos de Mapeador de Mensageria estão presentes na plataforma Java EE, nas APIs JAX-RS (RESTful Web Services) e JAX-WS (SOAP Web Services). Frequentemente o formato de mensagem gerada pelo framework, mesmo sendo compatível, é inadequada para uso no sistema (ex: pode conter dados desnecessários ou em formato mais complexo). Neste caso é comum usar um Tradutor de Mensagens para adequar a mensagem a um formato canônico.

## Aplicações

Um Mapeador de Mensageria pode ser usado para incluir serviços de mensageria em uma aplicação sem alterar a sua lógica de negócios.

### Mapeador de Mensageria em Java / JMS

A interface abaixo possui um método para gravar um objeto, e outro para localizar o objeto pela sua chave-primária.

```
196. public interface SimpleMapperFacade {  
197.     void persist(Product p);  
198.     Product select(Long pk) throws ProductNotFoundException;  
199.     void closeConnection();  
200. }
```

Um cliente irá trabalhar com objetos Java comuns, completamente desacoplado do sistema de mensageria:

```
201. public class ProductClient {  
202.     public static void main(String[] args) throws MapperException {  
203.         SimpleMapperFacade facade =  
204.             new JmsMapperFacade("request-queue", "response-queue");  
205.         Product p1 = new Product(5L, "G837", 56.99);  
206.         facade.persist(p1);  
207.         try {  
208.             Product p2 = facade.select(3L);  
209.             System.out.println("Produto encontrado: " + p2);  
210.         } catch (ProductNotFoundException e) {  
211.             e.printStackTrace();  
212.         } finally {
```

```

214.         facade.closeConnection();
215.     }
216. }
217. }
```

Assim como no exemplo do Gateway, não há dependência da API JMS, e desta vez mapeamos um objeto à mensagem (e não uma operação). A implementação da interface utiliza JAXB para converter o objeto a ser gravado em um documento XML que será enviado no corpo da mensagem. No destino, as informações serão extraídas do XML e usadas para gravar o objeto no banco. A obtenção de um objeto pela sua chave-primária é realizada através de uma Mensagem-comando que será usada para enviar a chave. Localizado o objeto, seus dados são devolvidos em uma Mensagem-documento contendo sua representação XML, que novamente usando JAXB será convertido em objeto.

```

218. public class JmsMapperFacade implements SimpleMapperFacade {
219.     Destination requestQueue;
220.     Destination replyQueue;
221.     Connection con;
222.
223.     public JmsMapperFacade(String requestQueueName, String replyQueueName)
224.             throws MapperException {
225.         try {
226.             Context ctx = new InitialContext();
227.             ConnectionFactory factory = (ConnectionFactory) ctx
228.                 .lookup("ConnectionFactory");
229.             con = factory.createConnection();
230.             requestQueue = (Destination) ctx.lookup(requestQueueName);
231.             replyQueue = (Destination) ctx.lookup(replyQueueName);
232.             con.start();
233.         } catch (JMSEException | NamingException e) {...}
234.     }
235.
236.     @Override
237.     public void closeConnection() {...}
238.
239.     @Override
240.     public void persist(Product p) {
241.         try {
242.             StringWriter writer = new StringWriter();
243.
244.             JAXBContext jctx = JAXBContext.newInstance(Product.class);
245.             Marshaller m = jctx.createMarshaller();
246.             m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
247.             m.setProperty(Marshaller.JAXB_ENCODING, "UTF-8");
248.             m.marshal(p, writer);
249.
250.             String payload = writer.toString();
251.
252.             Session session = con
253.                 .createSession(false, Session.AUTO_ACKNOWLEDGE);
```

```
254.         MessageProducer sender = session.createProducer(requestQueue);
255.         TextMessage message = session.createTextMessage(payload);
256.         message.setStringProperty("Command", "addProduct");
257.
258.         System.out.println("Sending persist request for " + p);
259.         sender.send(message);
260.
261.     } catch (Exception e) {...}
262. }
263.
264. @Override
265. public Product select(Long pk) throws ProductNotFoundException {
266.     try {
267.         Session session = con
268.             .createSession(false, Session.AUTO_ACKNOWLEDGE);
269.         MessageProducer requestor =
270.             session.createProducer(requestQueue);
271.
272.         Message getProdutoCmd = session.createMessage();
273.         getProdutoCmd.setStringProperty("Command", "getProduct");
274.         getProdutoCmd.setLongProperty("ProductID", pk);
275.         getProdutoCmd.setJMSReplyTo(replyQueue);
276.
277.         System.out.println("Sending select request for " + pk);
278.         requestor.send(getProdutoCmd);
279.
280.         MessageConsumer receiver = session.createConsumer(replyQueue);
281.
282.         System.out.println("Waiting for reply.");
283.         TextMessage reply = (TextMessage) receiver.receive();
284.         if(!reply.getJMSCorrelationID()
285.             .equals(getProdutoCmd.getJMSMessageID())) {
286.             System.out.println("Wrong correlation for
request/reply!");
287.         }
288.
289.         Object exception = reply.getObjectProperty("Exception");
290.         if(exception != null) {
291.             throw (ProductNotFoundException)exception;
292.         } else {
293.             StringReader reader = new StringReader(reply.getText());
294.
295.             JAXBContext jctx = JAXBContext.newInstance(Product.class);
296.             Unmarshaller m = jctx.createUnmarshaller();
297.             return (Product) m.unmarshal(reader);
298.         }
299.     } catch (Exception e) {...}
299. }
```

```
300. }
```

Uma outra possibilidade para este exemplo seria gravar os objetos diretamente em um banco de dados XML, ou mapear os objetos a JSON e gravar em um banco NoSQL.

## Mapeador de Mensageria em Apache Camel e Spring Integration

Camel não oferece uma solução completa de mapeamento objeto-mensageria mas é possível converter objetos em mensagens e vice-versa através de processadores e transformadores.

Spring Integration possui métodos que recebem mensagens e devolvem objetos contidos nas mensagens (payload ou cabeçalhos), com mapeamento objeto-XML automático.

## (49) Cliente transacional (Transactional Client)

### Ícone



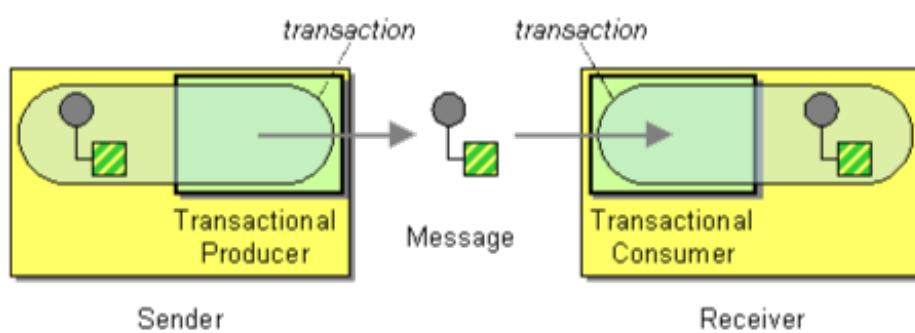
### Problema

“Como pode um cliente controlar suas transações com o sistema de mensageria?”

### Solução

“Use um Cliente Transacional (Transactional Client): faça com que a sessão do cliente com o sistema de mensageria seja transacional, para que o cliente possa especificar onde começa e onde termina uma transação”

### Diagrama



### Descrição

Um Cliente Transacional (Transactional Client) é um Produtor ou Consumidor de mensagens que inclui o envio ou recebimento de mensagens dentro de um contexto transacional, de forma que a operação seja tratada como uma unidade. Um Produtor transacional só transfere a mensagem para o

canal quando sua transação é cometida. Um Consumidor só remove a mensagem do canal quando sua transação é cometida.

O cliente deve ser capaz de determinar o escopo de sua transação. Sistemas de mensageria usam transações internamente, mas elas geralmente têm um escopo limitado a operações controladas pelo sistema ou métodos. As transações do sistema de mensageria terminam quando a mensagem é recebida ou consumida. Elas não são suficientes para incluir em um único escopo transacional uma Requisição e sua Resposta.

Transações geralmente são definidas como operações Atômicas, Consistentes, Isoladas e Duráveis (ACID). Mensagens são sempre atômicas, mas são duráveis apenas em sistemas que implementam entrega garantida (ex: Assinantes Duráveis). As operações de envio e recebimento devem ser isoladas de outros threads e seus resultados devem ser consistentes.

O catálogo EIP descreve alguns cenários em que seriam usados Clientes Transacionais:

- Sincronizar mensagens recebidas com mensagens enviadas: uma transação é iniciada para receber e processar a primeira mensagem, criar e enviar a segunda antes de cometer a transação, fazendo com que o recebimento e envio façam parte da mesma transação. Isto impede que a primeira mensagem seja removida do canal até que a segunda mensagem seja adicionada com sucesso ao seu canal.
- Garantir o envio (ou recebimento) de um grupo de mensagens: um grupo ou sequência de mensagens é enviada ou recebida dentro de um contexto transacional. Isto vai garantir que nenhuma mensagem seja adicionada ao canal até que tenha sido enviada, e que nenhuma mensagem seja removida do canal até que tenha sido recebida.
- Coordenação com workflow ou banco de dados: uma tarefa ou atualização de banco de dados que precisa ser associada a um envio ou recebimento deve ser incluída no mesmo contexto transacional que o envio ou recebimento. O objetivo é garantir que a alteração no banco de dados ou a execução da tarefa só será efetivada se a mensagem for recebida/removida, ou que a mensagem só será retirada ou adicionada em um canal havendo sucesso na atualização do banco ou execução da tarefa.

Não é possível incluir na mesma transação uma requisição e sua resposta, já que o envio requer que se cometa a transação, e se a transação não for cometida a mensagem nunca será enviada e consequentemente sua resposta nunca será recebida.

Um Consumidor Guiado por Eventos ignora clientes transacionais pois não tem acesso à transação que iniciou o envio (não tem como sinalizar um rollback, por exemplo).

## Aplicações

Aplicações que necessitam que várias mensagens façam parte de um escopo transacional. Exemplos:

- Resposta seguida de requisição (incluir o recebimento de uma resposta, seguida por uma requisição em uma transação);

- Uma sequência ou grupo de mensagens (todas as mensagens precisam ser enviadas, ou todas as mensagens precisam ser recebidas);
- Coordenação com workflow ou banco de dados (amarrar o envio ou recebimento de uma mensagem à execução de uma unidade de trabalho ou atualização de banco de dados).

## Cliente Transacional em Java / JMS

JMS oferece suporte programático a transações durante a criação da sessão. Se o primeiro argumento de `createSession()` for `true`, o envio ou recebimento de mensagens será transacional e só terá início quando a sessão for cometida. Em caso de falha, pode-se também chamar `rollback()`.

O exemplo abaixo recebe uma mensagem dentro de um contexto transacional e envia uma requisição na mesma transação. A mensagem não será consumida (removida do canal) enquanto a resposta não tiver sido enviada, portanto se o envio falhar, a mensagem será enviada para o DLQ.

```

301. public class SynchronizedMessageProducer implements MessageListener{
302.
303.     private Connection con;
304.     private Session session;
305.     private MessageProducer producer;
306.     private MessageConsumer consumer;
307.
308.     SynchronizedMessageProducer(Connection con, Destination inQueue)
309.                             throws JMSEException {
310.         // Transactional session
311.         session = con.createSession(true, Session.AUTO_ACKNOWLEDGE);
312.         consumer = session.createConsumer(inQueue);
313.         consumer.setMessageListener(this);
314.     }
315.
316.     @Override
317.     public void onMessage(Message request) {
318.         try {
319.             System.out.println("Received message: " + request);
320.             Destination outQueue = request.getJMSReplyTo();
321.             producer = session.createProducer(outQueue);
322.
323.             Message reply = session.createMessage();
324.             reply.setJMSCorrelationID(request.getJMSMessageID());
325.             reply.setStringProperty("Status", "OK");
326.
327.             System.out.println("Sending response" + reply);
328.             producer.send(reply);
329.
330.             System.out.println("Committing session");
331.             session.commit();
332.         } catch (Exception e) {

```

```

333.         System.out.println("Rolling back session!");
334.         try {
335.             session.rollback();
336.         } catch (JMSEException e1) {
337.             e1.printStackTrace();
338.         }
339.     }
340. }
341.
342. public static void main(String[] args) {
343.     Connection con = null;
344.     try {
345.         Context ctx = new InitialContext();
346.         ConnectionFactory factory = (ConnectionFactory) ctx
347.             .lookup("ConnectionFactory");
348.         con = factory.createConnection();
349.         Destination queue = (Destination) ctx.lookup("produtos");
350.
351.         new SynchronizedMessageProducer(con, queue);
352.         System.out.println("Willwait 60 seconds...");
353.         Thread.sleep(60000);
354.         System.out.println("Done.");
355.
356.     } catch (Exception e) {
357.         e.printStackTrace();
358.     } finally {
359.         if (con != null) {
360.             try {
361.                 con.close();
362.             } catch (JMSEException e) {
363.             }
364.         }
365.     }
366. }
367. }
```

Se for enviada uma mensagem sem cabeçalho JMSReplyTo para a fila “produtos”, não será possível enviar a resposta, a transação será desfeita e a mensagem recebida não será removida da fila (não será consumida).

As transações mostradas não são distribuídas. Para sincronizar um banco de dados com o sistema de mensageria é preciso usar transações XA. Isto é possível em Java usando JTA e a classe UserTransaction, que pode ser usada para demarcar um bloco de código contendo instruções de envio/recebimento JMS e inserções/atualizações/remoções no banco.

Em servidores de aplicação também pode-se usar demarcação declarativa e automática de transações em EJBs (Session e Message-Driven Beans). O ambiente do Spring também fornece um mecanismo de transações popular em aplicações Java.

## Cliente Transacional em Apache Camel

Camel recomenda o uso de transações declarativas do Spring para demarcação. No XML de configuração pode-se configurar um JmsTransactionManager para as conexões do broker usado e depois usá-lo para definir uma ou mais políticas de propagação transacional. As políticas transacionais podem ser selecionadas e usadas nas rotas. A configuração abaixo (da documentação Camel) mostra a definição de duas políticas (REQUIRED e REQUIRES\_NEW)

```
<bean id="jmsConnectionFactory"
      class="org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL" value="tcp://localhost:61616"/>
</bean>

<bean id="jmsTransactionManager"
      class="org.springframework.jms.connection.JmsTransactionManager">
    <property name="connectionFactory" ref="jmsConnectionFactory" />
</bean>

<bean id="PROPAGATION_REQUIRED"
      class="org.apache.camel.spring.spi.SpringTransactionPolicy">
    <property name="transactionManager" ref="jmsTransactionManager"/>
</bean>

<bean id="PROPAGATION_REQUIRES_NEW"
      class="org.apache.camel.spring.spi.SpringTransactionPolicy">
    <property name="transactionManager" ref="jmsTransactionManager"/>
    <property name="propagationBehaviorName"
value="PROPAGATION_REQUIRES_NEW"/>
</bean>

<bean id="PROPAGATION_NOT_SUPPORTED"
      class="org.apache.camel.spring.spi.SpringTransactionPolicy">
    <property name="transactionManager" ref="jmsTransactionManager"/>
    <property name="propagationBehaviorName"
value="PROPAGATION_NOT_SUPPORTED"/>
</bean>

...

```

Uma vez declaradas, elas podem ser instanciadas na configuração do RouteBuilder e usadas para aplicar regras de propagação transacional nas rotas:

```
368. public void configure() {
369.     ...
370.     Policy required = bean(SpringTransactionPolicy.class,
371.                               "PROPAGATION_REQUIRED"));
371.     Policy requirenew = bean(SpringTransactionPolicy.class,
372.                               "PROPAGATION_REQUIRES_NEW"));
372.
```

```

373.      from("activemq:queue:foo")
            .policy(requirenew)
            .to("activemq:queue:bar");
374.
375.      from("activemq:queue:foo")
            .policy(required )
            .to("activemq:queue:bar");
376.  }

```

## Cliente Transacional em Spring Integration

Em Spring, processos iniciados pelo usuário (chamada de métodos em Gateways, envio de uma mensagem a um Canal de Mensagens) podem ser configurados para serem incluídos em um contexto transacional usando anotações. Normalmente métodos de Gateways ou Ativadores de Serviços, ou outros que enviam ou recebem mensagens são anotados com `@Transactional` em aplicações Spring.

Mas o Spring Integration possui vários outros processos que são iniciados pelo sistema. Exemplo são os Pollers ou Schedulers, além dos Adaptadores de Canais que são disparados por eventos internos e externos. Eles podem ser configurados no próprio XML (exemplo da documentação):

```

<int:poller max-messages-per-poll="1" fixed-rate="1000">
    <transactional transaction-manager="txManager"
        isolation="DEFAULT"
        propagation="REQUIRED"
        read-only="true"
        timeout="1000"/>
</poller>

```

A sincronização de transações (ex: sincronizar um Adaptador de Canal com um banco de dados) pode ser realizada através de processadores de sincronização de transações, com call-backs que são chamados antes do commit, depois do commit e depois do rollback. A configuração também pode ser feita em XML (exemplo da documentação):

```

<int-file:inbound-channel-adapter id="inputDirPoller"
    channel="someChannel"
    directory="/foo/bar"
    filter="filter"
    comparator="testComparator">
    <int:poller fixed-rate="5000">
        <int:transactional transaction-manager="transactionManager"
            synchronization-factory="syncFactory" />
    </int:poller>
</int-file:inbound-channel-adapter>

<int:transaction-synchronization-factory id="syncFactory">
    <int:after-commit expression="payload.renameTo('/success/' + payload.name)"
        channel="committedChannel" />
    <int:after-rollback expression="payload.renameTo('/failed/' + payload.name)"
        channel="rolledBackChannel" />

```

```
</int:transaction-synchronization-factory>
```

## (51) Consumidor de sondagem (Polling Consumer)

### Ícone



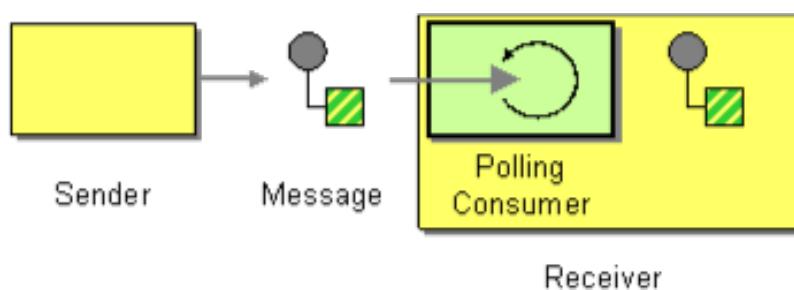
### Problema

“Como pode uma aplicação consumir uma mensagem, quando a aplicação estiver pronta?”

### Solução

“A aplicação deve usar um Consumidor de Sondagem (Polling Consumer), que faz uma chamada explícita quando deseja receber uma mensagem”

### Diagrama



### Descrição

A sondagem (polling) de um canal consiste em verificar periodicamente. Um Consumidor de Sondagem (Polling Consumer) conecta-se a um canal e espera um tempo até mensagem chegar ou atingir um timeout. Quando a mensagem chega, ele a consome, e pode então receber outras mensagens que ficam enfileiradas no canal até que ele as consuma.

Com um Consumidor de Sondagem não há risco de sobrecarregar o servidor, pois as mensagens são processadas no ritmo em que possam ser consumidas. Se o consumidor consome as mensagens mais rapidamente do que elas são produzidas, ele passará mais tempo esperando as mensagens chegarem e o ritmo será determinado pelo produtor. Se o produtor produzir mensagens mais rapidamente do que elas puderem ser consumidas, elas serão enfileiradas no canal e serão consumidas à medida em que o consumidor terminar de processar as mensagens anteriores. Desta maneira, o consumidor limita a taxa em que as mensagens são processadas.

Um Consumidor de Sondagem é um receptor síncrono, já que cada thread que consome mensagens bloqueia enquanto não chega uma mensagem no canal que está monitorando. Pode-se controlar a taxa de consumo de mensagens alterando o número de threads disponíveis, a taxa de sondagem e por quanto tempo cada thread permanece bloqueado.

## Aplicações

Um Consumidor de Sondagem é uma solução eficiente quando mensagens são consumidas regularmente e podem ser produzidas a um ritmo próximo daquele suportado pelo consumidor. Se o consumidor bloqueia threads por longos períodos ou realiza sondagens excessivas (ex: quando o fluxo de mensagens não é regular ou fraco), esta pode ser uma solução ineficiente. Em situações assim um Consumidor Ativado por Eventos pode ser uma solução melhor.

### Consumidor de Sondagem em Java / JMS

Em JMS o recebimento síncrono de mensagens pode ser realizado através do método `receive()` de `MessageConsumer`, que também pode receber um timeout em milissegundos. O `receive()` bloqueia o thread até que uma mensagem seja recebida. Em um Canal Ponto-a-Ponto a mensagem ficará disponível em um canal até que expire ou seja consumida. Pode-se implementar um Consumidor de Sondagem em JMS incluindo o `receive()` em um loop que sonde periodicamente o canal:

```

377. public class PollingMessageReceiver {
378.
379.     private Session session;
380.     private MessageConsumer consumer;
381.     private String name;
382.     private long delay;
383.
384.     public PollingMessageReceiver (Connection con, Destination queue,
385.                                     String name, long delay) throws
JMSEException {
386.         this.name = name;
387.         this.delay = delay;
388.         session = con.createSession(false, Session.AUTO_ACKNOWLEDGE);
389.         consumer = session.createConsumer(queue);
390.         consumer.start();
391.     }
392.
393.     public void receive() throws JMSEException {
394.         while(true) {
395.             TextMessage message = (TextMessage)consumer.receive();
396.             System.out.println(name + " received: " +
message.getText());
397.             try {
398.                 Thread.sleep(delay);
399.             } catch (InterruptedException e) {}
400.         }
401.     }
402.
403.     public void run(Executor thread) {
404.         thread.execute(new Runnable() {
405.             public void run() {
406.                 try {

```

```

407.                 receive();
408.             } catch (JMSEException e) {
409.                 e.printStackTrace();
410.             }
411.         }
412.     });
413. }
414.

415.     public static void main(String[] args) throws NamingException,
416. JMSEException {
417.     Context ctx = new InitialContext();
418.     ConnectionFactory factory =
419.             (ConnectionFactory)ctx.lookup("ConnectionFactory");
420.     Destination from = (Destination)ctx.lookup("inbound");
421.     Connection con = factory.createConnection();
422.     Executor thread = Executors.newFixedThreadPool(2);
423.     System.out.println("Waiting for messages... (^C to cancel)");
424.
425.     PollingMessageReceiver receiver1 =
426.         new PollingMessageReceiver (con, from, "Receiver 1", 500);
427.     receiver1.run(thread);
428.
429.     PollingMessageReceiver receiver2 =
430.         new PollingMessageReceiver (con, from, "Receiver 2", 1000);
431.     receiver2.run(thread);
432.   }
433. }
```

## Consumidor de Sondagem em Apache Camel

Na configuração de rotas Camel usa Consumidores Ativados por Eventos por default, que podem ser configurados em Processors. Consumidores de Sondagem podem ser criados usando a interface PollingConsumer que possui métodos receive() similares às existentes no MessageConsumer do JMS. O exemplo abaixo (da documentação Camel) ilustra o uso de um Consumidor de Sondagem em Camel:

```

Endpoint endpoint = context.getEndpoint("activemq:my.queue");
PollingConsumer consumer = endpoint.createPollingConsumer();
Exchange exchange = consumer.receive();
```

## Consumidor de Sondagem em Spring Integration

Spring permite configurar Pollers para várias situações (um uso típico é para determinar o fluxo de mensagens lidas em um Adaptador de Canal). Um PollingConsumer deve definir um trigger e pode definir outras propriedades:

```

PollableChannel channel = context.getBean("pollableChannel",
PollableChannel.class);
PollingConsumer consumer = new PollingConsumer(channel, handler);
```

```
IntervalTrigger trigger = new IntervalTrigger(30, TimeUnit.SECONDS)
trigger.setInitialDelay(5);
trigger.setFixedRate(true);
consumer.setTrigger(trigger);
consumer.setMaxMessagesPerPoll(10);
consumer.setReceiveTimeout(5000);
```

Pollers também podem ser configurados em XML:

```
<int:poller id="defaultPoller"
    default="true"
    max-messages-per-poll="5"
    fixed-rate="3000"/>
```

## (50) Consumidor ativado por eventos (Event-Driven Consumer)

### Ícone



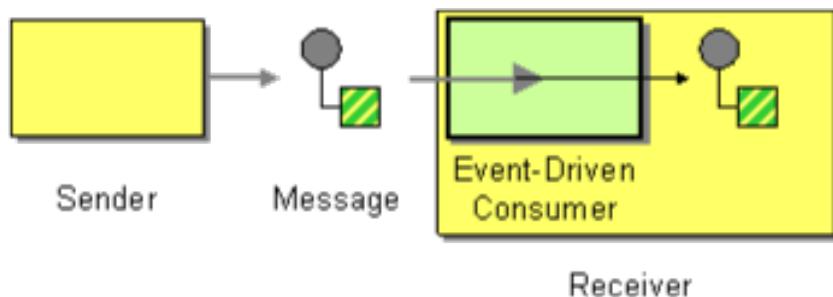
### Problema

“Como pode uma aplicação automaticamente consumir mensagens à medida em que se tornam disponíveis?”

### Solução

“A aplicação deve usar um Consumidor Ativado por Eventos (Event-Driven Consumer), que receba mensagens automaticamente à medida em que são entregues no canal”

### Diagrama



### Descrição

Um Consumidor Ativado por Eventos (Event-Driven Consumer) é um consumidor assíncrono. Ele não sonda canais esperando por mensagens. Quando uma mensagem chega em um canal ele é notificado por um evento.

Este tipo de consumidor é um Observer (padrão GoF) e precisa registrar-se como observador (listener) de um canal. Quando uma mensagem é recebida pelo canal, o canal notifica todos os interessados. A notificação pode conter a própria mensagem ou um token que o consumidor pode usar para recuperar a mensagem.

## Aplicações

Um Consumidor Ativado por Eventos é a solução ideal quando a produção de mensagens é irregular ou pouco frequente e as mensagens devem ser consumidas assim que forem entregues. Se o fluxo de mensagens for muito grande existe o risco de sobrecarregar o servidor. Neste caso um Consumidor de Sondagem poderá ser uma solução melhor.

### Consumidor Ativado por Eventos em Java / JMS

Um Consumidor Ativado por Evento pode ser configurado em JMS usando um MessageListener. O MessageListener precisa implementar o método onMessage() que irá receber a mensagem como um evento. Antes de iniciar o recebimento das mensagens, o MessageListener deve ser registrado no consumidor (associado a um Canal Ponto-a-Ponto ou de Difusão) e o recebimento de mensagens iniciado chamando o método start() de Connection (se o método for chamado antes de registrar o listener, mensagens podem se perder).

```

431. public class EventDrivenMessageReceiver implements MessageListener {
432.
433.     private Session session;
434.     private MessageConsumer consumer;
435.     private String name;
436.
437.     public EventDrivenMessageReceiver(Connection con, Destination queue,
438.                                         String name) throws JMSEException {
439.         this.name = name;
440.         session = con.createSession(false, Session.AUTO_ACKNOWLEDGE);
441.         consumer = session.createConsumer(queue);
442.         consumer.setMessageListener(this);
443.         con.start();
444.     }
445.
446.     @Override
447.     public void onMessage(Message msg) {
448.         try {
449.             TextMessage message = (TextMessage) msg;
450.             System.out.println(name + " received: " + message.getText());
451.         } catch (JMSEException e) {
452.             e.printStackTrace();
453.         }
454.     }
455.
456.     public static void main(String[] args) throws Exception {
457.         Context ctx = new InitialContext();

```

```

458.     ConnectionFactory factory = (ConnectionFactory) ctx
459.             .lookup("ConnectionFactory");
460.     Destination from = (Destination) ctx.lookup("inbound");
461.     Connection con = factory.createConnection();
462.
463.     System.out.println("Waiting 60 seconds for messages... (^C to
cancel)");
464.
465.     new EventDrivenMessageReceiver(con, from, "Receiver 1");
466.     new EventDrivenMessageReceiver(con, from, "Receiver 2");
467.
468.     Thread.sleep(60000);
469.
470. }
471. }
```

## Consumidor Ativado por Eventos em Apache Camel

Um Consumidor Ativado por Eventos é implementado por qualquer consumidor que implemente a interface Processor em uma rota.

```

from("jms:queue:inicio")
    .process(new Processor() {
        public void process(Exchange exchange) throws Exception {
            System.out.println(exchange.getIn().getBody(String.class));
        }
    });
}
```

Cada mensagem que passa pela rota dispara o método process().

## Consumidor Ativado por Eventos em Spring Integration

O SubscribableChannel de Spring Integration permite criar canais assináveis (exemplos da documentação do Spring Integration):

```

public interface SubscribableChannel extends MessageChannel {
    boolean subscribe(MessageHandler handler);
    boolean unsubscribe(MessageHandler handler);
}
```

A interface MessageHandler é similar a um MessageListener do JMS, que precisa implementar um método handleMessage(), que é similar ao onMessage() do JMS:

```
void handleMessage(Message<?> message) throws MessagingException
```

Um EventDrivenConsumer pode ser configurado passando um SubscribableChannel (registrado no XML do Spring) e um MessageHandler:

```

subscribableChannel.subscribe(messageHandler);
SubscribableChannel channel =
    context.getBean("subscribableChannel", SubscribableChannel.class);
```

```
EventDrivenConsumer consumer = new EventDrivenConsumer(channel, exampleHandler);
```

## (52) Consumidores concorrentes (Competing Consumers)

### Ícone



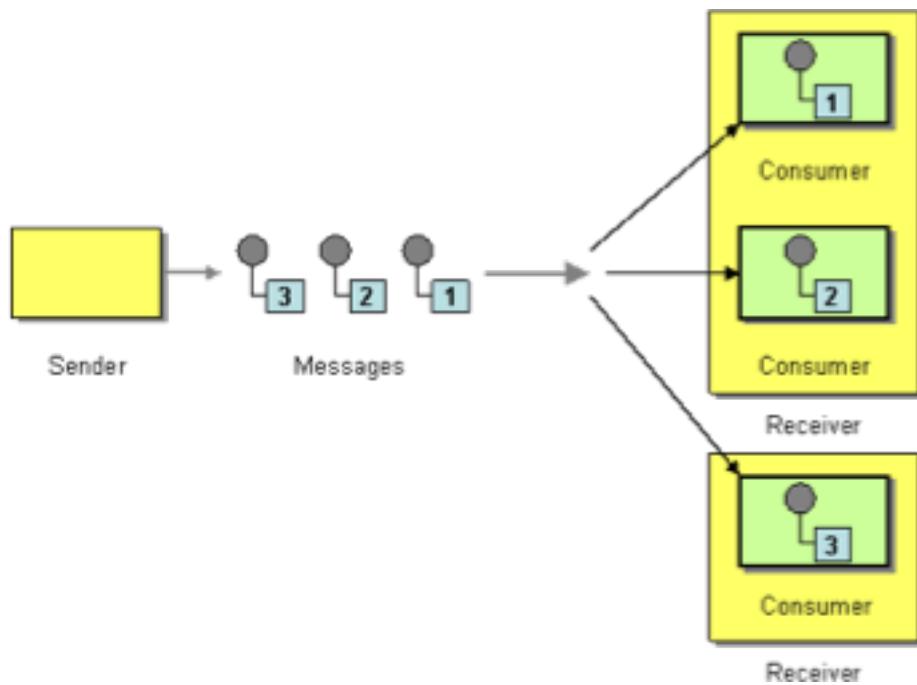
### Problema

“Como pode um cliente de mensageria processar múltiplas mensagens ao mesmo tempo?”

### Solução

“Crie múltiplos Consumidores Concorrentes (Competing Customers) em um único canal para que os consumidores possam processar múltiplas mensagens de forma concorrente”

### Diagrama



### Descrição

Consumidores Concorrentes (Competing Customers) são consumidores que recebem mensagens de um único Canal Ponto-a-Ponto. Quando uma mensagem é recebida pelo canal, qualquer um dos consumidores pode consumi-la. Cada cliente executa em um thread separado. É mais simples de implementar (e provavelmente mais eficiente) se cada consumidor for um Consumidor de Sondagem. Nesse caso, o número de Consumidores Concorrentes pode ser ajustado para a melhor eficiência.

A eficiência desta solução depende do algoritmo usado pelo sistema de mensageria para garantir o consumo da mensagem por um único consumidor. Um sistema pode verificar qual cliente chegou primeiro e internamente despachar a mensagem somente para ele, impedindo que os outros tentem consumir a mesma mensagem. Sistemas mais simples podem deixar o conflito acontecer (dois consumidores que pensam que estão consumindo a mesma mensagem) e deixar que o primeiro commit defina o vencedor (os outros terão que fazer rollback). Esse tipo de sistema pode tornar um Transactional Client muito ineficiente. Para não ficar dependente da implementação do sistema de mensageria, uma solução é usar um Despachante de Mensagens (Message Dispatcher) em substituição ou em conjunto com os Consumidores Concorrentes.

## Aplicações

Consumidores Concorrentes pode ser usado para implementar uma solução escalável de Consumidores de Sondagem, distribuindo o trabalho de consumir mensagens de uma fila por vários consumidores operando em paralelo.

### Consumidores Concorrentes em Java / JMS

Para implementar Consumidores Concorrentes em JMS configure dois ou mais consumidores para consumir do mesmo Canal Ponto-a-Ponto. Os dois consumidores do exemplo mostrado na seção sobre Consumidor de Sondagem são também Consumidores Concorrentes. As mensagens enviadas para o canal “inbound” serão disputadas entre eles:

```

472. public static void main(String[] args) throws NamingException,
JMSEException {
473.     Context ctx = new InitialContext();
474.     ConnectionFactory factory =
        (ConnectionFactory)ctx.lookup("ConnectionFactory");
475.     Destination from = (Destination)ctx.lookup("inbound");
476.     Connection con = factory.createConnection();
477.
478.     Executor thread = Executors.newFixedThreadPool(2);
479.     System.out.println("Waiting for messages... (^C to cancel)");
480.
481.     PollingMessageReceiver receiver1 =
        new PollingMessageReceiver(con, from, "Receiver 1", 500);
482.     receiver1.run(thread);
483.
484.     PollingMessageReceiver receiver2 =
        new PollingMessageReceiver(con, from, "Receiver 2", 1000);
485.     receiver2.run(thread);
486. }
```

### Consumidores Concorrentes em Apache Camel

Para configurar consumidores concorrentes para filas JMS em diferentes threads em Camel, use a opção concurrentConsumers:

```
from("jms:MyQueue?concurrentConsumers=5").bean(SomeBean.class);
```

Ou crie rotas em JVMs ou threads diferentes disputando os mesmos canais P2P.

## (53) Despachante de mensagens (Message Dispatcher)

ícone



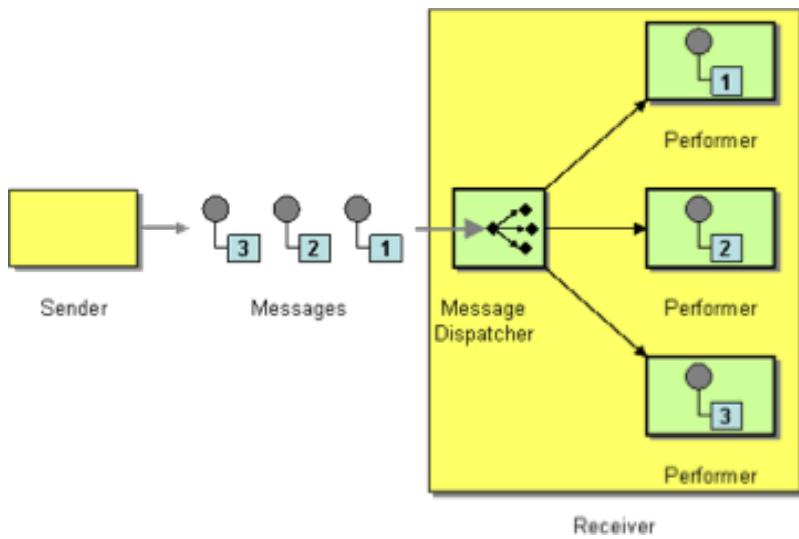
## Problema

“Como podem múltiplos consumidores em um único canal coordenarem o seu processamento de mensagens?”

## Solução

“Crie um Despachante de Mensagens (Message Dispatcher) em um canal que irá consumir mensagens de um canal e distribuí-las para diferentes realizadores”

## Diagrama



## Descrição

Um Despachante de Mensagens (Message Dispatcher) permite distribuir o consumo de mensagens a vários componentes que irão processá-las em paralelo, controlando qual componente irá receber cada mensagem recebida.

Já é possível distribuir o processamento de mensagens entre vários consumidores paralelos usando Consumidores Concorrentes, mas não é possível determinar qual consumidor irá consumir determinada mensagem. Se todos os consumidores forem iguais não faz diferença uma solução ou outra, mas um Despachante de Mensagens será necessário se houver consumidores especializados em consumir determinado tipo de mensagem.

Outros padrões podem ser usados para solucionar o mesmo problema. Consumidores Seletivos em um Canal Publica-Inscreve filtram as mensagens indesejadas e consomem apenas uma seleção. Um Roteador Baseado em Conteúdo redirecionando para Canais de Tipo de Dados também pode é uma solução possível. Mas existem situações em que essas soluções podem não ser as melhores. Canais de Tipos de Dados não são uma boa solução se houverem muitos tipos diferentes de mensagens, ou se os tipos forem determinados com base em critérios dinâmicos. Consumidores Seletivos selecionam mensagens com base em propriedades limitadas (geralmente apenas dados contidos no cabeçalho) e é preciso garantir que as expressões distribuam as mensagens de forma mutuamente exclusiva.

Um Despachante de Mensagens possui duas partes: o despachante propriamente dito, que consome todas as mensagens do canal, e o realizador (performer), que recebe mensagens repassadas pelo despachante para processamento. Um realizador pode ser criado na hora ou reutilizado a partir de um pool de objetos. Cada realizador executa seu próprio thread garantindo o processamento concorrente.

Uma solução equivalente a Consumidores Concorrentes, mas que não depende do mecanismo usado para despachar mensagens usada pelo sistema de mensageria, pode ser obtido com um Despachante de Mensagens repassando para uma coleção de realizadores equivalentes. Como existe apenas um consumidor, a solução pode ser usada tanto com um Canal Ponto-a-Ponto como com um Canal Publica-Inscreve, e o Consumidor pode ser de Sondagem ou Guiado por Eventos. Os realizadores podem ser implementados como Consumidores Ativados por Eventos.

## Aplicações

Um Despachante de Mensagens pode ser usado para distribuir o trabalho de consumir mensagens de uma fila por vários consumidores operando em paralelo. Difere do padrão Consumidor Concorrente porque mantém o controle sobre como as mensagens são consumidas e processadas.

### Despachante de Mensagens em Java / JMS

A implementação de um Despachante de Mensagens é muito semelhante à implementação de um Content-Based Router, com a diferença de que as mensagens não são roteadas a diferentes canais, mas despachadas a diferentes processadores que operam dentro do contexto do Terminal. Os processadores geralmente operam em paralelo, portanto cada despacho inicia um novo thread.

O exemplo abaixo mostra um Terminal de Mensageria que despacha as mensagens recebidas de um Canal Ponto-a-Ponto a diferentes processadores de acordo com seu tipo:

```
487. public class ExampleMessageDispatcher implements MessageListener {  
488.     private Session session;  
489.     private MessageConsumer consumer;  
490.     private MessageProducer invalidChannelProducer;  
491.  
492.     public ExampleMessageDispatcher(Connection con, Destination queue,  
493.                                         Destination invalidChannel) throws JMSEException {  
494.         session = con.createSession(false, Session.AUTO_ACKNOWLEDGE);  
495.         consumer = session.createConsumer(queue);  
496.         invalidChannelProducer = session.createProducer(invalidChannel);
```

```
497.         consumer.setMessageListener(this);
498.         con.start();
499.     }
500.
501.     @Override
502.     public void onMessage(Message msg) {
503.         try {
504.             Message message = (Message) msg;
505.             String type      = message.getStringProperty("Type");
506.             String filename = message.getStringProperty("Filename");
507.             System.out.println("Received message with file " + filename);
508.
509.             MessageProcessor processor;
510.             Executor thread = Executors.newFixedThreadPool(3);
511.
512.             if (type != null && type.equals("png")) {
513.                 processor = new ImageProcessor(message);
514.             } else if (type != null && type.equals("txt")) {
515.                 processor = new TextProcessor(message);
516.             } else if (type != null && type.equals("xml")) {
517.                 processor = new XMLProcessor(message);
518.             } else {
519.                 processor = null;
520.             }
521.
522.             if(processor == null) { // cannot process - send to invalid
channel
523.                 invalidChannelProducer.send(message);
524.             } else {
525.                 processor.run(thread);
526.             }
527.
528.         } catch (JMSException e) {
529.             e.printStackTrace();
530.         }
531.     }
532.
533.     public static void main(String[] args) throws Exception {
534.         Context ctx = new InitialContext();
535.         ConnectionFactory factory = (ConnectionFactory) ctx
536.             .lookup("ConnectionFactory");
537.         Destination from      = (Destination) ctx.lookup("inbound");
538.         Destination invalidTo = (Destination) ctx.lookup("invalid");
539.         Connection con = factory.createConnection();
540.
541.         System.out.println("Waiting for 60 seconds... (^C to cancel)");
542.
543.         new ExampleMessageDispatcher(con, from, invalidTo);
544.
```

```

545.         Thread.sleep(60000);
546.     }
547. }
```

## Despachante de Mensagens em Apache Camel

Para implementar um Despachante de Mensagens em Camel pode-se inserir um processador no meio da rota que inclua uma lógica de tomada de decisão para determinar para onde vai cada mensagem. Pode-se usar um Processor para delegar o processamento de cada mensagem despachada em processadores-filho (que rodam seu próprio thread).

O exemplo abaixo (da documentação do Camel) mostra como implementar um despachante com choice() e Processadores dedicados para mensagens recebidas por um endpoint que permite Consumidores Concorrentes. Apesar de usar choice(), os blocos when() não direcionam para diferentes canais mas diretamente a diferentes processadores. Apenas mensagens inválidas são roteadas (para um Invalid Message Channel):

```

from("jms:queue:myDestination?concurrentConsumers=5")
    .thread(5)
    .choice()
        .when(header("type").isEqualTo("A"))
            .process(myProcessorForTypeA)
        .when(header("type").isEqualTo("B"))
            .process(myProcessorForTypeB)
        .when(header("type").isEqualTo("C"))
            .process(myProcessorForTypeC)
        .otherwise()
            .to("jms:queue:myInvalidMessageQueue");
```

## Despachante de Mensagens em Spring Integration

Em Spring Integration o elemento <dispatcher> pode ser usado para que as mensagens de um canal sejam despachadas para os processadores através da execução de um thread. Essa configuração é chamada de Executor Channel.

```

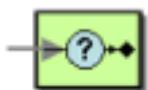
<int:channel id="inbound">
    <int:dispatcher task-executor="thread"/>
</int:channel>

<task:executor id="thread"
    pool-size="3"
    queue-capacity="3"
    rejection-policy="CALLER_RUNS" />

<int:service-activator input-channel="inbound" ref="processor"
method="process"/>
<bean id="processor" class="br.com...ExampleProcessor"/>
```

## (54) Consumidor seletivo (Selective Consumer)

### Ícone



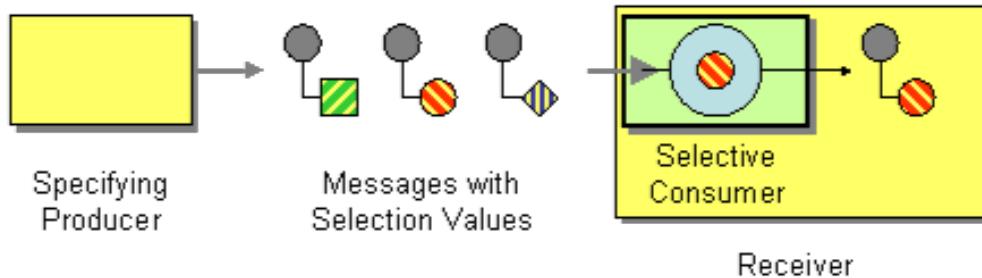
### Problema

“Como pode um consumidor selecionar as mensagens que deseja receber”

### Solução

“Implemente o consumidor como um Consumidor Seletivo (Selective Consumer), que filtra as mensagens entregues pelo canal para que só receba as mensagens que estejam de acordo com seu critério de filtro”

### Diagrama



### Descrição

Um Consumidor Seletivo (Selective Customer) utiliza uma expressão booleana para decidir se consome ou não uma mensagem recebida. A expressão utiliza dados que devem estar presentes no cabeçalho da mensagem (o produtor deve incluir essas propriedades antes do envio). Se o resultado for verdadeiro, a mensagem será consumida. Caso contrário, será descartada.

Se houver vários Consumidores Seletivos conectados a um canal, deve-se usar um Canal Publica-Inscreve para que todos os consumidores recebam todas as mensagens. Se as expressões combinadas de todos os consumidores não forem mutuamente exclusivas, uma mesma mensagem poderá ser consumida por mais de um consumidor. Se as expressões combinadas não abrangerem todas as combinações possíveis de propriedades de mensagens, existe o risco de mensagens não serem consumidas por nenhum consumidor. As mensagens devem ter um Prazo de Validade para que sejam removidas do canal caso não sejam selecionadas em determinado prazo.

Um Filtro de Mensagens pode ser usado como solução para o mesmo problema. Neste caso as mensagens sequer são entregues se não passam pelo filtro. Um Consumidor Seletivo recebe a mensagem, mas joga fora se ela não passar pelo filtro. Outras soluções mais ou menos equivalentes são Roteadores Baseados em Conteúdo e Canais de Tipo-de-Dados, que podem ser mais eficientes se

houverem poucas opções de filtragem. Caso contrário, um Consumidor Seletivo pode ser uma solução mais dinâmica e flexível.

## Aplicações

Um Consumidor Seletivo deve ser usado quando for necessário selecionar, dentre as mensagens recebidas, as que aderem a determinado critério baseado em suas propriedades, e descartar as demais.

### Consumidor Seletivo em Java / JMS

Um Consumidor Seletivo pode ser implementado em JMS usando seletores JMS. Um consumidor pode configurar um seletor declarando uma expressão baseada em um subconjunto da linguagem SQL92 (cláusula WHERE) para testar propriedades e cabeçalhos padrão, mas não tem acesso ao corpo da mensagem. O seletor é um String que deve ser passado como segundo argumento do método `createConsumer()` (ou `createDurableSubscriber()`), por exemplo:

```
Message consumer consumer =
    session.createConsumer(queue, "Tipo = 'xml' AND Length < 1000");
```

Os elementos que podem ser usados nos seletores são:

- Cabeçalhos padrão (propriedades que começam com JMS) exceto `JMSDestination`, `JMSExpiration` e `JMSReplyTo`. Exemplo: `JMSCorrelationID = 'A123'`
- Propriedades. Exemplo: `Type = 'png'`

Os tipos suportados são:

- Literais de string entre apóstrofes. Exemplo: `Type = 'xml'`
- Literais numéricos: Exemplo: `Length < 1000`
- Literais booleanos: `IsFragment = FALSE`

Os operadores suportados são:

- Parênteses ()
- Operadores lógicos: `AND`, `OR`, `NOT`
- Operadores aritméticos: `*`, `+`, `-`, `/`
- Operador `LIKE`. Exemplo: `Type LIKE 'jp%'`
- Operadores `IN`, `BETWEEN`, `IS NULL`, `IS NOT NULL`

Se um consumidor consome uma mensagem de um Topic ela será removida do canal passando ou não no seletor. Se o canal for um Queue, a mensagem não será consumida e permanecerá no canal.

O trecho abaixo (`EventDrivenSelectiveReceiver.java`) mostra a configuração de um consumidor que só consome mensagens que tenham um cabeçalho “Type” com o valor “xml” e cujo tamanho não exceda 1000 bytes.

```
548. public EventDrivenSelectiveReceiver(Connection con, Destination queue,
549.           String name) throws JMSEException {
550.     this.name = name;
551.     session = con.createSession(false, Session.AUTO_ACKNOWLEDGE);
552.     consumer = session.createConsumer(queue, "Type = 'xml' AND Length <
553.           1000");
554.     consumer.setMessageListener(this);
555.     con.start();
556. }
```

## Consumidor Seletivo em Apache Camel

Este padrão não é explicitamente implementado em Camel. Depende de suporte do provedor de mensageria usado. Se o provedor for JMS, pode-se usar os mesmos seletores JMS mostrados no exemplo anterior. O seletor pode ser incluído diretamente na URI do endpoint, mas precisa ser codificado de acordo com a sintaxe de URIs. Portanto, uma versão Camel do exemplo anterior seria:

```
from("jms:queue:myDestination?selector=Type%3D%27xml%27%20AND%20Length%3C1000")
    .process(myProcessor); // seletor: Type='xml'
```

## Consumidor Seletivo em Spring Integration

Em Spring Integration usar um `<filter>` dentro de um Message Handler Chain (`<chain>`) filtra as mensagens que serão passadas ao componente seguinte. Como os componentes do `<chain>` não têm canais entre eles (não é Pipes and Filters), o filtro é o seletor do componente seguinte, que é o Consumidor Seletivo. O seletor pode ser implementado como expressão (SpEL) ou por um bean que implemente a interface `MessageSelector`:

```
556. public ExampleSelector MessageSelector {
557.     @Override
558.     boolean accept(Message<?> message) {
559.         if(message.getHeaders().get("Type").equals("xml")) {
560.             return true;
561.         }
562.         return false;
563.     }
564. }
```

No exemplo abaixo (da documentação Spring) o componente `<header-enricher>` é um Consumidor Seletivo e recebe apenas as mensagens que passam pelo filtro “selector”

```
<chain input-channel="input" output-channel="output">
    <filter ref="selector" throw-exception-on-rejection="true"/>
    <header-enricher error-channel="customErrorChannel">
        <header name="foo" value="bar"/>
```

```

</header-enricher>
<service-activator ref="someService" method="someMethod"/>
</chain>
<bean id="selector" class="br.com...ExampleSelector"/>

```

## (55) Assinante durável (Durable Subscriber)

### Ícone



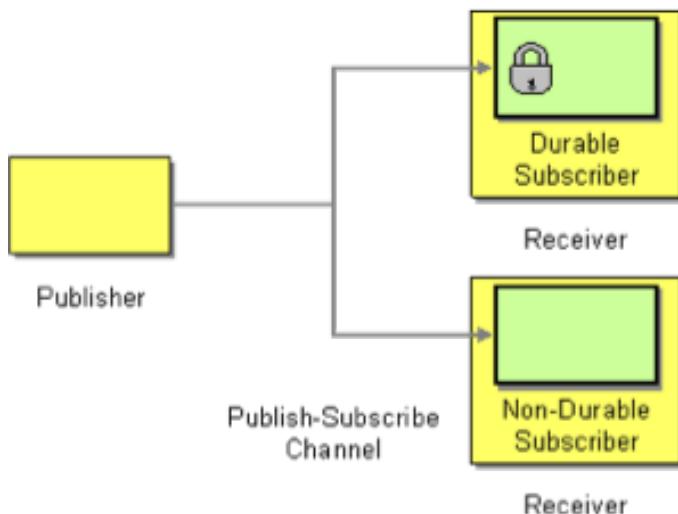
### Problema

“Como evitar que um assinante perca mensagens quando não estiver ativo?”

### Solução

“Use um Assinante Durável (Durable Subscriber) para fazer com que o sistema de mensageria guarde mensagens publicadas quando o assinante estiver desconectado”

### Diagrama



### Descrição

Em relação a um Canal Publica-Inscorre um consumidor pode estar inscrito ou não-inscrito. Se for um consumidor comum, inscrito é o mesmo que conectado, e não-inscrito é o mesmo que desconectado. O consumidor não receberá mensagens se estiver desconectado. Para um consumidor que é um Assinante Durável (Durable Subscriber) há três estados possíveis: inscrito e ativo (conectado), não-inscrito (desconectado) ou inscrito e inativo (desconectado). Uma vez inscrito em um canal, o Assinante Durável precisa explicitamente cancelar a inscrição para não receber mais mensagens, pois mensagens enviadas quando estiver desconectado (no estado inscrito e inativo) serão guardadas até

que ele torne-se ativo novamente. Desta forma, um Assinante Durável não perde mensagens enviadas quando está ausente.

Pode ser que o Assinante Durável não volte nunca mais, ou que demore muito tempo para voltar. Nesse caso é preciso que se controle o número de mensagens que podem ser guardadas, estabelecer um Prazo de Validade para as mensagens, para que elas não se acumulem, ou estabelecer um timeout dentro do qual o Assinante teria sua inscrição terminada automaticamente.

## Aplicações

Um Assinante Durável deve ser usado para consumidores-assinantes de um Canal Publica-Inscreve que não podem perder mensagens enviadas. Se estiverem inativos no momento do recebimento elas devem ser guardadas e enviadas novamente quando ele reconectar.

## Assinante Durável em Java / JMS

Um Assinante Durável é implementado em JMS durante a configuração de um Consumidor do tipo TopicSubscriber. Os consumidores são identificados por um ID que é usado para encerrar a assinatura (chamando o método unsubscribe()) quando ele não quiser mais receber mensagens. Em JMS 1.1 apenas um cliente ativo pode receber mensagens ao mesmo tempo. Em JMS 2.0 há métodos adicionais que permitem a criação de assinantes duráveis que compartilham canais.

No exemplo abaixo foram criados três consumidores que executarão com intervalos de 10 segundos. Nesse intervalo serão enviadas 10 mensagens, com intervalos de 2 segundos entre cada uma. É necessário definir um ID para a conexão (setClientID) e para cada consumidor durável.

```
565. public class DurableSubscriberExample {
566.     public static void main(String[] args) throws Exception {
567.         Context ctx = new InitialContext();
568.         ConnectionFactory factory = (ConnectionFactory) ctx
569.             .lookup("ConnectionFactory");
570.         Topic topic = (Topic) ctx.lookup("durable-topic");
571.         Connection con = factory.createConnection();
572.         String clientID = "Sub1";
573.         con.setClientID(clientID);
574.         Session session = con.createSession(false,
Session.AUTO_ACKNOWLEDGE);
```

O produtor abaixo criará 15 mensagens que serão enviadas para o Topic “durable-topic” de onde os consumidores irão receber mensagens:

```
575.         MessageProducer producer = session.createProducer(topic);
576.         ExecutorService executorService =
Executors.newSingleThreadExecutor();
577.         executorService.execute(new Runnable() {
578.             int mcount = 0;
579.
580.             public void run() {
581.                 try {
```

```

582.             while (mcount < 15) {
583.                 TextMessage m = session.createTextMessage("Message
"
584.                             + ++mcount);
585.                 producer.send(m);
586.                 Thread.sleep(2000);
587.             }
588.             System.out.println("Done sending messages!");
589.         } catch (Exception e) {
590.             e.printStackTrace();
591.         }
592.     }
593. });

```

O primeiro consumidor contará três mensagens recebidas antes de fechar. Enquanto estiver fechado estará inativo mas como é um Consumidor Durável espera não perder mensagens que forem enviadas durante sua inatividade:

```

594.     MessageConsumer consumer1 = session.createDurableSubscriber(topic,
595.                         clientID);
596.     consumer1.setMessageListener(new MessageListener() {
597.         int messageCount = 0;
598.
599.         @Override
600.         public void onMessage(Message msg) {
601.             try {
602.                 TextMessage message = (TextMessage) msg;
603.                 messageCount++;
604.                 System.out.println("Consumer1 (Durable) received: "
605.                                 + message.getText());
606.
607.                 if (messageCount == 3) { // stop after 3 messages
608.                     consumer1.close();
609.                     messageCount++;
610.                 }
611.
612.             } catch (JMSEException e) {
613.                 e.printStackTrace();
614.             }
615.         }
616.     });
617.
618.     con.start();
619.     Thread.sleep(10000); // wait 10 seconds for messages
620.     con.stop();

```

O próximo consumidor é um assinante comum não-durável. Ele irá receber mensagens do mesmo Topic. Como é não-durável só irá receber mensagens enquanto estiver funcionando.

```

621.         MessageConsumer consumer2 = session.createConsumer(topic);
622.         consumer2.setMessageListener(new MessageListener() {
623.             @Override
624.             public void onMessage(Message msg) {
625.                 try {
626.                     TextMessage message = (TextMessage) msg;
627.                     System.out.println("Consumer2 (Non-Durable) received:
"
628.                         + message.getText());
629.
630.                 } catch (JMSEException e) {
631.                     e.printStackTrace();
632.                 }
633.             }
634.         });
635.
636.         con.start();
637.         Thread.sleep(10000); // wait 10 seconds for messages
638.         con.stop();

```

O terceiro consumidor é durável e tem o mesmo ID do primeiro. Assim que ele entrar no ar ele deve receber não apenas as mensagens que continuam sendo enviadas (se houver) mas também as mensagens que foram enviadas no período em que esteve inativo.

```

639.         MessageConsumer consumer3 = session.createDurableSubscriber(topic,
640.                         clientID);
641.         consumer3.setMessageListener(new MessageListener() {
642.             @Override
643.             public void onMessage(Message msg) {
644.                 try {
645.                     TextMessage message = (TextMessage) msg;
646.                     System.out.println("Consumer3 (Durable) received: "
647.                         + message.getText());
648.
649.                 } catch (JMSEException e) {
650.                     e.printStackTrace();
651.                 }
652.             }
653.         });
654.         con.start();
655.
656.         executorService.shutdown();
657.
658.         Thread.sleep(10000); // wait 10 seconds for messages
659.         con.stop();
660.         con.close();
661.
662.     }
663. }

```

Este é o resultado da execução. Observe que o Consumer 3 (que tem o mesmo ID que o Consumer 1) recebe todas as mensagens que não foram recebidas pelo Consumer 1, além das mensagens que continuam chegando:

```
Consumer1 (Durable) received: Message 1
Consumer1 (Durable) received: Message 2
Consumer1 (Durable) received: Message 3
Consumer2 (Non-Durable) received: Message 6
Consumer2 (Non-Durable) received: Message 7
Consumer2 (Non-Durable) received: Message 8
Consumer2 (Non-Durable) received: Message 9
Consumer2 (Non-Durable) received: Message 10
Consumer3 (Durable) received: Message 4
Consumer3 (Durable) received: Message 5
Consumer3 (Durable) received: Message 6
Consumer3 (Durable) received: Message 7
Consumer3 (Durable) received: Message 8
Consumer3 (Durable) received: Message 9
Consumer3 (Durable) received: Message 10
Consumer3 (Durable) received: Message 11
Consumer2 (Non-Durable) received: Message 11
Consumer3 (Durable) received: Message 12
Consumer2 (Non-Durable) received: Message 12
Consumer3 (Durable) received: Message 13
Consumer2 (Non-Durable) received: Message 13
Consumer3 (Durable) received: Message 14
Consumer2 (Non-Durable) received: Message 14
Consumer3 (Durable) received: Message 15
Consumer2 (Non-Durable) received: Message 15
Done sending messages!
```

## Assinante Durável em Apache Camel e Spring Integration

O suporte de Camel e Spring Integration a assinantes duráveis depende do sistema de mensageria usado. Não há suporte nativo. Usando JMS como provedor, pode-se configurar uma assinatura durável em Camel através de opções do Endpoint:

```
from("jms:topic:durtest?clientID=testID&durableSubscriptionName=testSubs");
```

## (56) Receptor idempotente (Idempotent Receiver)

### Ícone



## Problema

“Como pode um receptor de mensagens lidar com mensagens duplicadas?”

## Solução

“Projete o receptor para que seja um Receptor Idempotente (Idempotent Receiver), que possa receber a mesma mensagem múltiplas vezes com segurança”

## Descrição

Um Receptor Idempotente (Idempotent Receiver) pode receber mensagens duplicadas sem que isto produza resultados inconsistentes ou indesejados.

Mensagens podem ser duplicadas em canais com implementações de Entrega Garantida, se o produtor não receber um aviso de recebimento. Também podem ser duplicadas se o próprio sistema de mensageria deixar de enviar um *acknowledgement* após o envio de uma mensagem. A remoção de mensagens duplicadas tem impacto na performance, portanto às vezes compensa permitir duplicatas e construir receptores que sejam imunes a elas.

Há duas maneiras de construir um Receptor Idempotente: tratar de eliminar mensagens duplicadas no receptor, ou definir a semântica da mensagem de forma a garantir a idempotência.

Para remover ou ignorar duplicatas, é preciso que o receptor mantenha um controle sobre quais mensagens já foram recebidas. Normalmente isto é feito conferindo o Identificador da Mensagem – uma propriedade que normalmente já é incluída nas mensagens pelo sistema de mensageria – e comparando-o com uma lista de identificadores de mensagens já recebidas.

Às vezes é possível rescrever uma mensagem de forma a tornar sua semântica idempotente. Por exemplo, uma instrução para adicionar ou inserir um bloco de texto em um arquivo não é idempotente, mas uma instrução para substituir um texto é idempotente pois depois de receber uma ou muitas mensagens o efeito é o mesmo. A desvantagem é que a mensagem será maior.

## Aplicações

Se mensagens duplicadas podem ocorrer em um sistema é preciso usar Receptores Idempotentes para lidar com os efeitos colaterais da duplicação.

## Receptor Idempotente em Java / JMS

A maior parte dos componentes são idempotentes por não manterem estado. Os agregadores e resequenciadores precisam manter estado e, dependendo de como utilizam as mensagens recebidas, podem ou não ser idempotentes. Agregadores que reconstruem mensagens recebidas devem ser idempotentes. Os Agregadores mostrados nos exemplos neste livro usavam um Set para reter mensagens, eliminando a possibilidade de duplicatas. Uma estratégia similar também pode ser usada nos Terminais guardando um Set de MessageIDs das mensagens que são recebidas. Isso poderia ser implementado através de um filtro de mensagens duplicadas que, conectado a um processador seria uma implementação de Receptor Idempotente.

O exemplo abaixo é um Filtro de Mensagens que elimina duplicatas. Ele pode ser usado como um componente na arquitetura Dutos e Filtros, mas também poderia ser adaptado como entrada de um Terminal, transformando-o em Receptor Idempotente. Se o flag removeDups for true, as mensagens duplicadas (as que têm o MessageID no repositório messageIDs) serão roteadas para o Canal de Mensagens Inválidas, caso contrário serão enviadas para a saída do filtro com uma propriedade de cabeçalho “Duplicate: true” para que outros componentes possam lidar com ela. Se a mensagem não tiver um ID no repositório messageIDs, sua MessageID é adicionada ao conjunto e a mensagem é enviada para a saída:

```

664. public class DupsFilter implements MessageListener {
665.
666.     private MessageProducer producer;
667.     private MessageProducer invalidProducer;
668.     private Session session;
669.
670.     private Set<String> messageIDs = new HashSet<>();
671.     boolean removeDups = false;
672.
673.     public DupsFilter(Connection con,
674.                         Destination in, Destination out, Destination
675.                         invalid,
676.                         boolean removeDups) throws JMSEException {
677.         this.removeDups = removeDups;
678.         session = con.createSession(false, Session.AUTO_ACKNOWLEDGE);
679.         MessageConsumer consumer = session.createConsumer(in);
680.         producer = session.createProducer(out);
681.         invalidProducer = session.createProducer(invalid);
682.         consumer.setMessageListener(this);
683.         con.start();
684.
685.     @Override
686.     public void onMessage(Message message) {
687.         try {
688.             String messageID = message.getJMSMessageID();
689.
690.             if (messageIDs.contains(messageID)) {
691.                 System.out.println("Duplicate found!");
692.                 if(removeDups) {
693.                     invalidProducer.send(message);
694.                 } else {
695.                     message.setBooleanProperty("Duplicate", true);
696.                     producer.send(message);
697.                 }
698.             } else {
699.                 System.out.println("Not a duplicate!");
700.                 messageIDs.add(messageID);
701.                 producer.send(message);
702.             }
703.         }
704.     }
705. }
```

```

700.          }
701.      } catch (JMSEException e) {
702.          e.printStackTrace();
703.      }
704.  }
705.
706.  public static void main(String[] args) throws Exception {
707.      Context ctx = new InitialContext();
708.      ConnectionFactory factory =
709.          (ConnectionFactory) ctx.lookup("ConnectionFactory");
710.      Connection con = factory.createConnection();
711.      Destination from = (Destination) ctx.lookup("inbound");
712.      Destination to = (Destination) ctx.lookup("outbound");
713.      Destination invalid = (Destination) ctx.lookup("invalid");
714.
715.      new DupsFilter(con, from, to, invalid, true);
716.
717.      System.out.println("Waiting 60 seconds for messages...");
718.
719.      Thread.sleep(60000); // Will wait one minute for files
720.      con.close();
721.  }
722. }
```

## Receptor Idempotente em Apache Camel

Camel suporta este padrão através da classe `IdempotentConsumer` que usa uma expressão para calcular um string de ID único para um Exchange, usando-o para eliminar mensagens duplicadas. O ID é comparado com valores disponíveis em um repositório. Se o ID já foi visto antes a mensagem é consumida, caso contrário o ID é adicionado e a mensagem é processada. Camel fornece vários tipos de repositórios (implementações da interface `IdempotentRepository`): `MemoryIdempotentRepository`, `FileIdempotentRepository`, `JPAMessageIdRepository`, `JdbcMessageIdRepository`, etc.

O exemplo abaixo (da documentação) mostra como usar um `MemoryIdempotentRepository` para eliminar mensagens duplicadas:

```

from("jms:queue:a")
    .idempotentConsumer(header("MyMessageId"),
        MemoryIdempotentRepository.memoryMessageIdRepository(100)
    )
    .process(new Processor() { ... });
```

Camel também permite manter as mensagens duplicadas na rota e apenas marcá-las, para que possam ser processadas posteriormente por outros componentes. Para isto deve usar a opção `skipDuplicate(false)`:

```

from("direct:start")
    .idempotentConsumer(header("messageId"))
```

```

.messageIdRepository(repo)
.skipDuplicate(false)
.filter(property(Exchange.DUPLICATE_MESSAGE).isEqualTo(true))
    .to("mock:duplicate") // duplicatas aqui!
    .stop()
.end()
.to("mock:result"); // sem duplicatas aqui

```

## Receptor Idempotente em Spring Integration

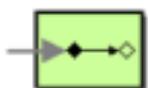
Em Spring Integration é fornecido como um componente interceptador (Advice) que pode ser aplicado a qualquer terminal de consumo de mensagens (através de <advice-chain>):

```
<idempotent-receiver endpoint="endpoint1, foo*"
    metadata-store="store"
    discard-channel="duplicates"
    key-expression="payload.invoiceNumber"/>
```

Mensagens duplicadas são enviadas ao discard-channel, que é opcional. Se ele não estiver presente, mensagens duplicadas irão passar mas elas conterão um cabeçalho “duplicateMessage” que permite que outros componentes lidem com o problema.

## (57) Ativador de serviço (Service Activator)

### Ícone



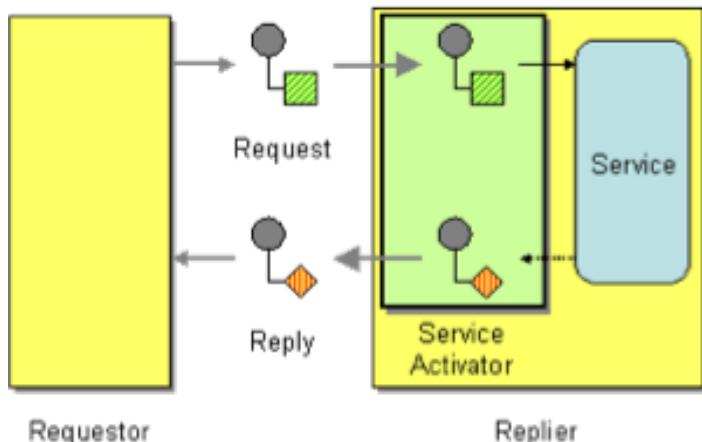
### Problema

“Como projetar um serviço que possa ser chamado de forma síncrona (sem usar mensageria) ou de forma assíncrona (usando tecnologias de mensageria)?”

### Solução

“Projete um Ativador de Serviço (Service Activator) que conecte as mensagens do canal ao serviço”

## Diagrama



## Descrição

Um Ativador de Serviço (Service Activator) permite que um serviço síncrono (ex: RPC) possa ser utilizado de forma assíncrona em um sistema de mensageria. Pode ser um Consumidor de Sondagem ou um Consumidor Ativado por Evento que recebe Mensagens-Comando. O Ativador de Serviço extrai o comando da mensagem recebida e utiliza-o para fazer uma chamada ao serviço síncrono.

Um Ativador de Serviço pode ser one-way (apenas envia uma requisição) ou two-way (envia uma requisição e devolve uma resposta). Serviços one-way são executados por Mensagens-Comando que não retornam dados. Um serviço two-way requer que o Ativador de Serviço chame o serviço síncrono, obtenha seus dados de retorno e os empacote em uma mensagem enviada para um canal de resposta.

Aplicações que desejem chamar o serviço diretamente, de forma síncrona (ex: clientes RPC), podem fazê-lo diretamente, sem passar pelo sistema de mensageria.

## Aplicações

Aplicações que precisam oferecer uma interface síncrona, mas desejam participar de mensageria.

Exemplo: uma aplicação EJB pode usar um Session Bean para oferecer um serviço síncrono para clientes remotos. Um Message-Driven Bean (Consumidor Ativado por Evento) configurado para receber mensagens em um canal pode ser usado para receber Mensagens-Comando que serão traduzidas em chamadas síncronas locais aos métodos do Session Bean. O Message-Driven Bean pode enviar o valor retornado pelo método em uma mensagem de resposta.

## Ativador de Serviço em Java / JMS

A interface de serviços abaixo oferece métodos para gravar um Product ou localizá-lo por sua chave primária. É uma interface síncrona. Uma das operações retorna valor ou exceção, a outra não retorna valor:

```
723. public class ProductService {
724.     public void persist(Product p) {
725.         ProductDatabase.addProduct(p);
```

```

726.    }
727.
728.    public Product select(Long pk) throws ProductNotFoundException {
729.        Product p = ProductDatabase.getProduct(pk);
730.        if (p == null) {
731.            throw new ProductNotFoundException("Product " + pk + " not
732.                found!");
733.        }
734.        return p;
735.    }

```

Podemos prover uma interface com o sistema de mensageria através de um Ativador de Serviço. A classe abaixo recebe Mensagens-comando para esse serviço. Se o comando for “getProduct” um mecanismo Requisição-Resposta pesquisará o produto no banco e devolverá ao cliente uma mensagem contendo o produto solicitado em XML ou uma mensagem contendo uma exceção, se o produto não for localizado. Se o comando for addProduct, uma representação em XML do produto estará no corpo da Mensagem-comando. Ele será decodificado e transformado em objeto para que possa ser gravado no banco.

```

736.    public class ProductServiceActivator implements MessageListener {
737.
738.        private Session session;
739.        private MessageProducer replyProducer;
740.        private MessageConsumer requestConsumer;
741.
742.        public ProductServiceActivator(Connection con, Destination
requestQueue)
743.            throws NamingException, JMSEException {
744.            session = con.createSession(false, Session.AUTO_ACKNOWLEDGE);
745.            requestConsumer = session.createConsumer(requestQueue);
746.            requestConsumer.setMessageListener(this);
747.        }
748.
749.        public void onMessage(Message message) {
750.            try {
751.                String command = message.getStringProperty("Command");
752.                System.out.println("Message received: " + command);
753.
754.                if (command.equals("getProduct")) {
755.                    Long pk = message.getLongProperty("ProductID");
756.                    String payload = null;
757.                    try {
758.                        Product p = new ProductService().select(pk);
759.
760.                        StringWriter writer = new StringWriter();
761.                        JAXBContext jctx =
JAXBContext.newInstance(Product.class);
762.                        Marshaller m = jctx.createMarshaller();

```

```

763.             m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
764.             m.setProperty(Marshaller.JAXB_ENCODING, "UTF-8");
765.             m.marshal(p, writer);
766.             payload = writer.toString();
767.         } catch (ProductNotFoundException e) {
768.             message.setObjectProperty("Exception", e);
769.             payload = "<exception>" + e.getClass().getName()
770.                         + "</exception>";
771.         }
772.
773.         Destination replyDestination = message.getJMSReplyTo();
774.         replyProducer = session.createProducer(replyDestination);
775.         TextMessage replyMessage = session.createTextMessage();
776.         replyMessage.setText(payload);
777.
778.         replyMessage.setJMSSCorrelationID(message.getJMSSessageID());
779.         replyProducer.send(replyMessage);
780.
781.         System.out.println("getProduct() reply was sent: "
782.                         + replyMessage.getText());
783.
784.     } else if (command.equals("addProduct")) {
785.         TextMessage textMessage = (TextMessage) message;
786.
787.         StringReader reader = new
StringReader(textMessage.getText());
788.         JAXBContext jctx = JAXBContext.newInstance(Product.class);
789.         Unmarshaller m = jctx.createUnmarshaller();
790.         Product p = (Product) m.unmarshal(reader);
791.
792.         new ProductService().persist(p);
793.         System.out.println("Product " + p + " was added!");
794.     } else {
795.         System.out.println("Send to invalid message channel!");
796.     }
797. } catch (JMSEException e) {
798.     e.printStackTrace();
799. } catch (JAXBException e) {
800.     e.printStackTrace();
801. }
802.
803. public static void main(String[] args) throws Exception {
804.     Context ctx = new InitialContext();
805.     Destination requestQueue = (Destination) ctx.lookup("request-
queue");
806.
807.     ConnectionFactory factory = (ConnectionFactory) ctx
808.         .lookup("ConnectionFactory");

```

```

809.         Connection con = factory.createConnection();
810.
811.         new ProductServiceActivator(con, requestQueue);
812.
813.         con.start();
814.         System.out.println("Service Activator started.");
815.
816.         Thread.sleep(30000);
817.         // close the connections
818.         System.out.println("Done.");
819.         con.close();
820.     }
821.
822. }
```

O cliente para esta aplicação é o mesmo que foi mostrado no exemplo de Mapeador de Mensageria.

## Ativador de Serviço em Camel

Usando beans, POJOs ou CXF (Web Services SOAP) é possível mapear uma mensagem a uma interface de serviços em Camel. O exemplo abaixo usa a URI bean (Bean Component):

```
from("jms:queue:inbound")
    .to("bean:produtoFacade?methodName=persist")
```

Pode-se também usar a sintaxe DSL, que é equivalente:

```
from("jms:queue:inbound")
    .bean(new ProdutoFacade(), "persist")
```

## Ativador de Serviço em Spring Integration

Um Ativador de Serviço é criado em Spring Integration usando o elemento <service-activator> e atributos que associam um canal a uma instância da interface de serviços. Se a interface contém apenas um método, ou se ele é anotado com @ServiceActivator não é necessário especificar um atributo identificando-o. Se houver mais de um ele deve ser identificado pelo atributo method:

```
<int:service-activator input-channel="inbound" ref="produtoFacade"
method="persist"/>
```

Se o método retorna um valor diferente de void, o Terminal irá tentar enviar uma mensagem de resposta ao Endereço de Resposta, que deve ser especificado no atributo "output-channel".

```
<int:service-activator input-channel="inbound" output-channel="produtoReply"
ref="produtoFacade" method="getProduto"/>
```

A interface de serviços também pode ser aninhada no <service-activator>:

```
<int:service-activator input-channel="inbound"
                      output-channel="produtoReply" method="getAllProdutos">
    <beans:bean class="br.com...ProdutoFacade"/>
```

```
</int:service-activator>
```

## Revisão

Padrões de integração de sistemas relacionados a terminais de mensageria:

- (47) Gateway de mensageria: desacopla o sistema de mensageria do restante da aplicação.
- (48) Mapeador de mensageria: mapeia objetos ao sistema de mensageria.
- (49) Cliente transacional: insere o consumo e produção de mensagens em um contexto transacional.
- (51) Consumidor de sondagem: consumidor que sonda periodicamente um canal para consumir mensagens.
- (50) Consumidor ativado por eventos: consumidor que é notificado quando novas mensagens são adicionadas ao canal.
- (52) Consumidores concorrentes: consumidores que disputam mensagens em um Canal Ponto-a-Ponto.
- (53) Despachante de mensagens: consumidor que recebe mensagens em um canal e despacha para diferentes processadores que os processam geralmente em paralelo.
- (54) Consumidor seletivo: consumidor que filtra as mensagens que irá consumir usando um seletor.
- (55) Assinante durável: assinante de um Canal de Difusão que poderá receber todas as mensagens enviadas para o canal do qual é assinante mesmo se estiver inativo durante o envio.
- (56) Receptor idempotente: um consumidor de mensagens que não produz resultados diferentes se receber mensagens duplicadas.
- (57) Ativador de serviço: um consumidor de mensagens que é capaz de executar operações em interfaces de serviços síncronos.

# Capítulo 9

# Gerenciamento

Todos os benefícios proporcionados pelos padrões de mensageria como a flexibilidade, baixo acoplamento, isolamento, etc. ajudam a construir soluções mais robustas, modulares, reutilizáveis, escaláveis, que operam de modo assíncrono e que podem ser distribuídas entre plataformas e sistemas diferentes. Por outro lado, isto tudo pode tornar muito mais difícil a monitoração, controle, depuração e teste da solução.

Os padrões do catálogo EIP para monitorar uma solução de mensageria tem um nível de abstração limitado ao gerenciamento de sistemas, que procura medir quantas mensagens são enviadas, quanto tempo leva para uma mensagem ser processada, etc. mas não inclui a inspeção do conteúdo da mensagem (exceto propriedades acessíveis pelo cabeçalho da mensagem). São classificados em três categorias:

- *Monitoração e controle*: inclui o (58) *Barramento de Controle (Control Bus)* – um ponto central de controle para gerenciar e monitorar o sistema, que é capaz de enviar comandos aos componentes (ex: para ligar ou desligar o modo de testes, por exemplo); e o padrão (59) *Desvio (Detour)*, que permite desviar o fluxo de controle para que passe por etapas adicionais, para medir performance, realizar validação e logging.
- *Observação e análise do tráfego*: inclui o padrão (60) *Escuta (Wire Tap)*, que permite observar o conteúdo das mensagens que circulam pelos canais sem interferir no seu fluxo; (61) *Histórico de Mensagens (Message History)* que mantém um log dos componentes que uma mensagem visitou; (62) *Repositório de Mensagens (Message Store)* que guarda um log das mensagens que passaram pelo sistema. Esses três padrões permitem analisar todo o fluxo assíncrono. O padrão (63) *Proxy Inteligente (Smart Proxy)* descreve uma solução para rastrear serviços Requisição-Resposta.
- *Testes e depuração*: pode-se periodicamente monitorar a saúde do sistema enviando uma mensagem que irá passar por vários componentes coletando dados de teste. O padrão (64) *Mensagem de Teste (Test Message)* descreve uma solução para essa questão. Para que outras mensagens não interfiram nos testes, pode-se usar um (65) *Purificador de Canal (Channel Purger)* para realizar um “clean” no canal antes de testar.

## (58) Barramento de controle (Control Bus)

### Ícone



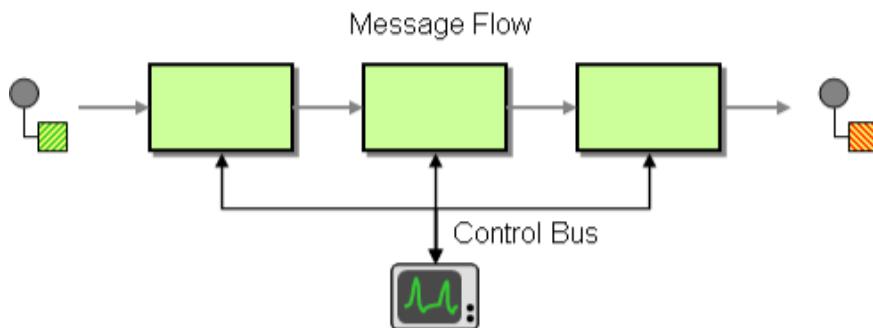
### Problema

“Como administrar um sistema de mensageria que é distribuído através de múltiplas plataformas e uma ampla área geográfica?”

### Solução

“Use um Barramento de Controle (Control Bus) para gerenciar um sistema de integração. O Barramento de Controle usa o mesmo mecanismo de mensageria usado pelos dados da aplicação, mas emprega canais separados para transmitir dados que são relevantes para o gerenciamento de componentes envolvidos no fluxo de mensagens”

### Diagrama



### Descrição

A inclusão de um Barramento de Controle (Control Bus) em um sistema de integração requer configurar *todos* os componentes processadores de mensagens para que tenham uma interface de controle, onde possam receber e enviar mensagens de controle. Durante seu funcionamento, cada componente irá receber e enviar mensagens normalmente, de acordo com o fluxo normal da aplicação, mas também poderá enviar e receber mensagens de e para o barramento de controle.

As mensagens de controle poderão ser mensagens de configuração automática da aplicação (ex: tabelas dinâmicas de roteamento), mensagens de “pulso” que informam que a aplicação ou componente está funcionando corretamente, mensagens de testes com dados mais detalhados sobre as operações, mensagens reportando erros e exceções, estatísticas e relatórios (que podem ser usados em um console para monitoração do sistema em tempo real).

## Aplicações

Um Barramento de Controle deve ser projetado e implantado em um sistema de integração para servir de serviço central de operações de gerenciamento do sistema. Isto é crítico principalmente em sistemas distribuídos geograficamente e que envolvem aplicações rodando em diferentes sistemas operacionais, redes e protocolos.

### Barramento de Controle em Java / JMS

No exemplo abaixo reconfiguramos os componentes usados para rotear arquivos (mostrados no capítulo 3) para que todos sejam também listeners de um topic onde são enviadas mensagens de controle. Os componentes filtram as mensagens que se destinam a eles usando Consumidores Seletivos. As mensagens possuem cabeçalhos identificando o componente ou componentes que devem receber a mensagem. A resposta é enviada para o canal de resposta informado pelo Barramento de Controle.

O código abaixo mostra como poderia ser usado um Barramento de Controle simples que solicita status de componentes que participam de uma rota. Neste exemplo o ID do componente é a sua classe:

```
Connection con = ...;
Destination controlTopic = (Destination) ctx.lookup("control-topic");
Destination replyQueue = (Destination) ctx.lookup("control-reply");

ControlBus control = new ControlBus(con, controlTopic, replyQueue);

control.requestStatus("ProductServiceActivator");
control.requestStatusAllComponents();
```

A classe ControlBus poderia ser implementada como um listener que espera mensagens de resposta dos componentes na fila de respostas “control-reply”. Os comandos acima enviam mensagens de controle para um Canal de Difusão “control-topic” que é assinado por todos os componentes que participam da monitoração. Quando uma resposta é recebida o status dela é impresso na console do ControlBus:

```
01.  public class ControlBus implements MessageListener {
02.
03.      private Session controlSession;
04.      private MessageConsumer statusReceiver;
05.      private MessageProducer requestSender;
06.      private Destination replyQueue;
07.
08.      public ControlBus( Connection con, Destination controlTopic,
09.                          Destination replyQueue) throws JMSEException {
10.
11.          this.replyQueue = replyQueue;
12.          controlSession = con.createSession(false,
Session.AUTO_ACKNOWLEDGE);
13.          requestSender = controlSession.createProducer(controlTopic);
```

```

14.         statusReceiver = controlSession.createConsumer(replyQueue);
15.         statusReceiver.setMessageListener(this);
16.         con.start();
17.     }
18.
19.     @Override
20.     public void onMessage(Message message) {
21.         System.out.println("Received response.");
22.         try {
23.             System.out.println("\nStatus response received for "
24.                             + message.getStringProperty("ComponentID"));
25.             System.out.println("Inbound channel: "
26.                             + message.getStringProperty("InboundChannel"));
27.             System.out.println("Outbound channel: "
28.                             + message.getStringProperty("OutboundChannel"));
29.             System.out.println("Timestamp: "
30.                             + message.getStringProperty("Timestamp") + "\n");
31.         } catch (JMSEException e) {
32.             e.printStackTrace();
33.         }
34.     }
35.
36.     public void requestStatus(String componentID) throws JMSEException {
37.         System.out.println("Will request status for "+componentID+":");
38.         Message controlMessage = controlSession.createMessage();
39.         controlMessage.setStringProperty("ComponentID", componentID);
40.         controlMessage.setJMSReplyTo(replyQueue);
41.         requestSender.send(controlMessage);
42.     }
43.
44.     public void requestStatusAllComponents() throws JMSEException {
45.         System.out.println("Will request status for all components");
46.         requestStatus("all");
47.     }
48. }

```

Para que possam participar da monitoração, cada componente precisa inicializar também um componente monitorável, associado a ele. O componente abaixo (ManagedEndpoint) pode ser conectado a um componente T para que ele possa enviar dados de status:

```

01.  public class ManagedEndpoint<T> {
02.
03.      private Session controlSession;
04.      private MessageConsumer controlIn;
05.      private MessageProducer controlOut;
06.
07.      private Destination componentInChannel;
08.      private Destination componentOutChannel;
09.      private T component;

```

```
10.
11.    public ManagedEndpoint(T component, Destination componentInChannel,
12.                           Destination componentOutChannel)
13.    {
14.        this.component = component;
15.        this.componentInChannel = componentInChannel;
16.        this.componentOutChannel = componentOutChannel;
17.    }
18.
19.    public void initControl(Connection con, Destination controlTopic)
20.                            throws JMSEException {
21.        String componentID = component.getClass().getName();
22.        controlSession = con.createSession(false,
23. Session.AUTO_ACKNOWLEDGE);
24.        String selector = "ComponentID = '" + componentID
25.                           + "' OR ComponentID = 'all'";
26.        controlIn = controlSession.createConsumer(controlTopic, selector);
27.        controlIn.setMessageListener(new MessageListener() {
28.
29.            @Override
30.            public void onMessage(Message message) {
31.                System.out.println("Received a control message.");
32.                try {
33.                    Destination replyChannel = message.getJMSReplyTo();
34.                    Message reply = controlSession.createMessage();
35.                    reply.setJMSCorrelationID(message.getJMSMessageID());
36.                    if(componentInChannel != null) {
37.                        reply.setStringProperty("InboundChannel",
38. componentInChannel.toString());
39.                    }
40.                    if(componentOutChannel != null) {
41.                        reply.setStringProperty("OutboundChannel",
42. componentOutChannel.toString());
43.                    }
44.                    reply.setStringProperty("ComponentID", componentID);
45.                    reply.setLongProperty("Timestamp", new
46. Date().getTime());
47.                    controlOut =
controlSession.createProducer(replyChannel);
48.                    System.out.println("Sending status to " +
replyChannel);
49.                    controlOut.send(reply);
50.                } catch (JMSEException e) {
51.                    e.printStackTrace();
52.                }
53.            }
54.        }
55.    }
56.
```

```

47.         }
48.     });
49.     con.start();
50.   }
51. }
```

Para que um componente possa ser monitorado por este Barramento de Controle ele precisa conectar e configurar o ManagedEndpoint informando seus canais de entrada e saída. Neste exemplo a monitoração será ativada quando o método setManaged() for chamado:

```

01.  public class ProductServiceActivator implements MessageListener,
                           ManagedComponent {
02.
03.    private Session session;
04.    private MessageProducer replyProducer;
05.    private MessageConsumer requestConsumer;
06.    private Destination requestQueue;
07.
08.    private boolean managed = false;
09.    private ManagedEndpoint<ProductServiceActivator> managedEndpoint;
10.
11.    public ProductServiceActivator(Connection con, Destination
requestQueue)
12.          throws JMSEException { ... }
13.
14.    @Override
15.    public void setManaged(Connection con, Destination controlTopic)
                           throws JMSEException {
16.      this.managed = true;
17.      managedEndpoint =
18.          new ManagedEndpoint<ProductServiceActivator>(this,
requestQueue, null);
19.      managedEndpoint.initControl(con, controlTopic);
20.    ...
21.  }
```

A configuração pode ser feita depois de inicializado o componente:

```

Destination controlTopic = (Destination) ctx.lookup("control-topic");
...
ProductServiceActivator activator = new ProductServiceActivator(con,
requestQueue);
activator.setManaged(con, controlTopic);
```

A execução do ControlBus em uma rota (usando exemplo do Service Activator do capítulo 8) monitorando o ProductServiceActivator e JmsMapperFacade produz a saída abaixo se ambos os componentes estiverem no ar no momento em que as mensagens forem enviadas:

```
Status response received for
br.com.argonavis.eipcourse.mgmt.ProductServiceActivator
Inbound channel: queue://request-queue
Outbound channel: null
Timestamp: 1444153822161

Status response received for br.com.argonavis.eipcourse.mgmt.JmsMapperFacade
Inbound channel: queue://request-queue
Outbound channel: queue://response-queue
Timestamp: 1444153895960
```

## Barramento de Controle em Apache Camel

O componente controlbus permite ligar, desligar, suspender, reiniciar rotas e obter o seu status. Por exemplo, o comando abaixo (da documentação do Camel) manda uma mensagem vazia para o endpoint do controlbus para iniciar uma rota:

```
template.sendBody("controlbus:route?routeId=foo&action=start", null);
```

Outros comandos são stop, suspend, resume e status. Para obter o status de uma rota use:

```
String status = template.requestBody("controlbus:route?routeId=foo&action=status",
                                     null, String.class);
```

O Barramento de Controle do Camel também permite controles mais elaborados usando JMS e linguagens de scripting.

## Barramento de Controle em Spring Integration

Em Spring Integration qualquer componente Spring com método anotado com @ManagedOperation (ou @ManagedAttribute) pode ser chamado pelo Control Bus. Por exemplo:

```
@Component
public class TestComponent {
    @ManagedOperation
    public void testMethod() {
        System.out.println("This is a test!");
    }
}
```

É preciso configurar o <control-bus> e um canal onde mensagens de controle possam ser enviadas.

```
<int:channel id="control-channel" />
<int:control-bus input-channel="control-channel"/>
```

As mensagens enviadas pelo canal “control-channel” podem chamar um método monitorado. Uma mensagem contendo um payload em SpEL, da forma "@componente.metodo()" pode ser usada para chamar o método.

```
Message msg = MessageBuilder.withPayload( "textComponent.testMethod" ).build();
```

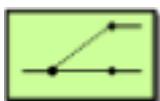
```
MessageChannel controle = springCtx.getBean("control-channel",
MessageChannel.class);
controle.send(msg);
```

O resultado da execução será

This is a test!

## (59) Desvio (Detour)

### Ícone



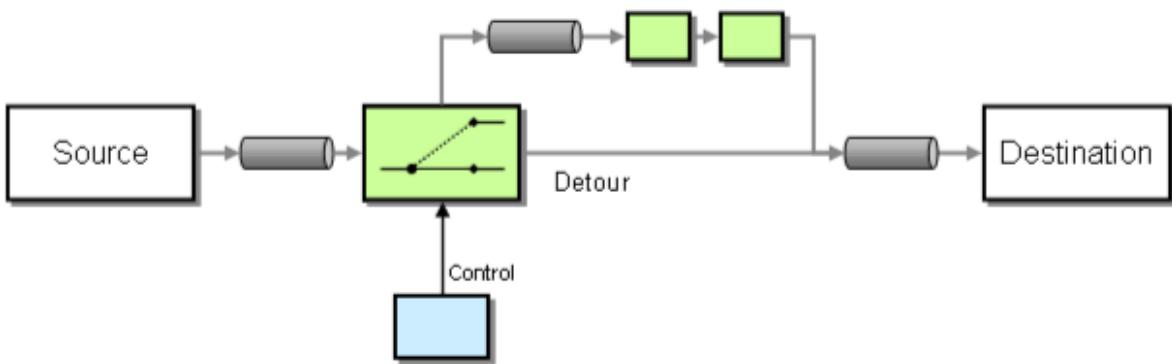
### Problema

“Como redirecionar uma mensagem através de etapas intermediárias para realizar funções de validação, testes e depuração?”

### Solução

“Construa um Desvio (Detour) com um Roteador Baseado em Conteúdo controlado pelo Barramento de Controle. Em um estado o roteador roteia mensagens recebidas através de etapas adicionais, enquanto que no outro ele roteia mensagens direto ao canal de destino”

### Diagrama



### Descrição

Um Desvio (Detour) serve para possibilitar a inclusão de caminhos alternativos para mensagens, tipicamente para validação, testes, depuração e outras tarefas que podem ter impacto no funcionamento do sistema e devem ser desligadas em produção. O Desvio pode ser ligado ou desligado com uma mensagem enviada a partir do Barramento de Controle.

## Aplicações

Um Desvio deve ser usado quando for necessário ter um caminho alternativo para testar mensagens em desenvolvimento, sem afetar a performance da aplicação em ambiente de produção.

### Desvio em Java / JMS

Um desvio de rota pode ser implementado por um Roteador Dinâmico, como o que foi mostrado no capítulo 6, onde em vez da nova rota ser calculada ela seria informada diretamente pelo Barramento de Controle. Usando o Barramento de Controle apresentado na seção anterior, podemos construir um mecanismo de desvio acrescentando um método que substitua o canal de saída de um componente por outro:

```
public void detour(String componentID, Destination detour) throws JMSEException {
    System.out.println("Will request that "+componentID+" detour to " + detour);
    Message controlMessage = controlSession.createMessage();
    controlMessage.setStringProperty("ComponentID", componentID);
    controlMessage.setJMSReplyTo(replyQueue);
    controlMessage.setObjectProperty("Detour", detour);
    requestSender.send(controlMessage);
}
```

Para isto os componentes teriam que ter métodos get/set para possibilitar a alteração dos seus canais de saída, e o componente de gerenciamento (a classe ManagedEndpoint mostrada anteriormente) teria que substituir o canal de saída pelo canal recebido na propriedade “Detour”:

```
Destination detour = (Destination) message.getStringProperty("Detour");
if(detour != null) {
    component.setOutChannel(detour);
    ...
}
```

### Desvio em Apache Camel

Um desvio pode ser implementado em Camel usando expressões em blocos choice() para roteamento, permitindo que a alteração do valor de uma variável altere o valor booleano da expressão. O exemplo abaixo (da documentação Camel) mostra como desviar uma rota usando um bean. Dependendo do valor retornado pelo método isDetour() haverá ou não desvio de rota:

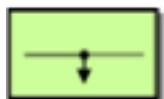
```
from("direct:start")
    .choice()
        .when()
            .method("controlBean", "isDetour")
            .to("mock:detour")
        .end()
    .to("mock:result");
```

## Desvio em Spring Integration

Spring Integration não implementa explicitamente este padrão, mas ele pode ser configurado como um roteador dinâmico de dois estados.

### (60) Escuta (Wire Tap)

#### Ícone



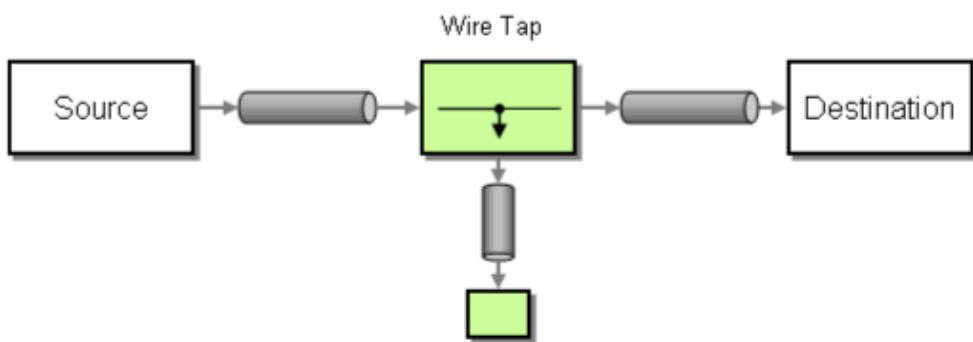
#### Problema

“Como inspecionar as mensagens que são transmitidas por um Canal Ponto-a-Ponto?”

#### Solução

“Insira uma Lista de Receptores (Recipient List) no canal que publica cada mensagem recebida para o canal principal e para um canal secundário”

#### Diagrama



#### Descrição

Uma Escuta (Wire Tap) é uma Lista de Receptores que publica as mensagens recebidas em dois canais de saída: um para inspeção e controle, e o outro para o fluxo normal da aplicação. Uma Escuta deve normalmente ser controlada por um Barramento de Controle, para que possa ser ligado e desligado.

Dependendo da implementação usada, pode haver um impacto de performance. Alguns sistemas fazem marshalling e unmarshalling da mensagem quando passa por um processador, e o ID da mensagem poderá mudar.

#### Aplicações

Uma Escuta pode ser implantada para monitorar todas as mensagens que passam por um canal, sem interferir no fluxo normal da aplicação.

## Escuta em Java / JMS

Uma escuta pode ser implementada em JMS usando RecipientList. Neste exemplo o WireTap age como uma bifurcação, transferindo a mensagem para seu canal de saída e enviando uma cópia para o canal da Escuta:

```

01.  public class WireTap implements MessageListener {
02.      private Session session;
03.      private Destination outChannel;
04.      private Destination wireTapChannel;
05.
06.      public WireTap(Connection con, Destination inChannel, Destination
outChannel,
                           Destination wireTap) throws
JMSEException {
07.          this.outChannel = outChannel;
08.          session = con.createSession(false, Session.AUTO_ACKNOWLEDGE);
09.          MessageConsumer consumer = session.createConsumer(inChannel);
10.          consumer.setMessageListener(this);
11.          con.start();
12.      }
13.
14.      @Override
15.      public void onMessage(Message message) {
16.          try {
17.              routeMessage(outChannel, message);
18.              routeMessage(wireTapChannel, message);
19.          } catch (JMSEException e) {
20.              e.printStackTrace();
21.          }
22.      }
23.
24.      public void routeMessage(Destination destination, Message message)
throws JMSEException {
25.          MessageProducer producer =
session.createProducer(destination);
26.          producer.send(message);
27.      }
28.
29.  }

```

No trecho abaixo (de `WireTappedBridge.java`) arquivos são lidos da pasta `/tmp/jms/inbox` e transferidos para a pasta `/tmp/jms/outbox`, mas como há uma Escuta no meio, ela desvia uma cópia para o canal “wiretap” onde há outro `FileOutboundAdapter` que transfere cópias dos arquivos para “`/tmp/jms/wiretap`”:

```

Destination fromChannel = (Destination) ctx.lookup("in-channel");
Destination toChannel   = (Destination) ctx.lookup("out-channel");
Destination wireTap     = (Destination) ctx.lookup("wiretap");

```

```

FileInboundAdapter adapter =
    new FileInboundAdapter(con, fromChannel, new File("/tmp/jms/inbox"), false);
List<File> files = adapter.loadFiles();
if (!files.isEmpty()) {
    List<Message> messages = adapter.createMessages(files);
    adapter.send(messages);
}

new WireTap(con, fromChannel, toChannel, wireTap);
new FileOutboundAdapter(con, toChannel, new File("/tmp/jms/outbox"));
new FileOutboundAdapter(con, wireTap, new File("/tmp/jms/wiretap"));

```

Uma outra maneira de construir uma Escuta em JMS é através da classe QueueBrowser. Ela permite analisar as mensagens enfileiradas em um Canal Ponto-a-Ponto sem consumi-las. A classe abaixo analisa as mensagens (assumindo que são TextMessage) de um canal e imprime o conteúdo de cada mensagem:

```

01.  public class BrowsingWireTap {
02.
03.      private Session session;
04.      private QueueBrowser browser;
05.
06.      public BrowsingWireTap(Connection con, Queue inQueue) throws
JMSEException {
07.          session = con.createSession(false, Session.AUTO_ACKNOWLEDGE);
08.          browser = session.createBrowser(inQueue);
09.          con.start();
10.      }
11.
12.      public void browse() throws JMSEException {
13.          Enumeration<Message> messages = browser.getEnumeration();
14.
15.          if (!messages.hasMoreElements()) {
16.              System.out.println("No messages in queue");
17.          } else {
18.              while (messages.hasMoreElements()) {
19.                  Message message = messages.nextElement();
20.                  System.out.println("Message: "
+ ((TextMessage)message).getText());
21.              }
22.          }
23.      }
24.
25.      public static void main(String[] args) throws Exception {
26.          Context ctx = new InitialContext();
27.          ConnectionFactory factory = (ConnectionFactory) ctx
28.              .lookup("ConnectionFactory");
29.          Connection con = factory.createConnection();

```

```

30.         Queue queue = (Queue) ctx.lookup("inbound");
31.
32.         BrowsingWireTap browser = new BrowsingWireTap(con, queue);
33.         browser.browse();
34.
35.         con.close();
36.     }
37. }
```

Executando o MockMessageProducer que cria 10 mensagens contendo XML, o resultado é:

```

Message:
<product><name>Product_1</name><price>52.63802370619917</price></product>
Message:
<product><name>Product_2</name><price>43.40573295067485</price></product>
Message:
<product><name>Product_3</name><price>67.24390274716473</price></product>
Message:
<product><name>Product_4</name><price>42.63552750797087</price></product>
Message:
<product><name>Product_5</name><price>82.39986521536899</price></product>
Message:
<product><name>Product_6</name><price>8.93304617236229</price></product>
Message:
<product><name>Product_7</name><price>47.048529714931774</price></product>
Message:
<product><name>Product_8</name><price>96.71567922456403</price></product>
Message:
<product><name>Product_9</name><price>22.710211705262285</price></product>
Message:
<product><name>Product_10</name><price>74.33832823496917</price></product>
```

Mas as mensagens ainda estão disponíveis na fila “inbound” pois não foram consumidas.

## Escuta em Apache Camel

Uma escuta em Camel é feita com o método DSL wireTap(), que recebe como argumento um canal para onde as cópias das mensagens serão enviadas:

```
from("jms:queue:inbound")
    .to("jms:queue:png-queue")
    .wireTap("file:/tmp/wiretap")
    .to("jms:queue:outbound");
```

## Escuta em Spring Integration

Uma Escuta pode ser adicionada a qualquer canal como um interceptador, para capturar mensagens sem afetar o fluxo normal. A configuração é simples. No exemplo abaixo todas as mensagens enviadas para o canal “inbound” serão interceptadas e uma cópia será redirecionada ao adaptador de log:

```

<int:channel id="inbound">
    <int:interceptors>
        <int:wire-tap channel="logger" />
    </int:interceptors>
</int:channel>

<int:logging-channel-adapter log-full-message="true" id="logger" level="INFO" />

```

## (61) Histórico da Mensagem (Message History)

### Ícone

Não tem.

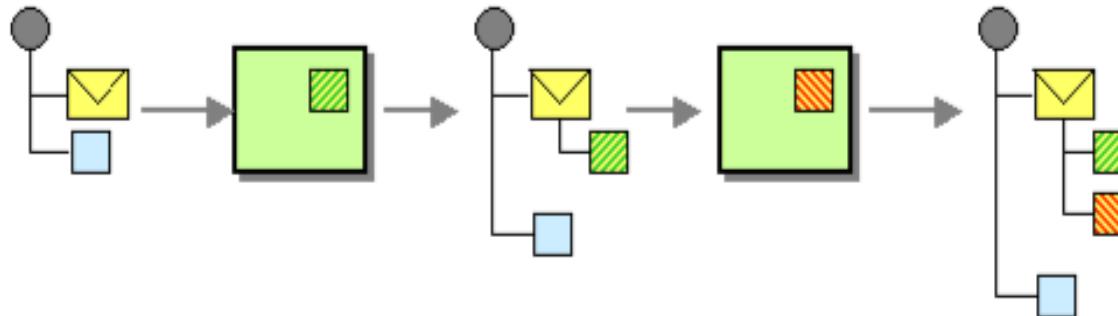
### Problema

“Como podemos efetivamente analisar e depurar o fluxo de mensagens em um sistema fracamente acoplado”

### Solução

“Conecte um Histórico (Message History) à mensagem. O histórico é uma lista de todas as aplicações pelas quais a mensagem passou desde que foi criada.”

### Diagrama



### Descrição

Um Histórico (Message History) é uma propriedade mantida no cabeçalho de uma mensagem que contém uma lista de todos os componentes por onde uma mensagem passou.

O histórico pode ser uma lista simples. Se uma mensagem se divide em várias (passa por um Divisor) é necessário copiar o histórico para cada sub-mensagem. Se for uma combinação de outras mensagens (passando por um Agregador) pode-se escolher um dos caminhos anteriores (perdendo dados de fluxos paralelos) ou guardar uma árvore hierárquica com todos os históricos anteriores. Um histórico também pode ser mantido em um repositório central, guardando uma referência para o histórico no cabeçalho da mensagem.

## Aplicações

Um Histórico deve ser incluído em uma mensagem quando for desejável saber por onde ela passou. O Histórico pode ser usado para detectar problemas de roteamento e loops infinitos.

### Histórico em Java / JMS

O método abaixo, incluído em ManagedEndpoint, grava em cada mensagem os Ids de cada componente por qual ela passou em um Histórico no cabeçalho “MessageHistory”.

```

01. public void saveHistory(T component, Message inMessage, Message
outMessage,
                           Destination outChannel) throws JMSEException {
02.     String history = null;
03.     if(inMessage != null) {
04.         history = inMessage.getStringProperty("MessageHistory");
05.     }
06.     if(history != null) {
07.         history += "," + component.getClass().getName() + "," +
outChannel;
08.     } else {
09.         history = component.getClass().getName() + "," + outChannel;
10.     }
11.     outMessage.setStringProperty("MessageHistory", history);
12. }
```

Ele pode ser chamado antes de cada envio, para gravar o componente que está fazendo o envio e o canal para onde a mensagem está sendo enviada (trecho de ProductServiceActivator.java):

```

...
replyMessage.setJMSCorrelationID(message.getJMSMessageID());

if(managed) {
    managedEndpoint.saveHistory(this, message, replyMessage, replyDestination);
    System.out.println("History for replyMessage: "
                       + replyMessage.getStringProperty("MessageHistory"));
}
replyProducer.send(replyMessage);
...
```

Se todos os componentes gravarem informações no cabeçalho “MessageHistory”, pode-se extrair da Mensagem as informações armazenadas e descobrir por onde ela passou. Na execução de ProductClient.java a mensagem de resposta contendo um Produto devolvida pelo Ativador de Serviço contém o seguinte histórico:

```

br.com.argonavis.eipcourse.mgmt.controlbus.JmsMapperFacade,
queue://request-queue,
br.com.argonavis.eipcourse.mgmt.controlbus.ProductServiceActivator,
queue://response-queue
```

## Histórico em Apache Camel

O Histórico é ligado por default. Para *desligar* configure no CamelContext:

```
camelContext.setMessageHistory(false);
```

Pode-se recuperar o histórico através da propriedade Exchange.MESSAGE\_HISTORY que mantém uma lista de objetos MessageHistory:

```
List<MessageHistory> list = exchange.getProperty(Exchange.MESSAGE_HISTORY,  
List.class);
```

A lista contém routeID, ProcessorID, timestamp e tempo transcorrido.

## Histórico em Spring Integration

Para ativar a gravação do Histórico de todas as mensagens em Spring Integration e rastrear todo componente que tiver um ID, é preciso apenas declarar o elemento <message-history> na configuração do contexto:

```
<int:message-history />
```

A mensagem agora gravará o histórico no cabeçalho MessageHistory.HEADER\_NAME como uma lista de Properties (pares nome=valor) para cada componente.

```
MessageHistory history =  
    message.getHeaders().get(MessageHistory.HEADER_NAME,  
MessageHistory.class);  
  
for(Properties props : history) {  
    System.out.println(props);  
}
```

A saída depende dos componentes e canais que tiverem ID por onde a mensagem passou:

```
{name=inbound-channel, type=channel, timestamp=1444106258}  
{name=xml-transformer, type=transformer, timestamp=1444106267}  
{name=outbound-channel, type=channel, timestamp=1444106291}
```

## (62) Repositório de mensagens (Message Store)

### Ícone



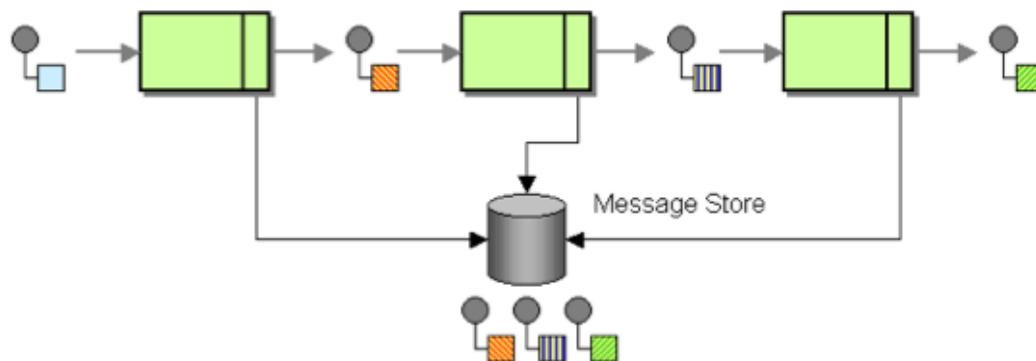
## Problema

“Como obter informações sobre mensagens sem interferir na natureza fracamente acoplada e transiente do sistema de mensageria?”

## Solução

“Use um Repositório de Mensagens (Message Store) para capturar informação sobre cada mensagem em um ponto central”

## Diagrama



## Descrição

Um Repositório de Mensagens (Message Store) é um log que guarda informações sobre as mensagens que passam por um sistema. Pode-se configurar o repositório para guardar diferentes níveis de detalhamento das informações (ex: mensagens inteiras, apenas os cabeçalhos das mensagens, apenas determinadas mensagens, etc.) As informações mantidas no Repositório podem expirar ou serem excluídas periodicamente para liberar espaço.

É diferente do Histórico (Message History) que fica em cada mensagem e deixa de existir quando a mensagem é consumida. O Repositório fica em uma parte central da aplicação e guarda uma cópia das mensagens (e os Históricos dessas mensagens, se houver)

## Aplicações

Um Repositório de Mensagens deve ser usado para manter um histórico de todas as mensagens que passam pelo sistema. Isto pode ser usado para monitoração, estatísticas, etc.

## Repositório em Spring Integration

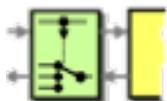
Spring Integration oferece uma interface e várias implementações de MessageStore (JDBC, Redis, MongoDB, etc.) que, uma vez configuradas, podem ser usadas em canais e componentes como Claim Check. Um componente pode ser associado a um Repositório de Mensagens durante a configuração do contexto:

```
<int:channel id="myQueueChannel">
    <int:queue message-store="refToMessageStore"/>
```

<int:channel>

## (63) Proxy inteligente (Smart Proxy)

### Ícone



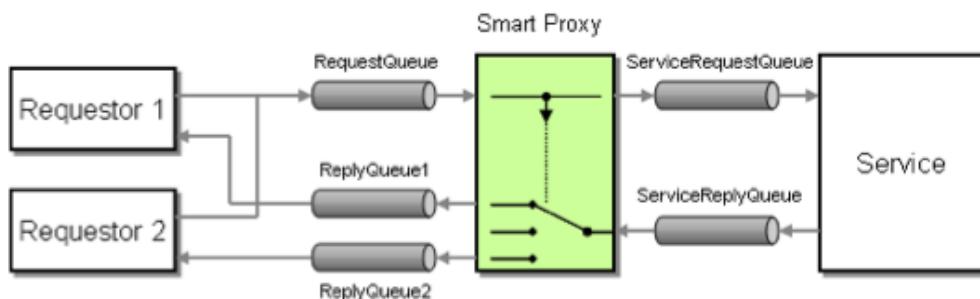
### Problema

“Como pode-se rastrear mensagens em um serviço que publica mensagens de retorno para um Endereço de Resposta especificado pelo solicitante?”

### Solução

“Use um Proxy Inteligente (Smart Proxy) para guardar o Endereço de Resposta fornecido pelo solicitante original e substitua-o com o endereço do Proxy Inteligente. Quando o serviço enviar a mensagem de resposta, redirecione-a para o Endereço de Resposta original”

### Diagrama



### Descrição

O Proxy Inteligente intercepta uma mensagem, guarda seu endereço de resposta e a substitui com o seu próprio endereço, de forma que quando a mensagem for processada pelo serviço, ela seja redirecionada de volta ao Proxy, que pode então ter acesso à mensagem de resposta. Ao receber a mensagem interceptada, o Proxy a redireciona para o endereço de resposta que havia guardado. O endereço de retorno e o endereço do proxy podem ser armazenados em cabeçalhos da própria mensagem. O endereço de retorno original também pode ser armazenado pelo Proxy, neste caso ele deverá utilizar um identificador para associar à mensagem quando ela retornar.

### Aplicações

Quando for necessário mensagens de resposta (em implementações de Requisição-Resposta) uma Escuta não é suficiente. Nesse caso um Proxy Inteligente poderá ser usado para rastrear as mensagens de resposta.

## Proxy Inteligente em Apache Camel

Camel possui um componente de proxy que pode ser usado para testar Requisição-Resposta. O ProxyBuilder cria um proxy para um POJO (exemplo da documentação):

```
public void testProxyBuilderProxyCallAnotherBean() throws Exception {
    OrderService service =
        new
    ProxyBuilder(context).endpoint("direct:bean").build(OrderService.class);
    String reply = service.submitOrderStringReturnString("World");
    assertEquals("Hello World", reply);
}
```

## Proxy Inteligente em Spring Integration

É possível interceptar requisições e respostas com Spring Integration inserindo Escutas nos canais de requisição e resposta e redirecionando para um canal de log comum:

```
<jms:inbound-gateway id="queue" connection-factory="connectionFactory"
    request-destination="requests"
    request-channel="request-begin"
    reply-channel="request-end"/>

<int:chain input-channel="request-begin" output-channel="request-end">
    ...
</int:chain>

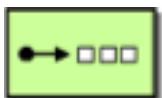
<int:logging-channel-adapter id="request-response-logger"
    log-full-message="true" level="DEBUG"
    logger-name="test.messages" />

<int:channel id="request-begin">
    <int:interceptors>
        <int:wire-tap channel="request-response-logger" />
    </int:interceptors>
</int:channel>

<int:channel id="request-end">
    <int:interceptors>
        <int:wire-tap channel="request-response-logger" />
    </int:interceptors>
</int:channel>
```

## (64) Mensagem de Teste (Test Message)

### Ícone



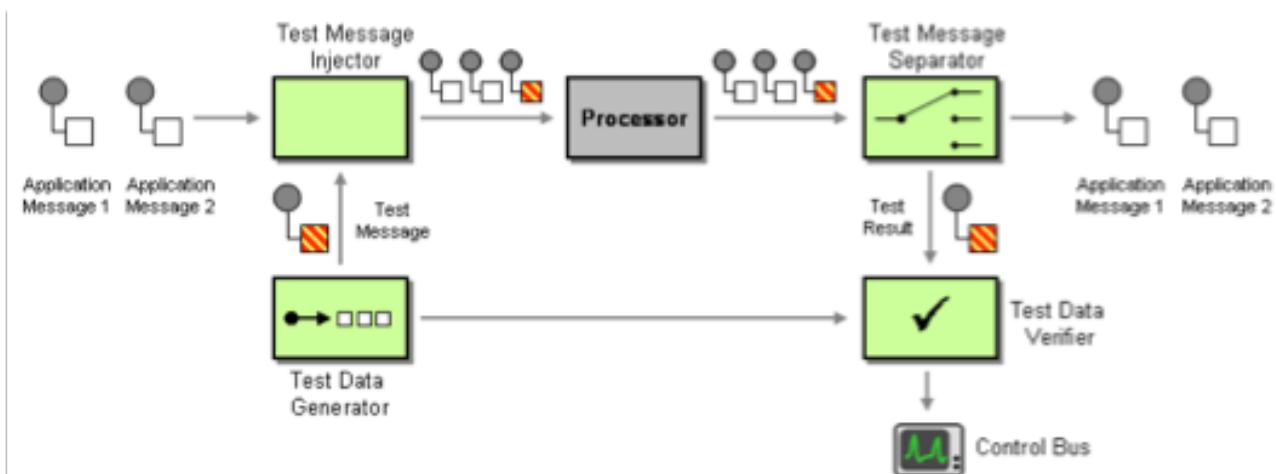
### Problema

“O que acontece se um componente está ativamente processando mensagens, mas corrompe mensagens de saída devido a um erro interno”

### Solução

“Use uma Mensagem de Teste (Test Message) para assegurar a saúde dos componentes de processamento de mensagens”

### Diagrama



### Descrição

Mensagens de Teste são mensagens que circulam pelo sistema coletando dados que serão usados no final para dar um diagnóstico sobre a saúde de cada componente do sistema. A solução depende de uma infraestrutura que consiste de:

Um gerador de dados de teste, que cria as mensagens que serão enviadas ao sistema para testar.

Um injetor de mensagens de teste, que envia dados de teste no meio das mensagens que são enviadas para o componente. O injetor tem o papel de marcar as mensagens de forma que elas sejam reconhecidas pelo sistema como sendo mensagens de teste. Pode ser um cabeçalho especial que identifique tais mensagens, ou um valor especial usado em cabeçalhos existentes.

Um separador de mensagens de testes que extrai as mensagens de teste do fluxo. Pode ser um Roteador Baseado em Conteúdo. Pode ser desnecessário se as mensagens tiverem informações de roteamento presentes no cabeçalho (ex: Endereço de Retorno).

Um verificador de dados de teste que compara os resultados coletados com os resultados esperados e gera um relatório com o diagnóstico final.

Os componentes do sistema devem ser projetados de tal maneira a reconhecer mensagens de teste e ignorá-las em operações que geram efeitos colaterais, ou em componentes *stateful*.

## Aplicações

Ambiente de testes e monitoração ativa. Uma implementação do padrão Mensagem de Teste pode ser usada para realizar teste ativo do sistema (quando for necessário obter mais do que dados do sistema, uso de CPU, número de mensagens processadas, tempo de processamento, etc.)

### Mensagem de Teste em Java / JMS

Pode-se usar um framework de testes como JUnit para comparar valores esperados com valores obtidos. Mensagens de teste podem ser redirecionadas a canais especializados onde componentes que irão comparar seu Histórico, cabeçalhos e payload com valores esperados produzindo um resultado PASS ou FAIL.

### Mensagem de Teste em Apache Camel

Camel fornece três componentes, configuráveis via endpoints, para mensagens de teste. O componente Mock é usado para testar mensagens recebidas através de asserções. O componente Test estende Mock para receber os formatos de mensagens esperadas, que serão comparadas com as mensagens de teste, em outro canal. O componente DataSet é usado para obter conjuntos de dados de um repositório. Os componentes podem ser usados com canais comuns, internos ou externos, ou com o canal Stub, que serve para representar um canal externo que não está disponível no momento dos testes.

A interface DataSet facilita a criação de conjuntos de dados usados para testes. Para usar, deve-se implementar a Interface e seus quatro métodos para retornar o tamanho do DataSet, ou estender a classe DataSetSupport. O DataSet deve ser configurado como um bean e guardado no Registry do Spring. Camel já fornece uma implementação pronta na classe SimpleDataSet que pode ser usada para gerar mensagens de teste estáticas. Por exemplo:

```
<bean id="teste" class="org.apache.camel.component.dataset.SimpleDataSet">
    <property name="defaultBody" value="<teste>Isto é um teste</teste>" />
    <property name="size" value="100" />
</bean>
```

Pode ser usado da forma:

```
from("dataset:teste").to("jms:queue:a")
from("jms:queue:a").to("dataset:teste")
```

Mock pode ser usado para realizar testes unitários, especificando asserções que devem ser cumpridas para que o teste passe como se o número correto de mensagens foi recebido, se a ordem está correta, se os cabeçalhos combinam com um predicado ou seletor, se o corpo é validado por um esquema, ou

uma expressão XPath, etc. O endpoint mock pode ser incluído normalmente em uma rota como qualquer outro endpoint:

```
from("jms:queue:inbound").to("mock:teste-inbox");
```

Os testes podem ser feitos com expressões realizadas em cada mensagem recebida (indexadas em ordem de chegada) ou usando métodos de MockEndpoint criados para determinar as expectativas das asserções: expectedMessageCount(...), expectedHeaderReceived(...), expectsNoDuplicates(...), etc.

```
MockEndpoint testEndpoint =
    context.resolveEndpoint("mock:teste-inbox", MockEndpoint.class);

testEndpoint.expectedMessageCount(3);
testEndpoint.message(0).header("Type").isEqualTo("xml");
testEndpoint.expectedBodiesReceived("<teste>1</teste>", "Teste 2",
    "<teste>3</teste>");
```

No final chama-se assertIsSatisfied() para verificar se as expectativas foram cumpridas:

```
testEndpoint.assertIsSatisfied();
```

Endpoints Mock mantém as mensagens na memória para sempre. Existem opções (retainFirst, retainLast) para especificar a política de retenção, e eliminar mensagens durante os testes. Após os testes um Purificador de Canal (Channel Purger) pode ser usado para esvaziar os canais de testes.

Test é um componente Mock que obtém as mensagens esperadas de outro canal. Isto é útil se as mensagens esperadas estão armazenadas em outro lugar (arquivo, banco de dados). Por exemplo, se a aplicação espera receber um payload idêntico ao arquivo /testdata/teste1.xml pode-se usar:

```
from("jms:queue:inbound").to("test:file:/testdata/teste1.xml");
```

Stub é um canal similar a SEDA que serve para criar uma versão mock de um canal. Pode ser usado como prefixo de qualquer endpoint real. Por exemplo, para redirecionar para um stub em vez de enviar para a fila JMS real, em um ambiente de testes, pode-se usar:

```
to("stub:jms:queue:outbox")
```

## Mensagem de Teste em Spring Integration

Pode-se usar JUnit para testar, mas Spring Integration fornece um pacote que facilita testes e evita a necessidade de casting e desempacotamento de cabeçalhos e payloads. Para usá-lo é preciso incluir suporte ao artifact spring-integration-test (via Maven).

O PayloadMatcher contém métodos que podem ser usados para comparar payloads, e vários métodos existem para comparar cabeçalhos. Pode-se construir testes como:

```
@Test
public void xmlFileShouldHaveXmlHeader() {
```

```

Message testMessage =
    MessageBuilder.withPayload("<product>1</product>")
        .setCorrelationId("A01").build();

inputChannel.send(testMessage);
Message reply = outputChannel.receive();
assertThat(outputChannel.receive(), hasPayload(product1XMLcontents));
assertThat(reply, hasCorrelationId("A01"));
assertThat(reply, hasHeaderKey("ObjectType"));
assertThat(reply, hasHeader("ObjectType", Product.class));
}

```

Spring Integration também suporta mocks e stubs aplicados a canais e componentes usando Mockito.

## (65) Purificador de canal (Channel Purger)

### Ícone



### Problema

“Como impedir que sobras de mensagens em um canal atrapalhem testes ou sistemas em execução?”

### Solução

“Use um Purificador de Canal (Channel Purger) para remover mensagens indesejadas de um canal”

### Diagrama



### Descrição

Um Purificador de Canal (Channel Purger) é um filtro que remove todas as mensagens de um canal. É usado para limpar o ambiente (fazer um clean) antes de testes ou de uma execução inicial do sistema.

Se houver mensagens importantes no canal que não podem ser excluídas, elas podem ser guardadas (ex: em um Message Store) para posterior verificação e reintrodução no canal.

## Aplicações

Um Purificador de Canal deve ser instalado em ambientes que fazem testes ativos (envio de mensagens de testes pelas rotas de integração) para permitir a neutralização do ambiente antes dos testes.

### Purificador de Canal em Spring Integration

A classe `org.springframework.integration.channel.ChannelPurger` permite remover todas as mensagens de um ou mais canais. O `ChannelPurger` também pode ser inicializado com um seletor (`MessageSelector`) para selecionar quais mensagens serão removidas.

```
ChannelPurger purger1 = new ChannelPurger(channel);

ChannelPurger purger2 = new ChannelPurger(channel1, channel2, channel3);

ChannelPurger purger3 = new ChannelPurger(new MessageSelector() {
    public boolean accept(Message<?> message) {
        return message.getHeaders().get("TestMessage") == false;
    }
}, channel);

List<Message<?>> purgedMessages = purger3.purge();
```

## Revisão

Padrões de integração de sistemas relacionados a gerenciamento do sistema:

(58) *Barramento de controle (Control Bus)*: um componente que publica mensagens de controle que são recebidas por todos os componentes, e que pode ser usado para monitorar o sistema, coletar dados, ligar e desligar recursos de teste, etc.

(59) *Desvio (Detour)*: uma rota paralela usada para desviar o fluxo de mensagens para outros componentes, geralmente usado para testes.

(60) *Escuta (Wire Tap)*: um componente que lê mensagens de um Canal Ponto-a-Ponto sem consumir suas mensagens.

(61) *Histórico da Mensagem (Message History)*: o histórico contendo uma lista de componentes e/ou canais por onde uma mensagem passou.

(62) *Repositório de Mensagens (Message Store)*: um repositório contendo todas as mensagens que passaram por um componente e/ou canal.

(63) *Proxy inteligente (Smart Proxy)*: um interceptador de mensagens Requisição-Resposta.

(64) *Mensagem de Teste (Test Message)*: uma mensagem construída para coletar dados que serão usados para realizar testes no sistema.

(65) *Purificador de Canal (Channel Purger)*: um consumidor que remove todas as mensagens de uma fila.

# Apêndice A

# Ambiente

## Apache ActiveMQ

Apache ActiveMQ é um dos mais populares servidores de mensageria. É um projeto de código aberto lançado sob a licença Apache 2.0. Suporta vários protocolos e clientes em diversas linguagens. Oferece suporte total aos padrões JMS 1.1 e J2EE 1.4 podendo ser integrado aos principais servidores de aplicação. Está disponível para download em [activemq.apache.org](http://activemq.apache.org), mas pode ser instalado através de serviços como apt-get / wget em Linux ou homebrew em MacOS.

O código de exemplos e exercícios disponibilizados estão pré-configurados para uso de ActiveMQ, mas eles devem funcionar em qualquer servidor compatível com JMS como HornetMQ, RabbitMQ, IBM MQ, etc. necessitando apenas configurá-los de acordo com cada servidor.

As instruções abaixo referem-se à instalação e configuração do ambiente usando ActiveMQ 5.x.

Uma vez instalado, a partir do diretório de instalação execute:

```
activemq start
```

Para acessar a interface de administração abra um browser em <http://localhost:8161/admin/>. O login/senha é *admin/admin*.

Para parar o servidor use:

```
activemq stop
```

A porta default do serviço (message broker) é 61616. Use `tcp://localhost:61616` para configurar clientes que precisam acessar o serviço, como clientes JNDI e JMS.

Para incluir suporte ao ActiveMQ em um projeto Java, a forma mais simples de baixar os JARs necessários é usar um serviço como Maven. A dependência abaixo baixa todos os JARs necessários para rodar o cliente:

```
<dependency>
    <groupId>org.apache.activemq</groupId>
    <artifactId>activemq-all</artifactId>
    <version>5.12.0</version>
</dependency>
```

Para configurar uma ConnectionFactory JMS em Spring usando o ActiveMQ, registre um bean com o ActiveMQConnectionFactory e a URL do serviço como propriedade:

```
<bean id="jmsFactory" class="org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL">
        <value>tcp://localhost:61616</value>
    </property>
</bean>
```

Nos exemplos JMS usados neste livro o acesso ao ActiveMQ sempre foi feito através das interfaces do JMS, que foi configurado através de JNDI. A instrução Java:

```
Context jndi = new javax.naming.InitialContext();
```

procura no Classpath por um arquivo jndi.properties que será usado para configurar o contexto JNDI e permitir a localização do ConnectionFactory (para criar conexões) e dos destinos (filas e topics). Neste arquivo duas linhas são essenciais para configurar a conexão:

```
java.naming.factory.initial =
org.apache.activemq.jndi.ActiveMQInitialContextFactory
java.naming.provider.url = tcp://localhost:61616
```

Para configurar topics e queues a sintaxe é:

```
<tipo-de-canal>.<nome-jndi> = <nome-activemq>
```

Ou seja, para configurar duas filas identificadas no ActiveMQ pelos nomes queue-a e queue-b, mas que seriam localizadas em JNDI usando fila1 e fila2, respectivamente, use:

```
queue.fila1 = queue-a
queue.fila2 = queue-a
```

Em Java pode-se agora acessar o queue-a usando:

```
Queue fila1 = jndi.lookup("fila1");
```

Para configurar um topic, use topic como prefixo. O exemplo abaixo registra um topic que tem um nome ActiveMQ igual ao nome JNDI:

```
topic.canal-difusao = canal-difusao
```

Veja mais sobre ActiveMQ em [activemq.apache.org](http://activemq.apache.org).

## Apache Camel

O framework Apache Camel foi apresentado no Capítulo 3 dentro do contexto dos padrões de integração de sistemas. Nesta seção algumas informações sobre configuração e componentes nativos.

A maneira mais fácil de integrar Camel em um projeto é usar uma ferramenta como Maven. A dependência abaixo deve ser configurada no POM de um projeto Maven para suportar Camel (a versão pode ser outra):

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-core</artifactId>
    <version>2.15.3</version>
</dependency>
```

Componentes adicionais que não fazem parte do pacote Core também precisam ser declarados se forem usados. Por exemplo para usar o componente JMS inclua:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-jms</artifactId>
    <version>2.15.3</version>
    <type>jar</type>
</dependency>
```

Para usar Camel é preciso criar um contexto *CamelContext*, configurar endpoints e componentes se necessário, configurar rotas e iniciar o contexto. O exemplo abaixo (da documentação) mostra a criação de um *DefaultCamelContext* (1), a configuração de endpoints “jms” (2), a configuração de uma rota (3), a inicialização do contexto (4), a criação de um *ProducerTemplate* para construir mensagens que serão enviadas para uma fila (5) e a criação e envio de mensagens através de do template (6):

```
01. CamelContext context = new DefaultCamelContext();
02.
03. ConnectionFactory cf
     = new ActiveMQConnectionFactory("tcp://localhost:61616");
04.         context.addComponent("jms",
JmsComponent.jmsComponentAutoAcknowledge(cf));
05.
06. context.addRoutes(new RouteBuilder() {
07.             public void configure() {
08.                 from("test-jms:queue:test.queue").to("file://test");
09.             }
10.         });
11.
12. context.start();
13.
14. ProducerTemplate template = context.createProducerTemplate();
15.
16. for (int i = 0; i < 10; i++) {
```

```

17.           template.sendBody("jms:queue:test.queue", "Test Message: " +
i);
18.       }

```

O Camel também pode ser configurado através de Spring. O arquivo de configuração de beans do Spring precisa conter o namespace e localização do schema do Camel como indicado abaixo. As rotas podem ser configuradas em um elemento <route> dentro do <camelContext>:

```

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd
           http://camel.apache.org/schema/spring
           http://camel.apache.org/schema/spring/camel-spring.xsd">

    <camelContext xmlns="http://camel.apache.org/schema/spring">
        <route>
            <from uri="direct:two"/>
            <to uri="log:two"/>
            <to uri="mock:result"/>
        </route>
    </camelContext>
</beans>

```

Camel fornece canais nativos que rotam no CamelContext. Eles não implementam nenhum tipo de persistência (se a máquina virtual falhar as mensagens serão perdidas) mas são eficientes e evitam a configuração de canais externos quando não é necessário.

- *Direct*: é um canal que representa uma ligação direta entre dois componentes (equivale a uma ponte com um Event-Driven). É uma ligação em único thread e limitada ao contexto do camel (CamelContext na mesma JVM). É útil para aplicar a arquitetura Dutos e Filtros, que garante baixo acoplamento, de forma mais eficiente e sem a necessidade de usar canais fixos externos (ex: JMS). Exemplo:

```
from("jms:queue:inbound").to("direct:canal1");
```

- *SEDA*: é uma sigla para “Staged Event-Driven Architecture” que representa uma arquitetura ativada por eventos em estágios, ou etapas, interligadas por filas, possibilitando controlar (via BlockingQueue) o fluxo de mensagens e evitando sobrecarga. SEDA usa threads diferentes para consumidor e produtor (diferentemente de Direct, que é single-threaded). A comunicação SEDA ocorre no mesmo CamelContext (na mesma máquina virtual). Exemplo”:

```
from("seda:inbound").to("direct:canal1");
```

- *VM*: é uma extensão do componente SEDA que permite comunicação entre instâncias de CamelContext rodando em máquinas virtuais diferentes (ex: aplicações distribuídas). Para

funcionar, é necessário que camel-core.jar esteja disponibilizado globalmente no sistema. O nome do canal usado pelo produtor e consumidor devem ser idênticos. Exemplo:

```
from("direct:canal1").to("vm:canal-remoto");
from("vm:canal-remoto").to("direct:canal2");
```

Camel também fornece Endpoints e Canais para testes (veja *Mensagem de Teste*, no capítulo 9).

Veja mais sobre Camel em [camel.apache.org](http://camel.apache.org).

## Spring Integration

Uma visão geral do Spring Integration foi apresentada no Capítulo 3. Nesta seção estão algumas informações sobre como configurar o ambiente para usar o framework.

Talvez a maneira mais fácil de incluir os JARs necessários em um projeto Spring Integration seja através de uma ferramenta como o Maven. Incluindo a dependência abaixo no POM o projeto terá o módulo essencial (Core) necessário para configurar componentes e rotas:

```
<dependencies>
    <dependency>
        <groupId>org.springframework.integration</groupId>
        <artifactId>spring-integration-core</artifactId>
        <version>4.2.0.RELEASE</version>
    </dependency>
</dependencies>
```

Dependências adicionais são necessárias caso o projeto utilize-se de adaptadores de canais e outros componentes que não fazem parte do Core. Por exemplo, para usar os adaptadores de canal que conectam JMS ao Spring Integration é preciso declarar no POM a dependência abaixo:

```
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-jms</artifactId>
    <version>4.2.0.RELEASE</version>
</dependency>
```

O Spring Integration faz parte do ecossistema Spring, portanto é natural que seja configurado como outros projetos do Spring (no Spring 4.0 já é possível configurar usando uma Java DSL com expressões lambda (Java 8). A configuração mais comum, portanto, é usando XML.

Componentes podem ser configurados como beans comuns do Spring (usando `<bean id="..." class="...">` e propriedades), mas o Spring Integration fornece namespaces XML que permitem que eles sejam configurados usando elementos XML, o que melhora a legibilidade dos arquivos de configuração do contexto. O módulo Core usa o namespace <http://www.springframework.org/schema/integration> e cada módulo extra deve declarar um namespace. É comum manter o Spring como namespace default, e usar “int” como prefixo do namespace do Spring Integration. Componentes extras são geralmente

declarados com prefixo “int-componente”, por exemplo “int-jms”, “int-file”, etc. O XML abaixo mostra um arquivo de configuração do Spring configurado para alguns componentes de Spring Integration:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:int="http://www.springframework.org/schema/integration"
       xmlns:int-file="http://www.springframework.org/schema/integration/file"
       xmlns:int-jms="http://www.springframework.org/schema/integration/jms"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/integration/jms
           http://www.springframework.org/schema/integration/jms/spring-integration-jms-4.0.xsd
           http://www.springframework.org/schema/integration/file
           http://www.springframework.org/schema/integration/file/spring-integration-file-
4.0.xsd
           http://www.springframework.org/schema/integration
           http://www.springframework.org/schema/integration/spring-integration-4.0.xsd">

    <int:gateway id="helloGateway"
                 service-interface="hello.SpringHello"
                 default-request-channel="channel-two" />

    <int:channel id="channel-two" />

    <int:service-activator input-channel="channel-two"
                           expression="payload + ' World'" />
</beans>
```

A configuração acima inclui um Gateway de Mensageria (veja Capítulo 8), um Canal de Mensagens (Capítulo 4) e um Ativador de Serviço (Capítulo 8). O Gateway poderia usar esta interface:

```
public interface SpringHello {
    String say(String what);
}
```

Para usar o gateway, o arquivo de configuração do contexto (beans.xml no exemplo abaixo) é tratado como registro Spring e o componente pode ser recuperado usando getBean():

```
public class HelloSpringIntegration {

    public static void main(String... args) throws Exception {
        ApplicationContext ctx =
            new ClassPathXmlApplicationContext("beans.xml");
        SpringHello test =
            ctx.getBean("helloGateway", SpringHello.class);
        System.out.println(test.say("Hello"));
    }
}
```

O resultado da execução é:

```
Hello World
```

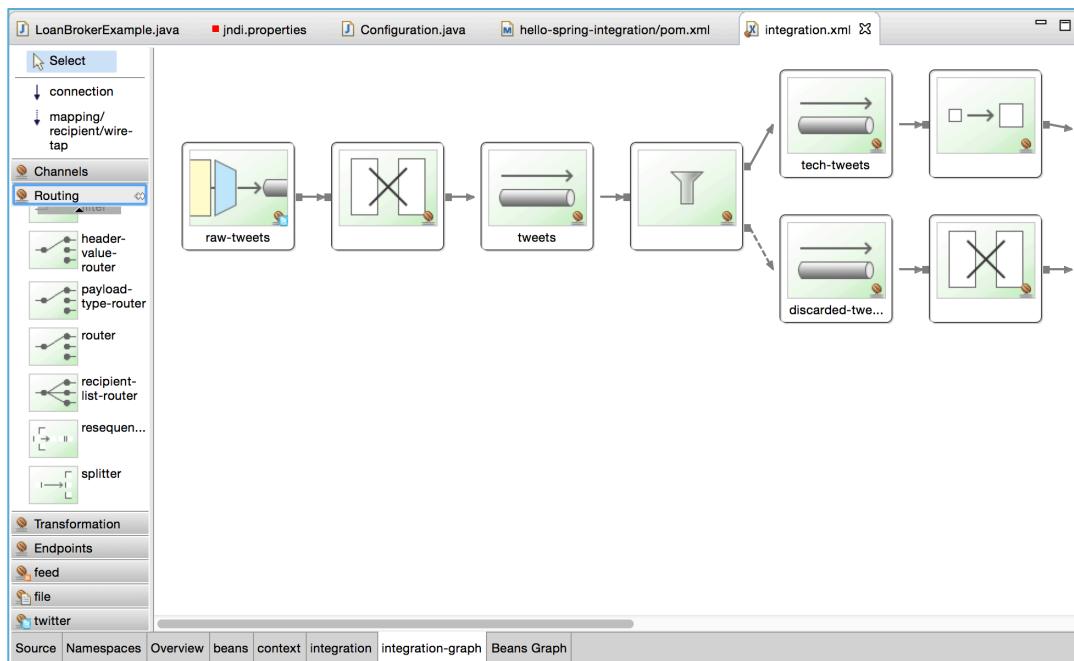
Veja mais sobre Spring Integration em <http://projects.spring.io/spring-integration/>

## Ferramentas do Eclipse

O IDE Eclipse possui duas ferramentas úteis para quem pretende construir aplicações usando Camel ou Spring Integration. Ambas são parte de pacotes maiores e incluem recursos como geração de código e modelagem usando ícones dos padrões de integração de sistemas.

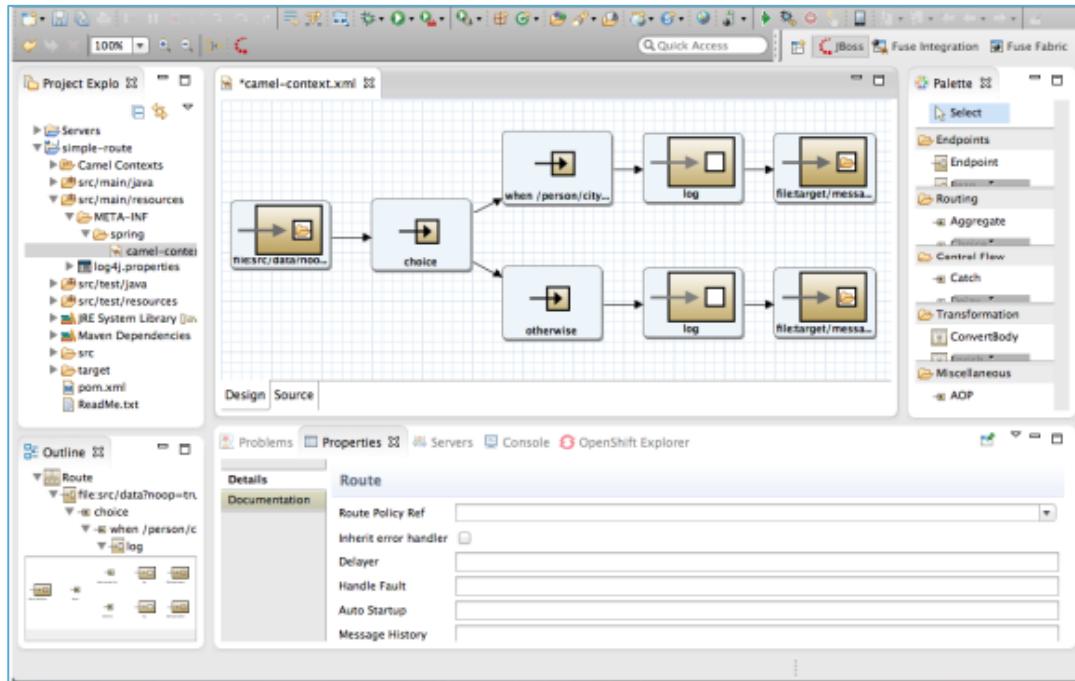
### STS

O Spring Tool Suite (STS) é um plugin Eclipse que oferece várias ferramentas para projetos Eclipse, dentre elas o suporte a Spring Integration. Além de assistentes e ajuda de contexto para API e esquemas XSD do Spring, o STS também possui um editor gráfico que mostra as rotas de um arquivo de configuração de contexto e também permite editar ou gerar rotas em XML:



### JBoss Tools

O JBoss Tools Integration Stack é parte do JBoss Tools (que precisa ser instalado antes) e oferece suporte a Camel e padrões do catálogo [EIP]. Assim como o STS também permite a geração e edição da configuração de rotas em XML via Spring usando um editor gráfico, além de assistentes e outros recursos para facilitar o desenvolvimento de aplicações Camel.



# Apêndice B

# Referências

## Livros

[EIP] Gregor Hohpe, Bobby Woolf, et al. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2004. Um resumo do livro com o catálogo de padrões está disponível em [www.eaipatterns.com](http://www.eaipatterns.com)

[GoF] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[PEAA] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003. Um catálogo online está disponível em: [martinfowler.com/eaaCatalog](http://martinfowler.com/eaaCatalog)

[POSA] Frank Buschmann et al. *Pattern-Oriented Software Architecture*. Wiley. Vol 1 (1996) & 2 (2000)

[J2EE] Deepak Alur et al. *Core J2EE Patterns: Best Practices and Design Strategies. 2nd Edition*. Prentice-Hall, 2003. Catálogo online em [corej2eepatterns.com](http://corej2eepatterns.com)

[Ibsen] Claus Ibsen. *Camel In Action*. Manning, 2011.

[Bloch] Joshua Bloch. *Effective Java, 2nd Edition*. Addison-Wesley, 2008.

[Stevens] W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, 1994.

[Kay] Michael Kay. *XSLT 2.0 and XPath 2.0 Programmer's Reference. 4th Edition*. Wrox Press, 2008.

[Daigneau] Robert Daigneau. *Service Design Patterns: fundamental design solutions for SOAP/WSDL and RESTful Web Services*. Addison-Wesley, 2012.

[Gupta] Arun Gupta, *Java EE 7 Essentials*. O'Reilly and Associates, 2013.

## Especificações

[JMS] Mark Hapner et al. *Java Message Service version 1.1*. Sun Microsystems. April, 2002.

[EJB] Kenneth Saks et al. *JSR 318: Enterprise JavaBeans, Version 3.1*. Sun Microsystems, 2009.

[JAXB] Kohsuke Kawaguchi. JSR-222 Java Architecture for XML Binding (JAXB) 2.2. Sun Microsystems, 2009.

[JAXP] Jeff Sutor et al. *JSR-206: Java API for XML Processing (JAXP) 1.3*. Sun Microsystems, 2003.

[XPath] James Clark et al. *XML Path Language (XPath) Version 1.0*. W3C Recommendation Nov 16, 1999. Disponível em [www.w3.org/TR/xpath/](http://www.w3.org/TR/xpath/)

[XSLT] James Clark et al. *XSL Transformations (XSLT) Version 1.0*. W3C Recommendation Nov 16, 1999. Disponível em [www.w3.org/TR/xslt/](http://www.w3.org/TR/xslt/)

## Artigos

[Waehner] Kai Waehner. *Integration Framework Comparison: Spring Integration, Mule ESB or Apache Camel*. Java Code Geeks, March, 2012. [www.javacodegeeks.com/2012/03/integration-framework-comparison-spring.html](http://www.javacodegeeks.com/2012/03/integration-framework-comparison-spring.html)

[Byars] Brandon Byars. Enterprise Integration using REST. Nov 18 2013. MartinFowler.com. Disponível em [martinfowler.com/articles/enterpriseREST.html](http://martinfowler.com/articles/enterpriseREST.html)

[Haines] Steven Haines. How-to: Open source Java projects: Spring Integration: Develop a robust message-passing architecture with Spring Integration. JavaWorld Magazine. April 10, 2014. Disponível em [www.javaworld.com/article/2142107/spring-framework/open-source-java-projects-spring-integration.html](http://www.javaworld.com/article/2142107/spring-framework/open-source-java-projects-spring-integration.html)

[Hohpe03] Gregor Hohpe. *An Asynchronous World*. Dr. Dobbs Magazine. July 01, 2003. Disponível em [www.drdobbs.com/an-asynchronous-world/184415001](http://www.drdobbs.com/an-asynchronous-world/184415001)

[Kolb] Pascal Kolb. *Realization of EAI Patterns with Apache Camel*. University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Student Thesis No. 2127 (2008). Disponível em [www2.informatik.uni-stuttgart.de/cgi-bin/NCCTRL/NCCTRL\\_view.pl?id=STUD-2127](http://www2.informatik.uni-stuttgart.de/cgi-bin/NCCTRL/NCCTRL_view.pl?id=STUD-2127)

[SEDA] Matt Welsh. *SEDA: An Architecture for Highly Concurrent Server Applications*. Harvard University. Updated May 2006. Disponível em [www.eecs.harvard.edu/~mdw/proj/seda/](http://www.eecs.harvard.edu/~mdw/proj/seda/)

## Documentação, tutoriais e referências diversas

[JavaEE] Eric Jendrock et. al *The Java EE 6 Tutorial. Chapter 47: Java Message Service Concepts*. Sun Microsystems, 2009. Disponível em [docs.oracle.com/javaee/6/tutorial/doc/](http://docs.oracle.com/javaee/6/tutorial/doc/)

[Camel] Apache Software Foundation. *Apache Camel. Book in One Page*. Documentação oficial do Camel em uma página. Atualizada em 2015. Disponível em [camel.apache.org/book-in-one-page.html](http://camel.apache.org/book-in-one-page.html)

[Camel-EIP] Apache Software Foundation. *Apache Camel. Enterprise Integration Patterns.* Documentação da implementação Camel dos padrões de integração de sistemas. Disponível em [camel.apache.org/enterprise-integration-patterns.html](http://camel.apache.org/enterprise-integration-patterns.html)

[Spring] Mark Fisher et al. *Spring Integration Reference Manual. 4.2.0.RELEASE.* Documentação oficial do Spring Integration. 2015. Disponível em [docs.spring.io/spring-integration/docs/4.2.0.RELEASE/reference/](http://docs.spring.io/spring-integration/docs/4.2.0.RELEASE/reference/)

[ActiveMQ] Apache Software Foundation. ActiveMQ Documentation. Disponível em [activemq.apache.org/getting-started.html](http://activemq.apache.org/getting-started.html)

[White] Jim White. *Spring Integration Tutorial.* Intertech, 2014. Ótimo tutorial passo-a-passo em 8 partes. Disponível em [www.intertech.com/Blog/spring-integration-part-1-understanding-channels/](http://www.intertech.com/Blog/spring-integration-part-1-understanding-channels/)

[DZone-Camel] Claus Ibsen. *The Top Twelve Integration Patterns for Apache Camel: implement in Java or Spring XML.* DZone Refcard #47. Disponível em [dzone.com/refcardz/enterprise-integration](http://dzone.com/refcardz/enterprise-integration)

[DZone-SI] Soby Chacko and Chris Schaefer. *Spring Integration: core components of the leading Java framework.* DZone Refcard #197. Disponível em [dzone.com/refcardz/spring-integration](http://dzone.com/refcardz/spring-integration)

[Fuse] RedHat/FuseSource. Fuse ESB Enterprise. Implementing Enterprise Integration Patterns. Version 7.0. July 2012. Disponível em [access.redhat.com/documentation/en-US/Fuse\\_ESB\\_Enterprise/7.0/](http://access.redhat.com/documentation/en-US/Fuse_ESB_Enterprise/7.0/)

# Apêndice C

# Exercícios

## Capítulo 3 – Mensageria

1. [Arquitetura] Uma aplicação local produz listas de preços em formato CSV que são depositados em uma pasta. Outra aplicação produz listas em formato XML que são enviadas para uma URL. Uma terceira aplicação produz listas em formato HTML (tabelas) também disponíveis em uma URL. Você precisa elaborar uma solução de integração que construa listas de produtos com preços obtidos das três listas em formato HTML. A solução não precisa fazer nada além de listar produtos e preços. Esboce uma solução usando padrões de integração.
2. [Message, Channel] A classe **MockData** contém um List com 10 documentos XML em formato String. São dados de teste. O método estático **getMockData()** retorna esta lista. Escreva um Produtor JMS que envie esses documentos XML para a fila “**inbound**” do ActiveMQ. Você vai precisar construir uma mensagem. A mensagem construída deve conter um cabeçalho “**Tipo**” com o valor “**xml**”. As configurações do ActiveMQ já estão prontas no arquivo **jndi.properties** (as filas do ActiveMQ têm o mesmo nome JNDI). Configure seu ambiente e verifique que o ActiveMQ recebe as mensagens na fila.
3. [Message, Channel] Escreva uma classe Java que consuma as mensagens da fila “**inbound**” e imprima seu conteúdo na saída padrão se elas tiverem um cabeçalho “**Tipo**” com o valor “**xml**”.
4. [Endpoints] A classe **FileInboundAdapter** lê arquivos da pasta **INBOX** e envia para a fila “**inbound**”. A classe **FileOutboundAdapter** consome mensagens de “**inbound**” e grava em arquivo na pasta **OUTBOX**. Configure os valores de **INBOX** e **OUTBOX** na sua máquina. Copie (não mova) os arquivos da pasta **arquivos** para a pasta **INBOX** e execute o **FileInboundAdapter**. Verifique na console do ActiveMQ se as mensagens estão na fila “**inbound**”. Alguns desses são XML, portanto se você rodar o exercício 3 deve conseguir imprimir seu conteúdo.
5. [Pipes and Filters] Copie novamente arquivos para a pasta **INBOX** e rode o **FileInboundAdapter**. Agora deve haver vários arquivos na fila “**inbound**”. Rode o

**FileOutboundAdapter.** Ele está esperando mensagens na pasta “**outbound**”, mas eles não estão lá. Elabore uma solução para integrar os dois. Pode ser em Camel, Spring Integration ou em JMS (use a classe **JMSChannelBridge**).

6. [Router] Escreva um roteador que analise o cabeçalho “**Tipo**” de cada mensagem de uma fila e redirecione para um destino diferente. Os tipos possíveis são “**png**”, “**xml**” e “**txt**”. Os destinos podem ser “**png-queue**”, “**xml-queue**” e “**txt-queue**”. Use JMS, Camel ou Spring Integration. Altere a classe que você criou no exercício 3 para que ela consuma das filas “**xml-queue**” e “**txt-queue**” e o **FileOutboundAdapter** para que consuma da fila “**png-queue**”
7. [Translator] A classe **JMSChannelBridge** simplesmente consome mensagens de um canal e copia em outro. Use-a como template para criar um **UpperCaseTransformer** que transforme mensagens de entrada em caixa-alta na saída. Conecte o componente para receber mensagens do “**txt-queue**”. Se quiser implemente em Camel ou Spring Integration.

## Capítulo 4 – Canais

1. [Arquitetura, Ponto-to-Point, Datatype Channel] Identifique que padrões de canais foram usados nos exercícios do capítulo anterior. Existe algum Datatype Channel? Point-to-Point Channel? Publish-Subscribe Channel? Alguma Ponte de Mensageria? Existe algum Adaptador de Canal? Identifique-os.
2. [Arquitetura, Publish-Subscribe] No capítulo anterior construímos uma solução de integração usando canais Ponto-a-Ponto que transferia documentos XML e texto para uma aplicação que imprimia o seu conteúdo, e arquivos PNG para uma pasta. Queremos agora continuar imprimindo os arquivos de texto (via GenericMessageConsumer), mas também queremos que os arquivos XML e de texto (transformados) **também** sejam enviados para a pasta **OUTBOX**. Que alterações precisamos fazer na aplicação? Desenhe a solução. Use a classe **MessagingBridge** para implementar quaisquer rotas to-from com **JMSChannelBridge**. Crie um ou mais topics se necessário. Você talvez precise alterar os canais usados por **FileOutboundAdapter** e **GenericMessageConsumer**. Se desejar, implemente a solução usando Camel.
3. [Guaranteed Delivery] Transfira mensagens para uma fila qualquer (rode por exemplo a classe que você escreveu no exercício 2 do capítulo anterior ou **MockMessageProducer**). Reinicie o ActiveMQ. Verifique se as mensagens ainda estão na fila. Se estiverem, o seu provedor foi configurado para usar *Guaranteed Delivery* por default. Nesse caso, configure suas mensagens com **DeliveryMode.NOT\_PERSISTENT**, e veja que elas não são mantidas após o restart. Se elas não estiverem mais disponíveis, faça o contrário, usando **DeliveryMode.PERSISTENT** para verificar que elas continuam no canal, mesmo após o restart.
4. [Invalid-Message Channel] Altere o Roteador que foi desenvolvido no exercício 3.6 do capítulo anterior (**JMSMessageHeaderRouter**) para que ele envie arquivos que não são PNG, XML ou TXT para a fila “**invalid-queue**”, de forma que a fila “**inbound**” permaneça sempre vazia após o roteamento.

5. [DLQ] Crie um componente intermediário que mude o Tempo de Vida das mensagens que passam por ele para zero. Inclua ele na sua rota e veja se mensagens são redirecionadas para o DLQ do ActiveMQ (Use Camel, Spring Integration ou o **JMSChannelBridge**). Uma solução semi-pronta está disponível em **ExpirationBridge** (envie para a fila inbound e espere a mensagem ser movida para a DLQ).
6. [Arquitetura] Que alterações precisariam ser feitas para um componente possa imprimir o nome e tamanho dos arquivos que estão na fila de arquivos (“**file-queue**”)? Como plugar este componente e continuar copiando os arquivos para a pasta **OUTBOX**?

## Capítulo 5 – Mensagens

1. [Mensagem-evento, Prazo de Validade] Transforme o **FileInboundAdapter** do capítulo anterior em um listener de eventos. Ele só deve começar a buscar arquivos na pasta **INBOX** quando receber uma notificação. A classe que constrói a notificação e envia deve criar uma mensagem com Prazo de Validade de 10 segundos. Ou seja, se o **FileInboundAdapter** não estiver no ar dentro desse prazo, a mensagem deve ser removida da fila para o DLQ pelo sistema.
2. [Requisição-Resposta, Mensagem-comando, Endereço de Resposta, ID de correlação] Um processador remoto de comandos recebe Mensagens em formato XML:

```
<command>
  <method class='CLASSE' name='METODO'>
    <params>
      <CLASSE>PARAMETRO</CLASSE>
      <CLASSE>PARAMETRO</CLASSE>
      ...
    </params>
  </method>
</command>
```

Por exemplo, o comando:

```
<command>
  <method class='com.exemplo.Operacoes' name='repetir'>
    <params>
      <java.lang.String>Hello World</java.lang.String>
      <java.lang.Double>7.0</java.lang.Double>
    </params>
  </method>
</command>
```

equivale ao método `Operacoes.repetir(String, Double)` (estático ou de instância). A resposta também é devolvida em formato XML.

```
<result><CLASSE>RESULTADO</CLASSE></result>
```

A classe pode ser o tipo de retorno ou a exceção. Utilize como base a classe **RequestingClient** que envia um comando e recebe uma resposta e construa uma solução que envie pelo menos quatro

requisições da classe **Operation** com valores diferentes. Imprima cada Requisição seguida de sua Resposta (como garantir isto?).

3. [Indicador de Formato] O arquivo **command.xsd** descreve o arquivo XML usado como Mensagem-comando nos exercícios anteriores. Descreva como você poderia realizar a validação das mensagens da requisição e resposta. Onde seriam colocados os componentes? Como a validação seria feita? Para onde devem ser enviadas as mensagens que não validarem?
4. [Arquitetura, Sequencia de mensagens] Que propriedades são necessárias para transmitir e processar uma sequencia de mensagens? Se a ordem das mensagens do grupo não for importante, quais podem ser omitidas? O que deve ser feito se todas as mensagens da sequencia não chegarem?

## Capítulo 6 – Roteadores

1. [CBR] Construa um **CBR** para separar mensagens XML de esquemas diferentes. Use uma expressão XPath para identificar o arquivo pelo conteúdo (ex: em `<a><b>texto</b></a>` para comparar se elemento **<b>** contém “texto” use `/a/b[text() = 'texto']`; para comparar se **<a>** tem um elemento filho **<b>** use `/a[name(*) = 'b']`
2. [Filtro de Mensagens] Construa uma solução equivalente à solução acima usando **Filtros de Mensagens**.
3. [Roteador Dinâmico] Crie um **roteador dinâmico** que envie mensagens para filas diferentes de acordo com um algoritmo aleatório (ex: `(int)(Math.random() * 3)` retorna 0, 1 ou 2)
4. [Lista de Receptores] Use uma **lista de receptores** para implementar um roteador equivalente aos roteadores dos exercícios 1 e 2, selecionando as rotas com base no conteúdo das mensagens em uma etapa anterior. Envie os arquivos também para a fila de saída.
5. [Divisor] Use o exemplo de **Sequencia de Mensagens** do capítulo anterior para implementar um **Splitter**. Guarde as informações necessárias nos cabeçalhos de cada mensagem para que seja possível reordenar a mensagem posteriormente.
6. [Agregador] Escreva um **agregador** que guarde a mensagem que tenha a linha mais longa do canal para onde foram enviadas as mensagens do exercício anterior.
7. [Agregador] Escreva um **agregador** para combinar todas as mensagens que o **Splitter** do exercício 5 enviou para a fila. Não se preocupe com a ordem.
8. [Resequenciador] Insira um **re-sequenciador** na rota acima para que os poemas seja reconstruído na ordem correta.
9. [Arquitetura, Roteador de Mensagens Compostas] Desenhe um **roteador de mensagens compostas** para o exercício anterior, que trate cada linha par do poema e ponha o texto em caixa-alta.

10. [Arquitetura, Espalha-Recolhe] Desenhe um **Espalha-Recolhe** que selecione a mensagem com o menor tamanho.

## Capítulo 7 – Tradutores

1. [Enriquecedor de Conteúdo] O documento **arquivos.xml** contém uma lista de dois arquivos. Use esse documento para carregar os dois arquivos e inclui-los na mensagem.
2. [Filtro de Conteúdo] Use o **XPathFilter** ou outro processador para remover os nós-filho desnecessários de uma mensagem que transporta o arquivo sol.xml. As mensagens que serão usadas precisam apenas do nome de cada planeta.
3. [Recibo] Adapte o roteador de arquivos que você desenvolveu no capítulo 3 para que a fila que recebe mensagens PNG substitua o conteúdo da mensagem por uma referência (nome ou caminho do arquivo) que outra parte da aplicação possa usar para recuperá-lo. Não precisa recuperar o arquivo, mas a referência deve ser impressa no destino.
4. [Normalizador, Modelo de Dados Canônico] Desenhe um **Normalizador** para lidar com as diferentes maneiras de registrar usuários usados por uma aplicação. Existem usuários que são um CSV com dois campos, e outros que são XMLs em esquemas diferentes, como listado abaixo:
  - a. <users>
 

```
<user>
            <name>Albert                           Einstein</name>
            <email>einstein@eismc2.org</email>
          </user>
          <user>...</user>
        </users>
```
  - b. <usuarios> xmlns="ns1">
 

```
<usuario><nome>Albert    Einstein</nome><email>einstein@eismc2.org</email>
          </usuario>
        </usuarios>
```
  - c. <users>
 

```
<user>
            <lastname>Einstein</lastname><firstname>Albert</firstname>
            <email>einstein@eismc2.org</email>
          </user>
          <user>...</user>
        </users>
```
  - d. Albert Einstein, einstein@eismc2.org
 Alan Turing, turing@computer.net
 ...

Estabeleça um cabeçalho padrão que informe o tipo original dos dados e guarde-os em um envelope ou usando um **Recibo**.

## Capítulo 8 – Terminais

1. Você precisa projetar um consumidor que irá processar diferentes tipos de mensagens. Você também tem à disposição processadores especializados para cada tipo. Rodando em paralelo com outros cem processadores, cada processador leva em média um segundo para processar cada mensagem. A taxa de chegada de mensagens é de 100 mensagens por segundo, ou seja, o sistema opera no limite da sobrecarga. Marque, dentre as opções abaixo, os padrões que você empregaria. Por que?
  - a. Consumidor de Sondagem (Polling Consumer)
  - b. Consumidor Ativado por Evento (Event-Driven Consumer)
  - c. Consumidores Concorrentes (Competing Consumers)
  - d. Despachante de Mensagens (Message Dispatcher)
2. Em um sistema de integração há um componente que seleciona as mensagens de acordo com a hora em que foram enviadas. Essa informação está guardada no cabeçalho de cada mensagem “Data-Envio”. O componente redireciona cada mensagem diretamente a processadores especializados dependendo da hora do dia. Que padrão é representado por este componente?
  - a. Filtro de Conteúdo (Content Filter)
  - b. Filtro de Mensagens (Message Filter)
  - c. Consumidor Seletivo (Selective Consumer)
  - d. Roteador Baseado em Conteúdo (Content-Based Router)
3. Marque verdadeiro ou falso
  - a. Mensagens persistentes enviadas para canais ponto-a-ponto permanecem disponíveis até que expirem, sejam consumidas ou removidas do canal pelo provedor
  - b. Mensagens persistentes enviadas para canais de difusão são entregues imediatamente a todos e apenas aos assinantes e consumidores que estiverem configurados para receber mensagens do canal no momento do envio;
  - c. Mensagens não-persistentes são mantidas em meio não persistente e poderão não serem entregues se o servidor sair do ar temporariamente, mesmo que os canais sejam ponto-a-ponto.
  - d. Mensagens persistentes serão perdidas se enviadas para canais de difusão quando a conexão ainda não tiver iniciado o consumo de mensagens, mas elas serão preservadas

se enviadas para canais ponto-a-ponto, mesmo que o consumo de mensagens ainda não tenha iniciado.

- e. Um receptor idempotente é um consumidor que filtra mensagens duplicadas.
- 4. [Ativador de Serviço] Use Camel ou Spring Integration para configurar um **Ativador de Serviço** para o exemplo mostrado em JMS (ProductService)

## Capítulo 9 – Gerenciamento

1. [Barramento de Controle, Desvio] Configure a rota que transfere arquivos para que ela envie informações de status ao **Barramento de Controle**. Configure um **desvio** que imprima nome e tamanho de cada arquivo e faça com que o desvio possa ser ativado pelo **Barramento de Controle**.
2. [Escuta] Inclua uma **Escuta** na rota que transfere arquivos e desvie-os para uma pasta local.
3. [Histórico] Configure os componentes da rota de arquivos para que gravem o **Histórico de Mensagens**. Imprima o histórico nos endpoints de saída.

## Extra – estudos de caso

Escolha dois projetos abaixo e esboce uma solução genérica usando padrões de integração. Escolha um deles e implemente uma prova de conceito, demo ou funcionalidade usando algum framework de mensageria.

- a) Uma loja virtual que vende livros, controla clientes, pedidos, fornecedores, sistema de estoque de produtos, com um sistema de acompanhamento do pedido
- b) Balsa que embarca carros, pessoas, ônibus, containers
- c) Pesquisa Web que solicita preço de passagem aérea e retorna o menor preço
- d) Banco de informações local que sincroniza com sistema externo diariamente (orbitas)
- e) Sistema que publica resultado de eleições em tempo real
- f) Sistema que distribui tarefas em calendário e as executa ou reagenda (um número determinado de vezes) quando chega a data.
- g) Linha de produção
- h) Filtro de dados de geolocalização
- i) Enricher de dados de geolocalização
- j) Um serviço de transporte (agregador de passageiros)

k) Um serviço de envio de notificações, anúncios, notícias