

# ***Integração de Sistemas Com Apache Camel***

Otávio R. Piske

# Índice

Introdução.....	5
Apache Camel.....	6
Conceitos.....	6
Rotas.....	6
Componentes e Endpoints.....	6
DSL.....	7
Contexto.....	8
Usando o Apache Camel.....	8
Inicialização.....	8
Configuração dos Componentes.....	10
Apache Active MQ.....	10
IBM WebSphere MQ.....	12
Definindo Rotas.....	14
Processadores.....	16
Gerenciando Excessões na Rota.....	17
Conversores de Tipos.....	18
Padrões de Integração de Sistemas.....	20
Estilos De Integração.....	20
Transferência de arquivos (file transfer).....	20
Banco de dados compartilhado (shared database).....	20
Invocação remota de procedimento (remote procedure invocation).....	21
Mensageria (messaging).....	21
Sistemas de mensageria.....	21
Canais (message channel).....	22
Mensagens (message).....	22
Tubos e filtros (pipes and filters).....	22
Implementação no Camel.....	23
Roteamento de mensagens (message routing).....	23
Tradutor (message translator).....	23
Terminadores (message endpoint).....	23
Canais de mensageria.....	24

Canal ponto a ponto (point-to-point channel).....	24
Publicação/assinante (publish/subscribe).....	24
Canal por tipo de dado (datatype channel).....	25
Canal de mensagem inválida (invalid message channel).....	25
Canal “dead letter” (dead-letter channel).....	26
Entrega garantida (guaranteed delivery).....	26
Adaptador de canal (channel adapter).....	27
Ponte de mensageria (messaging bridge).....	27
Barramento (message bus).....	27
Construção de mensagens.....	28
Mensagem de comando (command message).....	28
Mensagem de documento (document message).....	28
Mensagem de evento (event message).....	29
Requisição e resposta (request and reply).....	29
Endereço de retorno (return address).....	30
Identificador de correlação (correlation identifier).....	30
Sequência de mensagem (message sequence).....	31
Expiração de mensagem (message expiration).....	31
Identificador de formato (format identifier).....	32
Roteamento de mensagens.....	32
Roteador baseado em conteúdo (content-based router).....	32
Filtro de mensagem (message filter).....	32
Roteador dinâmico (dynamic router).....	33
Lista de receptores (recipient list).....	33
Divisor (splitter).....	34
Agregador (aggregator).....	34
Resequenciador (re-sequencer).....	34
Processador de mensagens compostas (composed message processor).....	35
Dispersão e recolhimento (scatter-gather).....	35
Lista de circulação (routing slip).....	36
Gerenciador de processos (process manager).....	36
Corretor de mensagens (message broker).....	36
Transformação de Mensagens.....	37
Enriquecedor de conteúdo (content enricher).....	37

Filtro de conteúdo (content filter).....	38
Gerenciamento de sistemas.....	38
Escuta (wire tap).....	38
Outros.....	38
Licenças.....	41
Referências.....	42

# Introdução

Nos últimos anos a demanda dos clientes por serviços diferenciados, bem como o aumento da concorrência, aumentou a pressão sobre as organizações para prover serviços cada vez mais integrados. Além disso, as oportunidades apresentadas por uma internet cada vez mais “social” trazem uma série de novos desafios para as companhias.

A consolidação de sistemas e a integração de organizações, estão entre as maneiras encontradas pelos executivos de negócios para atingir o objetivo de atender seus clientes e aproveitar as oportunidades apresentadas pela internet social.

Esse novo panorama de negócios, por consequência, adiciona ao trabalho de desenvolvedores e arquitetos requerimentos e preocupações referentes a integração entre sistemas. Anteriormente focava-se no desenvolvimento de aplicações isoladas – tanto em desktop quanto em web – ou, em casos mais complexos, aplicações em n-camadas. Hoje, contudo, há a necessidade de desenvolvê-las de tal forma que enfoque se dá, não nas funcionalidades, mas também nos serviços fornecidos pelas aplicações.

Neste e-book será apresentado o Apache Camel, um *framework* de código aberto, baseado em padrões de integração de sistemas (*Enterprise Integration Patterns* – EIP).

# Apache Camel

O Apache Camel é um *framework* Java para desenvolvimento regras de mediação e roteamento baseado nos padrões de integração de sistemas. O Apache Camel é um software livre desenvolvido pela fundação Apache e disponibilizado sob a licença *Apache License 2.0*.

## Conceitos

### Rotas

O núcleo do funcionamento do Camel está na declaração de rotas que consistem na declaração de 1 ou mais *endpoints* cujo esquema de ligação é definido de acordo com um dos padrões de integração.

Para declarar uma rota é necessário especializar a classe *RouteBuilder* e implementar o esquema de ligação entre os *endpoints* no método *configure*.

### Componentes e *Endpoints*

A ligação do Camel com os *endpoints* é feita através de componentes. Os componentes são a implementação de mais baixo nível do meio utilizado para a troca de mensagens através do Camel. A diferença entre um *endpoint* e um componente é que enquanto o primeiro declara um terminador da rota, o segundo determina qual é o protocolo ou tecnologia utilizada pelo terminador.

Os principais componentes disponibilizados pelo Camel são:

- AMQP: suporte ao protocolo AMQP.
- CXF: suporte a *web services* através do Apache CXF.

- Direct: para chamadas síncronas dentro de um mesmo contexto do Camel.
- File: para troca de dados com arquivos.
- FTP: para troca de dados através de FTP.
- HTTP: para acessar servidores HTTP remotos.
- IMAP: para receber e-mails através de IMAP.
- JMS: para troca de mensagens através de JMS.
- MINA: para suporte a *sockets* em baixo nível através do Apache Mina.
- Quartz: para troca agendada de mensagens usando o agendador Quartz.
- RabbitMQ: para troca de mensagens usando o RabbitMQ.
- SEDA: para troca de mensagens assíncrona dentro de um mesmo contexto do Camel.

Além dos componentes nativos, diversas organizações fornecem componentes nativos do Camel para seus produtos. Alguns dos componentes mais conhecidos são:

- ActiveMQ: para troca de mensagens via JMS usando o Apache ActiveMQ.
- Hibernate: para acesso a banco de dados usando o Hibernate..
- ZeroMQ: para troca de mensagens via JMS usando o Zero MQ.

O Camel fornece, ainda, a possibilidade de estender o suporte a outros componentes através da implementação de interfaces customizadas que podem ser facilmente adicionadas ao Camel.

## DSL

No Camel as rotas são declaradas usando uma linguagem de domínio específico (DSL – *Domain Specific Language*). O Camel fornece a possibilidade de criar rotas usando as seguintes DSLs:

- Java DSL
- Spring XML
- Blueprint XML
- Scala DSL
- Groovy DSL
- Annotations DSL

A totalidade dos conceitos explicados nesse livro são aplicáveis a todas ou, pelo menos, a grande maioria das DSLs suportadas pelo Camel. Por simplicidade, todas as referências e exemplos do livro são feitos utilizando a Java DSL.

## Contexto

Um contexto do Camel agrupa um conjunto de regras de roteamento. Além disso, também agrupa as instâncias de componentes e registros de conversores.

## Usando o Apache Camel

### Inicialização

Os passos para inicializar o Camel são bastante simples. O primeiro passo envolve instanciar um contexto – usualmente uma instância da classe *org.apache.camel.impl.DefaultCamelContext* que servirá para agrupar o conjunto de regras de roteamento da aplicação:

```
CamelContext camelContext = new DefaultCamelContext();
```



Em seguida, pode-se instanciar, configurar e adicionar os componentes no contexto. Para adiciona-los no contexto, usa-se o método *addComponent*. O primeiro dos parâmetros associa um nome para o componente. Desta forma, ao fazer a declaração da regra de roteamento, deve-se usar *nome:endereçamento* ou *nome://endereçamento* onde *nome* é o nome do componente e *endereçamento* é uma *String* de natureza específica ao componente que é utilizada para endereçar o local onde as mensagens são recebidas ou enviadas. Detalhes adicionais a respeito da configuração dos componentes podem ser encontrados no próximo tópico. O exemplo abaixo mostra como adicionar um componente do Apache ActiveMQ, associando-o ao nome “*activemq*”:

```
ActiveMQConfiguration aMQConfiguration = getActiveMQConfiguration();
ActiveMQComponent component = new ActiveMQComponent(aMQConfiguration);

camelContext.addComponent("activemq", component);
```

Visto que o Camel fornece diversos componentes na distribuição padrão, a adição de componentes ao contexto é um passo opcional que só precisa ser executado se a aplicação utilizar um componente externo ao Camel, como o Active MQ ou o IBM WebSphere MQ.

Neste ponto é recomendável, caso se faça necessário, adicionar os conversores customizados de tipo ao registro de conversores. Conversores de tipo serão explicados nos tópicos seguintes.

Em seguida, é possível adicionar as rotas ao contexto através do método *addRoutes*. Este método é bastante simples, recebendo somente um parâmetro – uma instância de classe que especializa a classe abstrata *org.apache.camel.builder.RouteBuilder*.

```
camelContext.addRoutes(new EvalServiceRoute(""));
```

Por fim, é possível inicializar o contexto chamando o método *start*.

```
camelContext.start();
```

É importante notar que a chamada ao método *start* não bloqueia a *thread* que faz a chamada

ao método, desta forma ao utilizar o Camel no modelo “*standalone*”, comportamento padrão pode fazer com que a aplicação termine a execução da aplicação. Uma solução simples para evitar este comportamento, é fazer com que a *thread* aguarde o término da execução do programa através da chamada ao método *join*:

```
Thread.currentThread().join();
```

## Configuração dos Componentes

O Camel oferece uma vasta gama de componentes na sua distribuição padrão. Ainda assim, muitas vezes é necessário estender o Camel para integra-lo com soluções de terceiros. O Apache ActiveMQ e o sistema de mensageria da IBM, WebSphere MQ, estão entre as necessidades mais comuns nesse sentido.

### Apache Active MQ

O Apache Active MQ é um sistema de mensageria de código aberto, escrito em Java, e mantido pela Fundação de Software Apache. É comumente utilizado como uma solução rápida e barata de mensageria.

O primeiro passo para utilizar o Active MQ em conjunto com o Camel é declarar a dependência do Active MQ no sistema de build. No Maven isso pode ser feito da seguinte maneira:

```
<dependency>
  <groupId>org.apache.activemq</groupId>
  <artifactId>activemq-camel</artifactId>
  <version>5.8.0</version>
</dependency>
```

Obs.: ao utilizar uma versão do Camel diferente da 2.10.3, é importante adicionar o camel-jms na lista de exclusão da dependência. Desta forma evita-se que o Camel lance uma exceção `NoSuchMethodError` devido a diferenças na interface binária da aplicação (Application Binary Interface - ABI).

```
<exclusions>
  <exclusion>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-jms</artifactId>
  </exclusion>
</exclusions>
```

A partir de então, pode-se criar uma instância do objeto de configuração do Active MQ (*org.apache.activemq.camel.component.ActiveMQConfiguration*) e adiciona-lo ao contexto do Camel. O exemplo abaixo mostra uma configuração simples de acesso a um *broker* Active MQ configurado com autenticação por usuário e senha:

```
private ActiveMQConfiguration getActiveMQConfiguration() {
    ActiveMQConfiguration configuration = new ActiveMQConfiguration();

    // Usuário
    configuration.setUserName("user");

    // Senha
    configuration.setPassword("password");

    // Endereço do broker do Active MQ
    configuration.setBrokerURL("tcp://broker:61616");

    return configuration;
}
```

Uma vez que o objeto de configuração foi instanciado, é possível utilizá-lo para instanciar um componente Active MQ:

```
ActiveMQConfiguration aMQConfiguration = getActiveMQConfiguration();
ActiveMQComponent component = new ActiveMQComponent(aMQConfiguration);
```

Por fim, basta adicionar o componente no contexto do Camel:

```
camelContext.addComponent("activemq", component);
```

## IBM WebSphere MQ

O IBM WebSphere MQ é um flexível sistema de mensageria de altíssima performance desenvolvido pela IBM. É costumeiramente utilizado por diversas corporações para implementar uma infraestrutura de mensageria escalável e robusta, em conjunto com diversas outras aplicações middleware da IBM como IBM DataPower e IBM WebSphere Message Broker.

A IBM não oferece um componente nativo para o Camel, porém tanto o Camel quanto o WebSphere MQ fornecem suporte a JMS, o que torna possível integrar ambos de maneira relativamente simples.

O primeiro passo envolve instanciar um objeto responsável por criar conexões com as filas MQ (QueueConnections). Isso é feito a partir de uma instância da classe *com.ibm.mq.jms.MQQueueConnectionFactory* (o qual implementa a *javax.jms.ConnectionFactory*):

```
MQQueueConnectionFactory mqCf = new MQQueueConnectionFactory();
mqCf.setHostName("server.hostname");

// 1415 costuma ser a porta padrão para o MQ
```

```
mqcCf.setPort(1415);

// Nome do queue manager
mqcCf.setQueueManager("queueManager");

// Nome do canal
mqcCf.setChannel("channel");

// Tipo do transporte
mqcCf.setTransportType(1);
```

Caso seja necessário utilizar o suporte a *pools* de conexão do MQ, é possível adicionar um toque através de uma chamada ao método *addConnectionPoolToken* da classe *MQEnvironment*:

```
MQEnvironment.addConnectionPoolToken()
```

Em seguida, utiliza-se o *MQQueueConnectionFactory* instanciado anteriormente para criar um novo objeto de configuração Jms (*org.apache.camel.component.jms.JmsConfiguration*):

```
JmsConfiguration jmsConfiguration = new JmsConfiguration(mqcCf);
```

Através desse objeto é possível configurar, entre outras coisas, o comportamento do conector JMS em relação ao modo de confirmação de recebimento (acknowledge). Por exemplo:

```
jmsConfiguration.setAcknowledgementMode(Session.AUTO_ACKNOWLEDGE);
```

A partir da instância do objeto de configuração JMS pode-se, então, instanciar um componente *JmsComponent* e associar a resolução de destino aos componentes de mais baixo nível do MQ:

```
JmsComponent jmsComponent = new JmsComponent(jmsConfiguration);
```

```
jmsComponent.setDestinationResolver(new DestinationResolver(){
    public Destination resolveDestinationName(Session session, String
destinationName, boolean pubSubDomain) throws JMSEException {
        MQQueueSession wmqSession = (MQQueueSession) session;
        return wmqSession.createQueue("queue:://" + destinationName + "?
targetClient=1");
    }
});
```

Por fim, basta associar a instância do componente JMS criado previamente ao contexto MQ:

```
camelContext.addComponent("wmq", jmsComponent);
```

Ainda que pareça complexo, esse mecanismo permite um extenso grau de controle sob os parâmetros de mais baixo nível, o que permite ao Camel integrar com os mais variados ambientes WebSphere MQ.

## Definindo Rotas

Conforme explicado anteriormente, as rotas do Camel são definidas através de uma DSL. No caso da Java DSL, para definir uma rota é necessário especializar a classe *RouteBuilder* e declarar a ligação entre os *Endpoints* através do método *configure*.

A classe *RouteBuilder* é permite um arranjo bastante flexível da declaração da rota, de modo que aplicações complexas podem implementar mecanismos para auxiliar na reutilização de código.

### Configurando Rotas

Uma rota pode ser definida através da Java DSL da seguinte forma:

```
from("especificação do endpoint")
    .to("especificação do endpoint");
```

Para entender melhor, é necessário desconstruir a declaração da rota. Desta forma:

- *from()*: é um método da Java DSL utilizado para definir o *endpoint* inicial da rota. É a partir das mensagens recebidas no *endpoint* especificado nessa declaração que o Camel inicia uma troca de mensagens. Essa troca de mensagens é o que o Camel chama de Exchange – representado por um objeto de mesmo nome.
- Especificação do *endpoint*: é uma String utilizada para especificar o endereçamento do *endpoint* e suas opções. Seu formato assemelha-se ao seguinte: <componente>:<endereçamento e opções do componente><opções>. Onde:
  - Componente: é um dos componentes do Camel, conforme citado no começo do capítulo.
  - Endereçamento: é um endereçamento específico ao componente.
  - Opções: opções do componente ou do endereçamento. Seu formado é: ? *opção1=valor1&opção2=valor2*. As opções e valores são específicas para cada componente.
- *to()*: é um método da Java DSL utilizado para definir o(s) *endpoint(s)* final(is) da rota. É opcional e pode estar presente ou não, de acordo com o padrão de integração utilizado.

Abaixo é mostrado um exemplo de uma declaração simples de rota utilizando a Java DSL:

```
@Override
public void configure() throws Exception {
    JaxbDataFormat readerFormat = FormatBuilder.getReaderFormat();
    JaxbDataFormat writerFormat = FormatBuilder.getWriterFormat();

    from("activemq:queue:sas.request?" +
        "concurrentConsumers=2&" +
```

```
        "maxConcurrentConsumers=4")
        .unmarshal(readerFormat)
        .process(new EvalServiceProcessor())
        .marshal(writerFormat);
    }
```

## Processadores

Um processador é uma classe com propósito de fazer manipulações complexas no conteúdo de um *exchange*. O *processor* pode ser utilizado, por exemplo, para implementar um *parser*, interface com um *bean*, enriquecer o *exchange* e qualquer outra operação de mais baixo nível que não possa ser feito através da Java DSL.

No Camel um processador é feito implementando a interface *Processor* e, então, implementando a lógica de processamento no método *process*.

```
public void process(Exchange exchange) throws Exception {
    RequestType requestType =
exchange.getIn().getBody(RequestType.class);

    EvalServiceBean bean = new EvalServiceBean();
    ResponseType responseType = bean.eval(requestType);

    exchange.getOut().setBody(responseType);
}
```

No exemplo acima o processador utiliza as rotinas do *Exchange* para obter um objeto Java resultante do parseamento de uma transação XML por um objeto *JaxbDataFormat*. O parseamento de dados será explorado com maiores detalhes nas próximas seções.



## Gerenciando Excessões na Rota

O Camel fornece dois mecanismos para gerenciar excessões na rota. Desta forma, é possível ter um bom nível de flexibilidade. Ambos são definidos durante a declaração da rota (no método *configure*).

O primeiro dos métodos faz uma espécie de armadilha (trap), capturando todas as excessões do tipo especificado em qualquer ponto da rota. Usando como exemplo a rota definida anteriormente, temos:

```
public void configure() throws Exception {
    JaxbDataFormat readerFormat = FormatBuilder.getReaderFormat();
    JaxbDataFormat writerFormat = FormatBuilder.getWriterFormat();

    onException(JAXBException.class)
        .handled(true)
        .to("log:net.orpiske.sas.service.exchanges.evalservice?
level=ERROR")
        .process(new EvalServiceErrorProcessor())
        .marshal(writerFormat);

    from("activemq:queue:sas.request?" +
        "concurrentConsumers=2&" +
        "maxConcurrentConsumers=4")
        .unmarshal(readerFormat)
        .process(new EvalServiceProcessor())
        .marshal(writerFormat);
}
```

O segundo método permite capturar as excessões somente em um determinado ponto da rota:

```
public void configure() throws Exception {
```

```

        JaxbDataFormat readerFormat = FormatBuilder.getReaderFormat();
        JaxbDataFormat writerFormat = FormatBuilder.getWriterFormat();

        /**
         * This is the default exchange declaration. We start from our
sas.request
         * queue with 2 concurrent consumers (maxing out at 4),
unmarshal the request
         * and declare the processor
         */
        from("activemq:queue:sas.request?" +
            "concurrentConsumers=2&" +
            "maxConcurrentConsumers=4")
            .doTry()
                .unmarshal(readerFormat)
                .process(new EvalServiceProcessor())
                .marshal(writerFormat)
            .doCatch(JAXBException.class)

.to("log:net.orpiske.sas.service.exchanges.evalservice?level=ERROR")
            .process(new EvalServiceErrorProcessor())
            .marshal(writerFormat);
    }

```

## Conversores de Tipos

Os conversores de tipos permitem a implementação de rotinas de conversão de dados entre o formato que é utilizado para a comunicação entre as aplicações.

```

public static JaxbDataFormat getReaderFormat() {
    return new JaxbDataFormat(PACKAGE_NAME);
}

public static JaxbDataFormat getWriterFormat() {

```

```
JaxbDataFormat writerFormat =  
    new JaxbDataFormat(PACKAGE_NAME);  
  
writerFormat.setPartClass(PACKAGE_NAME + ".ResponseType");  
writerFormat.setPartNamespace(new QName(NAMESPACE, "response"));  
  
    return writerFormat;  
}
```

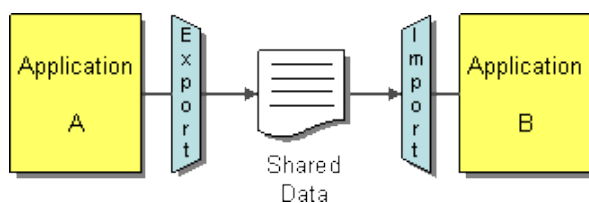
# Padrões de Integração de Sistemas

Os EIP foram desenvolvidos pelo engenheiro de software Gregor Hohpe, após anos de experiência desenvolvendo soluções de integração, e refletem algumas das melhores práticas para integração entre sistemas. A maioria dos mecanismos de integração coletados na biblioteca de EIP pode ser encontrado em diversas outras ferramentas de integração – além do Apache Camel – incluindo: IBM WebSphere MQ, Vitria, Apache ServiceMix, Apache Synapse e muitos outros.

Obs.: o nome dos padrões estão traduzidos por pura conveniência. Sempre que necessário, os padrões serão mencionados em inglês – seu idioma original e como são referenciados pelo próprio Apache Camel.

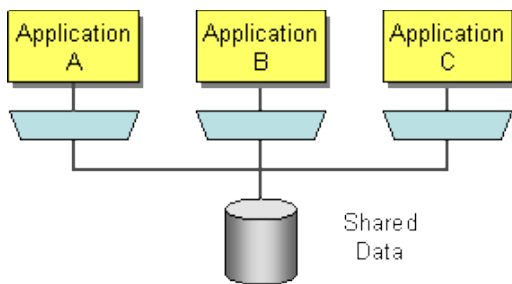
## Estilos De Integração

### Transferência de arquivos (*file transfer*)



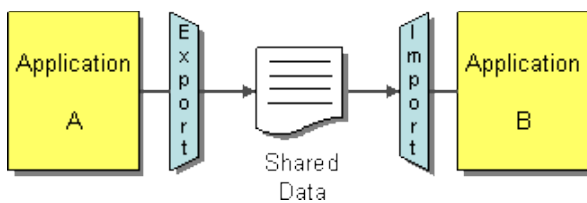
Está relacionado a produção, troca e consumo de arquivos como meio de integração.

### Banco de dados compartilhado (*shared database*)



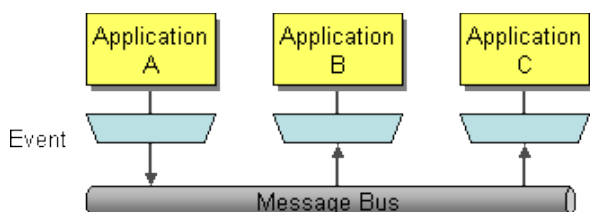
Refere-se ao uso de um banco de dados para compartilhamento de dados entre aplicações.

### Invocação remota de procedimento (*remote procedure invocation*)



É a troca de dados através da exposição e chamada de métodos remotos.

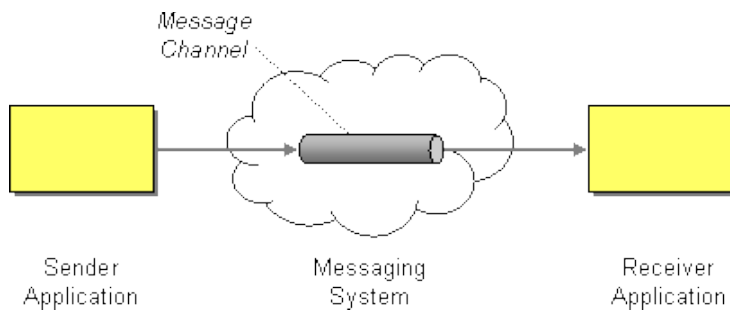
### Mensageria (*messaging*)



É a troca de informações através da utilização de um sistema comum que utiliza unidades de dados definidas como “mensagens”.

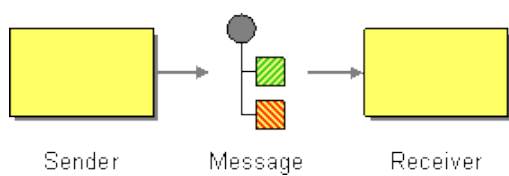
### Sistemas de mensageria

## Canais (*message channel*)



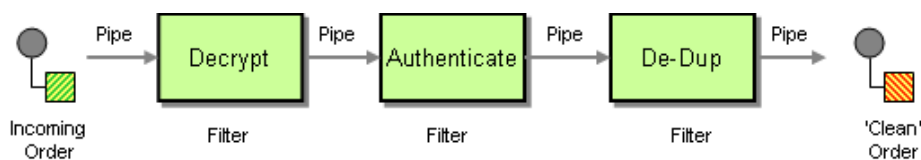
Virtualiza um meio de comunicação entre transmissor e o receptor.

## Mensagens (*message*)



É uma unidade de troca de dados entre sistemas. Pode ser interpretada como um envelope, da onde os dados devem ser extraídos para processamento pelos integrantes.

## Tubos e filtros (*pipes and filters*)

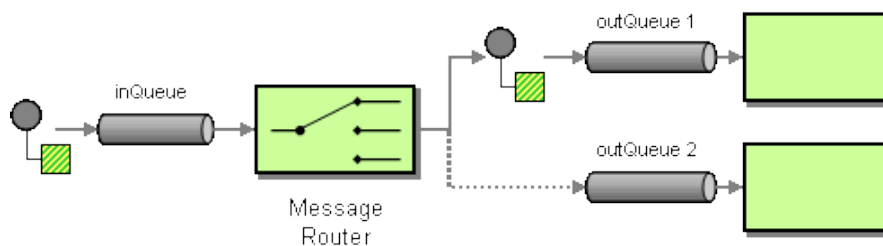


É uma abstração do processamento (incluindo, mas não limitado a filtragem) ordenado de mensagens em que cada passo do processamento é ligado por canais.

## Implementação no Camel

```
from("direct:a")  
    .pipeline("direct:x", "direct:y", "direct:z", "mock:result");
```

## Roteamento de mensagens (*message routing*)

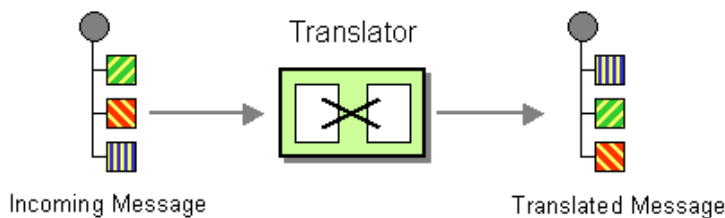


Em um ambiente complexo, formado por múltiplos integrantes, é o processo de encaminhar as mensagens para os receptores corretos.

## Implementação no Camel

```
from("direct:pedidos")  
    .choice()  
        .when(header("tipo").isEqualTo("passagens"))  
            .to("direct:filaDePassagens")  
        .when(header("tipo").isEqualTo("hoteis"))  
            .to("direct:filaDeHoteis")  
        .otherwise()  
            .to("direct:outros");
```

## Tradutor (*message translator*)

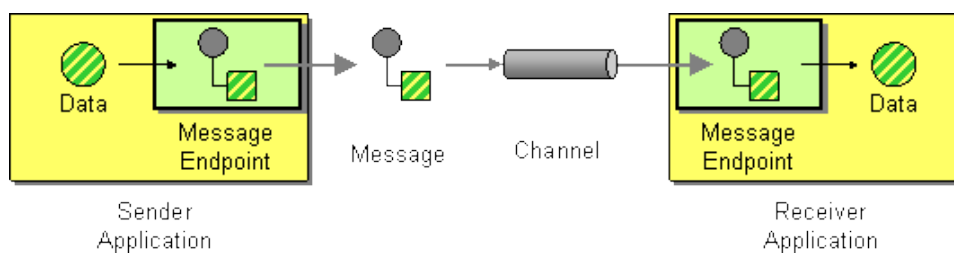


Diz respeito ao processo de modificação das mensagens de tal modo que cada integrante recebe a mensagem no formato que lhe é adequado.

## Implementação no Camel

```
from("direct:request")
    .process(new TransformationProcessor())
    .to("mock:resultado");
```

## Terminadores (*message endpoint*)

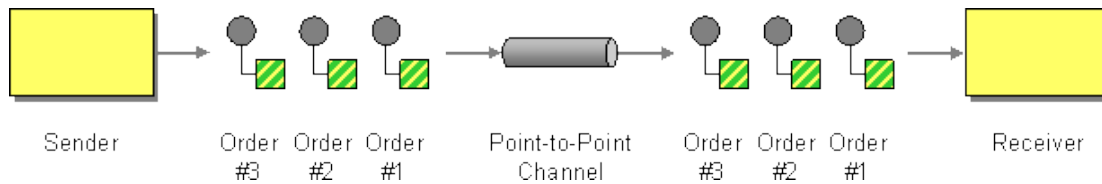


É o ponto final/inicial de conexão onde se inicia ou termina o transporte da mensagem. Serve como uma divisão lógica entre a camada da aplicação e a camada que opera em conjunto com sistema de mensageria.



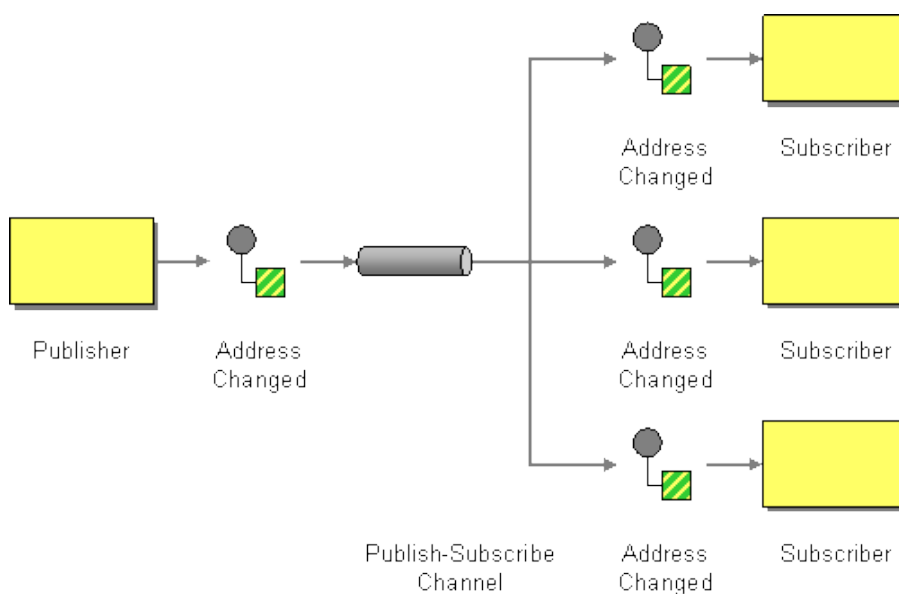
## Canais de mensageria

### Canal ponto a ponto (*point-to-point channel*)



É um tipo de ligação que garante que só um receptor receberá a mensagem.

### Publicação/assinante (*publish/subscribe*)

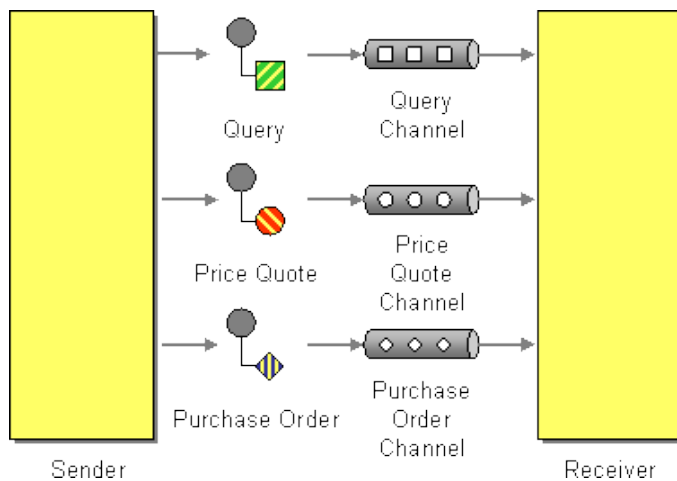


É um tipo de ligação em *broadcast* (anúncio), de tal forma que um dos integrantes publica uma mensagem e todos os outros integrantes recebem uma cópia da mesma.

### Implementação no Camel

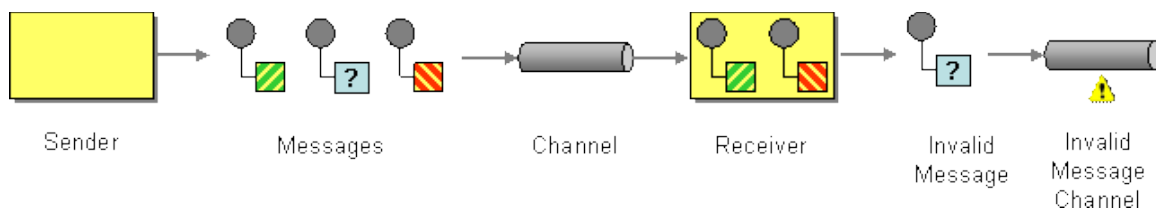
```
from("direct:a")
    .multicast()
    .to("mock:endpoint1", "mock:endpoint2");
```

### Canal por tipo de dado (*datatype channel*)



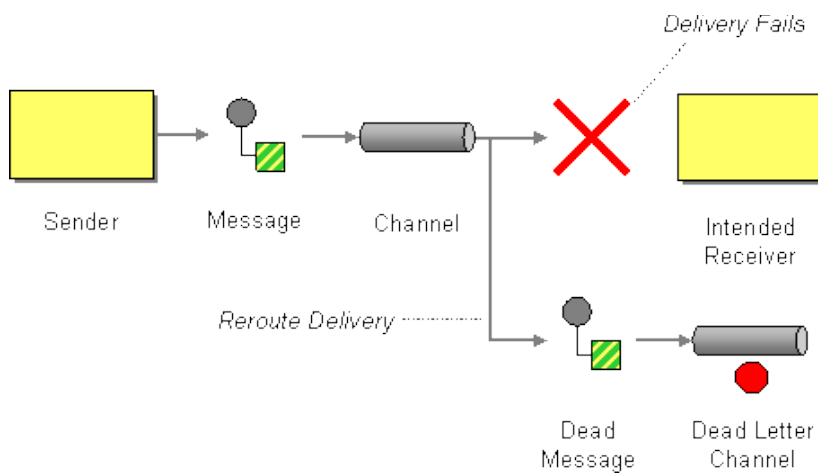
É uma forma de ligação que visa garantir que o receptor da mensagem será capaz de interpretá-la e processá-la.

### Canal de mensagem inválida (*invalid message channel*)



É um canal especial para onde as mensagens inválidas são encaminhadas.

### Canal “dead letter” (*dead-letter channel*)

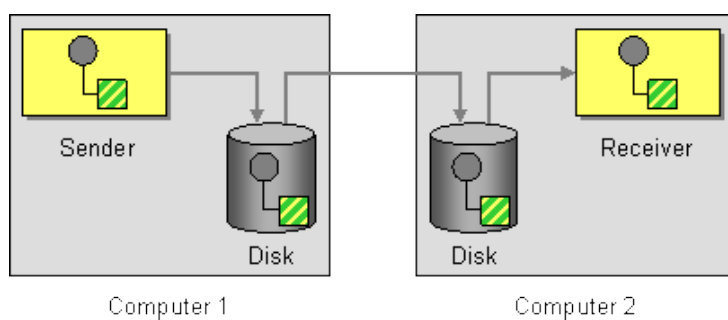


É um canal especial para onde vão as mensagens que não podem ser entregues.

## Implementação no Camel

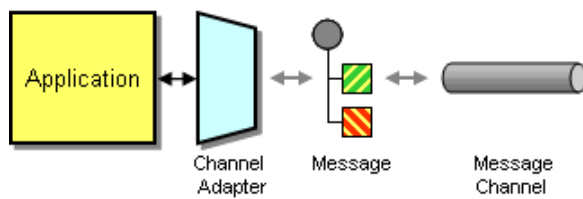
```
errorHandler(deadLetterChannel("jms:queue:dead")
    .maximumRedeliveries(3).redeliveryDelay(5000));
```

## Entrega garantida (*guaranteed delivery*)



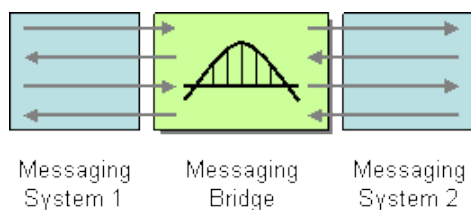
É uma funcionalidade do sistema de mensageria que garante a entrega da mensagem, através de um sistema de persistência, mesmo quando o próprio sistema de mensageria falha.

## Adaptador de canal (*channel adapter*)



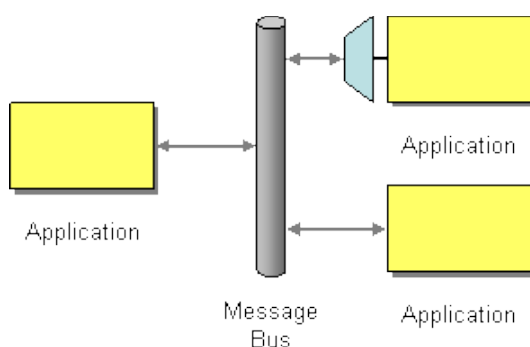
Fornece a possibilidade, através de APIs (*Application Programming Interfaces*), de que aplicações se acessem o canal.

## Ponte de mensageria (*messaging bridge*)



Permite que sistemas de mensageria se conectem de modo que as mensagens emitidas em um sistema estejam disponíveis em outro.

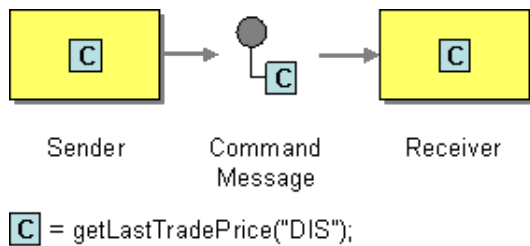
## Barramento (*message bus*)



É uma combinação de elementos de mensageria que garante o desacoplamento de sistemas mas garantindo que as aplicações conversem entre si através de uma infraestrutura comum.

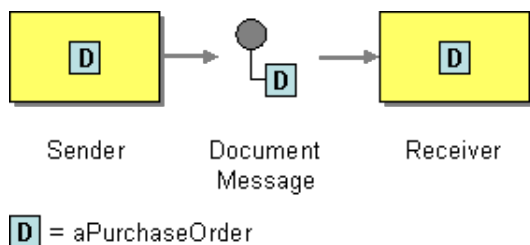
## Construção de mensagens

### Mensagem de comando (*command message*)



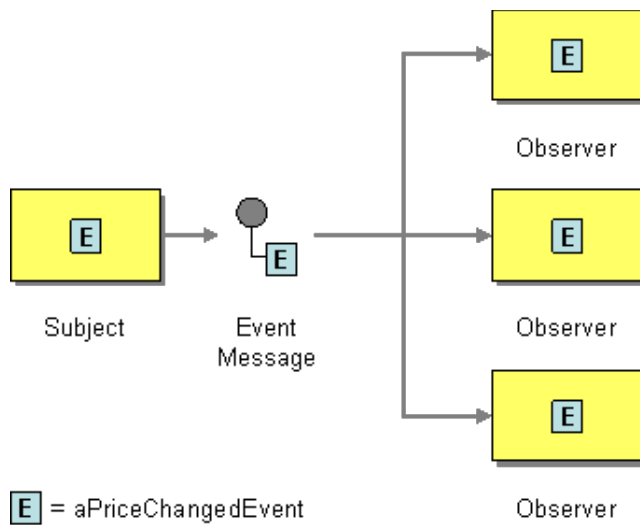
Uma mensagem de comando é uma mensagem, sem um tipo especificamente definido, que contém um comando.

### Mensagem de documento (*document message*)



Uma mensagem que cujos dados devem ser analisados (ou descartados) pelo receptor e que não têm uma ligação específica com uma invocação de método – ou seja, os dados contidos na mensagem não são referentes a uma chamada que deve ser invocada pelo receptor.

## Mensagem de evento (*event message*)

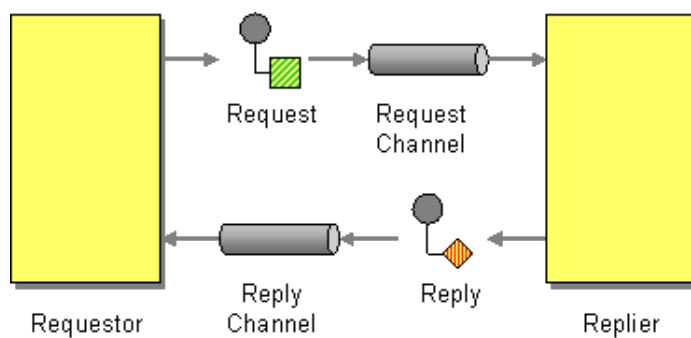


Uma mensagem cujos dados são referentes a um evento anunciado pelo emissor.

## Implementação no Camel

```
from("mq:fila1")
    .inOnly("mq:fila2");
```

## Requisição e resposta (*request and reply*)

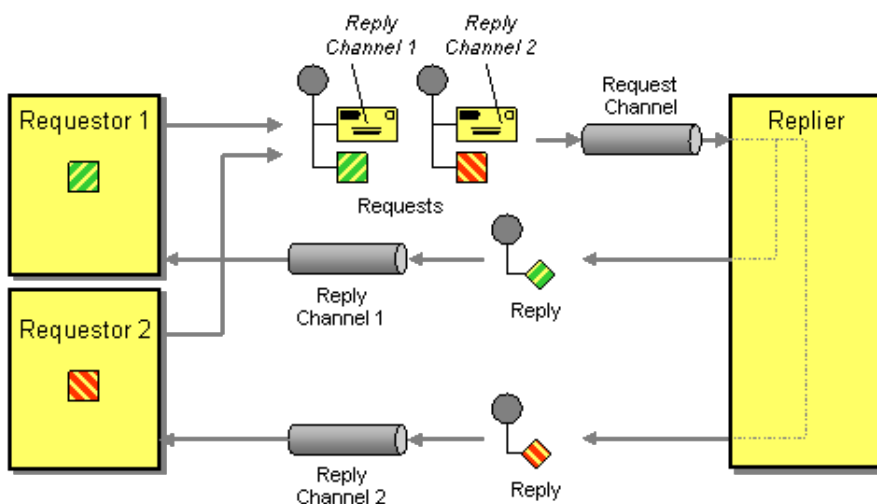


Requisição e resposta é um comportamento de troca de dados em que o emissor envia uma requisição, através de um canal específico, e recebe uma resposta, igualmente, em um canal específico para respostas.

## Implementação no Camel

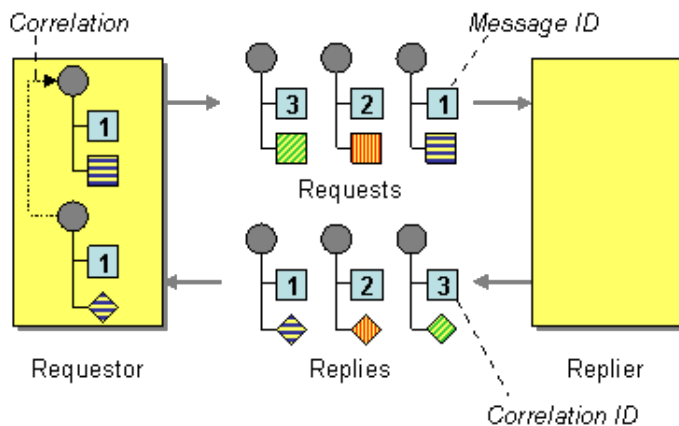
```
from("mq:fila1")  
    .process(new TransactionProcessor());
```

### Endereço de retorno (*return address*)



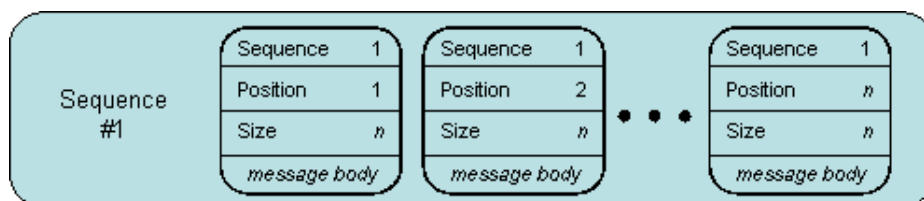
Nesse padrão, emitente da requisição informa o endereço de retorno – canal – da resposta. Desta forma é possível desacoplar o receptor de um canal resposta previamente definido.

## Identificador de correlação (*correlation identifier*)



O identificador de correlação fornece uma maneira de associar uma resposta a uma requisição.

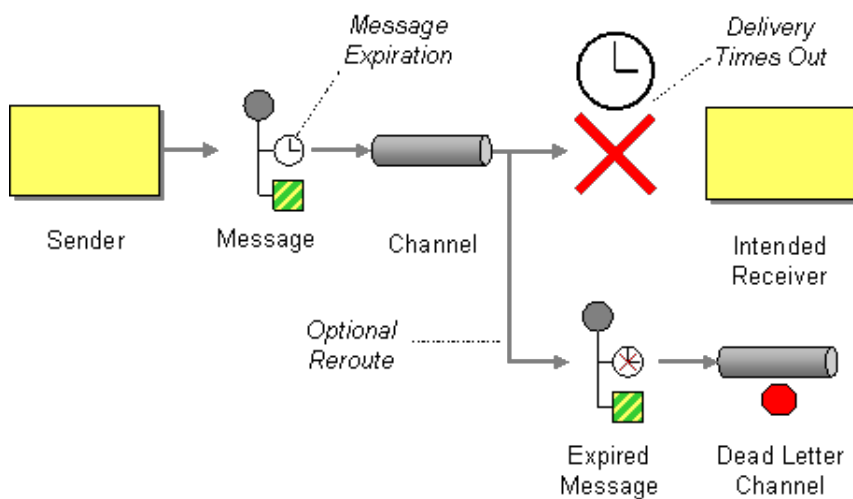
## Sequência de mensagem (*message sequence*)



Permite sequenciar uma mensagem de modo que grandes quantidades de dados, maiores do que seria possível transmitir em uma única mensagem, podem ser quebrados em partes menores e transmitidos através do canal.

## Expiração de mensagem (*message expiration*)





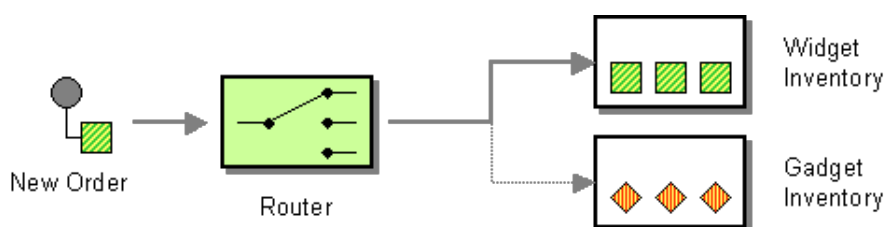
A expiração de mensagem permite indicar um tempo limite dentro do qual é aceitável consumir a mensagem.

## Identificador de formato (*format identifier*)

Fornece ao receptor uma maneira de identificar o formato da mensagem.

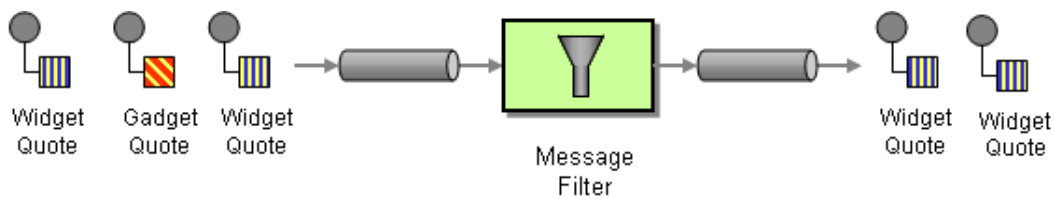
## Roteamento de mensagens

### Roteador baseado em conteúdo (*content-based router*)



Fornece uma solução para o problema de rotear uma mensagem de acordo com o seu conteúdo.

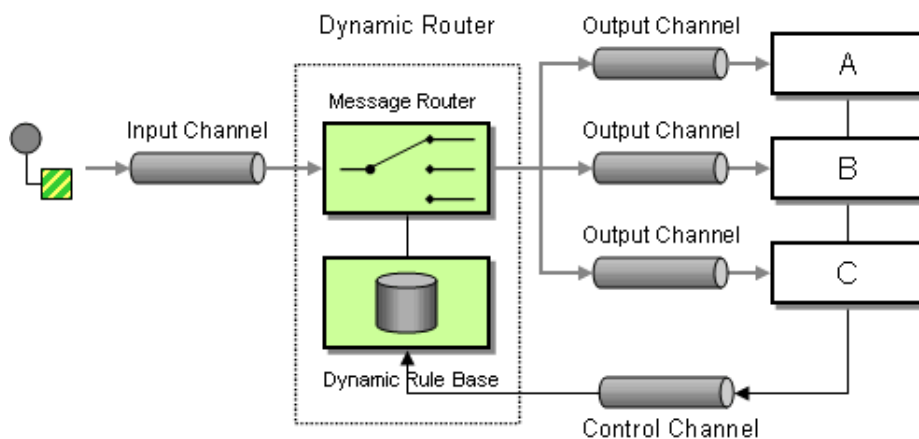
## Filtro de mensagem (*message filter*)



Esse padrão estabelece a possibilidade de filtrar os dados de uma mensagem de modo a eliminar conteúdo que seja inútil ao receptor.

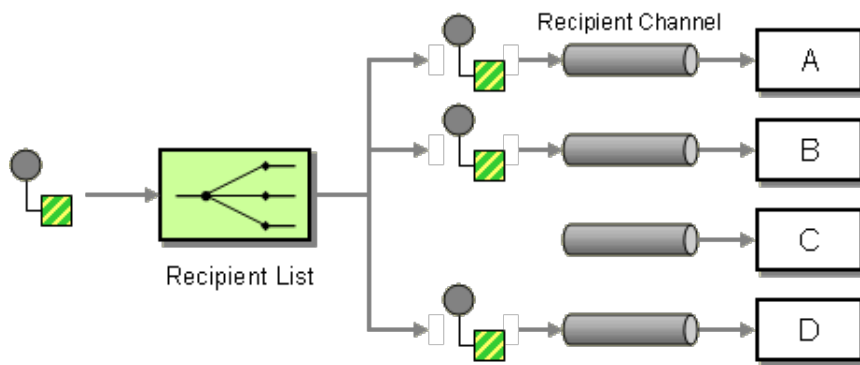
```
from("direct:fila")
    .filter(header("origin").isEqualTo("web"))
    .to("direct:webqueue");
```

## Roteador dinâmico (*dynamic router*)



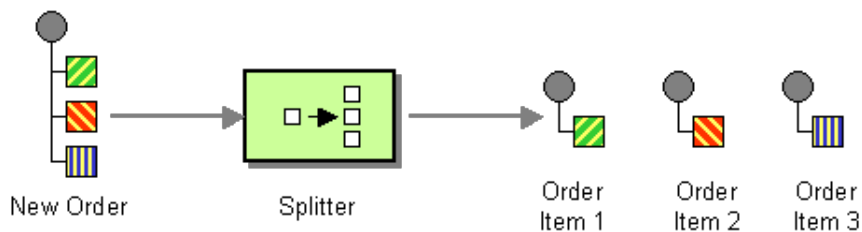
Esse padrão detalha uma solução para o problema de determinar o destino de uma mensagem em tempo de execução – ou seja, quando a mensagem está trafegando pela rota.

## Lista de receptores (*recipient list*)



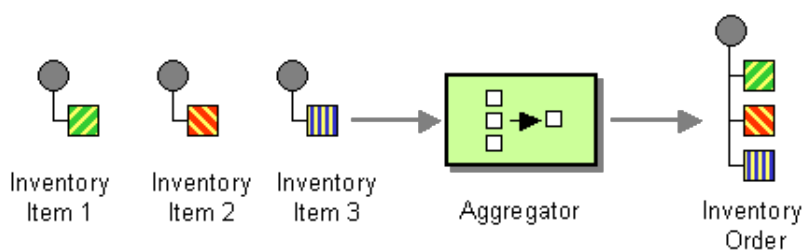
Esse padrão fornece uma solução para o problema de rotear dinamicamente uma mensagem para uma lista de recipientes.

## Divisor (*splitter*)



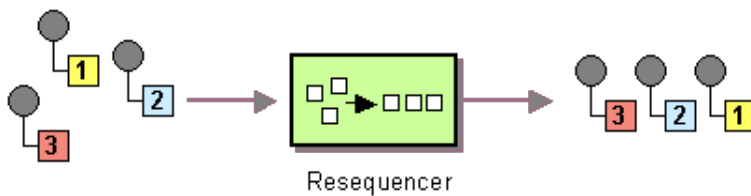
Refere-se a habilidade de dividir a mensagem em partes menores.

## Agregador (*aggregator*)



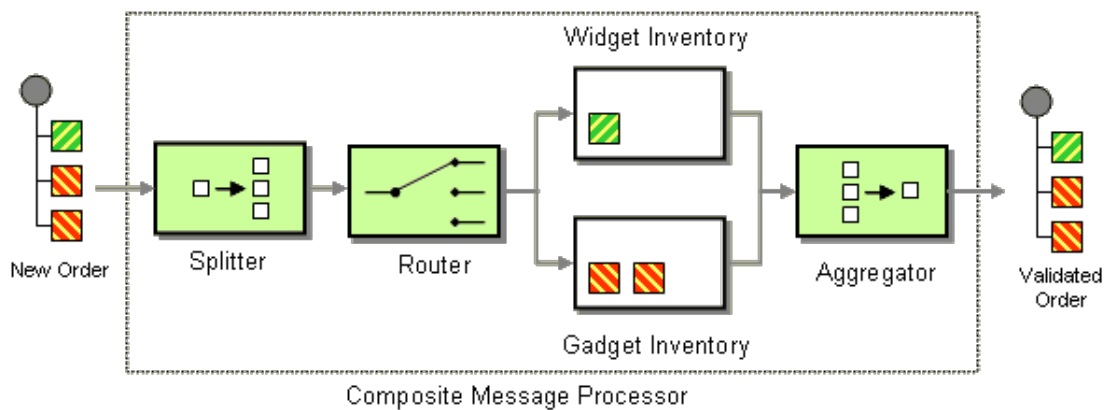
Enquanto que o *splitter* (divisor) está relacionado a habilidade de dividir a mensagem em partes menores, o agregador está relacionado a habilidade de coletar e armazenar mensagens de tal forma que elas podem ser agrupadas a uma única mensagem.

### Resequenciador (*re-sequencer*)



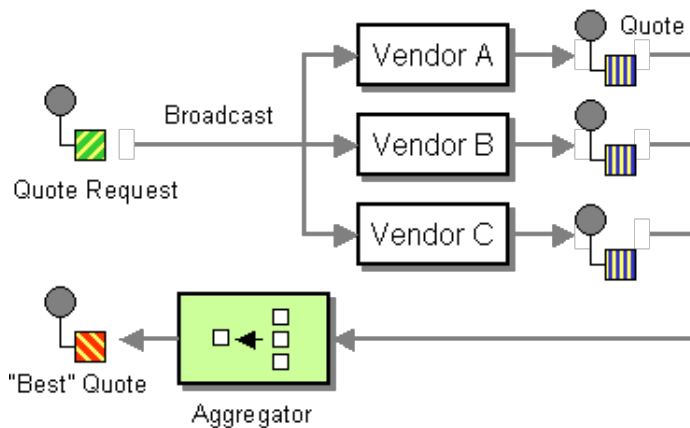
Este padrão fornece uma resposta a necessidade de ordenar as mensagens para processamento ou envio.

### Processador de mensagens compostas (*composed message processor*)



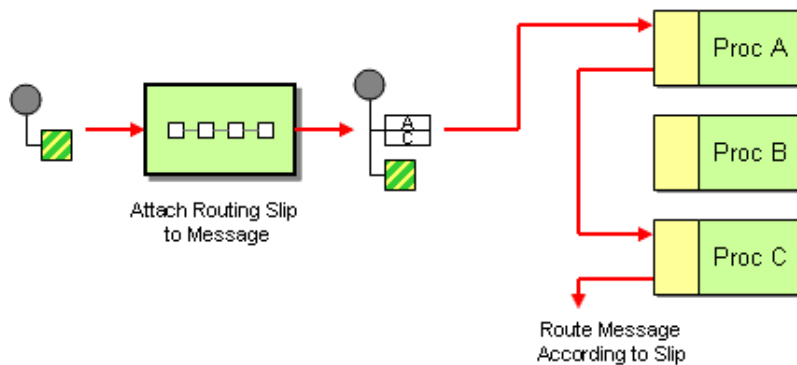
Detalha uma solução para processar mensagens compostas, de modo que cada sub-mensagem pode ser roteada para um destino específico e cujas respostas, posteriormente, podem ser agregadas em uma única mensagem.

## Dispersão e recolhimento (*scatter-gather*)



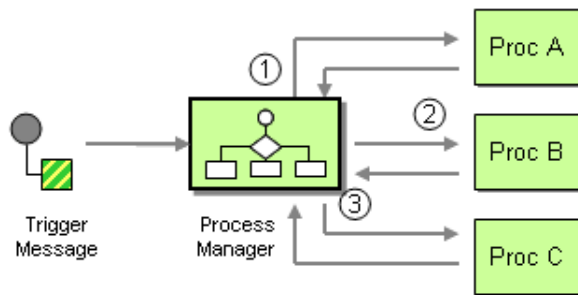
Esse padrão está relacionado a agregação das respostas de uma mensagem que fora previamente anunciada a múltiplos receptores (*broadcast*).

## Lista de circulação (*routing slip*)



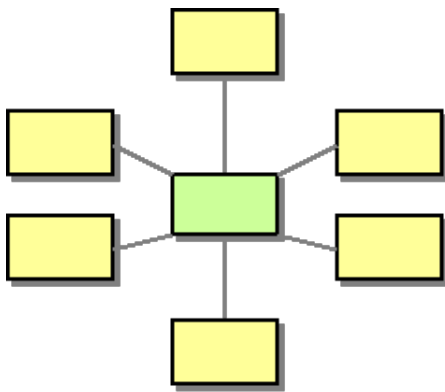
É uma informação de roteamento adicionada a mensagem de modo que permite definir a sequência de processamento em tempo de execução.

## Gerenciador de processos (*process manager*)



É um componente com a responsabilidade de gerenciar o estado, sequenciamento e determinar os próximos passos de processamento.

### Corretor de mensagens (*message broker*)

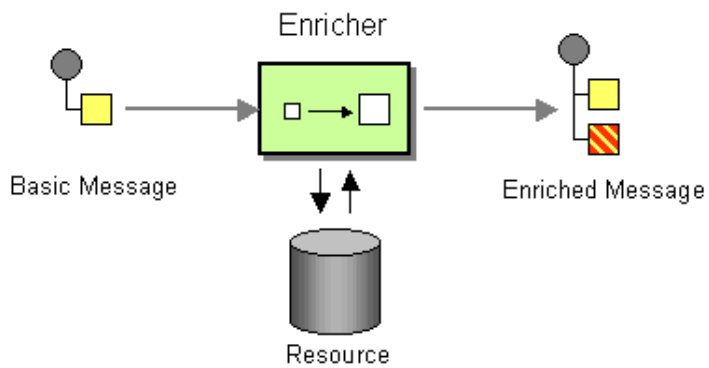


É um componente com a responsabilidade de orquestrar a execução das transações. Visa desacoplar o destinatário de uma mensagem de seu receptor. É elaborado a partir da implementação dos padrões de integração de sistemas.

### Transformação de Mensagens

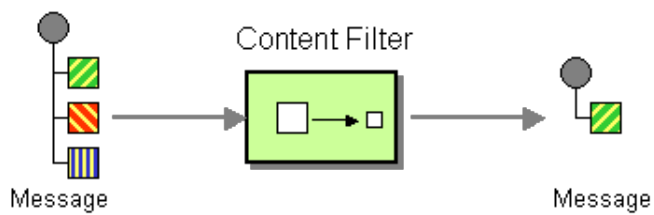
Os padrões de transformação de mensagens estão relacionados as transformações que podem ocorrer nas mensagens mediante seu processamento e roteamento.

## Enriquecedor de conteúdo (*content enricher*)



Fornece uma solução para a necessidade de incrementar a carga de dados de uma mensagem.

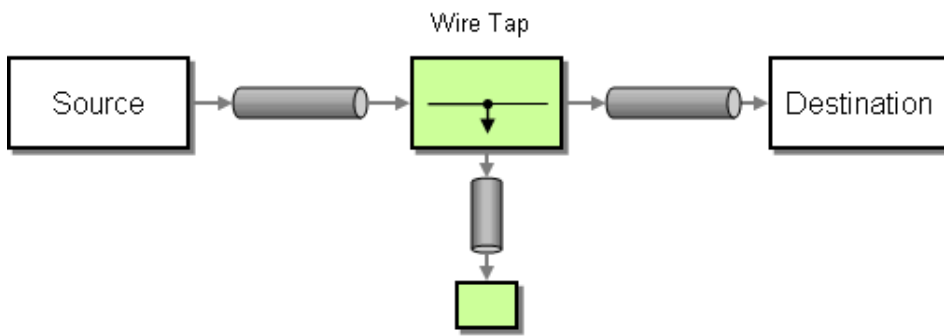
## Filtro de conteúdo (*content filter*)



Fornece uma solução para a necessidade de filtrar dados de uma mensagem, de modo que dados irrelevantes sejam removidos da transação.

## Gerenciamento de sistemas

### Escuta (*wire tap*)



Esse padrão detalha uma solução para a necessidade de inspecionar as mensagens trafegando em um canal, com a garantia de que elas não serão modificadas no processo.

## Outros

Os seguintes padrões, embora importantes, não são relevantes para um contexto introdutório à integração de sistemas com Apache Camel:

- Transformação de Mensagens
  - Normalizador (*normalizer*)
  - Modelo de dados canônico (*canonical data model*)
  - Invólucro de envelope (*envelope wrapper*)
  - Recibo (*claim check*)
- Pontos finais de mensageria
  - Portal de entrada de mensagens (*messaging gateway*)
  - Mapeamento de mensagens (*message mapper*)
  - Cliente transacional (*transactional client*)
  - Consumidor por captação (*polling consumer*)
  - Consumidor dirigido por evento (*event-driver consumer*)
  - Consumidor concorrente (*concurrent consumer*)
  - Despachante de mensagens (*message dispatcher*)



- Consumidor seletivo (*selective consumer*)
- Assinante persistente (*persistent subscriber*)
- Receptor idempotente (*idempotent receiver*)
- Ativador de Serviço (*service activator*)
- Gerenciamento de sistemas
  - Controle de barramento (*control bus*)
  - Histórico de mensagens (*message history*)
  - Desvio (*detour*)
  - Loja de mensagens (*message store*)
  - “Proxy” esperto (*smart proxy*)
  - Canal de expurgo (*channel purger*)

## Licenças

Todas as imagens da seção Padrões de Integração de Projetos disponíveis <http://www.eaipatterns.com/> são de propriedade de Gregor Hohpe e estão disponíveis sob a licença Creative Commons Attribution, conforme expresso em <http://creativecommons.org/licenses/by/3.0/>.

## Material

Este material, exceto seus exemplos, está disponível sob a licença Creative Commons Attribution 4.0 License, conforme expresso em <http://creativecommons.org/licenses/by/4.0/>.

O código fonte de exemplo está licenciado sob a Apache License 2.0 e encontra-se disponível nos seguintes repositórios:

- Data Collector Dispatcher: <https://github.com/orpiske/data-collector-dispatcher>
- MDM Broker: <https://github.com/orpiske/mdm-broker>
- MDM Types: <https://github.com/orpiske/mdm-types>
- SAS Commons: <https://github.com/orpiske/sas-commons>
- SAS Services: <https://github.com/orpiske/sas-service>
- SAS Types: <https://github.com/orpiske/sas-types>

## Referências

1. Hohpe, Gregor and Bobby Wolf; Enterprise Application Integration Patterns – Designing Building and Deploying Messaging Solutions. Addison-Wesley, 2003.
2. Chase, Nicholas; Understanding Web Services Specifications, Part 1: SOAP (<http://www.ibm.com/developerworks/webservices/tutorials/ws-understand-web-services1/ws-understand-web-services1.html>). IBM Developer Works, 2006.
3. Chase, Nicholas; Understanding Web Services Specifications, Part 2: Web Services Description Language (<http://www.ibm.com/developerworks/webservices/tutorials/ws-understand-web-services2/ws-understand-web-services2.html>). IBM Developer Works, 2006.
4. Apache Camel Documentation (<http://camel.apache.org>). Apache Software Foundation, 2014.
5. Apache Camel – Walk Through An Example (<https://camel.apache.org/walk-through-an-example.html>). Apache Software Foundation, 2014.
6. Apache Camel – Running Camel standalone and have it keep running (<http://camel.apache.org/running-camel-standalone-and-have-it-keep-running.html>). Apache Software Foundation, 2014.
7. Stack Overflow – Question 1846791: CamelContext.start() doesn't block (<http://stackoverflow.com/questions/1846791/camelcontext-start-doesnt-block>) . Stack Overflow, 2009.
8. MQ Version 7.0 Message Acknowledgement ([http://publib.boulder.ibm.com/infocenter/wmqv7/v7r0/index.jsp?topic=%2Fcom.ibm.mq.xms.doc%2Fconcepts%2Fxml\\_cmecack.html](http://publib.boulder.ibm.com/infocenter/wmqv7/v7r0/index.jsp?topic=%2Fcom.ibm.mq.xms.doc%2Fconcepts%2Fxml_cmecack.html)), IBM Corporation.
9. Apache Camel – JMS (<http://camel.apache.org/jms>). Apache Software Foundation, 2014.
10. Camel Users Mailing List: Camel and IBM MQ Series (<http://camel.465427.n5.nabble.com/Camel-and-IBM-MQ-Series-td476223.html#a476223>). Apache Software Foundation, 2009.
11. Camel Users Mailing List: IBM MQ and Camel (<http://camel.465427.n5.nabble.com/IBM-MQ-and-CAMEL-td5591661.html>). Apache Software Foundation, 2012.
12. Fuse Mediation Router: Adding a Component to the Camel Context ([http://fusesource.com/docs/router/2.8/deploy\\_guide/FMRDS.ACCC.html](http://fusesource.com/docs/router/2.8/deploy_guide/FMRDS.ACCC.html)). FuseSource, 2014.