Apache Camel Passo a Passo

Integração de Maneira Rápida, Simples e Livre

Otavio Rodolfo Piske

v2.0 beta 1

Sobre

O autor trabalha com sistemas back-ends e projetos de integração, com foco em C++ e Java, há mais de 13 anos. Focando nas áreas de testes e qualidade em geral, performance e empacotamento o autor tem contribuído com projetos de integração há vários anos. Atualmente participa ativamente do projeto Apache Camel onde participa como membro "*committer*" do projeto.

Postagens e publicações do autor podem ser encontradas em seus sites pessoais www.orpiske.net (em inglês) e www.angusyoung.org (em português), bem como no seu perfil no Twitter @otavio021.

Códigos e Exemplos

Esse e-book vem acompanhado de alguns códigos de exemplo. Os códigos mostrados em várias partes desse e-book estão descritos em maiores detalhes em cada um dos exemplos disponíveis no seguinte repositório do Github: https://github.com/integracao-passo-a-passo/camel-passo-a-passo.

Saiba Mais

O objetivo deste e-book é fornecer um material inicial para apresentar o Camel e seus projetos para os falantes do Português. Para aprofundar o conhecimento a respeito do Camel, o autor recomenda o excelente livro "Camel In Action" bem como a documentação oficial do projeto disponível em http://camel.apache.org.

Introdução

Nos últimos anos a explosão de APIs e a popularização de arquiteturas micro-serviços tem feito com que cada vez mais desenvolvedores tenham que criar soluções que consumam, processem e troquem dados com múltiplos sistemas. Além disso, as oportunidades apresentadas por uma internet cada vez mais "social" trazem uma série de novos desafios para as companhias.

Esse novo paradigma de desenvolvimento aliado a prazos cada vez mais curtos, pressiona engenheiros de software e arquitetos a encontrarem soluções de integração cada vez mais ágeis, robustas e extensíveis.

A popularização de soluções de código aberto, que se tornaram padrões de facto, para os *frameworks* de desenvolvimento trouxeram inúmeras soluções para resolves os desafios de integração. Este mini e-book apresentará o Apache Camel, o canivete suíço da integração de sistemas, bem como diversos sub-projetos relacionados ao Camel, como o Camel Kafka Connector, o Camel K e o Camel Quarkus.

O que é o 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, disponibilizado sob a licença Apache License 2.0. Ele é desenvolvido pela comunidade de desenvolvedores do Apache Camel, suportados pela Fundação de Software Apache (Apache Software Foundation - ASF). A projeto é um dos maiores e mais ativos dos projetos suportados pela fundação.

Usando o Camel, é possível criar rotas que descrevem como interligar sistemas, aplicar padrões de integração enterprise (EIPs) para troca e mediação de trocas de dados e filtrar e transformar os dados em trânsito. Isso permite resolver problemas como *"como fazer com que os dados sejam movidos para o sistema A ou sistema B baseado no conteúdo que está sendo transportado"* ou *"como combinar diferentes pedaços de dados e envia-los de uma só vez para uma API REST"*. O Camel oferece ferramentas para que problemas como esse sejam resolvidos de maneira simples, reutilizável e performática.

Padrões de Integração Corporativa

Os padrões de integração corporativa, do inglês Enterprise Integration Patterns (EIPs), são uma biblioteca de padrões arquiteturais e funcionais. Eles fornecessem um conjunto de soluções pré-definidas para os desafios de interligar aplicações.

Embora inicialmente voltadas para o mundo corporativo, interagindo com aplicações corporativas, esse contexto foi ampliado para os dias atuais. Com a popularização de aplicações sociais, aprendizado de máquina, big data e muitas outras disciplinas.

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 ESB, Mule, Apache ServiceMix e muitos outros.

Estilos De Integração

Nome	Nome em Inglês	Descrição
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 (RPI)	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

Nome	Nome em Inglês	Descrição
Canais	Message channel	Virtualiza um meio de comunicação entre transmissor e o receptor. Isso significa que tanto o transmissor quanto o receptor não precisam, necessariamente, saber qual aplicação está emitindo ou recebendo o evento. Isso favorece o desacoplamento arquitetural entre as aplicações.
Mensagens	Message channel	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.

Nome	Nome em Inglês	Descrição
Tubos e filtros	Pipes and filters	Uma abstração do processamento (incluindo, mas não limitado a filtragem) ordenado de mensagens em que cada passo do processamento é ligado por canais.
Roteamento de mensagens	Message routing	Processo de encaminhar as mensagens para os receptores corretos.
Tradutor de mensagens	Message translator	Processo de modificação das mensagens de tal modo que cada integrante recebe a mensagem no formato que lhe é adequado.
Terminadores	Message endpoint	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

Nome	Nome em Inglês	Descrição
Canal ponto a ponto	Point-to-point channel	Tipo de ligação que garante que só um receptor receberá a mensagem.
Publicação/assinante	Publish/subscribe	Tipo de ligação em que um dos integrantes publica uma mensagem e todos os outros integrantes recebem uma cópia da mesma.
Canal por tipo de dado	Datatype channel	Forma de ligação que visa garantir que o receptor da mensagem será capaz de interpreta-la e processa-la.
Canal de mensagem inválida	Invalid message channel	Canal para onde as mensagens inválidas são encaminhadas.
Canal “dead letter”	Dead-letter channel	Canal para onde vão as mensagens que não podem ser entregues.
Entrega garantida	Guaranteed delivery	É uma propriedade do sistema de mensageria que garante a entrega da mensagem. Pode ser feita através de um sistema de persistência, para garantia de entrega mesmo quando o próprio sistema de mensageria falha.
Adaptador de canel	Channel adapter	Fornece interfaces, através de APIs (*Application Programming Interfaces*), para 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.

Nome	Nome em Inglês	Descrição
Barramento	Message bus	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

Nome	Nome em Inglês	Descrição
Mensagem de comando	Command message	Mensagem, sem um tipo especificamente definido, que contém um comando.
Mensagem de documento	Document message	Mensagem cujo conteúdo deve ser analisado (ou descartado) pelo receptor e que não têm uma ligação específica com uma invocação de método.
Mensagem de evento	Event message	Uma mensagem cujos dados são referentes a um evento anunciado pelo emissor.
Requisição e resposta	Request/reply	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.
Endereço de retorno	Return address	Padrão onde o 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	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	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

Nome	Nome em Inglês	Descrição
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	Estabelece a possibilidade de filtrar os dados de uma mensagem de modo a eliminar conteúdo que seja inútil ao receptor.
Roteador dinâmico	Dynamic router	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	Fornece uma solução para o problema de rotear dinamicamente uma mensagem para uma lista de recipientes.
Divisor	Splitter	Habilidade de dividir a mensagem em partes menores.
Agregador	Aggregator	O agregador está relacionado a habilidade de coletar e armazenar mensagens de tal forma que elas podem ser agrupadas a uma única mensagem.
Re-sequenciador	Re-sequencer	Fornece uma reposta a necessidade de reordenar as mensagens para processamento ou envio.
Processador de mensagens compostas	Composed message processor	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	Está relacionado a agregação das respostas de uma mensagem que fora previamente anunciada a múltiplos receptores (*broadcast*).
Lista de circulação	Routing-slip	Informação de roteamento adicionada a mensagem que permite definir a sequência de processamento.
Gerenciador de processos	Process manager	Componente com a responsabilidade de gerenciar o estado, sequenciamento e determinar os próximos passos de processamento.
Corretor de mensagens	Message broker	Componente com a responsabilidade de orquestrar a execução das transações. Desacopla 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.

Nome	Nome em Inglês	Descrição
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:

Tipo	Padrão
Transformação de mensagens	○Normalizador (normalizer)
	○Modelo de dados canônico (canonical data model)
	○Invólucro de envelope (envelope wrapper)
	Recibo (claim check)
Pontos finais de mensageria (endpoints)	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*)

Tipo	Padrão
	Consumidor seletivo (*selective consumer*)
	Assinante persistente (*persistent subscriber*)
	Receptor idempotente (*idempotent receiver*)
	Ativador de Serviço (*service activator*)
Gerenciamento de sistemas	Controlle 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*)

Conceitos Básicos Sobre o Apache Camel

Rotas

As rotas são o principal mecanismo de funcionamento do Camel. Através delas é possível definir as regras de ligação entre diferentes pontos de interconexão (*endpoints*).

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.

O Camel conta, atualmente em sua versão 3.4.4, com suporte a mais de 300 tipos diferentes de componentes. Através destes, é possível ligar componentes que vão desde os mais tradicionais protocolos de mensageria até sistemas de CRM como Salesforce. Alguns dos principais componentes disponibilizados pelo projeto são:

Dica:

Todos os componentes do Camel contam com uma enorme gama de opções, sendo possível ajusta-los para os mais diversos propósitos.

Componente	Descrição
AMQP	Suporte ao protocolo AMQP 1.0 utilizado por sistemas de mensageria como Apache Artemis, Apache Qpid, Azure Service Bus e outros.
CXF	Suporte a web services e REST APIs através do projeto Apache CXF.
Direct	Para chamadas síncronas dentro de um mesmo contexto do Camel.
File	Para trocas de dados com arquivos, através da leitura e gravação de arquivos em locais pré-definidos.
FTP	Suporte para troca de dados através do protocolo FTP.
HTTP	Suporte para troca de dados através do protocolo HTTP.
JMS	Suporte para troca de dados através de mensageria compatível com o padrão JMS.
Netty	Para comunicação de mais baixo nível via sockets através do projeto Netty.
SJMS2	Suporte para troca de dados através de mensageria compatível com o padrão JMS utilizando uma implementação simples.

Componente	Descrição
Quartz	Para troca periódica ou pré-agendada de dados através do projeto Quartz.
Seda	Para troca de mensagens assíncrona dentro de um mesmo contexto do Camel.

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.

Linguagens de Domínio Específico - DSLs

No Camel os padrões de integração e as rotas são declaradas usando um linguagens de domínio específico (DSL – *Domain Specific Language*). O Camel fornece a possibilidade de criar rotas usando as seguintes DSLs:

DSL	
Java DSL	Definição de rotas usando construtores fluentes em Java
Spring XML	Definição de rotas usando Spring XML
Blueprint XML	Definição de rotas usando Blueprint XML para OSGI
Rest DSL	Para definição de serviços REST
Annotations DSL	Definição de rotas usando anotações em Java Beans

Os 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

Gerando um Novo Projeto

O Camel conta com diversos arquétipos (*archetypes*) do Apache Maven que podem ser utilizados para gerar os projetos.

Inicializar os projetos dessa maneira facilita a padronização de características comuns aos projetos, configura as dependências mínimas necessárias para começar o projeto, provê facilidades e evita todo o trabalho de ajustar manualmente arquivos e diretórios de acordo com o padrão utilizado pelo Maven.

O seguinte comando pode ser usado para gerar um projeto Camel usando a Java DSL:

```
mvn archetype:generate -B -DarchetypeGroupId=org.apache.camel.archetypes
-DarchetypeArtifactId=camel-archetype-java -DarchetypeVersion=3.4.4
-DgroupId=camel-passo-a-passo -DartifactId=primeiro-app-camel
-Dversion=1.0.0-SNAPSHOT -Dpackage=primeiro.app.camel
```

Dica:

O Camel fornece arquétipos adicionais para inicializar outros tipos de projetos. Não deixe de conferir quais estão disponíveis na versão atual!

O projeto pode ser compilado e empacotado em no formato jar usando os alvos *clean* e *package* do Maven.

```
mvn clean package
```

Da mesma forma, para executar o projeto podemos usar o alvo exec:java do Maven passando como argumento o nome qualificado da classe MainApp (que é criada automaticamente pelo arquétipo).

```
mvn exec:java -Dexec.mainClass="primeiro.app.camel.MainApp"
```

A execução desse projeto simples deve projeto inicial deve fazer com o que algumas mensagens de log sejam mostradas na tela até as seguintes mensagens sejam mostradas na tela:

```
[1) thread #1 - file://src/data] route1                              INFO
Other message
[1) thread #1 - file://src/data] route1                              INFO  UK
message
```

A partir desse ponto o programa fica em espera de mais dados e nada mais é mostrado. A partir desse ponto é possível abortar a execução do programa.

Inicialização

Explorando o projeto que foi gerado é possível identificar duas classes Java. A primeira delas, a *MainApp*, executa a inicialização do projeto criando uma instância de um wrapper de execução fornecido pelo próprio Camel. Esse wrapper facilita a execução do Camel como uma aplicação Java padrão sem a necessidade de runtimes adicionais e é definido na classe *org.apache.camel.main.Main*.

Esse wrapper permite uma execução limpa do Camel, sem a necessidade de uso de sleeps, busy spins e outros métodos rudimentares para controlar a execução do Camel.

Nesta classe podemos ver o uso de uma segunda classe auto-gerada, a MyRouteBuilder. Esta classe, uma especialização da classe abstrata *org.apache.camel.builder.RouteBuilder* é utilizada para configurar a rota utilizada pelo programa.

Ela é adicionada as propriedades de configuração do *wrapper,* o que é feito através dos métodos encadeados *configure().addRoutesBuilder()*, para que seja possível executa-la quando o programar rodar.

A configuração da rota dentro do método *configure* na classe *MyRouteBuilder* provê uma rota bastante simples mas que apresenta diversos conceitos importantes do Camel. De modo geral, a rota copia arquivos de um diretório para outro baseado no seu conteúdo. A rota do projeto de exemplo é a seguinte:

```
from("file:src/data?noop=true")
     .choice()
          .when(xpath("/person/city = 'London'"))
               .log("UK message")
               .to("file:target/messages/uk")
          .otherwise()
               .log("Other message")
               .to("file:target/messages/others");
```

Avaliando linha a linha a definição desta rota, temos o seguinte:

```
from("file:src/data?noop=true")
```

Um endpoint inicial *from* é declarado usando o componente file, utilizado para ler, monitorar arquivos e diretórios em disco, usando como entrada o diretório *src/data* que é parte da estrutura de diretórios do projeto recém criado. O componente é configurado com a opção *noop*, para evitar que os arquivos sejam movidos ou removidos do diretório.

Na linha seguinte, o código utiliza o método *choice* da DSL para aplicar uma condição na rota e definir que os dados devem seguir um rumo ou outro de acordo com uma expressão *xpath.*

```
.choice()
          .when(xpath("/person/city = ‘London’"))
              // … código
          .otherwise()
              // … código
```

Dependendo do resultado dessa expressão mostra uma mensagem de log e copia o arquivo para diretórios diferentes. O código usa o método log para acessar o *logger* configurado como parte da configuração inicial do projeto e definido no arquivo *log4j.properties*. Em seguida, define os terminadores da rota usando o método *to*. O terminador usa, novamente, o componente *file* para definir o diretório onde os arquivos serão salvos (*target/messages/uk* caso a expressão *xpath* resulte em verdadeiro ou *target/messages/others* caso contrário).

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 algumas das necessidades mais comuns nesse sentido.

Apache ActiveMQ Artemis

O Apache ActiveMQ Artemis é um sistema moderno de mensageria de código aberto, escrito em Java, e mantido pela Fundação de Software Apache. Atualmente encontra-se em desenvolvimento constante sendo trabalhando como o sucessor natural do tradicional Apache ActiveMQ e a próxima evolução em termos de mensageria tradicional. É comumente utilizado como uma solução de mensageria prática e de alta performance. Suportando dois mecanismos de persistência diferentes, baseados em NIO ou AIO, o Artemis consegue prover um misto de robustez, baixa latência e boa velocidade mesmo quando atuando em modo persistente.

Antes de criarmos o projeto com o Camel, precisamos criar uma instância de um *broker* Artemis. Esse passo pode ser realizado facilmente através de um container. O projeto Artemis fornece, junto com seu código fonte, instruções para a criação de containers. Podemos, também, utilizar containers fornecidos pela comunidade. Por exemplo:

```
docker run -it --rm -p 8161:8161 -p 61616:61616 -p 5672:5672 -e
ARTEMIS_USERNAME=admin -e ARTEMIS_PASSWORD=admin vromero/activemq-
artemis:2.15.0-alpine
```

Com isso temos uma instância do Artemis ouvindo nas portas 5672 (protocolo *AMQP 1.0*), 61616 (*OpenWire*, *CORE*, *AMQP*, *Stomp*) e 8161 (*HTTP/admin*). Podemos acessar a interface de administração para verificar se a instância está rodando com sucesso. Para isso basta acessar o endereço http://localhost:8161.

Uma vez que o Artemis esteja rodando com sucesso podemos seguir adiante e criar um projeto simples usando o Camel. Para isso, executamos o seguinte comando:

```
mvn archetype:generate -B -DarchetypeGroupId=org.apache.camel.archetypes
-DarchetypeArtifactId=camel-archetype-java -DarchetypeVersion=3.4.4
-DgroupId=camel-passo-a-passo -DartifactId=activemq-app-camel
-Dversion=1.0.0-SNAPSHOT -Dpackage=activemq.app.camel
```

Assim como no caso do primeiro projeto criado anteriormente, o resultado do comando acima será um projeto básico com o Camel e alguns dos arquivos mínimos necessários para a

execução. Modificaremos esse projeto para que ele envie os arquivos para filas diferentes na nossa instância do Apache Artemis.

Para a comunicação com o nosso *broker* Artemis, podemos utilizar o componente *Simple JMS 2*, também conhecido como *SJMS2*. Esse é um componente que implementa boas práticas do padrão *Jakarta Messaging API* (*JMS*) e fornece um mecanismo simples para troca de dados com soluções e protocolos suportados por este padrão.

O primeiro passo para modificar nosso projeto para este propósito é adicionar o componente *camel-sjms2* na lista de dependências do projeto. Fazemos isso modificando o arquivo *pom.xml* e adicionando o seguinte na lista de dependências:

```
<dependency>
     <groupId>org.apache.camel</groupId>
     <artifactId>camel-sjms2</artifactId>
</dependency>
```

Precisamos, também, de uma provedor para os "*Connection Factories*". Uma implementação concreta que provê o código necessário para a comunicação com o broker JMS. Para tal, podemos usar diversos projetos como QPid JMS, Artemis JMS e outros. Neste exemplo, usamos o QPid JMS. Desta forma, precisamos adicionar a seguinte dependência:

```
<dependency>
    <groupId>org.apache.qpid</groupId>
    <artifactId>qpid-jms-client</artifactId>
    <version>0.54.0</version>
</dependency>
```

A partir de então, pode-se criar uma instância do objeto de configuração da *connection factory*, configura-la com os parâmetros de conexão adiciona-lo ao contexto do Camel. Podemos adicionar os códigos a seguir no método configure da classe *MyRouteBuilder*.

O exemplo abaixo mostra uma configuração simples de acesso a um *broker* Active MQ configurado com autenticação por usuário e senha (admin/admin, os mesmos utilizados para rodar a instância do Artemis):

```
     JmsConnectionFactory connFactory = new
JmsConnectionFactory("amqp://localhost:5672");

     // Usuário e senha da instância do broker Artemis
     connFactory.setUsername("admin");
     connFactory.setPassword("admin");
```

Uma vez que o objeto de configuração foi instanciado, é possível utiliza-lo para instanciar um novo componente *Simple JMS 2*:

```
     Sjms2Component component = new Sjms2Component();

     component.setConnectionFactory(connFactory);
```

Então adicionamos o componente no contexto do Camel, para que ele mapeie os *endpoints sjms2* para esse componente configurado previamente:

```
      // Mapeamos endpoint sjms2 para o componente previamente
configurado
      getContext().addComponent("sjms2", component);
```

Por fim, podemos modificar a rota para que redirecione os arquivos XML para as filas uk ou others, baseado no conteúdo dos arquivos.

```
from("file:src/data?noop=true")
    .choice()
        .when(xpath("/person/city = 'London'"))
            .log("UK message")
            .to("sjms2://queue/uk")
        .otherwise()
            .log("Other message")
            .to("sjms2://queue/others");
```

Para executar o projeto, enviando as mensagens para o Artemis, podemos rodar o seguinte comando:

```
mvn clean package && mvn exec:java
-Dexec.mainClass="activemq.app.camel.MainApp"
```

Podemos checar se as mensagens foram enviadas com sucesso para o Artemis, acessando a interface de administração e verificando a existência de, pelo menos, 1 mensagem em cada um das filas "*uk*" e "*others*".

 **Dica:**

Alguns dos connection factories comumente utilizados com o SJMS2 são:
- org.apache.activemq.artemis.jms.client.ActiveMQConnectionFactory
- org.apache.activemq.ActiveMQConnectionFactory

 **Dica:**

Fique atento para importar corretamente as classes utilizadas neste exemplo:
- *import org.apache.camel.component.sjms2.Sjms2Component;*
- *import org.apache.qpid.jms.JmsConnectionFactory;*

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 URI utilizada para especificar o endereçamento do *endpoint* e suas opções, incluindo, por exemplo, caminhos e diretórios quando necessário. 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 formato segue o padrão utilizado para a parte "*query*" utilizadas em URI. Ou seja, assemelhasse ao seguinte formato: *?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);
}
```

Manipulando Dados em Trânsito

Durante o roteamento e a mediação dos dados em trânsito, uma necessidade bastante comum é a de manipular, transformar e enriquece-los. O Camel fornece diversos mecanismos

para esse fim, desde extensões disponíveis na própria Java DSL até processadores com capacidade de manipular diretamente os dados em trânsito.

Adicionar cabeçalhos (readers) através do método *setHeader* é um exemplo be comum desse tipo de manipulação:

```
from("direct:start")
      .setHeader(Exchange.CONTENT_TYPE, constant("application/x-www-form-
urlencoded"))
      .setHeader(Exchange.HTTP_METHOD, constant("POST"))
      …
      .to("http://meu.endpoint.rest")
```

Processadores

A forma mais flexível de manipular dados em trânsito é através de processadores. Um processador é uma classe com propósito de fazer manipulações complexas no conteúdo de um exchange. Esse processador, conhecido como *processor* no jargão do Camel, 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 *org.apache.camel.Processor* e, então, implementando a lógica de processamento no método *process*.

```
public void process(Exchange exchange) throws Exception {
       Status status = exchange.getIn().getBody(Status.class);
}
```

No exemplo acima o processador, utilizado como parte de uma rota responsável por obter dados do Twitter, utiliza as rotinas do Exchange para obter um objeto Status que representa uma postagem do Twitter.

Gerenciamento de Erros na Rota

O Camel fornece três mecanismos para gerenciar excessões na rota. Desta forma, é possível ter um bom nível de flexibilidade para gerenciar erros e excessões da maneira mais adequada para a aplicação.

Requerimentos específicos de cada aplicação fazem com que seja necessário avaliar com cautela a melhor estratégia para gerenciar os erros. Detalhes como gerenciamento de transações, rastreamento e monitoramento são algumas das razões pelas quais é preciso pensar com cautela na estratégia adequada para tratamento de erros.

No Camel o gerenciamento de erros é definido durante a configuração da rota e pode ser aplicado tanto para toda a extensão da rota, quanto para a determinadas partes. Não sendo definido nenhum mecanismo, comportamento padrão do Camel será o de logar as excessões e parar a execução da rota para o dado em trânsito.

Os mecanismos de tratamento de erro do Camel são ativados sempre que uma excessão não tratada ocorre em algum ponto da rota.

Gerenciando Erros via DSL

O primeiro mecanismo disponível permite o tratamento de erros em partes específicas da rota usando um forma similar ao *try/catch/finally* do Java.

Ao definir a rota é possível usar os métodos *doTry*, *doCatch*, *doFinally* e *end* para capturar e tratar excessões em pontos específicos da rota. Usando a Java DSL, uma rota definida através desse mecanismo seria parecida com o seguinte:

```
public void configure() throws Exception {
      from("componente:rota")
      .doTry()
            // código da rota que pode ou não lançar excessões
      .doCatch(ClasseDaExcessao.class)
            // tratamento da rota, por exemplo, logar o erro
      .doFinally()
            /* Bloco opcional contento código da rota que será
             * aplicado em qualquer circunstância
             */
      .end()
}
```

Gerenciando Erros via Gerenciador de Erros (Error Handler)

O segundo mecanismo disponível fornece uma maneira de tratar qualquer tipo de erro lançado durante a execução da rota. O Camel já provê, por padrão, alguns tratadores de erros para situações comuns. Entre essas funcionalidades estão, por exemplo, a possibilidade de gerenciar erros em rotas transacionadas ou enviar os dados em trânsito para um canal "*dead letter*".

Para fazer o tratamento de erros dessa forma é necessário definir um gerenciador de erros através do método *errorHandler*. Por exemplo, para usar o gerenciador de erros que redirecionaria o dado em trânsito para um canal *"dead letter"*:

```
public void configure() throws Exception {
      errorHandler(deadLetterChannel("direct:errors"));
}
```

Gerenciamento de Erros por Tratamento de Excessão na Rota

O último 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 {
      onException(ClasseDaExcessao.class)
            .handled(true)
            // código da rota de excessão

      // declaração da roda
}
```

Conversores de Dados

Os conversores de dados permitem a implementação de rotinas de conversão de dados entre o formato que é utilizado para a comunicação entre as aplicações. O Camel contém uma série de conversores de tipos que podem ser configurados para facilitar a conversão de dados ao longo das rotas.

Configurando Formato de Dados

Um uso comum para os conversores de dados é serializar ou de-serializar dados em formato JSON. O conversor JacksonDataFormat é um dos conversores fornecidos por padrão com o Camel.

Para configurar e usar esse conversor em uma rota, é possível fazer o seguinte:

```
public void configure() throws Exception {
      JacksonDataFormat format = new JacksonDataFormat();
      format.setUnmarshalType(MinhaClasse.class);

      from("endpoint")
            .unmarshal(format)
            // código da rota
}
```

Dica:

O Apache Camel contém diversos tipos de conversores de dados. Alguns deles são específicos para determinados componentes. Não deixe de conferir a documentação oficial do projeto antes de implementar o seu próprio conversor.

Camel Quarkus

O que é o Quarkus?

O Quarkus é um framework Java otimizado para uso em containeres. Além de suportar os frameworks e padrões mais populares, o Quarkus destaca-se por facilitar a criação de aplicações com que consomem menos memória e tem um tempo de inicialização bastante reduzido. Essas duas características, entre outras, fazem com que o Quarkus seja um framework ideal para aplicações em nuvem.

Além de ser otimizado para containeres, o Quarkus também é otimizado para tornar o desenvolvimento prazeroso para o desenvolvedor. Funcionalidades como codificação em tempo real ("live coding"), configuração unificada e facilidade de geração de executáveis nativos tornam o desenvolvimento com Quarkus rápido e prático.

O Apache Camel está entre os diversos frameworks suportados pelo Quarkus. Através do Camel Quarkus, o módulo de componentes de integração do Camel suportado pelo Quarkus, os desenvolvedores podem criar aplicações de integração que usufruem das funcionalidades desses dois frameworks.

Começando com o Camel Quarkus

Assim como nos exemplos anteriores, podemos utilizar arquétipos do Maven para gerar um projeto Camel Quarkus de exemplo.

Para criar um projeto de exemplo, consumindo mensagens de um tópico do Apache Kafka, podemos executar os seguintes passos:

```
mvn io.quarkus:quarkus-maven-plugin:1.9.2.Final:create
-DprojectGroupId=camel-passo-a-passo -DprojectArtifactId=kafka-camel-
quarkus -Dextensions=camel-quarkus-log,camel-quarkus-kafka
-DclassName=kafka.app.quarkus.KafkaRoute
```

Isso criará um projeto de exemplo do Quarkus com todos os módulos (no caso, *camel-quarkus-log* e *camel-quarkus-kafka*) e dependências pré-configurados, bem como uma class *KafkaRoute* onde podemos codificar a rota. A criação do projeto através desse arquétipo criará, entretanto, uma classe “*resource*" padrão do do Quarkus ao invés de uma rota do Camel. Para transforma-la em uma rota, o processo é bastante simples.

A primeira dessas alterações é anotar a classe com a anotação *@ApplicationScoped*. Essa anotação é necessária para que a injeção da configuração, feita através da anotação *@ConfigProperty*, funcione corretamente. De modo geral, não seria necessário e, até mesmo, não recomendável, pois esse escopo tem um ciclo de vida mais complexo e causa um aumento no tempo de inicialização. Posteriormente definimos uma variável membro do tipo *String* que irá conter o endereço do Kafka. Essa variável será anotada com *@ConfigProperty* para permitir que a configuração do endereço do Kafka seja feita através de qualquer um dos mecanismos de configuração suportados pelo Quarkus.

O segundo passo é fazer com que a rota especialize a classe *RouteBuilder*. Esse passo já é conhecido e aplicado ao usar Camel com outros frameworks. Conforme requerido pela super-classe, precisamos definir um método *configure* onde será definido a lógica da rota. Utilizaremos o método fromF para passar parâmetros de configuração da rota sem a necessidade de concatenar *strings*.

```
import javax.enterprise.context.ApplicationScoped;

import org.apache.camel.builder.RouteBuilder;
import org.eclipse.microprofile.config.inject.ConfigProperty;

@ApplicationScoped
public class KafkaRoute extends RouteBuilder {

   @ConfigProperty(name = "kafka.bootstrap.address")
    String bootstrapAddress;

    @Override
    public void configure() throws Exception {
        fromF("kafka:exemplo?brokers=%s", bootstrapAddress)
            .to("log:info ${body}");
    }
}
```

Para testar o exemplo, será necessário subir uma instância do Apache Kafka. Este processo costuma ser um pouco trabalhoso. Para simplificar, podemos utilizar o Docker Compose com o template disponível nos exemplo kafka-camel-quarkus do ebook.

Primeiramente rodamos o docker compose para subir o Kafka e o Zookeeper:

```
docker-compose up
```

Assim que estiverem rodando, podemos abrir outro terminal e rodar o código de exemplo. Note como o endereço do Kafka é passado como argumento para o comando. Essa é apenas uma das muitas outras formas suportadas pelo Quarkus. Entre essas formas, inclui-se a possiblidade de configurar o parâmetro no arquivo application.properties, passar o valor através de variáveis de ambiente e utilizar diferentes valores de acordo com o ambiente em execução (desenvolvimento, testes ou produção). Para executar o código, usamos o comando:

```
mvn -Dkafka.bootstrap.address=localhost:9092 clean compile quarkus:dev
```

O último passo é executar o cliente de linha de comando do Kafka para inserir algumas mensagens no tópico *exemplo*. Note que o comando abaixo é levemente mais complicado do que o necessário para evitar o trabalho de procurar manualmente o nome do container do Kafka na saída do comando docker ps:

```
docker exec -it $(docker ps --format '{{.Names}}' --filter name=^kafka-
camel-quarkus_kafka) /opt/kafka/bin/kafka-console-producer.sh --
bootstrap-server localhost:9092 --topic exemplo
```

No terminal, você pode digitar as mensagens a serem consumidas pelo código de exemplo:

```
>Olá mundo!
>
```

Elas serão consumidas e mostradas no terminal onde o código de exemplo está rodando:

```
2020-11-15 12:36:22,954 INFO  [info ${body}] (Camel (camel-1) thread #0 -
KafkaConsumer[exemplo]) Exchange[ExchangePattern: InOnly, BodyType:
String, Body: Olá mundo!]
```

Para sair da console do cliente do Kafka, utilize *Ctrl-D*. Para terminar a execução do código de exemplo e o docker compose, utilize *Ctrl+C* em ambos.

Licenças

Creditos

Créditos da imagem na capa: "Camel" by @Doug88888 is licensed under CC BY-NC-SA 2.0 .

Material

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 – JMS (https://camel.apache.org/components/latest/jms-component.html). Apache Software Foundation, 2014.