## Gravata

Sempre nos deparamos com desafios em relação à integração entre sistemas, como fazer de maneira ágil e também de maneira simples e clara para futuras manutenções e porque não reutilizáveis. Foi em face destas dificuldades que em 2007 o James Strachan criou o Apache Camel desde então o crescimento do framework vem em crescimento constante. Apache Camel é um framework de integração de código aberto baseado em conhecidos [padrões empresariais de integração (EIP)](http://www.devmedia.com.br/artigo-java-magazine-70-siga-a-rota-do-camelo/13549), o uso de padrões pode simplificar o entendimento das integrações e com uma grande vantagem: poucas linhas de código!

Introdução

Hoje integração entre sistemas é tão popular no mundo dos desenvolvedores que não é difícil nos depararmos com uma integração por mais simples que seja o sistema. E por mais simples que seja as integrações geralmente usamos diversos frameworks para atender todas as integrações, fazendo com que o projeto fique confuso e de difícil manutenção.

A insatisfação dos desenvolvedores em relação à diversidade de frameworks existentes nos projetos é nítida, dificuldades de implementação devido às complexidades e grande quantidade de linhas de código acaba tornando a implementação demorada, com isso acaba se refletindo também na insatisfação gerencial dos projetos. Quem nunca já ouviu uma solicitação do gerente e/ou cliente para acrescentar uma determinada integração ou também alterar uma determinada integração já existente, o que vem à mente? – Justo aquela integração que tem “um milhão de linhas”? vai demorar muito isso e como vamos testar agora?

A ideia de criar o Apache Camel foi justamente diminuir a extensa lista de frameworks para integrações existentes nos projetos, isso com uma menor quantidade de linhas de código, desenvolvendo de modo ágil e que mesmo assim fique de maneira clara, mantendo a possibilidade de manutenção ágil.

O objetivo deste artigo é apresentar uma visão geral do Apache Camel: características, operações, interfaces de monitoramento e API. Na primeira parte serão apresentadas a arquitetura do Camel. E no segundo tópico apresenta a instalação, configuração e o monitoramento Camel. Por último, serão apresentadas a API do Camel e exemplos de utilização em uma aplicação.

Características do Apache Camel

O Apache Camel é um mecanismo de roteamento e mediação de mensagens, que nos permitem criar nossas próprias regras de processamento. Ele nos oferece um alto nível de abstração que nos permitem interagir com sistemas distintos independente do protocolo ou tipo de dados que os sistemas utilizam. O Camel foi baseado na publicação do livro ‘Enterprise Integration Patterns’ onde no framework Camel foi implementado mais de 50 padrões existente na publicação do autores Gregor Hohpe e Bobby Woolf, todos os padrões implementados no framework pode ser visto no site do Apache Camel (<http://camel.apache.org/enterprise-integration-patterns.html>).

Para atender todos os padrões existente foram implementados mais de 80 protocolos e conversores de dados, fazendo com que o framework tenha uma característica de implementação ágil. Para entender um pouco mais como toda essa “mágica” funciona, devemos conhecer basicamente duas características muito importantes: DSL(Domain Specific Languages), modelo de troca de mensagem e RouteBuilder.

DLS, primeiramente é importante saber o que é, de maneira simplista uma Domain Specific Language (DSL) é uma linguagem de programação de expressividade limitada, focada num domínio particular. Alguns poucos frameworks possui essa capacidade de DSL, expondo apenas XMLs como definição de regras para rotas. No caso do Camel existem DSLs especificas para algumas linguagens de programação(Java, Scala) e/ou frameworks que se integram a ele(Spring, Groovy). Para facilitar o entendimento do que é uma DSL segue alguns exemplos de implementação bem conhecida dos leitores: CSS (Definição de estilo),SQL (consulta em bases de dados relacionais), todas essas são linguagens de domínio particular.

Segue um exemplo de DSL utilizando diferentes linguagens, que podemos obter o mesmo funcionamento:

Código

* **Java DSL**

from("file:dados/recebidos") .to("file:dados/processados1?fileName=${date:now:yyyyMMdd}-${file:name}") .to("file:dados/processados2"); ■

* **Spring DSL**

<route>

<from uri="file:dados/recebidos"/>

<to uri="file:dados/processados1?fileName=${date:now:yyyyMMdd}-${file:name}"/>

<to uri="file:dados/processados2"/>

</route>

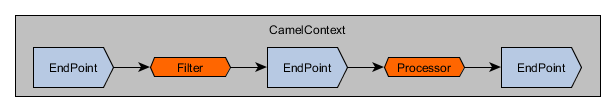
* **Scala DSL**

from "file:dados/recebidos" -> ""file:dados/processados1?fileName=${date:now:yyyyMMdd}-${file:name}"" -> "file:dados/processados2"

No Camel a DLS é utilizada para implementar dos padrões do livro e podemos ver que com a utilização DLS Camel nos possibilita em desenvolver concentrando-se apenas na integração necessária, independente da linguagem utilizada. As facilidades de orquestração das integrações se tornam nítidas e simples de manter, no exemplo Código 1 é esperado qualquer tipo de arquivo no diretório dados/recebidos e enviado para dois endpoints com características diferentes, um com a data o processamento e outro sem qualquer alteração. E como já foi falado no artigo a facilidade integração é tão impressionante que caso seja necessário enviar o arquivo recebido para mais um local seria necessário apenas criar mais uma linha com o destino, isso independente dos protocolos existentes no Camel, por exemplo poderíamos utilizar um endpoint para um FTP ou até mesmo um mecanismo de mensagem, como o apache activeMQ. Tudo isso graças ao modelo de conversões do Camel, onde não precisamos nos preocupar com a conversão de dados entre os protocolos.

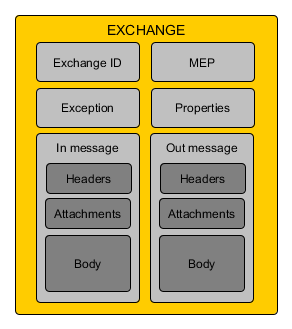
Arquitetura do apache camel

O Camel possui uma arquitetura bem simples baseada em uma grande biblioteca de componentes, onde tais componentes são orquestrados utilizando DSL para configurar rotas, regras de processadores de filtro(Filter Processor) e processadores de rota(Route Processor). Onde deve-se haver um contexto camel criado para que seja possível instanciar o ponto de acesso das rotas que sãos configuradas pelas rotas e pelos endpoints, que são construídos pelos componentes existentes no Camel, podendo ser utilizados tanto para receber quanto para enviar mensagens. Na Figura 1 pode-se visualizar um fluxo básico de uma mensagem entre endpoints e processors de filtro e rotas.



Figura

A finalidade dos processadores são diversas, com eles podemos realizar transformações, enriquecimentos das mensagens, interceptações, validações entre outras. A utilização dos componentes camel faz com que a troca de informações entre os endpoints sejam transparentes, digo, não há qualquer necessidade de codificação específica nas trocas de mensagens entre os componentes. As mensagens trocadas entre as rotas e processadores possui um padrão conhecido como MEP (message exchange patterns) onde podemos por meio de cabeçalhos diferenciar o modo de troca da mensagem que são: ‘one-way message’ e ‘request-response message’. Na ilustramos o formato onde há as troca de mensagem entre as rotas e processadores, de maneira geral podem trocar informações utilizando os headers, no caso de erros no processamento a mensagem de erro é encapsula. Podemos notar que existem duas estruturas idênticas: In message: de uso obrigatório onde contem a mensagem requisitada no corpo e Out message: utilizado para encapsular o retorno/resposta da mensagem.



Figura

Instalação, configuração e monitoramento.

Vamos agora exercitar um pouco mais na pratica como que realmente funciona o desenvolvimento utilizando o camel, no exemplo que será discutido iremos esperar um arquivo .csv como entrada, onde cada linha do arquivo será tratado individualmente de acordo com o conteúdo.

Inicialmente temos que configurar as dependências necessárias do camel e do componente utilizado para a leitura do .csv, .

<dependency>

<groupId>org.apache.camel</groupId>

<artifactId>camel-core</artifactId>

<version>2.12.1</version>

</dependency>

<dependency>

<groupId>org.apache.camel</groupId>

<artifactId>camel-csv</artifactId>

<version>2.12.1</version>

</dependency>

Código

Como já foi dito para que seja possível iniciar as rotas é necessário primeiramente criar um contexto camel onde serão adicionadas a rotas desejadas, lembrando também que o interessante de criar uma rota para cada responsabilidade é poder reutiliza-las, no Código 3 é demonstrado como criamos um contexto de maneira que seja configurada e adicionadas as rotas desejadas.

Linha 1: **public** **static** **void** main(String[] args) **throws** Exception {

Linha 2: // Criação da contexto camel.

Linha 3: CamelContext camelContext = **new** DefaultCamelContext();

Linha 4: // adicionando configuração das rotas

Linha 5: camelContext.addRoutes(**new** RouteBuilder() {

Linha 6: @Override

Linha 7: **public** **void** configure() **throws** Exception {

Linha 8: ....

Linha 9: }

Linha 10:}

Código

Com a classe main criada, agora é necessário criar a primeira rota que será responsável por ler um arquivo .csv de um determinado diretório e também “quebrar” o conteúdo do arquivo em diversos registros, isso porque queremos processar separadamente cada linha do arquivo (Código 4). Para que possamos identificar o arquivo no diretório utilizamos uma rota com o endpoint do tipo file que é responsável pela manipulação do arquivo, o endpoint é formado pela seguinte sintaxe: file:directoryName[?options], onde colocamos o diretório que desejamos ler o arquivo, com alguns possíveis parâmetros, em nosso exemplo utilizaremos o parâmetro *noop* que quanto é igual a true o arquivo de origem não será apagado após o processamento e também o parametro *include* onde configuramos uma expressão regular para ler apenas arquivos .csv, essa configuração pode ser vista na linha 3, na próxima linha temos o *routeId* utilizado apenas para nomear a rota criada, caso não seja dado um nome na rota o camel irá nomear automaticamente como route1,route2 e assim por diante para as demais rotas criadas no contexto.

Linha 1: //ROTA 1: Responsável por receber o arquivo csv

Linha 2: // e também separar o conteudo do arquivo em várias linha

Linha 3: from("file:dados/recebidos?noop=true&include=.\*.csv$")

Linha 4:.routeId("recebidos-csv")

Linha 5: .unmarshal().csv()

Linha 6: .split(body(ArrayList.**class**))

Linha 7: .to("direct:processa");

Código

Na linha 5, agora iremos transformar e formatar o conteúdo do arquivo para um conteúdo csv, isso significa o que exatamente? Significa que cada linha do arquivo será transformado em um array de String, logo na linha seguinte é utilizado mais um padrão (*Splitter*) onde utilizaremos mais uma facilidade do camel para tratamento de dados, ou seja, na linha 6 cada linha do arquivo será um objeto em um ArrayList. Com o ArrayList contendo o conteúdo do arquivo em mãos agora é só mandar para ser processado utilizando o conector *direct.* O *endpoint direct* se refere a uma chamada síncrona para qualquer consumidor, desde que o produtor da mensagem esteja enviado um *message exchange*, o *direct* só pode se conectar em uma rota do mesmo camel context.

Linha 1: // ROTA: Separa os tipos de registros

Linha 2: from("direct:processa")

Linha 3: .routeId("processa-item")

Linha 4: .choice()

Linha 5: .when(simple("${body[2]} == 'download'"))

Linha 6: .to("direct:processaItemDownload")

Linha 7: .otherwise()

Linha 8: .to("direct:processaItem")

Linha 9: .end();

Código

Após ler o arquivo e transformar o conteúdo em um *ArrayList* com as linhas do arquivo, agora ficou fácil separar o processamento específico de cada tipo de informação (Código 5). Iremos utilizar um tipo de seletor de informações o *choice*, que é uma estrutura condicional bem simples, é passado um conjunto de informação onde o tipo de informação pode ser verificado *n* vezes, e caso contrário é executado um *endpoint* padrão definido. Na linha 2, é recebida a lista com os registros onde a primeira validação é se o tipo do registro é de *download*, lembrando que na primeira rota criada o *ArrayList* é criado com a seguinte estrutura *ArrayList<String[]>*.