03

Conectando endpoints HTTP

Downloads

Caso queira começar o treinamento a partir dessa aula, pode baixar o projeto aqui (https://s3.amazonaws.com/caelum-online-public/camel/camel-stage-cap3.zip). Só baixe este arquivo, se não tiver feito os exercícios dos capítulos anteriores.

Revisão do capitulo anterior

Vimos como preparar a nossa mensagem na rota para funcionar com o serviço web. Transformamos, filtramos e dividimos o conteúdo, tudo isso do alto nível que a Camel DSL provê. Sempre seguimos as boas práticas definidas pelos padrões de integração.

```
from("file:pedidos?delay=5s&noop=true").
    split().
        xpath("/pedido/itens/item").
    filter().
        xpath("/item/formato[text()='EBOOK']").
    marshal().
        xmljson().
    log("${id} - ${body}").
    setHeader(Exchange.FILE_NAME, simple("${file:name.noext}-${header.CamelSplitIndex}.json")).
to("file:saida");
```

A seguir, vamos chamar o serviço web para entregar o JSON gerado.

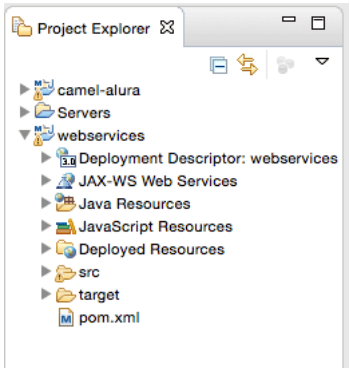
Configuração do serviço web

Para podermos testar o serviço vamos instalar o Tomcat e importar um projeto, o primeiro passo é instalar o Tomcat. Nesta aula, usamos o Tomcat 8 disponível no site do Apache (http://tomcat.apache.org/download-80.cgi).

Caso esteja com dúvidas de como instalar o Tomcat e configurar o projeto, preparamos um exercício dedicado para essa tarefa.

Vamos baixar e importar o projeto de web services. Trata-se de um projeto web e segue os mesmos passos de importação do projeto Camel:

1. Baixe o projeto webservices.zip (https://s3.amazonaws.com/caelum-online-public/camel/webservices.zip) e extraia o ZIP.
2. No Eclipse: *File > Import*
3. Depois, *Maven > Existing Maven Project*
4. Escolha a pasta do projeto (`webservices`) e confirme a importação.
5. Adicione o projeto no Tomcat.



Atenção: A configuração inicial do projeto pode demorar pois o Maven precisa baixar as dependências.

Testando o serviço

Vamos subir o Tomcat e testar o serviço pelo navegador:

http://localhost:8080/webservices/ebook/item (http://localhost:8080/webservices/ebook/item)

Você deve ver no navegador: **service ebook ok**

Configurando a chamada HTTP

Se o serviço web está rodando, podemos implementar a chamada HTTP com Camel. Para tal, o Camel possui um componente com o nome `http4`. O número `4` inserido no nome do componente é devido ao fato do Camel usar a biblioteca HTTP Client na versão 4 (http://camel.apache.org/http4.html), por baixo dos panos.

Sabendo disso, já podemos testar, mas como configurar a URI? Faremos isso de forma simples, já que ele segue a mesma sintaxe das URIs no navegador:

```
from("file:pedidos?delay=5s&noop=true").
    split().
        xpath("/pedido/itens/item").
    filter().
        xpath("/item/formato[text()='EBOOK']").
        marshal()
        .xmljson().
       log("${id} \n ${body}").
to("http4://localhost:8080/webservices/ebook/item");
```

O componente `http4` não faz parte do componente principal `camel-core`. Isso é diferente do componente `file` que já vem embutido no `camel-core`. Para usar o `http4`, devemos importar a dependência de maneira que ela faça parte do classpath. Basta colocar a dependências no arquivo `pom.xml`, já que usamos o Maven. Nós fizemos isso no arquivo `pom.xml` e você não precisa se preocupar.

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-http4</artifactId>
    <version>${camel-version}</version>
</dependency>
```

Agora podemos testar o código e executar. Verifique o console do Tomcat para ver se o serviço web realmente recebeu a chamada. Deverá aparecer **quatro** chamadas HTTP POST mostrando o JSON, por exemplo:

```
Ebook (HTTP) recebendo POST  {"formato":"EBOOK","quantidade":"1","livro":{"codigo":"ARQ","titulo
```

É importante mencionar que se o serviço web falha também falhará o envio da mensagem. Ou seja, o Camel só considera a entrega da mensagem com sucesso se o serviço web devolve 200 (algum status da categoria 2xx). Mais adiante, detalharemos o assunto de tratamento de exceções, fique tranquilo.

Configurando o método HTTP

Repare que o Camel enviou um POST automaticamente. Esse é o padrão do Camel quando há um corpo (*body*) da mensagem. Podemos deixar a nossa intenção explícita usando um cabeçalho específico:

```
from("file:pedidos?delay=5s&noop=true").
    split().
        xpath("/pedido/itens/item").
    filter().
        xpath("/item/formato[text()='EBOOK']").
        marshal()
        .xmljson().
       log("${id} \n ${body}").
       setHeader(Exchange.HTTP_METHOD, constant(org.apache.camel.component.http4.HttpMethods.POS
to("http4://localhost:8080/webservices/ebook/item");
```

Observe que usamos o método `constant()` no valor do header, responsável por definir o método HTTP. Como o Camel já tinha usado um POST, o resultado será o mesmo se executarmos novamente.

Repare também que usamos outra constante, mas da classe `Exchange` para definir a chave do cabeçalho. A classe `Exchange` é fundamental no Camel, não só para o uso de constantes como também para trabalhar com a mensagem em si. Uma mensagem no Camel é do tipo `Exchange`. Veremos ainda mais sobre esta classe.

Usando GET com query params

E se nosso serviço fosse um HTTP GET? Nas integrações no mundo web, ambos os métodos HTTP vão aparecer, você pode ter certeza disso.

Nós sabemos alterar o método HTTP, trocando `HttpMethods.POST` por `HttpMethods.GET`. No entanto, se usarmos GET, trabalharemos com os parâmetros da requisição para passar informações ao serviço. O nosso serviço web espera os parâmetros via GET, podemos testar isso no navegador:

```
http://localhost:8080/webservices/ebook/item?ebookId=ARQ&pedidoId=2451256&clienteId=edgar.b@abc
```

Verifique o console do Tomcat para ver se o envio dos parâmetros funcionou:

> Ebook (HTTP) recebendo GET /webservices/ebook/item - Ebook ID: ARQ, Cliente ID: edgar.b@abc.com, Pedido ID: 2451256

Para a definição dos parâmetros, devemos usar um cabeçalho da mensagem mais uma vez. Neste caso, o cabeçalho se chama `Exchange.HTTP_QUERY` e como valor recebe os seguintes parâmetros:

```
setHeader(Exchange.HTTP_QUERY,    constant("clienteId=breno@abc.com&pedidoId=123&ebookId=ARQ"))
```

Podemos testar a rota e executá-la novamente, só que os parâmetros não devem ser fixos na query - e devem vir de cada XML do pedido.

##Usando properties da rota

Falta pouco para completar a rota usando um HTTP GET. Devemos extrair as informações do XML para definir a id do pedido, do cliente e do ebook.

Novamente, o XPath vai ser útil neste trabalho, por exemplo, para descobrir o id do pedido usaremos: `xpath("/pedido/id/text()`

A questão ainda é onde vamos guardar este valor? Poderíamos usar o próprio cabeçalho da mensagem:

```
setHeader(“pedidoId”, xpath("/pedido/id/text())
```

Apesar de ser funcional, não é a melhor opção, pois os headers normalmente definem a meta-informação sobre a mensagem e o protocolo usado (por exemplo, a definição do método HTTP). Para resolver isso podemos usar o método `setProperty` que serve para guardar dados da rota:

```
setProperty("pedidoId", xpath("/pedido/id/text()"))
```

Uma vez guardada uma propriedade, podemos recuperá-la por meio da *Expression language* usando o método `simple()`:

```
simple(“${property.pedidoId}”)
```

Colocando tudo isso na nossa rota (3 propriedades e EL):

```
from("file:pedidos?delay=5s&noop=true").
    setProperty("pedidoId", xpath("/pedido/id/text()")).
    setProperty("clienteId", xpath("/pedido/pagamento/email-titular/text()")).
    split().
        xpath("/pedido/itens/item").
    filter().
        xpath("/item/formato[text()='EBOOK']").
    setProperty("ebookId", xpath("/item/livro/codigo/text()")).
    log("${id} \n ${body}").
    marshal().
        xmljson().
```

```
    setHeader(Exchange.HTTP_QUERY,
            simple("clienteId=${property.clienteId}&pedidoId=${property.pedidoId}&ebookId=${pro
to("http4://localhost:8080/webservices/ebook/item");
```

A ordem é importante nesse momento. Observe que as duas primeiras propriedades vem do XML do pedido completo. A terceira propriedade (`ebookId`) é definida para cada item, antes do `marshal().xmljson()` e depois, do `filter()` .

O que aprendemos?

- trabalhar com o componente `http4` ;
- enviar GET e POST;
- definir params da requisição GET;
- definir e recuperar propriedades na rota.