03

Validação de mensagens e tratamento de erros

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-cap6.zip). Baixe este arquivo, se não tiver feito os exercícios dos capítulos anteriores.

Revisão do capítulo anterior

Transformamos um XML usando um template XSLT, para criar a mensagem SOAP baseado no XML do pedido. Todo esse processamento ficou na sub-rota `soap`:

```
from("direct:soap").
    routeId("rota-soap").
to("xslt:pedido-para-soap.xslt").
    log("Resultado do template: ${body}").
    setHeader(Exchange.CONTENT_TYPE,constant("text/xml")).
to("http4://localhost:8080/webservices/financeiro");
```

Para executar a requisição HTTP POST, usaremos o conhecido componente `http4`.

Conhecendo o Schema

Na integração, estamos trabalhando com sistemas externos dos quais não temos controle. Ou seja, a qualquer momento algum sistema externo pode ser alterado de forma que cause problemas na nossa rota. Por isso, é fundamental validar todos os dados que recebemos. Se fazemos alguma mudança no XML do pedido, provavelmente, algo pode falhar na rota (por exemplo, as expressões XPath).

Vimos que o mundo XML já vem com padrões ao redor como XPath e XSLT. Para validar um XML, também há um padrão: o **XSD** (*XML Schema Definition*). XSD ou *Schema* é um outro XML que descreve formalmente como se compõe um XML, quais são os elementos, atributos e tipos relacionados.

Vejamos um exemplo concreto de um Schema XML descrevendo uma parte do pedido:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema version="1.0" xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="pedido" type="pedido"/>

  <xs:complexType name="pedido">
    <xs:sequence>
         <xs:element name="id" type="xs:positiveInteger" minOccurs="1"/>
      <xs:element name="dataCompra" type="xs:dateTime" minOccurs="1"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Um detalhe que normalmente pode dar um nó na cabeça é o fato dos Schemas, que servem para definir XML, serem definidos em XML. Isso é possível porque há um Schema de Schemas. O leitor pode notar que, na segunda linha do XML acima, há a indicação do Schema para definir outros Schemas.

```
<xs:schema version="1.0" xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

Além dos tipos simples que já existem no XML e que podem ser utilizados dentro de um Schema, há a possibilidade de definir novos tipos, que são os chamados **tipos complexos**. Um tipo complexo pode ser baseado em outros tipos complexos e/ou em tipos simples. Vejamos um exemplo que define um tipo complexo, chamado `tipo_formato` :

```
<xs:simpleType name="tipo_formato">
        <xs:restriction base="xs:string">
          <xs:enumeration value="EBOOK"/>
          <xs:enumeration value="IMPRESSO"/>
        </xs:restriction>
</xs:simpleType>
```

No Schema `tipo_formato` , está definido que todo formato é composto de um valor do tipo string. Os valores possíveis estão declarados na enumeração.

Dessa maneira, podemos definir todo o XSD do pedido, sempre dividindo em subtipos:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema version="1.0" xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="pedido" type="pedido"/>

  <xs:complexType name="pedido">
    <xs:sequence>
      <xs:element name="id" type="xs:positiveInteger" minOccurs="1"/>
      <xs:element name="dataCompra" type="xs:dateTime" minOccurs="1"/>
      <xs:element name="itens" minOccurs="1" maxOccurs="unbounded">
        <xs:complexType>
            <xs:sequence>
              <xs:element name="item" type="itemCompra" minOccurs="1" maxOccurs="unbounded" />
            </xs:sequence>
        </xs:complexType>
      </xs:element>

      <xs:element name="pagamento" type="pagamento" minOccurs="1"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="itemCompra">
    <xs:sequence>
      <xs:element name="formato" type="tipo_formato" minOccurs="0"/>
      <xs:element name="quantidade" type="xs:positiveInteger" minOccurs="1"/>
      <xs:element name="quantidadeEstoque" type="xs:positiveInteger" minOccurs="0"/>
      <xs:element name="livro" type="livro" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="livro">
```

```
    <xs:sequence>
      <xs:element name="codigo" type="xs:string" minOccurs="0"/>
      <xs:element name="titulo" type="xs:string" minOccurs="0"/>
      <xs:element name="tituloCurto" type="xs:string" minOccurs="0"/>
      <xs:element name="nomeAutor" type="xs:string" minOccurs="0"/>
      <xs:element name="imagem" type="xs:string" minOccurs="0"/>
      <xs:element name="valorEbook" type="xs:decimal" minOccurs="0"/>
      <xs:element name="valorImpresso" type="xs:decimal" minOccurs="0"/>
      <xs:element name="descricao" type="xs:string" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="pagamento">
    <xs:sequence>
      <xs:element name="status" type="status_pagamento_type" minOccurs="1"/>
      <xs:element name="valor" type="positive_decimal_type" minOccurs="1"/>
      <xs:element name="titular" type="xs:string" minOccurs="1"/>
      <xs:element name="email-titular" type="xs:string" minOccurs="1"/>
    </xs:sequence>
  </xs:complexType>

  <xs:simpleType name="status_pagamento_type">
    <xs:restriction base="xs:string">
      <xs:enumeration value="CANCELADO"/>
      <xs:enumeration value="CONFIRMADO"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="tipo_formato">
    <xs:restriction base="xs:string">
      <xs:enumeration value="EBOOK"/>
      <xs:enumeration value="IMPRESSO"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="positive_decimal_type">
  <xs:restriction base="xs:decimal">
    <xs:minInclusive value="0"/>
  </xs:restriction>
</xs:simpleType>
</xs:schema>
```

Não é incomum que XSD seja fornecido pelo serviço que estamos executando.

Validando o XSD com Camel

Com o XSD pronto, podemos usá-lo na rota para validar o XML, antes de qualquer sub-rota. O Camel vem pronto para isso e possui um componente de validação. Vamos validar antes de chamar o `multicast`:

```
from("file:pedidos?delay=5s&noop=true").
    routeId("rota-pedidos").
    to("validator:pedido.xsd").//nova validação
    multicast().
        to("direct:soap").
```

```
            log("Chamando soap com ${body}").
        to("direct:http");
```

Para focarmos na validação e simplificar um pouco vamos desabilitar o `multicast`. Ou seja, para testar apenas a validação não vamos chamar as sub-rotas.

```
from("file:pedidos?delay=5s&noop=true").
    routeId("rota-pedidos").
    to("validator:pedido.xsd"); //ponto virgula aqui
//    multicast().
//        to("direct:soap").
//            log("Chamando soap com ${body}").
//        to("direct:http");
```

Agora podemos rodar sem se preocupar com os serviços web. Vamos verificar se todos os arquivos XML estão na pasta `entrada` e executar.

Recebemos um erro de validação do XML:

```
ERROR 13:37:55.507 - Failed delivery for (MessageId: ID-MacBook-Pro-de-Nico-local-53866-1448897
errors: [
org.xml.sax.SAXParseException: cvc-complex-type.2.4.a: Invalid content was found starting with
]. Exchange[4_pedido.xml]
```

As exceções do Camel são bastante verbosas, mas analisando com calma percebemos que o XML `4_pedido.xml` não passou pela validação. Analisando detalhadamente descobriremos uma `SAXParseException` que acusa o elemento `valor`:

```
Invalid content was found starting with element 'valor'. One of '{status}' is expected.
```

Aparentemente falta o elemento `status`!

Tratamento de exceções

Realmente falta o elemento `status` no XML. Ele é essencial para as outras rotas, ou seja, não podemos continuar com esse XML na rota. Analisando o console, perceberemos que o Camel tentou várias vezes ler e validar o XML. Para deixar mais claro e mostrar este comportamento do Camel, novamente, faremos uma pequena alteração na rota. Adicionaremos um `log` e um `delay` na rota:

```
from("file:pedidos?delay=5s&noop=true").
    log("${file:name}"). //logando nome do arquivo
    routeId("rota-pedidos").
    delay(1000). //esperando 1 segundo
to("validator:pedido.xsd");
```

Isso faz que Camel atrase por um segundo a execução, tempo suficiente para entendermos o processamento.

Ao executar novamente, veremos que entre os `stacktraces` sempre aparece o nome do arquivo `4_pedido.xml` até o Camel terminar. O Camel tenta várias vezes entregar a mensagem, sem sucesso. Temos uma mensagem *venenosa* que não pode ser entregue. Isso é motivo suficiente para configurar um tratamento de exceção específico do Camel.

ErrorHandler e DeadLetterChannel

O tratamento de exceções é definido usando um `errorHandler` , porém, por padrão ele não faz muita coisa a não ser logar as exceções e tentar novamente. Nestes casos, é boa prática definir o próprio `errorHandler` por meio da Camel DSL:

```
errorHandler( aqui vem mais configurações  );
```

Cuidado: O errorHandler deve ser configurado antes de qualquer rota, senão, o Camel nem subirá.

No `errorHandler` , vamos configurar o tratamento seguindo as boas práticas dos padrões de integração usando um `deadLetterChannel` . O nome `deadLetter` vem da mensageria (por exemplo, JMS), onde há uma fila especial para receber mensagens que não podem ser entregues.

O `deadLetterChannel` é um `errorHandler` que possui uma série de configurações para serem personalizadas.

O Camel segue a ideia de *Dead Letter* e chamou isso de `deadLetterChannel` . O `deadLetterChannel` recebe a mensagem venenosa e pode enviá-la para uma fila JMS ou para outro *endpoint*. Por exemplo, guardaremos a mensagem venenosa em uma pasta usando o componente `file` (ou outro endpoint). Por exemplo:

```
errorHandler(
    deadLetterChannel("file:erro"));//mensagem venenosa será gravada na pasta erro
```

Ao executar, notaremos a diferença na hora. As exceções no console sumiram e a pasta `erro` recebe o XML com o pedido que não passou pela validação.

Personalizando o deadLetterChannel

O `deadLetterChannel` tem várias configurações que podem ser alteradas pela Camel DSL. Por exemplo, definiremos que uma mensagem venenosa deve ser entregue novamente. Mas isso faz sentido? A validação será a mesma neste caso. No entanto, existem casos no qual a rede gerou algum problema no IO e, depois, de um tempo volta a funcionar. Desta forma, faz sentido tentar entregar novamente a mensagem pois a entrega pode funcionar.

Usando a Camel DSL, podemos configurar quantas vezes queremos tentar a entrega após a falha e qual será o intervalo de tempo entre as tentativas:

```
errorHandler(
    deadLetterChannel("file:erro").
        maximumRedeliveries(3).//tente 3 vezes
            redeliveryDelay(5000) //espera 5 segundo entre as tentativas
);
```

Definimos três tentativas com um `delay` de 5 segundos. Para acompanhar o trabalho do Camel, podemos *plugar* uma chamada de um método para cada tentativa. Isso é feito usando um `Processor` que será chamado com o método `onRedelivery(...)`:

```
errorHandler(
    deadLetterChannel("file:erro").
        maximumRedeliveries(3).
            redeliveryDelay(5000).
        onRedelivery(new Processor() {
            @Override
            public void process(Exchange exchange) throws Exception {
                //chamado para cada tentativa
                System.out.println("Redelivery");
            }
        })
);
```

Neste processor, imprimiremos quantas tentativas já foram executadas e o máximo de tentativas:

```
//apenas o metodo process
public void process(Exchange exchange) throws Exception {
    int counter = (int) exchange.getIn().getHeader(Exchange.REDELIVERY_COUNTER);
    int max = (int) exchange.getIn().getHeader(Exchange.REDELIVERY_MAX_COUNTER);
    System.out.println("Redelivery - " + counter + "/" + max );
}
```

Caso desejarmos verificar o stacktrace com a exceção, podemos habilitar ele adicionando o método `logExhaustedMessageHistory(true)`. O código completo do `erroHandler`:

```
errorHandler(
    deadLetterChannel("file:erro").
        logExhaustedMessageHistory(true).
        maximumRedeliveries(3).
            redeliveryDelay(5000).
        onRedelivery(new Processor() {
            @Override
            public void process(Exchange exchange) throws Exception {
                int counter = (int) exchange.getIn().getHeader(Exchange.REDELIVERY_COUNTER);
                int max = (int) exchange.getIn().getHeader(Exchange.REDELIVERY_MAX_COUNTER);
                System.out.println("Redelivery - " + counter + "/" + max );
            }
        })
);
```

O que aprendemos?

- XSD serve para descrever XML;
- Como validar XML com XSD na rota;
- Tratamento padrão de exceções pelo Apache Camel;
- Personalizar o tratamento pelo `errorHandler` próprio;

- Usar o `deadLetterChannel` com `redelivery`;
- Usar um processor no método `onRedelivery`.