02

Trabalhando com cabeçalhos

Aluno, você está começando nesse capitulo? Não tem problema, você pode baixar o projeto que usaremos nesse capítulo aqui (https://s3.amazonaws.com/caelum-online-public/soap/stage/estoquews-cap3.zip). **Você só precisar baixar o arquivo se você não fez os exercícios do capítulo anterior.**

Revisão

No último capítulo vimos como personalizar as mensagem SOAP a partir das anotações do JAX-B e JAX-WS. Vimos que os métodos do mundo Java se tornam `operation` s e os parâmetros e retornos são as mensagens no WSDL. Essas `operation` s fazem parte de um elemento chamado `portType` . De certa forma, a classe Java é *traduzida* para XML.

Nova funcionalidade: Cadastrar itens

Vamos aumentar um pouco as possibilidades do nosso serviço e criar uma nova funcionalidade de cadastrar itens no sistema. Chamaremos o método `cadastrarItem` que recebe o `Item` a cadastrar como parâmetro. Na classe `EstoqueWS` adicionaremos:

```
public Item cadastrarItem(Item item) {
  System.out.println("Cadastrando " + item);
  this.dao.cadastrar(item);
  return item;
}
```

Estamos retornando o mesmo item. Parece que não faz sentido, mas normalmente esse item ganha a ID do banco de dados. E ao retorná-lo estamos informando essa ID. Como já fizemos antes, vamos deixar o WSDL mais expressivo usando as anotações `@WebMethod` , `@WebParam` e `@WebResult` :

```
@WebMethod(operationName="CadastrarItem")
@WebResult(name="item")
public Item cadastrarItem(@WebParam(name="item") Item item) {
  System.out.println("Cadastrando " + item);
  this.dao.cadastrar(item);
  return item;
}
```

Já podemos publicar o serviço e testar o resultado:

http://localhost:8080/estoquews?wsdl (http://localhost:8080/estoquews?wsdl)

Repare que no elemento `portType` aparece mais uma `operation` :

```
<portType name="EstoqueWS">
  <operation name="TodosOsItens">
     <input wsam:Action="http://ws.estoque.caelum.com.br/EstoqueWS/TodosOsItensRequest" message="tns
     <output wsam:Action="http://ws.estoque.caelum.com.br/EstoqueWS/TodosOsItensResponse" message="t
  </operation>
```

```
    <operation name="CadastrarItem">
       <input wsam:Action="http://ws.estoque.caelum.com.br/EstoqueWS/CadastrarItemRequest" message="tr
       <output wsam:Action="http://ws.estoque.caelum.com.br/EstoqueWS/CadastrarItemResponse" message="
    </operation>
  </portType>
```

Trabalhando com cabeçalhos

Para cadastrar um novo item no nosso sistema é preciso se autenticar. Podemos pensar que há uma auditoria automática quando alguém altera um dado no sistema e por isso devemos saber quem está solicitando a alteração. Para nosso exemplo não importa muito como a administração de usuários funciona, mas na web é muito comum que um usuário seja identificado através de um *token*. Um *token* é nada mais do que um *hash* gerado para um cliente. No mundo Java Web existe o `JSESSIONID` que representa um *token* utilizado em aplicações Web. O padrão de autenticação e autorização OAuth também usa um *token*. Enfim, o nosso sistema não vai reinventar a roda e também usará um token!

No nosso sistema já temos uma classe preparada para este objetivo, chamada `TokenUsuario` :

```
public class TokenUsuario {

  private String token;
  private Date dataValidade;

  //get e set
```

Queremos receber o token do usuário na requisição SOAP que cadastra um item, mas será que faz sentido misturar os dados do item e o token do usuário? Normalmente não faz e o SOAP já propõe uma forma de separar esses dados. Já vimos que existe para tal o elemento `Header` . Se usarmos ele, a mensagem SOAP deve ficar parecida com a abaixo:

```
<soapenv:Envelope ...>
 <soapenv:Header>
   <tokenUsuario>
       <dataValidade>2015-08-30T00:00:00</dataValidade>
       <token>123131AF!@DF12334a</token>
   </tokenUsuario>
   <soapenv:Header>
   <soapenv:Body>
        <!-- body com o item omitido -->
   </soapenv:Body>
</soapenv:Envelope>
```

Para adicionar um elemento no `Header` , basta criar mais um parâmetro no método `cadastrarItem` e configurá-lo com a anotação `@WebParam` . A anotação possui um atributo `header` que indica que o parâmetro deve ser adicionado ao cabeçalho:

```
@WebMethod(operationName="CadastrarItem")
@WebResult(name="item")
public Item cadastrarItem(@WebParam(name="tokenUsuario", header=true) TokenUsuario token, @WebParam(
  System.out.println("Cadastrando " + item + ", " + token);//imprimindo o token tbm
  this.dao.cadastrar(item);
  return item;
}
```

Ao alterar a classe `EstoqueWS` e rodar o serviço, não há mudanças no `portType` e sim no elemento **binding**. O *binding* define detalhes sobre a codificação dos dados e como se monta a mensagem SOAP. Nessa seção do WSDL está definido que usaremos o protocolo HTTP por baixo dos panos, entre várias outras configurações como `Document` e `literal`. O próximo capítulo veremos mais sobre elas. O que importa agora é o `input` da `operation` `CadastrarItem`, lá está definido que há um `soap:header`:

```
<!-- seção bindings-->
<operation name="CadastrarItem">
   <soap:operation soapAction=""/>
    <input>
       <soap:body use="literal" parts="parameters"/>
       <soap:header message="tns:CadastrarItem" part="tokenUsuario" use="literal"/>
     </input>
     <output>
       <soap:body use="literal"/>
    </output>
</operation>
```

Testando o Header

Vamos atualizar o SoapUI e criar um novo request, nele já deve aparecer o `Header`. Abaixo um exemplo do request já com dados preenchidos:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:ws="http://ws.estc
   <soapenv:Header>
      <ws:tokenUsuario>
         <token>AAA</token>
         <dataValidade>2015-12-31T00:00:00</dataValidade>
      </ws:tokenUsuario>

   </soapenv:Header>
   <soapenv:Body>
      <ws:CadastrarItem>
         <item>
            <codigo>MEA</codigo>
            <nome>MEAN</nome>
            <tipo>Livro</tipo>
            <quantidade>5</quantidade>
         </item>
      </ws:CadastrarItem>
   </soapenv:Body>
</soapenv:Envelope>
```

Ao submeter podemos ver deve aparecer no console do Eclipse o token do usuário.

```
TokenUsuario [token=AAA, dataValidade=Thu Dec 31 00:00:00 BRST 2015]
```

Verificando o token

Um vez que recebemos o token do usuário vamos verificar a validade do token. Em ambientes que são puramente baseados em serviços poderia ter um outro serviço com a responsabilidade de administrar usuários. Ainda não sabemos como chamar um serviço SOAP a partir do código Java, por isso preparamos no nosso sistema uma classe DAO que recebe o token e verifica a existência e validade dele. Vamos pensar que nossa aplicação administra os usuários e os tokens. Portanto, não é preciso chamar um serviço externo, ok?

A classe `TokenDao` possui apenas um método, `ehValido`, que recebe o token do usuário:

```
public class TokenDao {

  public boolean ehValido(TokenUsuario usuario) {
    return //devolve true ou false
  }
```

No método `cadastrarItem` do serviço chamaremos o método `ehValido`:

```
  @WebMethod(operationName="CadastrarItem")
  @WebResult(name="item")
  public Item cadastrarItem(@WebParam(name="tokenUsuario", header=true) TokenUsuario token, @WebPara

    System.out.println("Cadastrando " + item + ", " + token);

    //novo
    boolean valido = new TokenDao().ehValido(token); //o que faremos se o token for invalido?

    this.dao.cadastrar(item);
    return item;
  }
```

A pergunta que não quer calar é: *O que faremos se o token for invalido?* Com certeza, não tem como continuar com a execução. Como falamos antes, devemos saber quem está acessando para cadastrar um item no estoque. Isso parece ser um momento bom para interromper o fluxo comum e jogar uma exceção.

Trabalhando com exceções

Se alguém tenta cadastrar um item sem ter um token válido, vamos lançar um exceção. Chamaremos a exceção de `AutorizacaoException`:

```
boolean valido = new TokenDao().ehValido(token);

if(!valido) {
  throw new AutorizacaoException("Token invalido");
}
```

Como a exceção não existe ainda, é preciso criá-la. O Eclipse ajuda nesse sentido e cria a classe automaticamente estendendo `Exception`:

```
public class AutorizacaoException extends Exception {
```

```
  //esse numero eh relacionado com a serializacao do java.io mas nao importa nesse contexto
  private static final long serialVersionUID = 1L;

  public AutorizacaoException(String msg) {
    super(msg);
  }

}
```

A `AutorizacaoException` é do tipo *checked* e exige um tratamento explicito, por isso o código na classe `EstoqueWS` para de funcionar. Vamos adicionar o tratamento na assinatura do método:

```
@WebService()
public class EstoqueWS {

  private ItemDao dao = new ItemDao();

  @WebMethod(operationName="CadastrarItem")
  @WebResult(name="item")
  public Item cadastrarItem(
    @WebParam(name="tokenUsuario", header=true) TokenUsuario token,
    @WebParam(name="item") Item item) throws AutorizacaoException {

  //código omitido
```

Repare o `throws AutorizacaoException` , deixamos explícito que pode acontecer uma `AutorizacaoException` .

Fault no WSDL

No mundo SOAP não existem exceções e sim **Faults**. Uma exceção no mundo Java é traduzido para um *Fault*. Ao publicar o serviço podemos ver no WSDL que há uma nova mensagem com o nome da exceção:

```
<message name="AutorizacaoException">
  <part name="fault" element="tns:AutorizacaoException"/>
</message>
```

Essa mensagem é utilizada no elemento `portType` do WSDL. Além do `input` e `output` temos um elemento :

```
<portType name="EstoqueWS">

  <!-- operacao TodosOsItens omitida -->

  <operation name="CadastrarItem" parameterOrder="parameters tokenUsuario">
    <input message="tns:CadastrarItem"/>
    <output message="tns:CadastrarItemResponse"/>
    <fault message="tns:AutorizacaoException" name="AutorizacaoException" >
  </operation>
</portType>
```

Estrutura de um Fault

No WSDL um *SoapFault* é também uma mensagem que é associada no `portType` através do elemento `fault`. Vamos testar o serviço e enviar uma mensagem com token errado para causar uma exceção no lado servidor. Vamos colocar no elemento `token` do `Header` um valor inválido, por exemplo:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:ws="http://ws.esto
   <soapenv:Header>
      <ws:tokenUsuario soapenv:mustUnderstand="1">
         <token>errado</token>
         <dataValidade>2015-12-31T00:00:00</dataValidade>
      </ws:tokenUsuario>
   </soapenv:Header>
  <!-- body omitido -->
</soapenv:Envelope>
```

Ao submeter recebemos a resposta com o `Fault`, isto é, dentro do `Body` tem agora um `Fault`:

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <S:Fault xmlns:ns4="http://www.w3.org/2003/05/soap-envelope">
         <faultcode>S:Server</faultcode>
         <faultstring>Token invalido</faultstring>
         <detail>
            <ns2:AutorizacaoException xmlns:ns2="http://ws.estoque.caelum.com.br/">
               <message>Token invalido</message>
            </ns2:AutorizacaoException>
         </detail>
      </S:Fault>
   </S:Body>
</S:Envelope>
```

Um `Fault` possui um `faultcode` que indica se o problema foi do servidor ou do cliente, o `faultstring` com uma mensagem mais amigável e um `detail` que é a instância da exceção serializada em XML.

Personalizando Fault

Como tudo no JAX-WS podemos e devemos personalizar o `Fault` que tem ainda uma cara de exceção Java, que pode ser algo confuso para outras plataformas. A anotação responsável pelo `Fault` se chama `WebFault` e deve ser usado no nível da classe. Vamos chamar o exceção de `AutorizacaoFault`. Além disso, podemos adicionar um método chamado `getFaultInfo` na nossa classe que será usado pelo JAX-B para definir o conteúdo do elemento `detail` do `Fault`:

```
@WebFault(name="AutorizacaoFault")
public class AutorizacaoException extends Exception {

  private static final long serialVersionUID = 1L;

  public AutorizacaoException(String msg) {
    super(msg);
  }

  public String getFaultInfo() {
    return "Token invalido";
```

```
    }
}
```

Deixamos o getter com uma mensagem fixa, mas poderiamos ter algum atributo que define os detalhes do `Fault`. Ao testar o serviço aparece na resposta SOAP o nome `AutorizacaoFault` com a informação `Token invalido`:

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <S:Fault xmlns:ns4="http://www.w3.org/2003/05/soap-envelope">
         <faultcode>S:Server</faultcode>
         <faultstring>Autorizacao falhou</faultstring>
         <detail>
            <ns2:AutorizacaoFault xmlns:ns2="http://ws.estoque.caelum.com.br/">Token invalido</ns2:A
         </detail>
      </S:Fault>
   </S:Body>
</S:Envelope>
```

Ainda está um pouco inseguro com os *Faults* do mundo SOAP? Isso então é uma boa hora de praticar! Nos exercícios vamos consolidar o aprendizado e veremos com mais detalhes os cabeçalhos e Faults. Mãos a obra!

O que você aprendeu nesse capítulo?

- Cabeçalhos servem para guardar informações dados da aplicação
- o elemento Header vem antes do Body
- A anotação @WebParam serve para definir o Header
- Exceptions são mapeadas para Faults