02

Usando SOAP no servidor de aplicação

Nesse capítulo criaremos um novo projeto que importará um WSDL pronto. Você pode baixar este WSDL aqui (https://s3.amazonaws.com/caelum-online-public/soap/EstoqueWSServiceCap5.wsdl).

Revisão

No último capítulo vimos os detalhes do WSDL concreto, falamos muito sobre as seções binding e service. Nessas seções se encontram as regras que definem a codificação, protocolo e endereço do nosso serviço. Vimos que existem os estilos RPC e Document, além da codificação *encoded* e *literal*. Hoje em dia, a grande maioria dos serviços usam Document/literal. Porém para atender o estilo RPC foi criado o Document/literal/Wrapped.

Já criamos o nosso serviço e definimos os métodos mais importantes. Ou seja, o contrato já está pronto. Mas para deixar perfeito falta acessar o banco de dados, fazer o controle de transações e publicar o serviço de maneira robusta. E justamente aqui entra o servidor de aplicação, o papel dele é fornecer estes recursos de infraestrutura para a aplicação.

Já criamos o contrato do serviço, então vamos partir desse contrato para criar a implementação em uma aplicação que roda dentro servidor. Imagine que a equipe de integração definiu o WSDL e a sua tarefa agora é fornecer a implementação. Mãos a obra!

JAX-WS no servidor Wildfly

O servidor de aplicação também já vem com uma implementação JAX-WS. Para utilizar essa implementação vamos criar uma aplicação web. Aqui não há novidade, basta criar um projeto web no Eclipse para rodar no JBoss Wildfly que já temos instalado. Mas quem tiver com dúvida, pode olhar no primeiro exercício onde explicamos como instalar e configurar o JBoss através do Eclipse.

Uma vez o projeto criado podemos começar criar o nosso serviço. Como falamos, vamos usar o contrato como ponto de partida. Para a gente realmente usar o mesmo WSDL, baixar o arquivo: aqui.

https://s3.amazonaws.com/caelum-online-public/soap/EstoqueWSServiceCap5.wsdl (https://s3.amazonaws.com/caelum-online-public/soap/EstoqueWSServiceCap5.wsdl)

CXF e wsdl2java

O primeiro passo é copiar o arquivo para a pasta `src` do projeto. Uma vez copiado podemos pensar na criação das classes. O JAX-WS vai nos ajudar nessa tarefa. Na verdade a implementação do JAX-WS que o JBoss incorpora que se chama CXF (é uma das mais populares do mercado). Ela possui uma ferramenta para criar automaticamente as classes do Servidor ou do cliente a partir do WDSL chamada, não por acaso, de **wsdl2java**. Vamos alimentar usar o WSDL para gerar as classes e subir o serviço. Como criamos o serviço nos primeiros capítulos, poderíamos criar a classe manualmente, mas isso pode ser muito trabalhoso com contratos mais complexos. Então vamos usar essa ferramenta para agilizar nosso trabalho.

Como já falamos essa ferramenta se chama wsdl2java e é do CXF. O JBoss quer ainda mais facilitar e criou um pequeno script para acessar o wsdl2java. Esse script está na pasta `bin` do Wildfly. Veja o comando, que recebe a pasta de saida e o WSDL:

```
sh wsconsume.sh -k -n -o /Users/nico/workspace/estoque-web/src /Users/nico/workspace/estoque-web/src
```

As opções são:

- -k - keep, para manter o código fonte
- -o - para definir a pasta de saida
- -n - não compilar (pois o Eclipse vai compilar)

Ao executar na linha de comando podemos ver que o script realmente usa a ferramenta wsdl2java:

```
Loading FrontEnd jaxws ...
Loading DataBinding jaxb ...
wsdl2java -exsh false -d /Users/nico/workspace/estoque-web/src -verbose -allowElementReferences file
wsdl2java - Apache CXF 2.7.11
```

Vamos dar uma olhada no nosso projeto para ver quais classes foram criadas.

Repare que a maioria das classes são bem familiares. Temos classes do nosso modelo como `Item`, `TipoItem` ou `Filtro` e temos classes pra cada mensagem, por exemplo: `CadastrarItem` e `CadastrarItemResponse`, ou `TodosOsItens` e `TodosOsItensResponse`. Além disso, temos duas classes ligada a nosso serviço: `EstoqueWS` e `EstoqueWSService`.

Vamos abrir a classe `EstoqueWS`:

```
@WebService(targetNamespace = "http://ws.estoque.caelum.com.br/", name = "EstoqueWS")
@XmlSeeAlso({ObjectFactory.class})
public interface EstoqueWS {

    @WebResult(name = "itens", targetNamespace = "")
    @Action(input = "http://ws.estoque.caelum.com.br/EstoqueWS/TodosOsItensRequest", output = "http
    @RequestWrapper(localName = "TodosOsItens", targetNamespace = "http://ws.estoque.caelum.com.br/'
    @WebMethod(operationName = "TodosOsItens")
    @ResponseWrapper(localName = "TodosOsItensResponse", targetNamespace = "http://ws.estoque.caelur
    public br.com.caelum.estoque.ws.ListaItens todosOsItens(
        @WebParam(name = "filtros", targetNamespace = "http://ws.estoque.caelum.com.br/")
        br.com.caelum.estoque.ws.Filtros filtros

    );

    @SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)
    @WebResult(name = "CadastrarItemResponse", targetNamespace = "http://ws.estoque.caelum.com.br/",
    @Action(input = "http://ws.estoque.caelum.com.br/EstoqueWS/CadastrarItemRequest", output = "http
    @WebMethod(operationName = "CadastrarItem")
    public CadastrarItemResponse cadastrarItem(
        @WebParam(partName = "parameters", name = "CadastrarItem", targetNamespace = "http://ws.esto
        CadastrarItem parameters,
        @WebParam(partName = "tokenUsuario", name = "tokenUsuario", targetNamespace = "http://ws.est
        TokenUsuario tokenUsuario
    ) throws AutorizacaoFault;
}
```

É um festival de anotações e configurações, olhando com calma podemos ver que foram já usadas a grande maioria das configurações. O interessante é que foi criado uma interface baseado no elemento `portType` do WSDL e foi gerado esse código. A nossa tarefa agora é criar a implementação dessa interface.

Em um primeiro momento, poderíamos pensar que a implementação já foi gerada pois temos uma classe com o nome `EstoqueWSService`. No entanto, essa classe serve para a criação de um cliente. Usaremos ela no próximo capítulo.

Contract First Design

Repare que estamos fazendo justamente o contrário dos capítulos anteriores. Agora estamos partindo de um WDSL e as classes foram geradas. Essa forma de criar um serviço se chama de **contract first** (ou *implementation last*). Dessa forma valorizamos mais o contrato e a implementação é apenas um detalhe. Quando criarmos primeiro as classes e consequentemente o contrato é gerado, há um certo risco de nosso WSDL ficar com ruídos da implementação sendo pouco expressivo. Ao valorizar o contrato minimizarmos este problema.

Implementação como detalhe

Para realmente publicar o serviço, falta implementar a interface gerada. Vamos criar uma nova classe com o nome `EstoqueWSImpl`. Já no diálogo do Eclipse vamos definir a interface.

Na classe criada devemos utilizar anotação `@WebService` para referenciar a interface:

```
@WebService(endpointInterface="br.com.caelum.estoque.ws.EstoqueWS")
public class EstoqueWSImpl implements EstoqueWS {
```

Nos métodos vamos fazer algo bem simples e devolver um item fixo. Porém, fique a vontade para melhorar o código:

```
@Override
public ListaItens todosOsItens(Filtros filtros) {
    System.out.println("Chamando todos os Itens");
    ListaItens listaItens = new ListaItens();
    listaItens.item = Arrays.asList(geraItem());
    return listaItens;
}

@Override
public CadastrarItemResponse cadastrarItem(CadastrarItem parameters, TokenUsuario tokenUsuario) thro
    System.out.println("Chamando cadastarItem");
    CadastrarItemResponse resposta = new CadastrarItemResponse();
    resposta.setItem(geraItem());
    return resposta;
}

//método auxiliar
private Item geraItem() {
    Item item = new Item();
    item.codigo = "MEA";
    item.nome = "MEAN";
    item.quantidade = 5;
    item.tipo = "Livro";
    return item;
}
```

Publicando o serviço no Wildfly

Tudo pronto para publicar? Garanta que você associou o projeto com o servidor JBoss Wildfly.

Vamos subir o servidor e ficar de olho no console, onde iremos visualizar o endereço do WSDL:

```
address=http://localhost:8080/estoque-web/EstoqueWSImpl
implementor=br.com.caelum.estoque.ws.EstoqueWSImpl
serviceName={http://ws.estoque.caelum.com.br/}EstoqueWSImplService
portName={http://ws.estoque.caelum.com.br/}EstoqueWSImplPort
```

Repare que o nome da nossa implementação faz parte da URL que também aparece no WSDL.

```
<wsdl:service name="EstoqueWSImplService">
  <wsdl:port binding="tns:EstoqueWSImplServiceSoapBinding" name="EstoqueWSImplPort">
     <soap:address location="http://localhost:8080/estoque-web/EstoqueWSImpl"/>
  </wsdl:port>
</wsdl:service>
```

A nossa classe não possui um nome elegante, vamos corrigir isso já através dos atributos `serviceName` e `portName` da anotação `@WebService` :

```
@WebService(endpointInterface="br.com.caelum.estoque.ws.EstoqueWS",
serviceName="EstoqueWS",
portName="EstoqueWSPort")
```

Para melhorar a URL basta na verdade o `EstoqueWS` , mas vamos melhorar o WSDL também e colocar um nome mais bonito no *port*. Mexemos na classe `EstoqueWSImpl` e não na interface, pois o WSDL concreto está sendo definido pela implementação, a parte abstrata vem da interface `EstoqueWS` .

Só falta publicar o serviço para testar o WSDL pelo SoapUI. O novo endereço é:

http://localhost:8080/estoque-web/EstoqueWS?wsdl (http://localhost:8080/estoque-web/EstoqueWS?wsdl)

O que aprendemos nesse capítulo?

- Como gerar as classe do servidor com wsdl2java
- *Contract first* significa criar o WSDL primeiro
- Como publicar um serviço no JBoss Wildfly
- CXF é uma outra implementação popular do JBoss