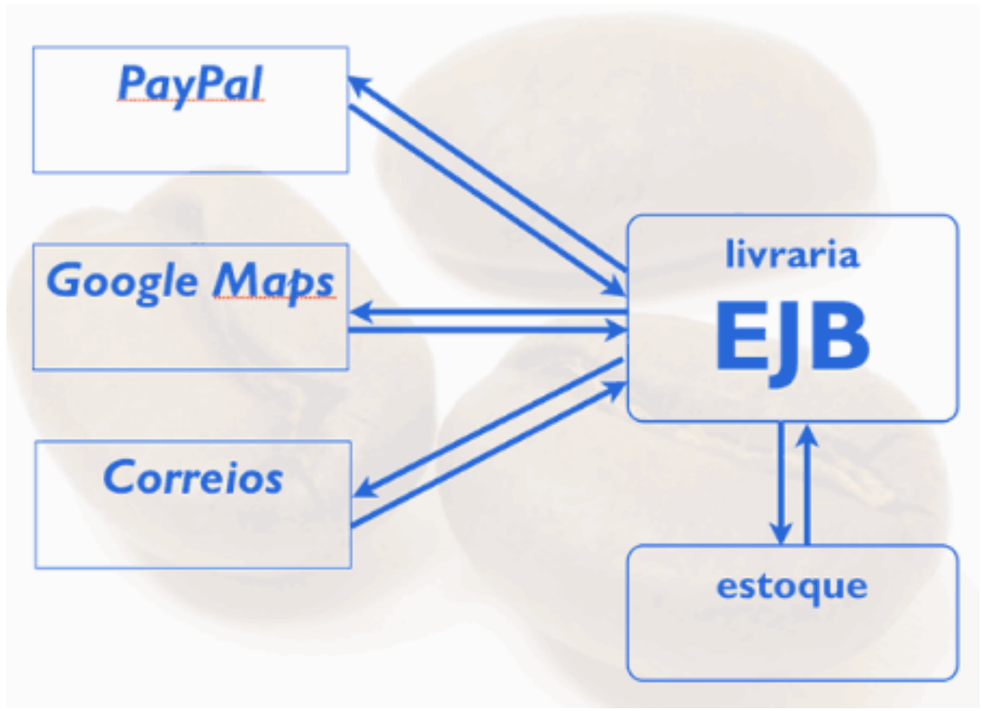
01 **Integração com Web Services**

Transcrição

No último capítulo vimos como é fácil usar novos serviços através de interceptadores. Um interceptador é uma classe que possui apenas um método. Nesse método podemos executar algum código antes e depois de uma chamada concreta de um Session Bean. Isso torna os interceptadores candidatos ideais para tratamento de erro personalizado, monitoramento ou auditoria.

Introdução ao Web Service

A troca de informações faz parte de maioria dos sistemas. É muito raro um sistema funcionando isoladamente. Há sistemas de terceiros, como Google Maps ou PayPal que precisam ser integrados, inclusive há na mesma empresa aplicações separadas que precisam trocar informações.



Os EJBs também atendem esses requisitos e fornecem uma forma de integração com outros sistemas. O mais comum é usar Web Services para essa

tarefa, pois usam o protocolo HTTP e seguem o padrão XML para troca de informação. Veremos nesse capítulo como usar Web Services com EJB baseados nos padrões SOAP e WSDL.

Serviços com SOAP/WSDL

Vamos criar um serviço que devolve os livros cadastrados em nosso sistema. O primeiro passo é criar uma classe com o nome `LivrariaWS` dentro do pacote `br.com.caelum.livraria.webservice`. Ela também será um Session Bean Stateless através da anotação @Stateless:

```
@Stateless
public class LivrariaWS {
}
```

COPIAR CÓDIGO

A implementação do serviços é simples. Vai ter um método apenas que procura livros pelo nome/titulo. O método então devolve uma lista de livros e se chama `getLivrosPeloNome` recebendo uma `String`. No método vamos imprimir o nome do parâmetro para acompanhar o trabalho de serviço:

```
@Stateless
public class LivrariaWS {

    public List<Livro> getLivrosPeloNome(String nome) {
```

```
        System.out.println("LivrariaWS: procurando pelo nome " + nome);

        //aqui usaremos o DAO para executar a pesquisa

        return null;
    }
}
```

COPIAR CÓDIGO

Web Services com JAX-WS

Por enquanto deixaremos o método vazio, mais para frente usaremos o DAO para realmente executar a pesquisa. Contudo o esboço da classe já basta para começarmos com Web Services.

Para realmente publicar essa funcionalidade como serviço falta uma configuração. É preciso anotar a classe com `@WebService` para o container EJB publicar o serviços usando os padrões SOAP e WSDL:

```
@WebService
@Stateless
public class LivrariaWS {
```

COPIAR CÓDIGO

A anotação `@WebService` faz parte de um outro padrão Java, o JAX-WS. O EJB usa o JAX-WS para publicar os Web Services.

Publicando o serviço

Vamos atualizar a aplicação através de um *Full publish*. Ao carregar as classes o container EJB encontrará a anotação `@WebService` e providenciará o serviço.

Podemos ver no console no Eclipse uma saída diferente referente à classe `LivrariaWS`. O JBoss mostra o endereço do serviço no console, algo como:

```
http://localhost:8080/livraria/LivrariaWS
```

COPIAR CÓDIGO

Vamos copiar o endereço e colar no navegador. Ao executar a URI pelo navegador recebemos como resposta um XML. Ela já é uma mensagem SOAP, mas apresentando uma mensagem de erro. Repare que a mensagem possui os elementos `<soap:Fault>` com um `<faultcode>` e `<faultstring>` indicando que algo saiu errado.

O problema é que uma chamada de um Web Service SOAP é por padrão uma requisição do tipo POST. Ao chamar a URI do serviço pelo navegador enviamos uma requisição do tipo GET, algo inválido. Então é preciso enviar um POST com a mensagem SOAP para chamar o nosso serviço.

Mas como saberemos criar uma mensagem SOAP?

Interface do serviço - WSDL

O serviço foi automaticamente publicado com outro serviço que descreve detalhadamente o primeiro. Esse arquivo define as mensagens SOAP, os tipos e os nomes de elementos. É a definição do serviço ou a *interface do serviço*. Podemos ver a interface pela mesma URI adicionando o parâmetro `?wsdl` :

```
http://localhost:8080/livraria/LivrariaWS?wsdl
```

COPIAR CÓDIGO

O WSDL também é um XML e foi gerado baseado na nossa classe `LivrariaWS` . Ou seja, se alterarmos, por exemplo, o nome da classe, o WSDL também mudará. O Importante é que o WSDL foi feito para ser interpretado por outras ferramentas e não apenas seres humanos. Se uma outra aplicação deseja consumir os dados do serviço, uma ferramenta "olhará" o WSDL primeiro.

Vamos utilizar uma vez a ferramenta mais famosa do mercado para se testar um Web Service, o *SoapUI*.

Testando o serviço com SoapUI

Para baixar o SoapUI bastar acessar o link (http://soapui.org (http://soapui.org)). Na página vamos diretamente para os downloads e baixar a versão mais atual, aqui a versão 5.

Há vários downloads para os sistema operacionais específicos. O Importante é escolher o download correto.

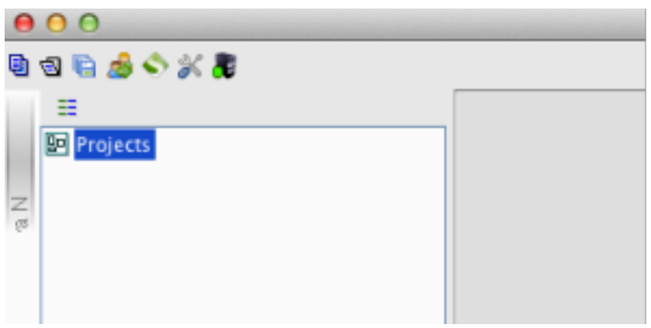
Já baixamos o ZIP do SoapUI, só falta extrai-lo. Dentro da distribuição há uma pasta `bin` com vários scripts que inicializam a ferramenta. O SoapUI na verdade é nada mais nada menos do que uma aplicação Java, os scripts facilitam sua inicialização.

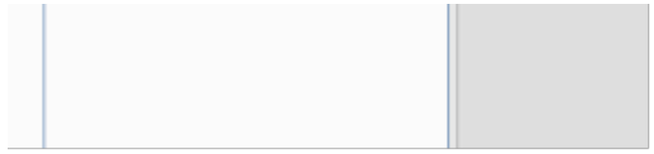
Vamos subir o SoapUI pelo terminal e entrar na pasta da distribuição, mas na subpasta `bin`. Nos sistemas UNIX basta rodar o arquivo `soapui.sh`, no mundo windows é o arquivo `soapui.bat`.

```
cd SoapUI-5.0.0/bin/
sh soapui.sh
```

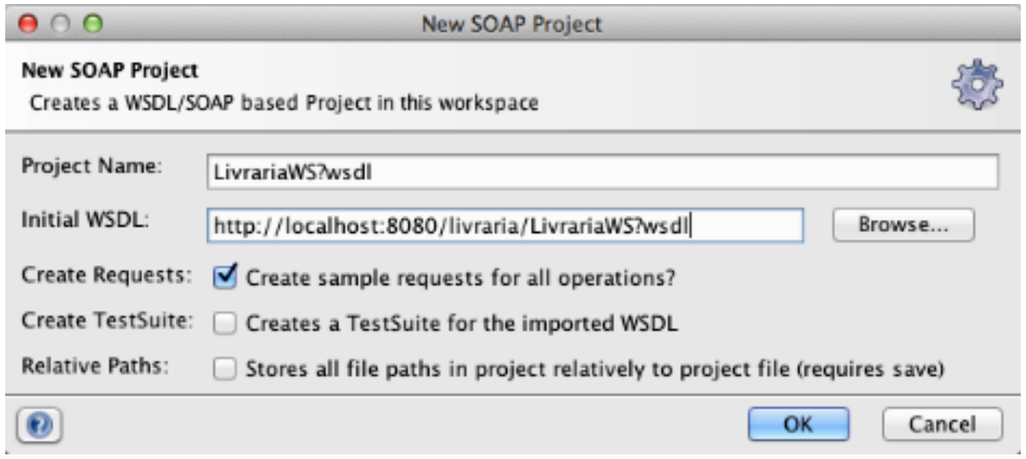
COPIAR CÓDIGO

Com o SoapUI aberto, vemos no lado esquerdo o gerenciador de projetos. É nele que criaremos um novo *SOAP Project*.





Na janela vamos colar a URI do WSDL do nosso serviço, o nome do projeto é automaticamente preenchido e podemos confirmar o diálogo. Ao confirmar, o SoapUI analisa o WSDL e cria automaticamente o cliente SOAP para testar o nosso serviço.



No lado esquerdo foi criado um projeto e um item com o nome do método `getLivrosPeloNome`. Abaixo desse item encontramos um elemento `request`. Ao abrir o `request` podemos ver a mensagem SOAP gerado pelo SoapUI. Como já falamos o SOAP também é um XML que é nada mais do um envelope com pelo menos um elemento `body`. Dentro do `body` ficam os dados de envio. Há mais detalhes sobre o SOAP, porém a mensagem com seus detalhes são geradas por ferramentas como o SoapUI:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:web="http://webservice.livraria.caelum.com.br/">
   <soapenv:Header/>
   <soapenv:Body>
      <web:getLivrosPeloNome>
         <!--Optional:-->
         <arq0>?</arq0>
      </web:getLivrosPeloNome>
   </soapenv:Body>
</soapenv:Envelope>
```

COPIAR CÓDIGO

No lugar de ? devemos colocar o nome do livro. Este elemento `<arg0>` representa o parâmetro do nosso método. Vamos colocar o nome *arquitetura* e enviar uma requisição para o nosso serviço. Basta clicar no botão verde na janela do `request`. Como resposta recebemos também um XML SOAP vazio:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body>
      <ns2:getLivrosPeloNomeResponse xmlns:ns2="http://webservice.livraria.caelum.com.br/"/>
   </soap:Body>
</soap:Envelope>
```

COPIAR CÓDIGO

Isso faz sentido pois o nosso serviço realmente não devolve nada. Agora implementaremos a pesquisa antes de continuar com o SOAP.

Implementação do Serviço

No Eclipse, na classe `LivrariaWS` usaremos o `LivroDao` para delegar a pesquisa. Como já fizemos antes, injetamos o `LivroDao` na classe para poder utilizá-la. No DAO faremos realmente a pesquisa, chamaremos o método `livrosPeloNome(..)`. Como o método não existe ainda, o código não compila. Geraremos o método pelo Eclipse mesmo.

O Eclipse já abre a classe `LivroDao` e gera o esboço do método. Nele faremos a pesquisa usando JPQL. Com o `EntityManager` em mãos basta executar uma *Query*. O JPQL é simples e fará um *select* na entidade `Livro` filtrando pelo `titulo` na clausula *where*. Repare que a query é parametrizada.

O retorno do método `createQuery(..)` é a query em si. Através dela passamos o valor do parâmetro `pTitulo` definido no JPQL. Por fim executamos a query chamando o método `getResultList()` para retornar a lista de livros:

```
@Stateless
public class LivroDao {

    @PersistenceContext
    private EntityManager manager;

    //outros métodos omitidos
```

```
    public List<Livro> livrosPeloNome(String nome) {

        TypedQuery<Livro> query = this.manager.createQuery(
                "select l from Livro l " +
                "where l.titulo like :pTitulo", Livro.class);
        query.setParameter("pTitulo", "%" + nome + "%");

        return query.getResultList();
    }

}
```

COPIAR CÓDIGO

E a nossa classe `LivrariaWS`:

```
@WebService
@Stateless
public class LivrariaWS {

    @Inject
    LivroDao dao;

    @WebResult(name="autores")
    public List<Livro> getLivrosPeloNome(@WebParam(name="titulo") String nome) {

        System.out.println("LivrariaWS: procurando pelo nome " + nome);
        return dao.livrosPeloNome(nome);
    }

}
```

COPIAR CÓDIGO

Se tudo estiver compilando podemos atualizar a aplicação.

Depois do deploy voltaremos para o SoapUI. Ainda temos a mensagem SOAP disponível, vamos reenviar a requisição. Agora sim, recebemos uma

resposta com o conteúdo. Dentro do elemento `body` da mensagem SOAP aparecem os dados sobre o livro e autor.

Personalizando a mensagem SOAP

Analisando as mensagens SOAP mais detalhadamente fica claro que elas não são muito expressivas. Por exemplo, na mensagem de ida qual é o significado do `<arg0>` ? Como desenvolvemos o serviço sabemos que ele representa o titulo/nome do livro. Mas para quem não conhece a implementação? Igualmente não está muito claro o significado do elemento `<retorno>` na resposta.

Vamos melhorar a expressividade da mensagem SOAP alterando o nosso serviço. No Eclipse, na classe `LivrariaWS` usaremos anotações para melhorar a mensagem. A primeira anotação é relacionada com o parâmetro do método. Ao usar `@WebParam` fica claro para o container EJB que queremos usar aquele nome na mensagem SOAP. Outro ponto é o retorno. Para dar um nome ao elemento que representa o retorno usa-se a anotação `@WebResult` :

Segue uma vez a implementação completa:

```
@WebService
@Stateless
public class LivrariaWS {

    @Inject
    LivroDao dao;

    @WebResult(name="livros")
    public List<Livro> getLivrosPeloNome(@WebParam(name="titulo") String nome) {

        System.out.println("LivrariaWS: procurando pelo nome " + nome);
        return dao.livrosPeloNome(nome);
    }
}
```

COPIAR CÓDIGO

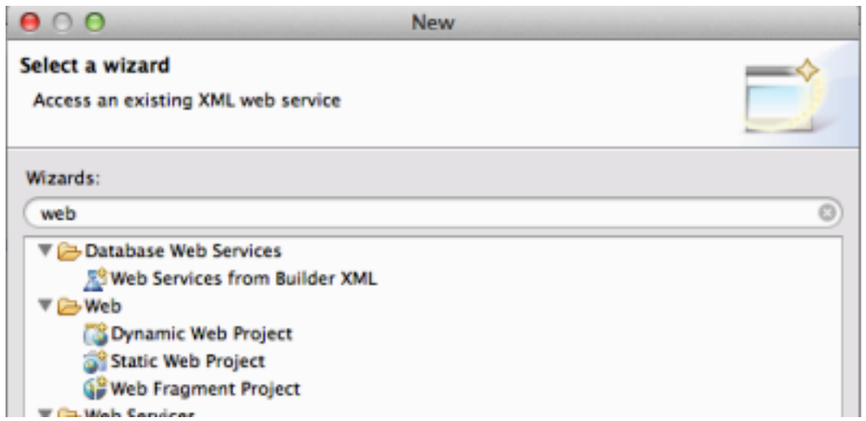
Voltando ao SoapUI não podemos esquecer de trocar o `arg0` por `titulo` . Uma vez feito, vamos enviar a requisição de novo. Na resposta aparece também o elemento `autores` .

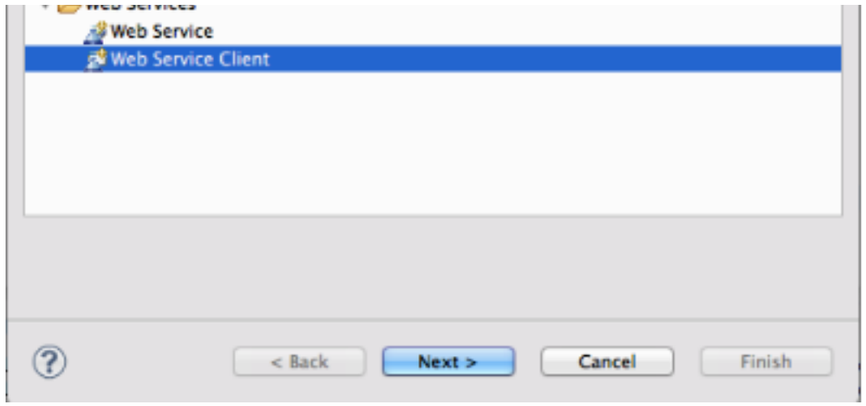
Cliente Java

Conseguimos testar o nosso serviço web com a ajuda do SoapUI. A ferramenta leu o WSDL e criou todo o cliente, incluindo a mensagem SOAP. Agora queremos usar o serviço através de uma aplicação Java.

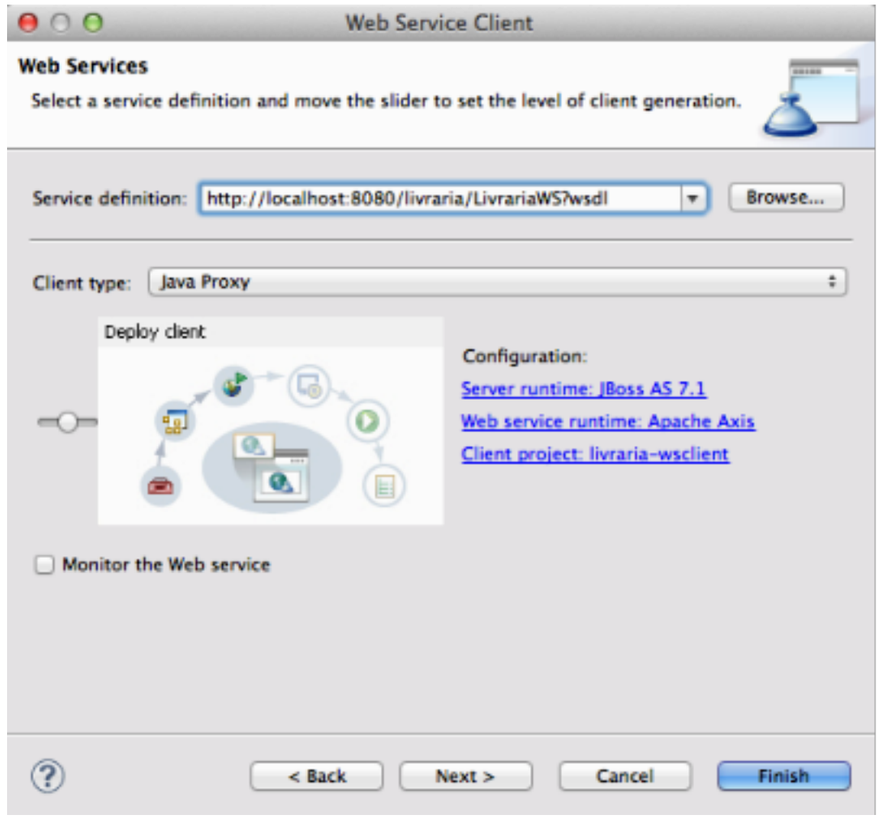
A instalação do JRE já vem com todas as classes necessárias para consumir um web service (especificação JAX-WS que trata SOAP/WSDL). No caso da JRE da Oracle já vem uma ferramenta, o `wsimport`, que consegue gerar classes que acessam o serviço de uma maneira transparente. No curso usaremos o Eclipse para criar um novo projeto e gerar automaticamente as classes do cliente para chamar o serviço.

Criaremos um novo projeto chamando-o de *livraria-wsclient*. Uma vez criado e com o projeto selecionado no *Project Explorer* entramos no mesmo menu para criar um novo *Web Service Client*.





O Eclipse mostra um novo dialogo que permite definir o localização do WSDL. Lembrando o WSDL tem todas as informações de que o cliente precisa. Copiaremos a URI do WSDL pelo navegador e colamos no elemento *Service Definition*. Pronto, já podemos finalizar este passo.



O Eclipse não só adicionou todas as bibliotecas no projeto como também gerou as classes Java para fazer uma chamada ao web service. Repare que entre as classes temos um `Livro e Autor`, o modelo do serviços, e temos outras classes bem parecidas com as do serviço.

Vamos usar essas classes para chamar o web service. Para tal criaremos uma nova classe `TesteRequestSoapComJava`, também já com método `main`

gerado. No método `main` vamos usar a interface `LivrariaWS` que foi gerado pelo Eclipse. Através dessa interface vamos referenciar um objeto que sabe fazer a chamada remota e gerar SOAP. Esses tipos de objetos são chamados de *Proxies*, no nosso caso `LivrariaWSProxy`:

```
LivrariaWS cliente = new LivrariaWSProxy();
```

COPIAR CÓDIGO

Uma vez criado o proxy podemos executar a chamada remota usando o método `getLivrosPeloNome(..)`. Repare que parece que fazemos uma chamada local, mas na verdade será feita uma requisição HTTP que enviará o SOAP. O resultado dessa chamada é um array de livros:

```
Livro[] livros = cliente.getLivrosPeloNome("Arquitetura"); //chamada HTTP com SOAP
```

COPIAR CÓDIGO

Por fim, faremos um laço para imprimir os dados do livro: o título e o nome do autor:

```
for (Livro livro : livros) {
    System.out.println(livro.getTitulo());
    System.out.println(livro.getAutor().getNome());
}
```

COPIAR CÓDIGO

Segue uma vez o código completo:

```
public class TesteRequestSoapComJava {

    public static void main(String[] args) throws RemoteException {


        LivrariaWS cliente = new LivrariaWSProxy();

        Livro[] livros = cliente.getLivrosPeloNome("Arquitetura");

        for (Livro livro : livros) {
            System.out.println(livro.getTitulo());
```

```
            System.out.println(livro.getAutor().getNome());
        }
    }

}
```

COPIAR CÓDIGO

Ao executar, podemos ver que aparecem os dados no console do Eclipse. Repare que esse código encapsula todos os detalhes sobre SOAP/WSDL. De alto nível chamamos um web service que é uma das grandes vantagens de Web Services SOAP.

Para saber mais

Para gerar as classes (também chamados *stubs*) com a ferramenta `wsimport` executamos na linha de comando:

```
wsimport -s src  -p br.com.caelum.livraria.clientews
        http://localhost:8080/livraria/LivrariaWS?wsdl
```

COPIAR CÓDIGO

Repare os parâmetros que passamos:

- `-s` - diretório dos arquivos .java gerados
- `-d` - diretório dos arquivos .class gerados
- `-p` - pacote das classes geradas

O comando deve ser executado a partir da raiz do projeto.

[0] http://www.soapui.org (http://www.soapui.org)