Desenvolvimento de Web Services RESTful e SOAP-WSDL utilizando as Implementações de Referência JAX-RS e JAX-WS

Ricardo Ramos de Oliveira

Curso das Tecnologias de Web Services

Ricardo Ramos de Oliveira

Orientadora: *Prof^a. Dr^a. Renata Pontin de Mattos Fortes*

Curso de Web Services do Instituto de Ciências Matemáticas e de Computação — ICMC/USP.

USP - São Carlos
Novembro/2011

Resumo

A tecnologia de web services é uma das tecnologias mais promissoras em termos de disponibilização de serviços na rede, além de solucionar o problema de integração de aplicações heterogêneas na web. Devido à crescente popularidade dos web services, eles fornecem uma base importante para novas oportunidades de negócios. São aplicações que satisfazem as necessidades de uma ampla variedade de clientes e negócios. Exemplos de sua utilização podem ser encontrados em aplicações B2B e B2C, *e-learning*, e assim por diante. Porém os web services são altamente vulneráveis e sujeitos a constantes mudanças. Assim, eles oferecem um novo desafio para a Engenharia de Software.

Sumário

Lista de Figuras

Capítulo

1

Introdução aos Web Services

1.1 Histórico Geral dos Web Services

Desde a introdução dos microcomputadores, através do lançamento do "revolucionário" Personal Computer (PC) em 1981 pela IBM, os programas eram executados apenas em uma máquina local. Porém, com o surgimento da internet, houve a necessidade de se conectar diversos computadores em redes; nesse momento surgiu a arquitetura conhecida como cliente-servidor. Nessa arquitetura de rede, existem duas entidades básicas na rede: o servidor e os clientes. Na computação, um servidor é um sistema de computação que fornece serviços a uma rede de computadores. Esses serviços podem ser de natureza diversa, como por exemplo, arquivos e correio eletrônico. Os computadores que acessam os serviços de um servidor são chamados clientes (Kurose, 2003).

Com o aumento do poder de processamento dos microcomputadores, os fabricantes de programas começaram a desenvolver redes cada vez mais poderosas, sistemas operacionais mais rápidos e flexíveis, redes locais (LANS - *Local Area Networks*) e redes amplas (WANs - *Wide Area Networks*). Esta arquitetura mostrou-se mais flexível devido à utilização dos computadores em rede. Sendo assim, as aplicações necessitavam se comunicar entre si de forma dinâmica através dos web services, com a distribuição de recursos através da internet. Podemos perceber que, ao contrário das aplicações web convencionais da arquitetura cliente-servidor, que foram projetadas para dar suporte entre as interações

aplicação-usuário (B2C), a tecnologia dos web services foi desenvolvida para realizar interações aplicação-aplicação (B2B), embora também possa ser utilizada para interações com o usuário, pois operações com web services podem ser realizadas através do navegador. Todavia, de acordo com a W3C, seu foco é a interação aplicação-aplicação.

Atualmente no cenário mundial a disponibilidade de recursos de serviços é diversificada e flexível, onde o mundo da internet é composto por serviços cooperativos, a integração de ferramentas é uma área promissora da Engenharia de Software.

1.2 Conceitos e Definições sobre Web Services

Antes de definir um web service é importante destacar que um web service é composto por duas estruturas: o **serviço** (W3C, 2004) e a **descrição do serviço** (W3C, 2007).

O **serviço** consiste num módulo de software instalado numa plataforma computacional com acesso à rede e oferecido pelo "provedor de serviços". Um serviço existe para ser usado por um consumidor, podendo funcionar também como um cliente de outro serviço.

A **descrição do serviço** (DS) contém os detalhes da interface e da implementação de um serviço, o que inclui os tipos de dados, operações, informação de ligação (*binding*), e localização de rede. Pode ainda incluir metadados e informação de categorização para facilitar as atividades de descoberta e utilização por consumidores do serviço. A DS pode ser publicada num registrador de serviço para tornar o respectivo serviço conhecido em um determinado contexto.

A definição mais aceita pela comunidade da Tecnologia da Informação (TI) é da World Wide Web Consortium (W3C) onde os web services são um sistema de software projetado para apoiar interações máquina-para-máquina interoperáveis pela rede, fornecendo uma interface descrita em um formato processável por máquina (especificamente a WSDL). Outros sistemas interagem com o web service de maneira prescrita por sua descrição usando mensagens SOAP, normalmente transmitidas através de HTTP com uma serialização XML em conjunto com outros padrões da web (W3C, 2004). Todos esses protocolos engoblando o UDDI são responsáveis por estabelecer uma simples conexão entre os web services. Cada protocolo é descrito a seguir:

- XML[1] (eXtensible Markup Language) é uma linguagem genérica e padronizada de marcação, isto é, uma linguagem capaz de descrever uma organização lógica, estruturada dos dados através de *tags* definidas. O XML além de ser popular e amplamente aceito é a base principal para o desenvolvimento dos web services (BASCI, 2009). Exemplo de um Livro em XML:

[1] http://www.w3.org/TR/2008/REC-xml-20081126/

```
<livro>
  <autor>Orwell</autor>
  <titulo>1984</titulo>
  <id>42</id>
</livro>
```

As *tags* são os elementos entre os símbolos < e >, são defenidas pelo próprio programador, além disso, não existe nenhum mecanismo em XML, para que 1984 do exemplo, seja um número em vez de uma palavra com quatro caracteres. As linguagens de XML *Schema* adicionam esta funcionalidade, ou seja, os tipos de dados no documento XML(Campbell et al., 2003).

- JSON[2] (JavaScript Object Notation) é um conjunto de chaves e valores, que podem ser interpretada por qualquer linguagem. Além de ser um formato de troca de dados leve[3], é fácil de ser entendido e escrito pelos programadores. Estas propriedades fazem do JSON uma linguagem ideal para o intercâmbio de dados como XML (Jun et al., 2008). Exemplo de um Livro em JSON:

```
{
    "autor": "George Orwell",
    "titulo": "1984",
    "id": 42
}
```

- SOAP[4] (Simple Object Access Protocol): é um protocolo padrão de troca de mensagens estruturado em XML que possibilita à comunicação entre serviços. Ele define os componentes essenciais e opcionais das mensagens transmitidas entre os serviços através do protocolo HTTP. Exemplo de um envelope SOAP:

```
<soapenv:Envelope>
   <soapenv:Header/>
   <soapenv:Body>
      <calc:soma>
         <calc:num1>5</calc:num1>
         <calc:num2>3</calc:num2>
```

[2]http://www.json.orgljson-en.html

[3]Em Tecnologia da Informação (TI), a expressão leve é às vezes aplicada a um programa, protocolo, ou qualquer outro dispositivo com um menor número de componentes e que seja relativamente mais simples ou mais rápido.

[4]http://www.w3c.org/TR/soap12-part1/

```
      </calc:soma>
   </soapenv:Body>
</soapenv:Envelope>
```

- WSDL[5] (Web Services Description Language): é um modelo e um formato XML para descrever web services, possibilitando a separação entre a funcionalidade oferecida de sua descrição abstrata. A descrição do web service define a sua interface, ou seja, o conjunto de operações possíveis entre o provedor e cliente do serviço, bem como as mensagens que serão trocadas entre eles. Exemplo de um arquivo WSDL:

```
<xs:element name="soma">
    <xs:complexType>
        <xs:sequence>
            <xs:element minOccurs="0" name="num1" type="xs:int"/>
            <xs:element minOccurs="0" name="num2" type="xs:int"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="somaResponse">
    <xs:complexType>
        <xs:sequence>
            <xs:element minOccurs="0" name="return" type="xs:int"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

- UDDI[6](Universal Description, Discovery, and Integration): oferece um mecanismo para que os clientes possam encontrar um determinado web service. O UDDI pode ser considerado o DNS para os web services, contendo informações sobre o provedor do serviço, sua localização, além das descrições do serviço (WSDL) (Wang et al., 2004). Os repositórios de serviços podem ser públicos ou privados, sendo que tanto o registro quanto a consulta de serviço são realizadas por meio de arquivos XML. Exemplo do UDDI:

```
<import namespace="http://www.getquote.com/StockQuoteService-interface"
   location="http://www.getquote.com/wsdl/SQS-interface.wsdl"/>
```

[5]http://www.w3c.org/TR/2003/WD-wsdl12-20030611/
[6]http://uddi.org/pubs/uddi-v3.00-published-20070719.htm

```
<service name="StockQuoteService">
  <documentation>Stock Quote Service</documentation>

<port name="SingleSymbolServicePort"
   binding="interface:SingleSymbolBinding">
   <documentation>Single Symbol Stock Quote Service</documentation>
    <soap:address location="http://www.getquote.com/stockquoteservice"/>
</port>
```

Os padrões descritos nesta seção fazem parte da primeira geração de web services. A segunda geração também conhecida como "WS-*"representa um complemento fornecendo novos recursos e funcionalidades para suprir as limitações da geração anterior. Dentre estas especificações, estão incluídas WS-Security, WS-ReliableMessaging, WS-Transaction entre outras (Erl, 2004).

A pilha de protocolos associados aos web services é formada por protocolos de rede responsáveis pela definição, localização, implementação e troca de mensagens. Podemos dividi-la também em quatro camadas: busca, descrição, comunicação e transporte conforme mostra a Figura 1.1. No topo da pilha encontra-se a descoberta dos serviços, com o protocolo UDDI. Na segunda camada está situada a descrição dos serviços com o protocolo WSDL. A terceira camada refere-se à troca de mensagens XML, com os protocolos XML-RPC, SOAP e XML. Na última camada está situado o transporte, com os protocolos HTTP, SMTP, FTP e BEEP.

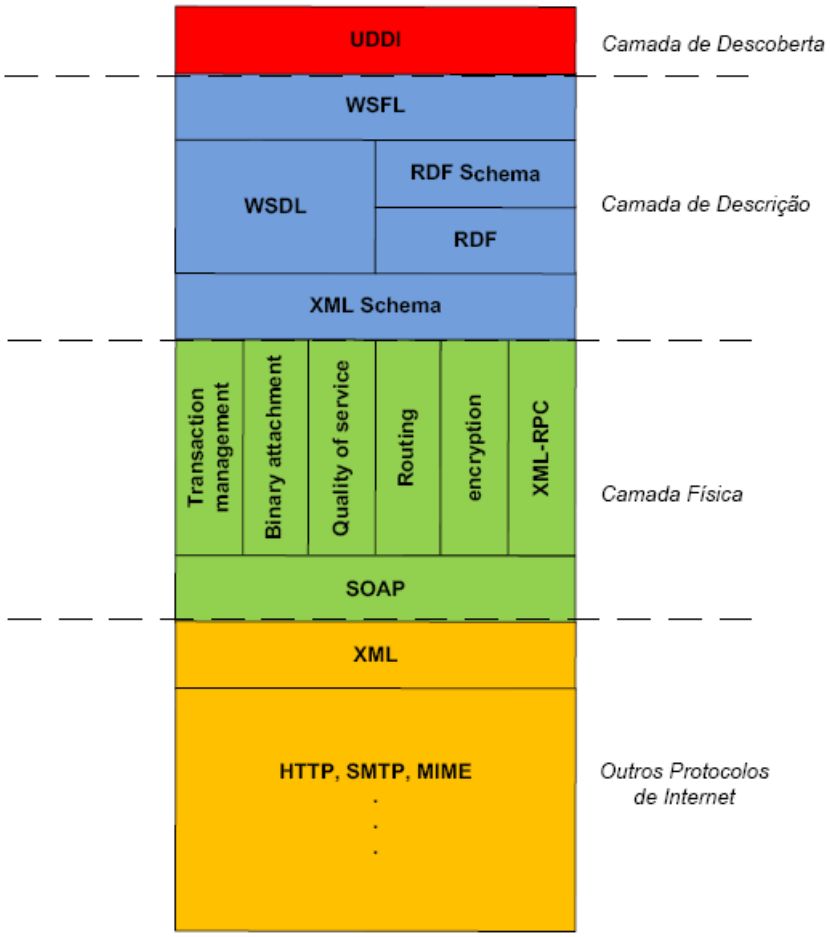


Figura 1.1: Pilha conceitual de web services (Brooks, 2002)

Para este curso, a pilha conceitual proposta pela IBM foi adotada por apresentar os conceitos de forma genérica e completa, englobando os padrões básicos e as principais especificações emergentes (IBM, 2009).

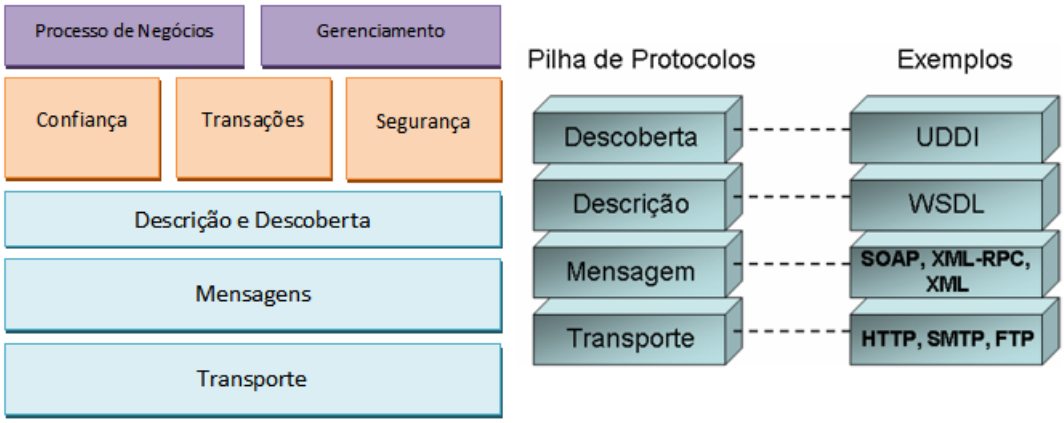


Figura 1.2: Pilha conceitual de web services (IBM, 2009)

1.3 Caracterísitcas dos Web Services

Uma vez que um navegador tem apresentado o seu pedido HTTP, que tem uma tarefa bastante fácil. É preciso tornar a resposta de forma que um ser humano pode entender. Ele não precisa descobrir o que a resposta HTTP significa: que é o trabalho do *browser* ou navegador. Um cliente do serviço web não tem esse luxo. É programado com antecedência, por isso tem de ser tanto o navegador da web que obtém os dados, e os "humanos"que decide o significado dos dados. Clientes do serviço web deve automaticamente extrair o significado das respostas HTTP e tomar decisões com base nesse sentido (Richardson e Ruby, 2007).

Mas o objetivo principal deste capítulo é para você pensar sobre a World Wide Web como um meio de ligação entre um e outro programa de computador, nas mesmas condições em que conecta os seres humanos a cada outros seres humanos. Sendo assim, podemos descrever as características dos web services como:

- é um sistema que utiliza o protocolo HTTP, do mesmo jeito que uma aplicação web, com *requests* e *responses*;
- tipicamente utiliza XML ou outros formatos de arquivos, como JSON, para transferência de dados;
- geralmente utilizado para integrar sistemas diferentes, ou disponibilizar uma série de serviços de uma aplicação, como o Twitter[7] ou Google[8].

A arquitetura cliente e servidor baseia-se em duas entidades: clientes e os servidores de serviços. Como ilustrado na Figura 1.3.

[7]Buscando os últimos tweets da @oliviamunn: http://api.twitter.com/1/statuses/-user_timeline.xml?screen_name=oliviamunn

[8]http://soapclient.com/xml/googleSearch.wsdl

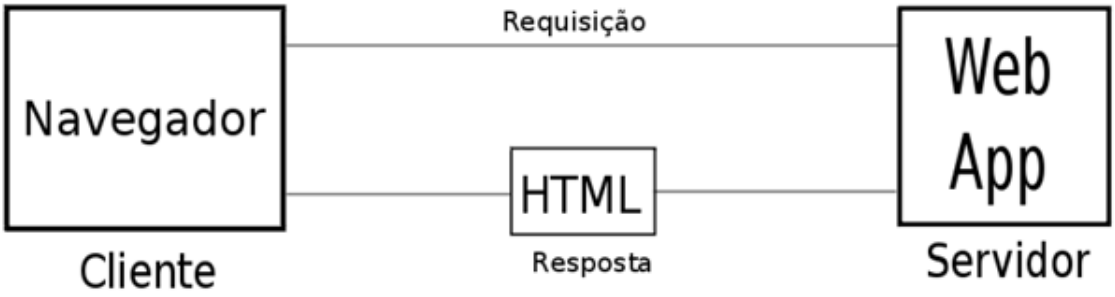


Figura 1.3: Arquitetura Cliente Servidor

Já arquitetura baseada em serviços possui duas entidades: consumidores e provedores de serviços. Um provedor possui características semelhantes a um servidor que disponibiliza serviços na rede, enquanto consumidores são considerados clientes que utilizam os serviços disponibilizados por provedores de serviços. Uma entidade também pode assumir ambos os papéis ao mesmo tempo, caracterizando a composição de serviços. Uma interação básica entre tais entidades é ilustrada na Figura 1.4. Nessa figura um programa do consumidor de serviço envia, pela rede, uma requisição ao provedor de serviço, o qual retorna, pela rede, a resposta aquela requisição para o devido consumidor de serviço.

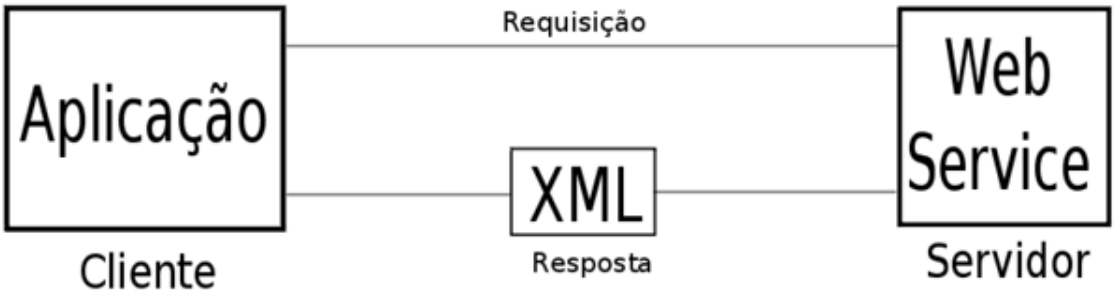


Figura 1.4: Arquitetura de Web Services

No entanto, é possível desenvolver aplicações que utilizem os dois tipos de arquitetura, a Figura 1.5 mostra uma aplicação que utiliza a arquitetura cliente-servidor na comunicação do *browser* com um sistema empresarial no *front-end* e ao mesmo tempo utiliza a arquitetura de web services no seu *back-end* para validar o número de cartões de crédito através da comunicação estabelecida pelo sistema empresarial com o serviço web de validação de cartões de crédito, por meio de arquivos XML.

Os web services constituem uma tecnologia emergente da Arquitetura Orientada a Serviços (SOA). Sua grande popularidade ocorreu devido à adoção de protocolos e padrões abertos, tais como HTTP e XML, visando solucionar o problema de integrar aplicativos de sistemas heterogêneos presentes em tecnologias como CORBA, DCOM e RMI (Stal, 2002). O principal objetivo dos web services é oferecer a interoperabilidade entre os sistemas escritos em diferentes linguagens de programação, desenvolvidos por fornecedores distintos e em sistemas operacionais diversos possam se comunicar (Meng et al., 2009).

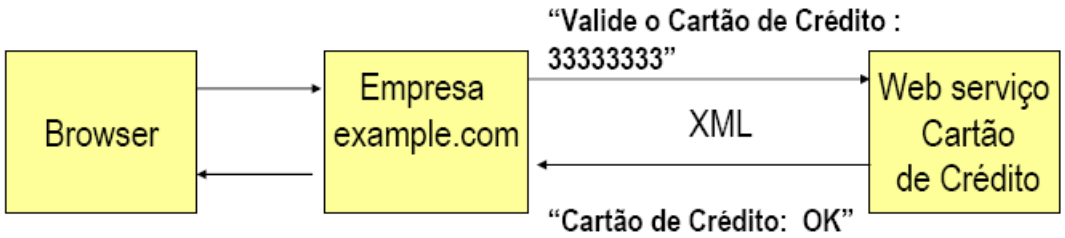


Figura 1.5: Validação de Cartão de Crédito

Por meio de uma interface XML, especificada em WSDL (*Web Service Description Language*), a comunicação entre web services pode ser feita. A WSDL é apenas um dos diversos padrões da W3C[9] que, depois de implementados, têm permitido um avanço na comunicação entre aplicações distribuídas. Entre esses padrões destacam-se o WS-security, WS-Addressing e WS-Reliable [10] considerados fundamentais para o desenvolvimento de aplicações distribuídas utilizando web services. A construção de aplicações distribuídas em um ambiente dinâmico como a internet é muito complexo principalmente quando o foco de um aplicativo deve considerar a manutenibilidade entre a interação das entidades envolvidas em uma Arquitetura Orientada a Serviços (cliente, provedor, *broker*, repositório de informações sobre serviços).

[9]World Wide Web Consortium (W3C) http://www.w3.org/
[10]http://www.oasis-open.org/specs/

1.4 A importância dos Web Services

Atualmente, as tecnologias de rede baseada na web têm se tornado cada vez mais importantes para soluções de Tecnologia da Informação (TI). Além disso o governo federal vem utilizando web services como ferramenta de apoio[11]. Dentre os web services do governo federal podemos destacar [12]:

- **Web Service do Sistema de Informações Organizacionais do Governo Federal (SIORG):** o serviço WebServiceSiorg foi desenvolvido com um propósito específico para atendimento a determinada necessidade pontual. Portanto, as consultas disponibilizadas pelo mesmo deverão ser avaliadas e pensadas em um contexto mais geral, mais integrador para que as informações geradas pelo sistema sejam melhor utilizadas. O seu objetivo principal é disponibilizar as informações organizacionais do Governo Federal utilizando tecnologias web services como SOAP e WSDL[13].

- **Consulta ao Sistema Integrado de Administração de Recursos Humanos (SIAPE):** o objetivo do web service SIAPE é possibilitar o acesso, em tempo real, por parte dos Sistemas Autorizados, aos dados dos servidores ativos, aposentados e pensionistas, que recebem seus pagamentos pelo SIAPE, utilizando um serviço disponível na Internet. O arquivo de descrição WSDL do projeto pode ser acessado no endereço: https://www1.siapenet.gov.br/ WSSiapenet/services/ConsultaSIAPE?wsdl.

- **Exportação do Sistema de Informações Gerenciais e Planejamento (SIGPlan):** o objetivo principal desse web service é disponibilizar operações de consulta a diversos dados de programas, de ações orçamentárias e de ações não orçamentárias contemplados no Módulo de Monitoramento do SIGPlan. O arquivo de descrição WSDL do projeto pode ser acessado no endereço: https://www1.siapenet.gov.br/ WSSiapenet/services/ConsultaSIAPE?wsdl.

- **Infrasig Sistema de Informações Gerenciais e Planejamento (SIGPLAN):** o objetivo principal é realizar a integração dos sistemas de orçamento de diversos

[11] http://www.governoeletronico.gov.br/acoes-e-projetos/e-ping-padroes-de-interoperabilidade/catalogo-de-interoperabilidade

[12] http://catalogo.governoeletronico.gov.br/pasta_servico/folder_summary_list_view

[13] Ambiente de Homologação:http://hom.siorg.redegoverno.gov.br/gestao/webservice/WSSiorg.asmx e Ambiente de Produção:http://www.siorg.redegoverno.gov.br/gestao/webservice/WSSiorg.asmx

órgãos da União com o SIGPLAN. As operações desse web service são para gravação de dados no SIGPLAN[14].

- **Consulta a Títulos do Tesouro:** O web service do Tesouro foi implementado com o intuito de atualizar a base de dados da Consulta a Títulos Públicos do Tesouro, mantida no portal da STN – Secretaria do Tesouro Nacional – com informações para a compra de títulos públicos ofertados no sistema Tesouro Direto, hospedado na CBLC – Companhia Brasileira de Liquidação e Custódia[15].

Além disso empresas como a Amazon e o Google permitem o acesso ao seu *back-end* do sistema de TI por meio de interfaces de web services. A IBM tem muitos clientes que começaram a usar os seus web services para reduzir o custo de negócios e resolver os problemas de integração. Esta tendência leva a sistemas de informações totalmente conectados, mas também provoca uma série de problemas que os desenvolvedores devem enfrentar (Wei-Chung Hu e Jiau, 2010). Um desses problemas está relacionado com a manutenibilidade dos web services, como exemplo mais atual podemos citar a manutenção do sistema Portal da Nota Fiscal Eletrônica (NF-e) da Secretaria da Fazenda do estado de São Paulo no dia 13/03/2011[16].

Os web services são altamente distribuídos e constituídos por meio de diferentes organizações. Além disso, seus componentes evoluem rapidamente. Sendo assim, os sistemas web services impõem uma complexidade adicional em processos de manutenção de software.

1.5 Estrutura do Curso

Este trabalho está organizado em quatro capítulos. Este capítulo introduziu os principais conceitos relacionados com os web services. O Capítulo 2 apresenta a tecnologia dos web services SOAP-WSDL. No Capítulo 3 será apresentado a tecnologia dos web services RESTful. Cada capítulo apronfuda no desenvolvimento das abordagens dos web services. Sendo assim, em cada capítulo será mostrado como desenvolver os web services cliente e provedores de serviço. Por fim, são listadas as referências bibliográficas utilizadas ao longo deste curso.

[14]Ambiente de Homologação:http://homsigplan.serpro.gov.br/infrasig/infrasig.asmx e Ambiente de Produção: http://www.sigplan.gov.br/infrasig/infrasig.asmx

[15]https://webservice.tesouro.fazenda.gov.br/TitulosTN.asmx

[16]http://www.nfe.fazenda.gov.br/portal/Default.aspx

Capítulo

2

Web Services SOAP-WSDL

2.1 Introdução aos serviços Web JAX-WS

A API de Java para serviços Web XML (JAX-WS) , JSR 224, o JAX-WS é uma parte importante da plataforma Java EE 6. Uma sequência da versão da API de Java para RPC 1.1 (JAX-RPC) com base em XML, o JAX-WS simplifica a tarefa de desenvolvimento de serviços Web utilizando a tecnologia Java. Ela enfoca alguns dos problemas em JAX-RPC 1.1 fornecendo suporte a múltiplos protocolos, como SOAP 1.1, SOAP 1.2, XML, e fornecendo um recurso para dar suporte a protocolos adicionais junto com HTTP. JAX-WS usa JAXB 2.0 para vinculação de dados e dá suporte a personalizações para controlar interfaces de ponto de extremidade de serviço geradas. Com suporte a anotações, JAX-WS simplifica o desenvolvimento do serviço Web e reduz o tamanho de arquivos JAR do tempo de execução[1].

Esta seção demonstra os conceitos básicos sobre o uso do IDE Netbeans no desenvolvimento de um serviço Web JAX-WS. Após criar o serviço Web, você escreve três clientes de serviço Web que usam serviço Web em uma rede, denominado "consumo" de um serviço Web. Os três clientes são uma classe Java em uma aplicação Java SE, um servlet e uma página JSP em uma aplicação Web. Um tutorial mais avançado que focaliza os clientes é Desenvolvendo clientes de serviço Web JAX-WS.

[1] `http://netbeans.org/kb/docs/websvc/jax-ws_pt_BR.html`

2.1.1 Provedor de serviço Hello World

Vamos criar um novo projeto no Netbeans para isso siga os seguintes passos vá em Arquivo -> Novo Projeto -> Java Web -> Aplicação Web, escolha o nome **WSHelloWorld** que será o nosso web service. Em seguida escolha o diretório da aplicação com o Tomcat 7.0 escolha a versão JavaEE 6 Web. Após criar um novo projeto clique com o botão direito em cima do seu projeto e escolha a opção Serviço Web como mostrado na Figura 2.1.

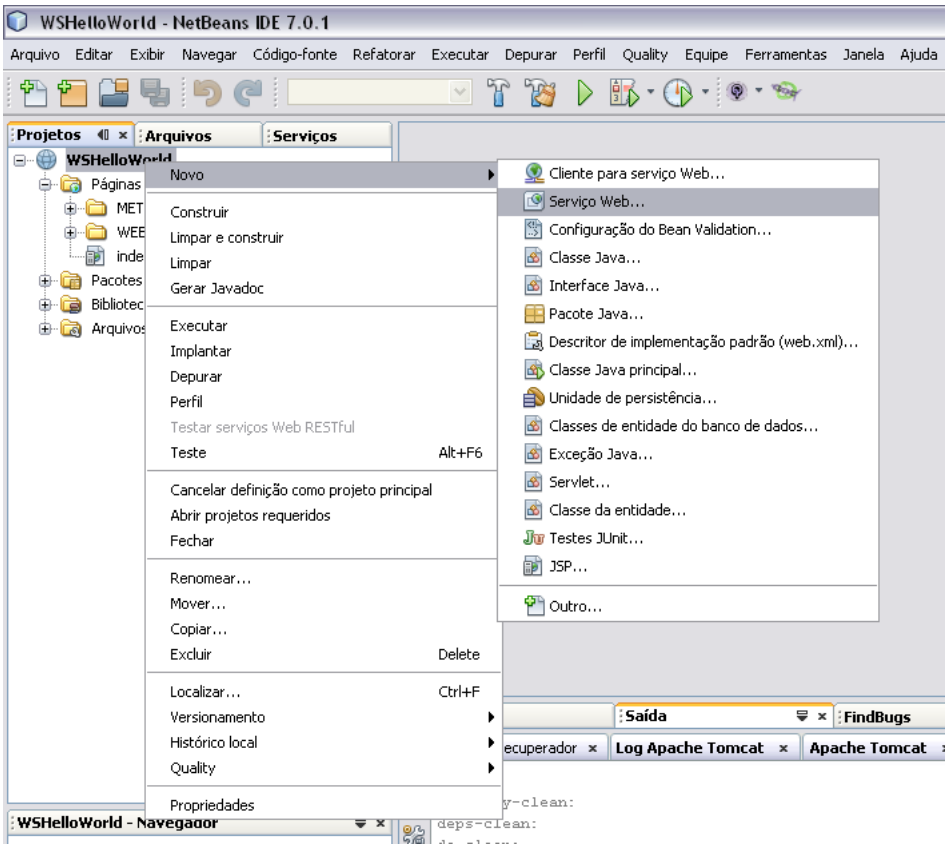


Figura 2.1: Serviço Web

Logo em seguida dê o nome de **HelloWorld** para o servico web. Como podemos visualizar na Figura 2.2

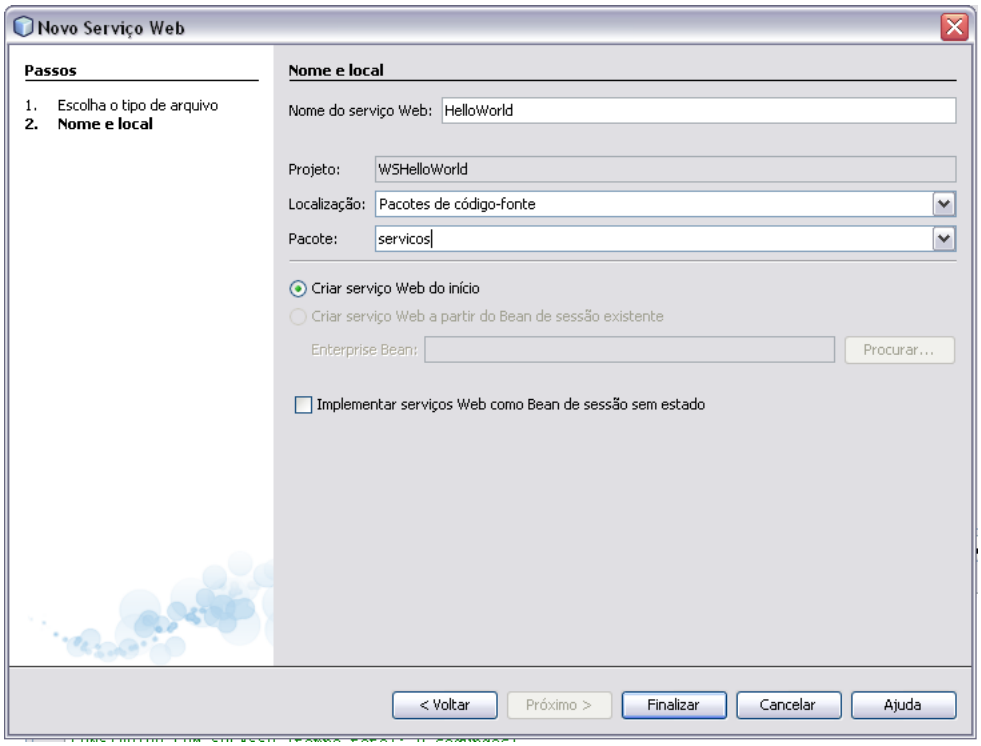


Figura 2.2: Novo Serviço Web

Em seguida o próprio Netbeans informará que o tomcat não oferece suporte a especificação JSR-109 responsavel por implementar e consultar serviços da Web no ambiente Java EE[2]. Clique em "SIM" que o Netbeans criará o arquivo sun-jaxws.xml e usará a pilha de serviços web da implementação METRO.

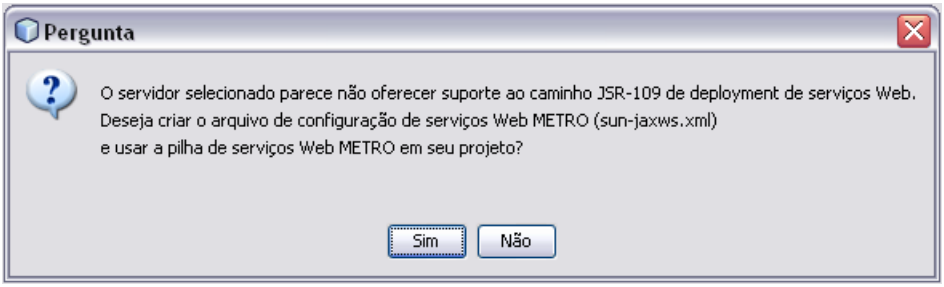


Figura 2.3: Especificação JSR-109

Após seguir todos os passos o próprio Netbeans já gera o código do serviço web Hello World conforme mostrado na Figura 2.4.

Após criado o serviço podemos acessá-lo por meio da sua interface WSDL. Além disso, é uma boa forma de saber se o serviço está funcionando corretamente para poder testá-lo. Lembrando que uma interface WSDL como descrita no Capítulo 1.1 possui duas partes:

[2] `http://publib.boulder.ibm.com/infocenter/radhelp/v7r5/index.jsp?topic=%2Forg.eclipse.jst.ws.doc.user%2Fconcepts%2Fcjsr109.html`

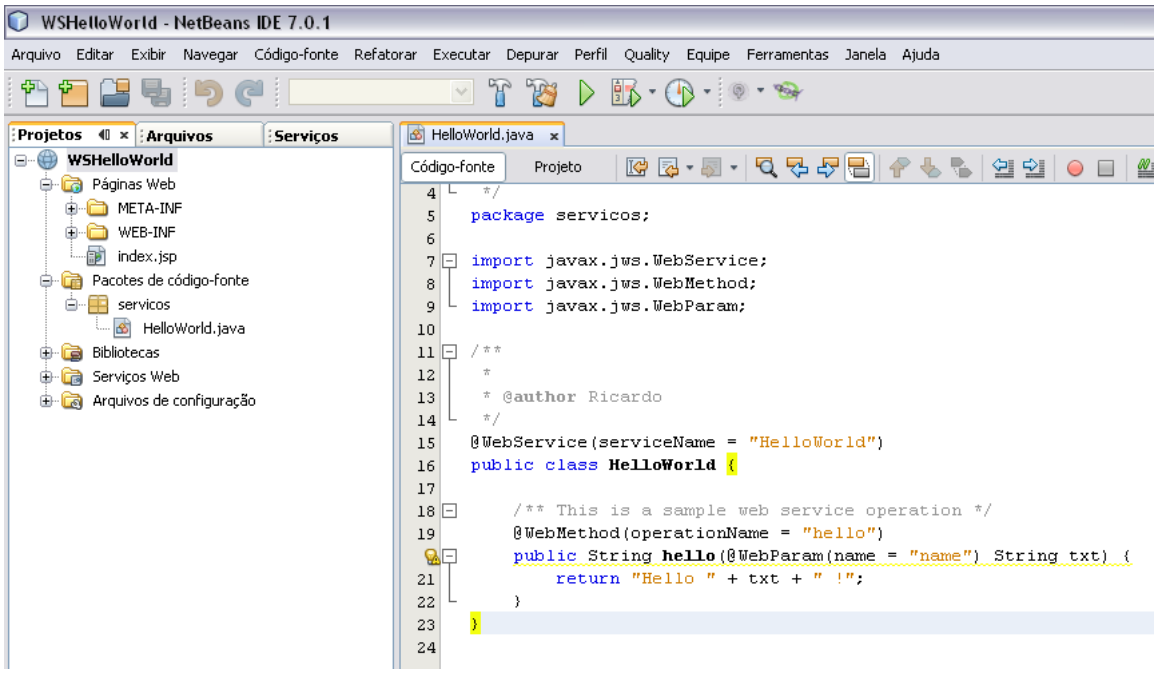


Figura 2.4: Servico HelloWorld

Descrição Abstrata e a **Descrição Concreta**. A Figura 2.5 ilustra a estrutura de um documento WSDL.

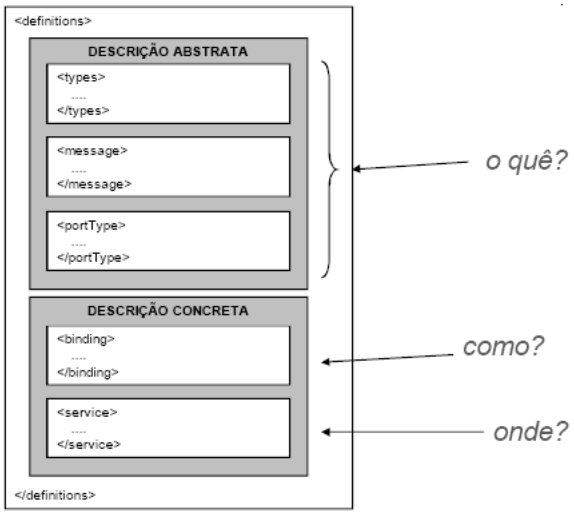


Figura 2.5: Estrutura do Arquivo WSDL

Para poder acessar o arquivo WSDL do nosso Hello World precisamos clicar com o botão direito em cima do nosso serviço HelloWorld e clicar na opção Gerar e Copiar WSDL, conforme mostra a Figura 2.6.

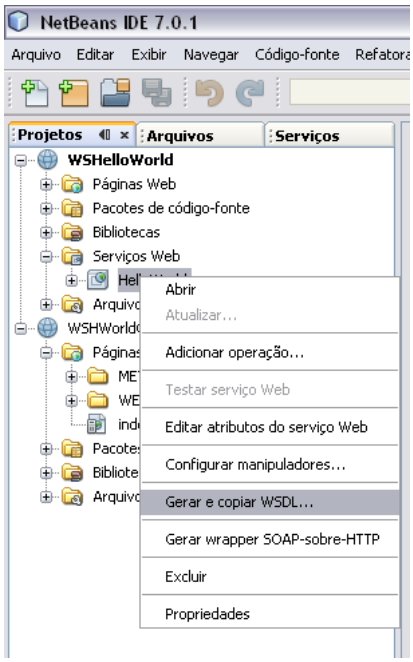


Figura 2.6: Gerar e Copiar arquivo WSDL

Logo em seguida escolha a pasta web da sua aplicação pois é lá que os usuário poderam acessar e consultar o arquivo WSDL da sua aplicação. Conforme exibido na Figura 2.7.

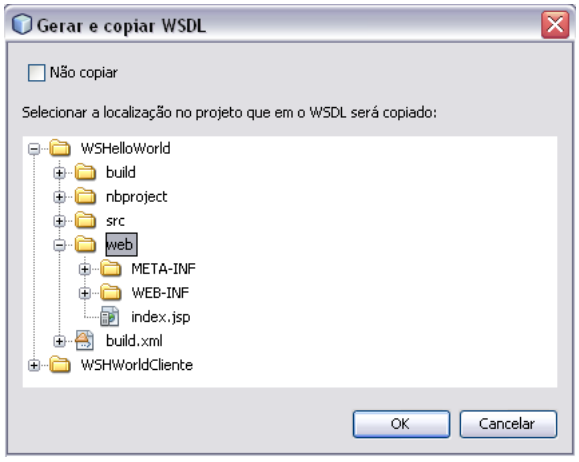


Figura 2.7: Pasta Web onde será gerado o Arquivo WSDL

O Netbeans vai gerar o arquivo wsdl e um arquivo xml schema e mais dois pacotes chamados resources e servicos.jaxws, conforme ilustrado na Figura 2.8

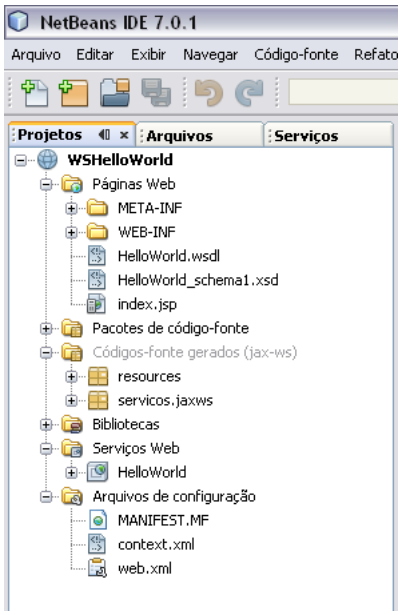


Figura 2.8: Arquivos e Pacotes gerados pelo JAX-WS

Para terminar e testar o nosso provedor de serviço web Hello World basta implantá-lo pelo tomcat e logo em seguida acessar a seguinte URL no seu *browser* ou navegador web: `http://localhost:8080/WSHelloWorld/HelloWorld?wsdl`. Conforme ilustrado na Figura 2.9.


```
- <!--
     Published by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is JAX-WS RI 2.2-hudson-740-.
  -->
- <!--
     Generated by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is JAX-WS RI 2.2-hudson-740-.
  -->
- <definitions targetNamespace="http://servicos/" name="HelloWorld">
  - <types>
    - <xsd:schema>
        <xsd:import namespace="http://servicos/" schemaLocation="http://localhost:8080/WSHelloWorld/HelloWorld?xsd=1"/>
      </xsd:schema>
    </types>
  - <message name="hello">
      <part name="parameters" element="tns:hello"/>
    </message>
  - <message name="helloResponse">
      <part name="parameters" element="tns:helloResponse"/>
    </message>
  - <portType name="HelloWorld">
    - <operation name="hello">
        <input wsam:Action="http://servicos/HelloWorld/helloRequest" message="tns:hello"/>
        <output wsam:Action="http://servicos/HelloWorld/helloResponse" message="tns:helloResponse"/>
      </operation>
    </portType>
```

Figura 2.9: Visualização do Arquivo WSDL

2.1.2 Cliente de serviço Hello World

Agora vamos criar o cliente para o nosso serviço web da seguinte forma, crie um novo projeto web com o nome **WSHWorldCliente**. Após criar o novo projeto clique com o botão direito em cima do projeto e vá em Novo -> Cliente para serviço Web. Assim como mostrado na Figura 2.10.

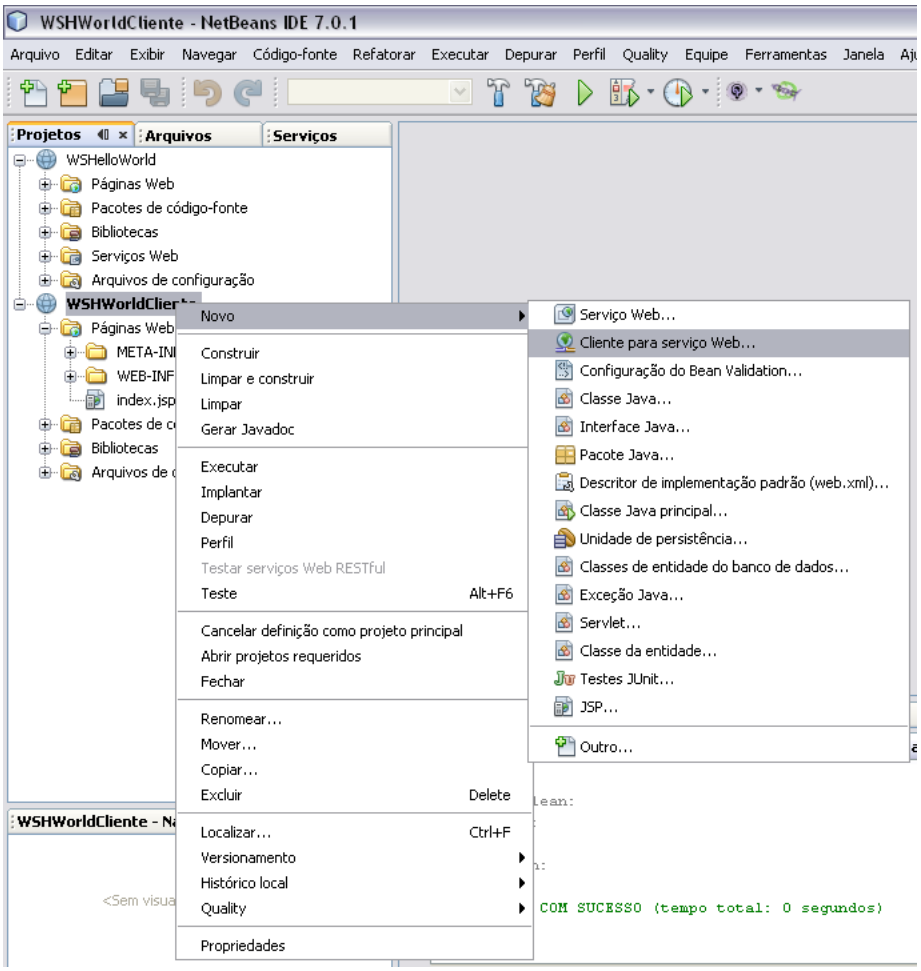


Figura 2.10: Novo Cliente HelloWorld em Java

Em seguida o Netbeans pedirá a URL do arquivo de descrição WSDL. Preencha os dados conforme ilustrado na Figura 2.11.

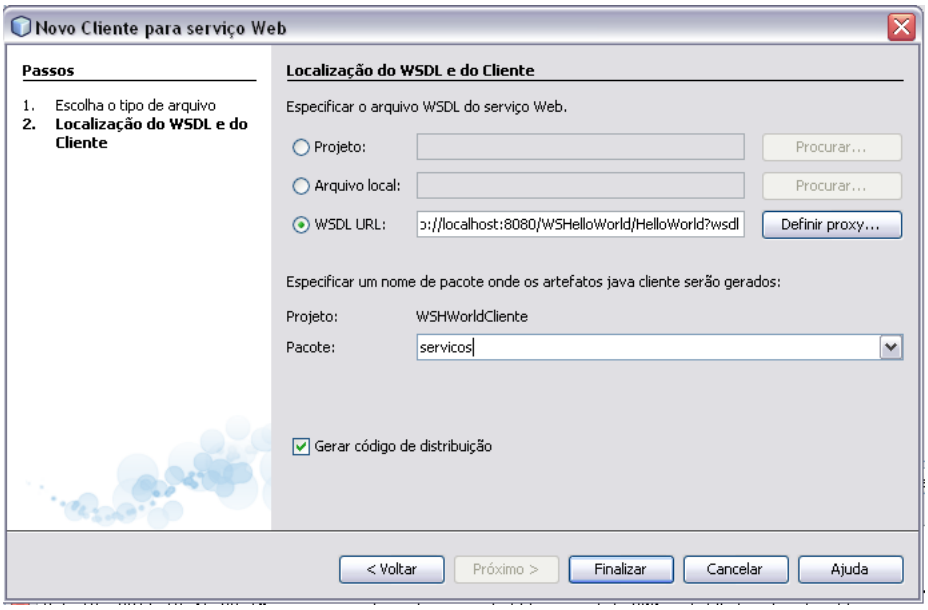


Figura 2.11: Endereço do Arquivo WSDL

Após seguir todos os passo o Netbeans vai gerar todo o código no cliente na pasta servicos, conforme mostrado na Figura 2.12.

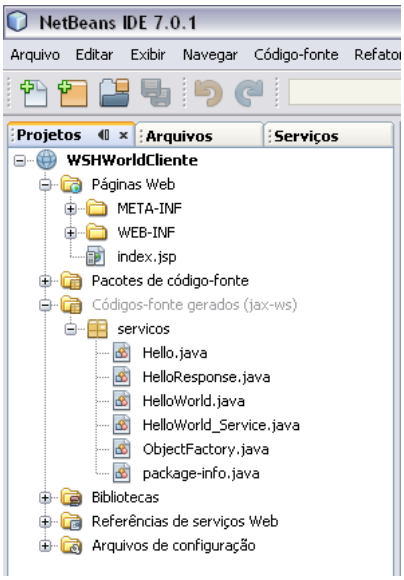


Figura 2.12: Implementação da Calculadora em SOAP-WSDL

O Código Final da página JSP do Cliente Hello World utilizando o JAX-WS consumidor do SOAP-WSDL ficará como mostrado na Figura 2.13.

Acesse a seguinte URL no seu *browser*: `http://localhost:8080/WSHWorldCliente/` você verá a imagem conforme ilustrado na Figura 3.16.

index.jsp | HelloWorld.java

```
1  <%--
2      Author      : Ricardo Ramos de Oliveira
3  --%>
4
5  <%@page import="servicos.HelloWorld"%>
6  <%@page import="servicos.HelloWorld_Service"%>
7  <%@page contentType="text/html" pageEncoding="UTF-8"%>
8  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
9     "http://www.w3.org/TR/html4/loose.dtd">
10 <html>
11     <head>
12         <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
13         <title>Hello World</title>
14     </head>
15     <body>
16         <%
17             //cria a comunicação com o Web Service
18             HelloWorld_Service hs = new HelloWorld_Service();
19
20             //recupera uma classe que implemente a interface do nosso Web Service
21             HelloWorld hw = hs.getHelloWorldPort();
22
23             //exibindo na pagina por meio do método disponível pelo web service
24             out.print(hw.hello("<h1>Ricardo Ramos de Oliveira</h1>")+"<hr>");
25         %>
26     </body>
27 </html>
28
```

Figura 2.13: Código JSP do Cliente Hello World

Figura 2.14: Visualização do Hello World pelo Browser

2.1.3 Provedor de serviço Calculadora

Agora vamos criar um novo projeto web como o nome **WSCalculadora** provedor Calculadora da mesma forma como criamos o HelloWorld. Para isso vamos seguir os seguintes passos:

Crie o novo projeto

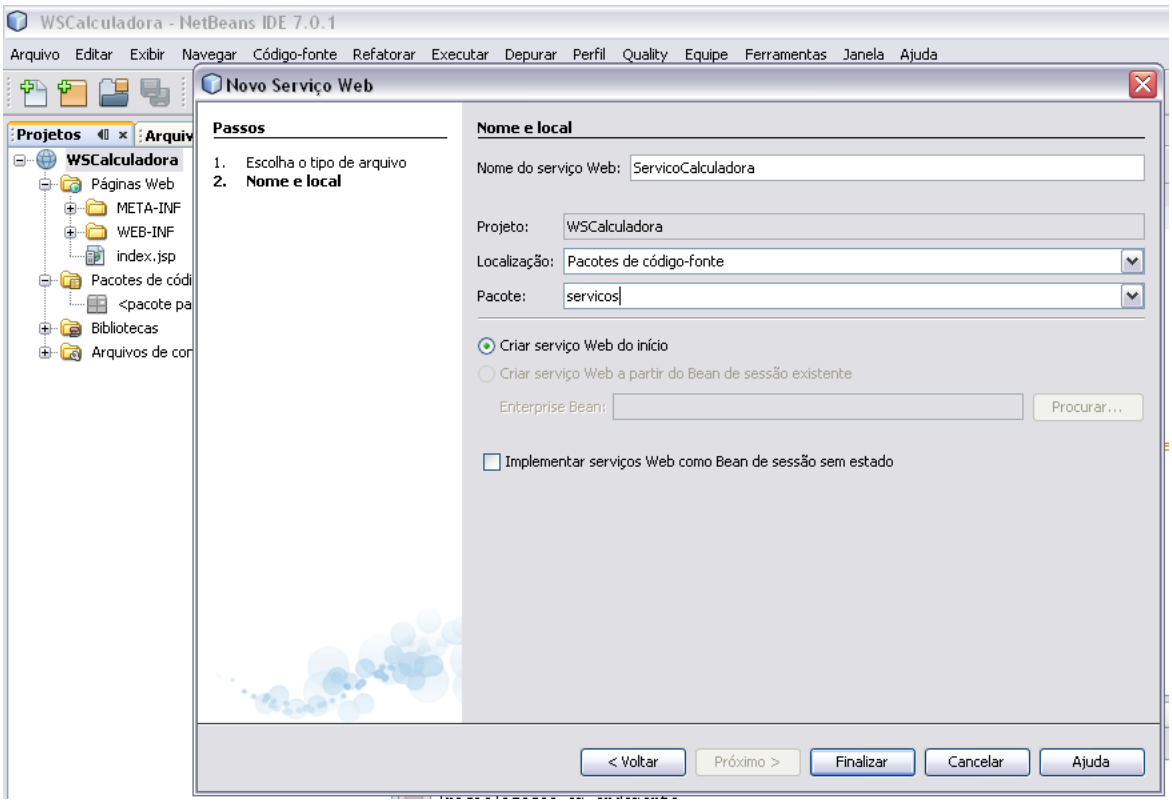


Figura 2.15: Criando o Servico Web Calculadora

Adicione as operções da Calculadora

O Neteans possui também uma interface gráfica para fazer a assinatura das operções, porém não a implementa.

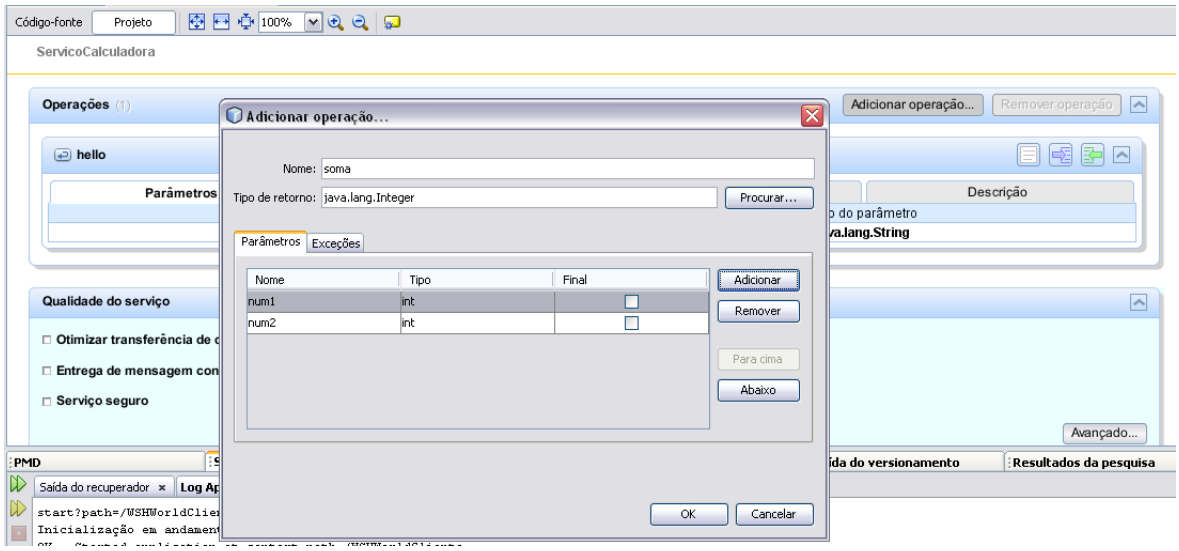


Figura 2.16: Utilizando a Interface Gráfica do Netbeans

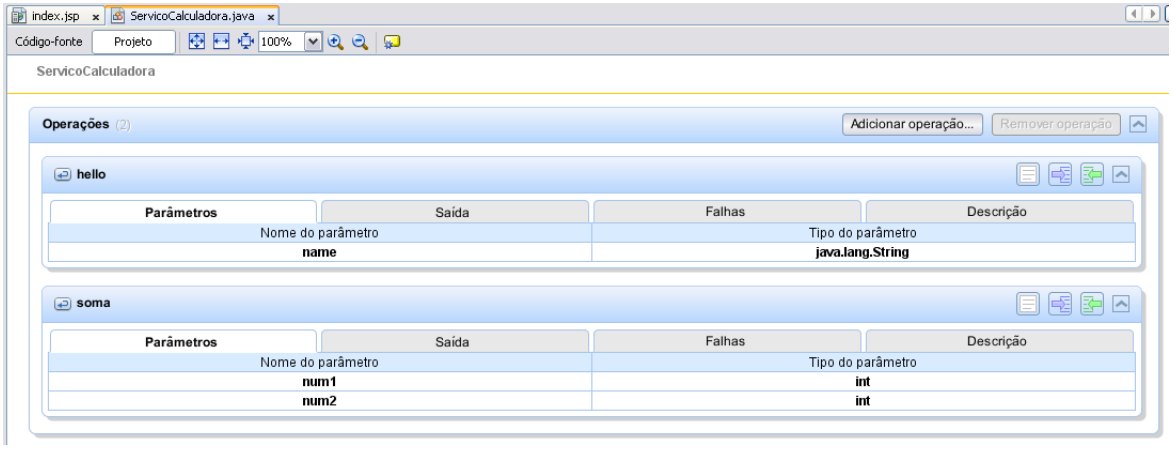


Figura 2.17: Adicionando Operções pela Interface do Netbeans

Implemente os Servico Calculadora

```java
/**
 *
 * @author Ricardo
 */
@WebService(serviceName = "ServicoCalculadora")
public class ServicoCalculadora {

    @WebMethod(operationName = "soma")
    public int soma(@WebParam(name = "num1") int num1,
                    @WebParam(name = "num2") int num2) {
        return num1 + num2;
    }

    @WebMethod(operationName = "subtracao")
    public int subtracao(@WebParam(name = "num1") int num1,
                         @WebParam(name = "num2") int num2) {
        return num1 - num2;
    }

    @WebMethod(operationName = "multiplicacao")
    public int multiplicacao(@WebParam(name = "num1") int num1,
                             @WebParam(name = "num2") int num2) {
        return num1 * num2;
    }

    @WebMethod(operationName = "divisao")
    public int divisao(@WebParam(name = "num1") int num1,
                       @WebParam(name = "num2") int num2) {
        if(num2 != 0){
           return num1 / num2;
        }
        return 0;
    }
}
```

Figura 2.18: Implementação do Web Service SOAP-WSDL para a Calculadora

Acessando o Servico Calculadora

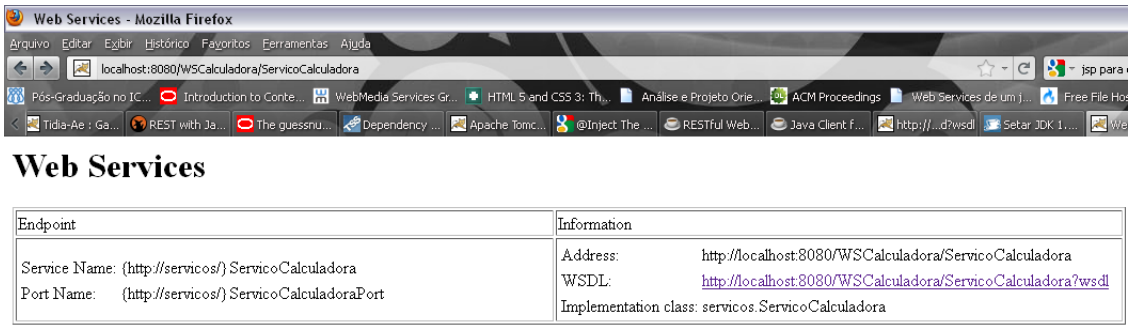


Figura 2.19: Acesso o Servio Web pelo Browser

Clique no link: `http://localhost:8080/WSCalculadora/ServicoCalculadora?wsdl` para acessar o arquivo WSDL do serviço calculadora.

2.1.4 Cliente de serviço Calculadora

Construa o Cliente da seguinte forma como ilustrado nas Figuras 2.20 e 3.34.

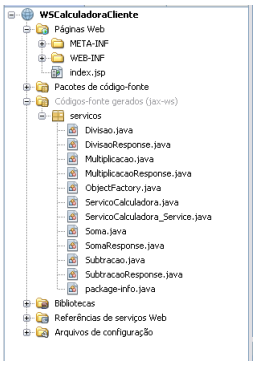


Figura 2.20: Cliente do Serviço Web Calculadora

index.jsp

```
1  <%--
2      Author     : Ricardo Ramos de Oliveira
3  --%>
4
5  <%@page import="servicos.*"%>
6  <%@page contentType="text/html" pageEncoding="UTF-8"%>
7  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
8     "http://www.w3.org/TR/html4/loose.dtd">
9  <html>
10     <head>
11         <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
12         <title>Calculadora</title>
13     </head>
14     <body>
15         <%
16             //cria a comunicação com o Web Service
17             ServicoCalculadora_Service scs = new ServicoCalculadora_Service();
18
19             //recupera uma classe que implemente a interface do nosso Web Service
20             ServicoCalculadora sc = scs.getServicoCalculadoraPort();
21
22             //exibindo na pagina por meio do método disponível pelo web service
23             out.print("Soma = 1 + 1 = " + sc.soma(1, 1) + "<hr>");
24             out.print("Subtração = 9 - 14 = " + sc.subtracao(9, 14) + "<hr>");
25             out.print("Multiplicação = 3 * 3 = " + sc.multiplicacao(3, 3) + "<hr>");
26             out.print("Divisão = 8 / 2 = " + sc.divisao(8, 2) + "<hr>");
27         %>
28     </body>
29 </html>
30
```

Figura 2.21: Código JSP do cliente Calculadora

Ao executar o código do cliente será exibida a seguinte página web como mostrado na Figura 2.22.

A comunicação entre os web services é feita por meio do protoclo SOAP e pode ser visualizada também através do programa Wireshark[3]. A Figura 2.23 a seguir mostra um

[3]http://www.wireshark.org/

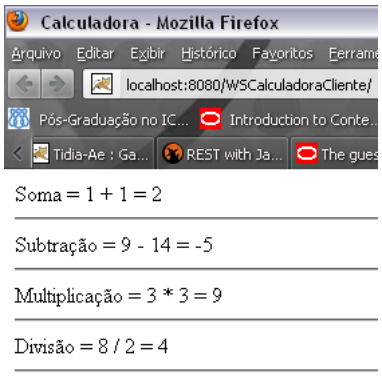


Figura 2.22: Visualização dos Calculos obtidos pelo cliente Web Service

exemplo de captura de pacotes com envelopes SOAP mais detalhes pode ser obtido em `http://www.jroller.com/gmazza/entry/soap_calls_over_wireshark`.

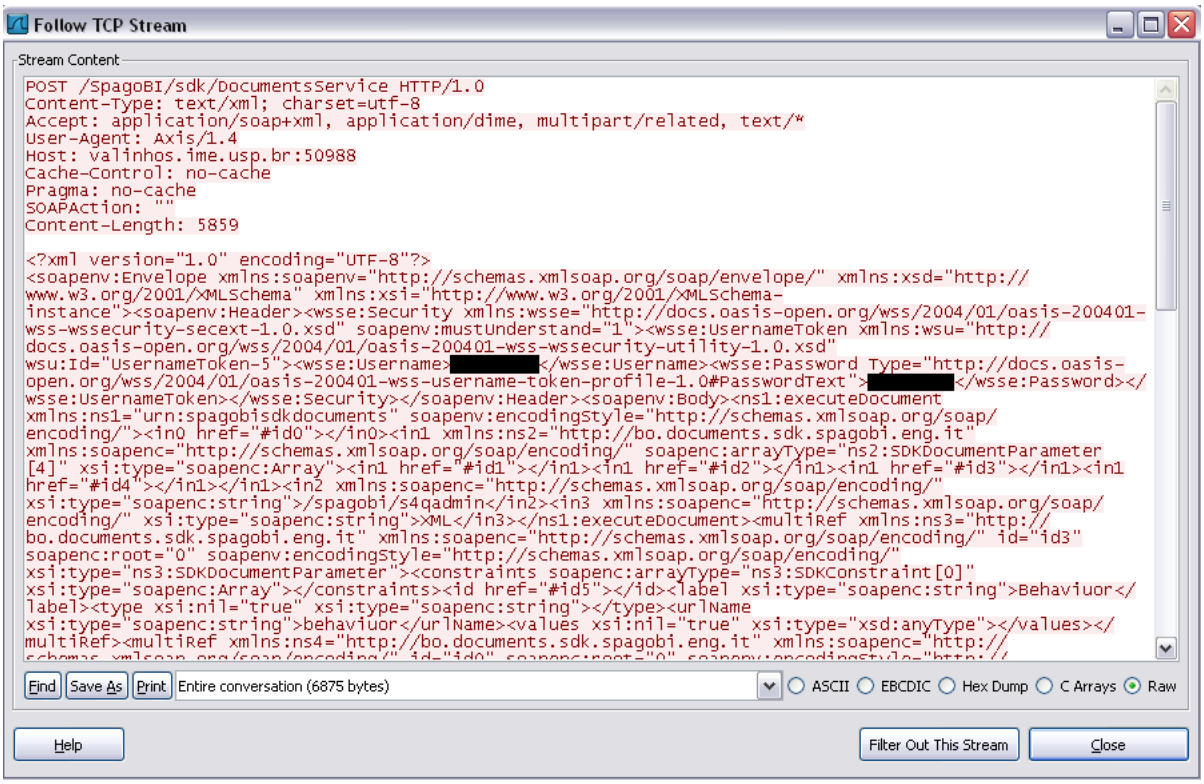


Figura 2.23: *Tracing* SOAP com o Wireshark

2.1.5 Provedor de serviço Biblioteca

Agora vamos fazer um exemplo bem simples de CRUD. Primeiro vamos precisar de uma entidade, um bean para conter as informações que nós queremos manipular. Veja o exemplo abaixo:

```
package br.com.k19.models;

import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class Livro {

    private int id;
    private String titulo;
    private String autor;

    //getters e setters

}
```

Figura 2.24: Classe Livro

Note a presença da anotação **@XmlRootElement** na definição da nossa classe. Ela é fundamental para que o Jersey conheça a representação do objeto livro em XML. Desse modo os atributos irão se tornar nós no momento em que o XML for enviado ao cliente. E do mesmo jeito, quando chegar um XML no corpo de uma requisição, o METRO vai saber converter esse XML em um objeto do tipo Livro.

O próximo passo é criar o nosso web service. No mundo real, geralmente você vai ter uma tabela "Livro" em alguma base de dados e vai desejar buscar a informação quando for requisitada. Aqui para que o exemplo fique mais simples e a gente mantenha o foco apenas na funcionalidade do METRO, vamos utilizar um Map para simular a nossa base de dados. Veja o exemplo abaixo:

```java
@WebService(serviceName = "ServicosSOAPBiblioteca")
public class ServicosSOAPBiblioteca {

    // vamos utilizar um Map estático para
    // "simular" uma base de dados
    static private Map<Integer, Livro> livrosMap;

    static {
        livrosMap = new HashMap<Integer, Livro>();

        Livro livro1 = new Livro();
        livro1.setId(1);
        livro1.setTitulo("Livro de Java");
        livro1.setAutor("Deitel");

        livrosMap.put(livro1.getId(), livro1);

        Livro livro2 = new Livro();
        livro2.setId(2);
        livro2.setTitulo("Livro de C");
        livro2.setAutor("Dennis Ritch");

        livrosMap.put(livro2.getId(), livro2);
    }

    @WebMethod(operationName = "listarLivros")
    public List<Livro> getLivros() {
        return new ArrayList<Livro>(livrosMap.values());
    }
}
```

Figura 2.25: Serviço Web em SOAP para uma Biblioteca

Para testar, utilize o programa SOAPUI, que o resultado será um XML parecido com esse:

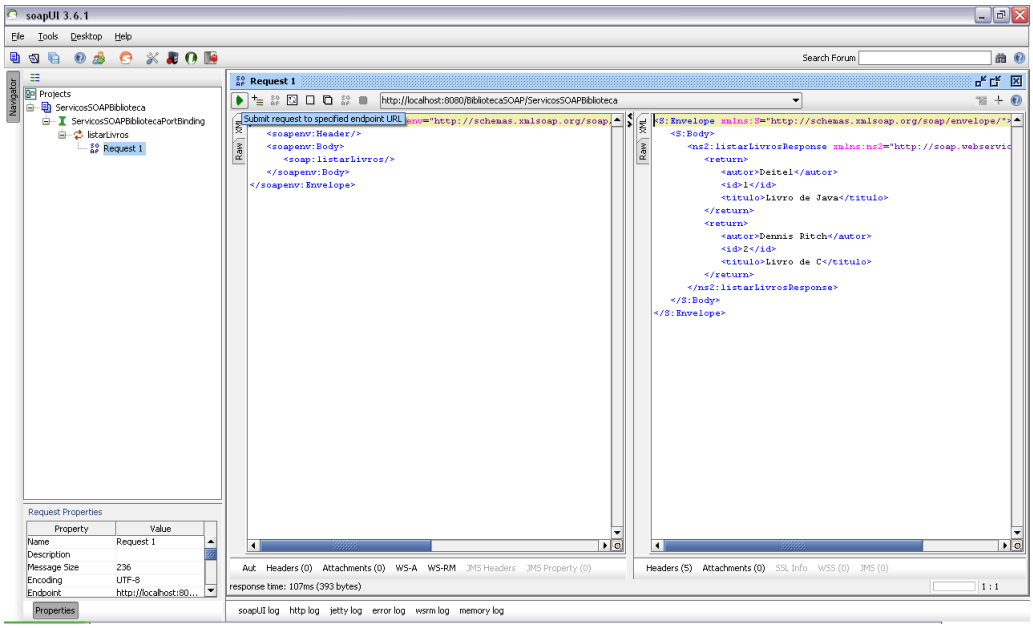


Figura 2.26: Testando o Serviço SOAP para Listar os Livros da Biblioteca

E caso queiramos listar apenas um livro em vez de todos? Também podemos incluir dentro da anotação @WebMethod a anotação @WebParam no parâmetro igual ao exemplo abaixo:

```
@WebMethod(operationName = "procurarLivroPorId")
public Livro getlivro(@WebParam(name = "id") int id) {
    return livrosMap.get(id);
}
```

Figura 2.27: Testando o Serviço SOAP para Listar os Livros da Biblioteca

Agora que conseguimos listar os livros, precisamos também de uma maneira de salvar, modificar e excluir um Livro. Veja o exemplo abaixo:

```
@WebMethod(operationName = "salvarLivro")
public String salvarlivro(Livro livro) {
    livro.setId(livrosMap.size() + 1);
    livrosMap.put(livro.getId(), livro);
    return livro.getTitulo() + " adicionado.";
}

@WebMethod(operationName = "atualizarLivro")
public String atualizarlivro(Livro livro, @WebParam(name = "id") int id) {
    Livro livroAtual = livrosMap.get(id);
    livroAtual.setTitulo(livro.getTitulo());
    livroAtual.setAutor(livro.getAutor());
    return livro.getTitulo()+ " atualizado.";
}

@WebMethod(operationName = "removerLivro")
public String removelivro(@WebParam(name = "id") int id) {
    livrosMap.remove(id);
    return "livro removido.";
}
```

Figura 2.28: Código para Salvar, Modificar e Excluir um Livro

Exercicio

Faça os testes no programa SOAP UI para os métodos salvar, modificar, getlivro (procurarLivroPorId) e excluir um Livro.

Após terminar completar o desenvolvimento do serviço web SOAP para a biblioteca a hierarquia de pastas do seu projeto deve estar parecida com a da Figura 2.35.

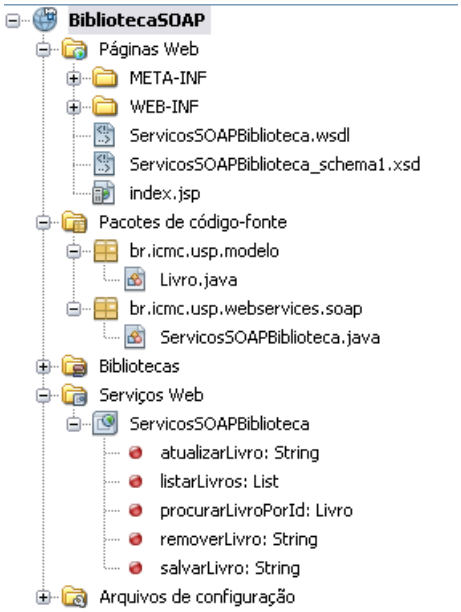


Figura 2.29: Hierarquia de Pastas do Projeto Biblioteca

2.1.6 Cliente de serviço Biblioteca

O provedor já esta pronto, precisamos agora construi e implementar o nosso cliente SOAP-WSDL para consumir o serviço web da biblioteca. Crie um projeto web com o nome ClienteSOAPBiblioteca, logo em seguida crie também o pacote br.icmc.usp.webservices.soap no seu projeto web.

Com botão direito em cima da raiz do seu projeto, crie um novo cliente para serviço web por meio do próprio Netbbeans como mostrado na Figura xxx.

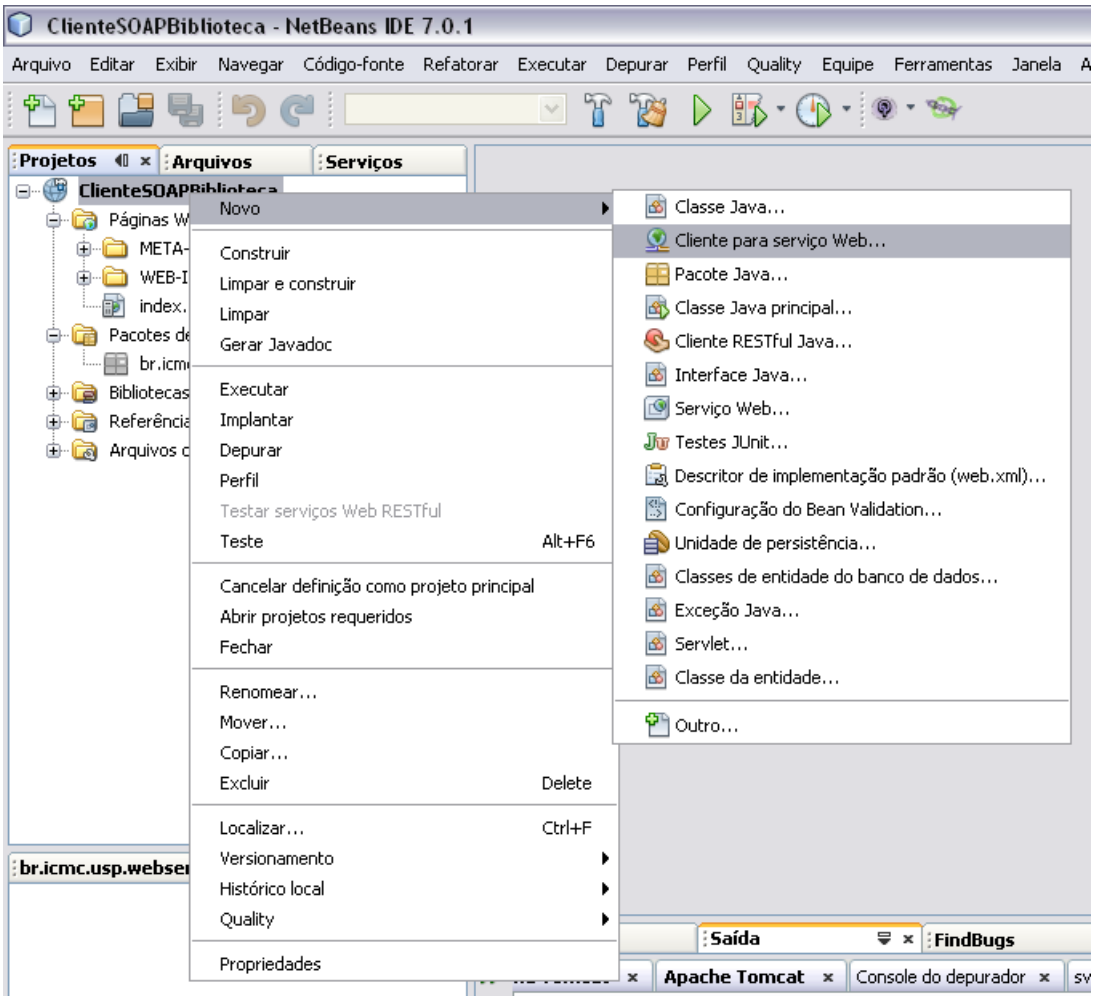


Figura 2.30: Hierarquia de Pastas do Projeto Biblioteca

para isso utilize o enredeço da URL do arquivo WSDL do servidor:
`http://localhost:8080/BibliotecaSOAP/ServicosSOAPBiblioteca?wsdl`

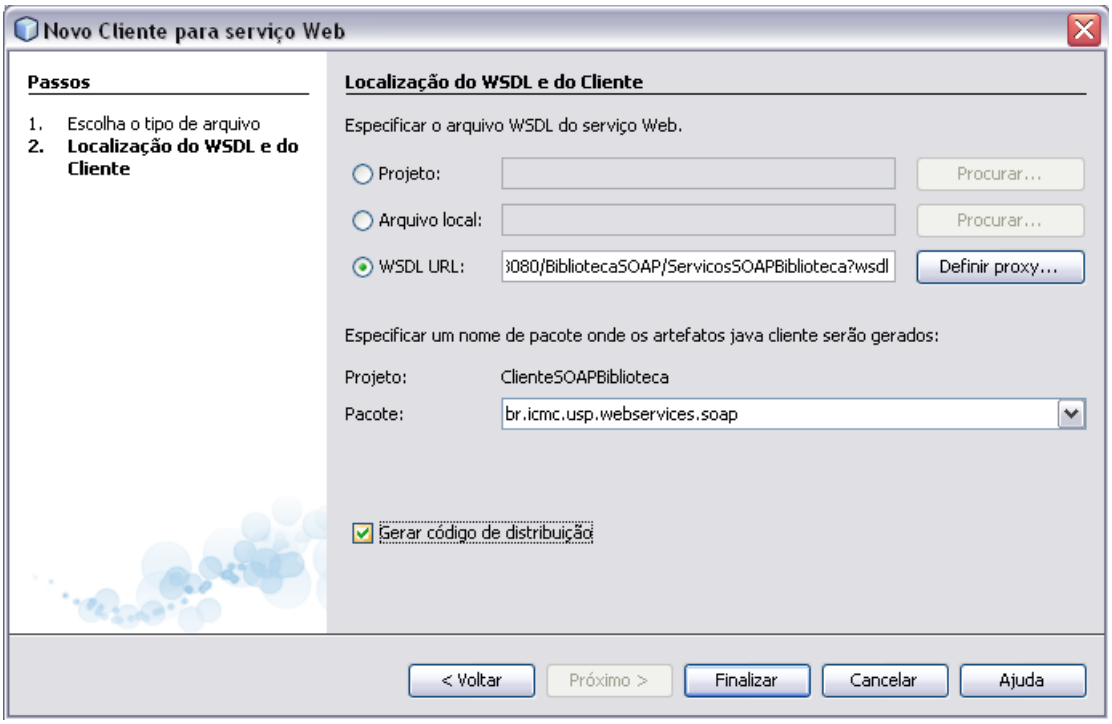


Figura 2.31: Hierarquia de Pastas do Projeto Biblioteca

Após esses passo será gerada automaticamente as classes por meio do API JAX-WS. Logo em seguida podesmo observar que o próprio JAX-WS criou duas classes importantes na pasta de código fontes gerados os nomes da classe são exatamente: ServicosSOAPBiblioteca e ServicosSOAPBiblioteca_Service

Agora podemos começar a desenvolver o nosso cliente SOAP para a nossa biblioteca, primeiro crie uma classe com o nome: ClienteSOAPLivro dentro do pacote br.icmc.usp.webservices.soap, como mostrado na Figura xxx.

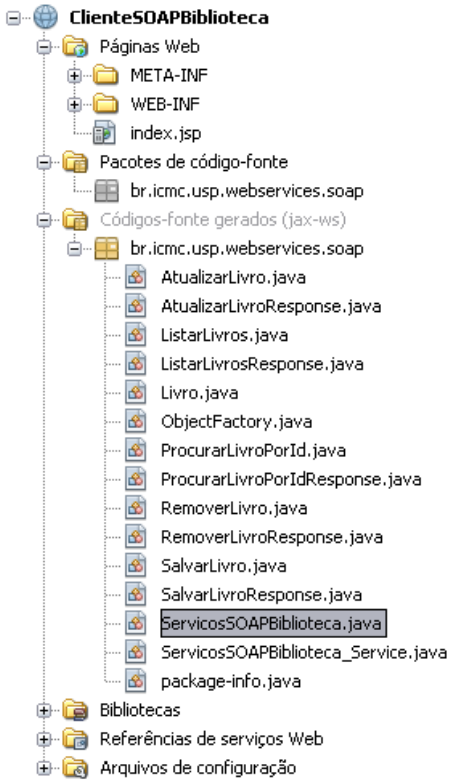


Figura 2.32: Hierarquia de Pastas do Projeto Biblioteca

Os componentes responsáveis pela parte da interação entre clientes e serviços através de mensagens SOAP são denominados *stubs*. Os *stubs* realizam toda a parte de criação de mensagens SOAP, configuração do protocolo de transporte e geram as conexões necessárias para permitir o envio e recebimento das mensagens SOAP envolvidas no acesso ao serviço. A Figura xxx contém o código-fonte do cliente que executa a chamada dos métodos do serviço publicado. A classe ClienteSOAPLivro utiliza os stubs gerados pelo JAX-WS por meio do arquivo WSDL do serviço, facilitando o desenvolvimento do serviço. Na verdade, os *stubs* são responsáveis por criar as requisições do SOAP a partir de anotações JAXB e convertendo a resposta em SOAP de volta para objetos Java.

Agora precisamos testar o nosso cliente SOAP-WSDL, para isso vamos criar um outro pacote na nossa aplciação com o nome br.icmc.usp.webservices.teste e dentro do pacote crie uma classe de teste com o nome ClienteSOAPLivroTeste como mostrado na Figura xxx.

O resultado da saída após a execução do teste da Figura xxx será identico ao mostrado na Figura xxx a seguir.

Após construir o teste para listar todos os livros, constru e desenvolva os testes para getLivro por meio do id, salvar, editar e remover um livro.

```
package br.icmc.usp.webservices.soap;

import java.util.List;

public class ClienteSOAPLivro {

    private ServicosSOAPBiblioteca_Service scs;
    private ServicosSOAPBiblioteca sc;

    public ClienteSOAPLivro() {
        //cria a comunicação com o Web Service
        scs = new ServicosSOAPBiblioteca_Service();
        //recupera uma classe que implemente a interface do nosso Web Service
        sc = scs.getServicosSOAPBibliotecaPort();
    }
    public List<Livro> getTodosLivros() {
        return sc.listarLivros();
    }

    public Livro getLivro(Integer id){
        return sc.procurarLivroPorId(id);
    }

    public void inserirLivro(Livro Livro) {
        sc.salvarLivro(Livro);
    }

    public void atualizarLivro(Livro Livro) {
        sc.atualizarLivro(Livro, Livro.getId());
    }

    public void removerLivro(Integer id) {
       sc.removerLivro(id);
    }
}
```

Figura 2.33: Hierarquia de Pastas do Projeto Biblioteca

2.1.7 Exercicios

Todos os serviços Web devem ser implementados usando um banco de dados de sua preferência.

Exercício Fácil

1 - Vamos imaginar o seguinte cenário: um jornal quer disponibilizar um serviço para distribuir notícias pela internet. Sendo assim, as pessoas podem colocar informações confiáveis no seu site, e toda vez que o pessoal da redação adicionar uma nova notícia, ela aparece em todos os sites que consomem o serviço. Com base nesse cenário, crie um serviço web para listar noticias. Obs: A classe Notícia é composta por dois atributos título e texto da notícia.

```
package br.icmc.usp.webservices.teste;

import br.icmc.usp.webservices.soap.ClienteSOAPLivro;
import br.icmc.usp.webservices.soap.Livro;
import java.util.List;

public class ClienteSOAPLivroTeste {

    public static void main(String[] args) {
        ClienteSOAPLivro cliente = new ClienteSOAPLivro();
        List<Livro> livros = cliente.getTodosLivros();
        for(Livro livro: livros){
            System.out.println("#----------------------------------------#");
            System.out.println("Id do Livro: " + livro.getId());
            System.out.println("Título do Livro: " + livro.getTitulo());
            System.out.println("Autor do Livro: " + livro.getAutor());
            System.out.println("#----------------------------------------#");
        }
    }
}
```

Figura 2.34: Hierarquia de Pastas do Projeto Biblioteca

```
run:
#----------------------------------------#
Id do Livro: 1
Título do Livro: Livro de Java
Autor do Livro: Deitel
#----------------------------------------#
#----------------------------------------#
Id do Livro: 2
Título do Livro: Livro de C
Autor do Livro: Dennis Ritch
#----------------------------------------#
CONSTRUÍDO COM SUCESSO (tempo total: 2 segundos)
```

Figura 2.35: Hierarquia de Pastas do Projeto Biblioteca

Exercício Desafio

2 - Implemente um web service SOAP-WSDL que ofereça o serviço de CRUD (Create, Read, Update e Delete) para cada uma das tabelas do banco de dados abaixo.

Exercício Extra

3 - Crie um web service SOAP-WSDL para consultar e listar todos os Orgãos do Governo Federal pelo SIORG (Sistema de Informações Organizacionais do Governo Federal).

Arquivo WSDL: `http://hom.siorg.redegoverno.gov.br/gestao/webservice/WSSiorg.asmx?WSDL`

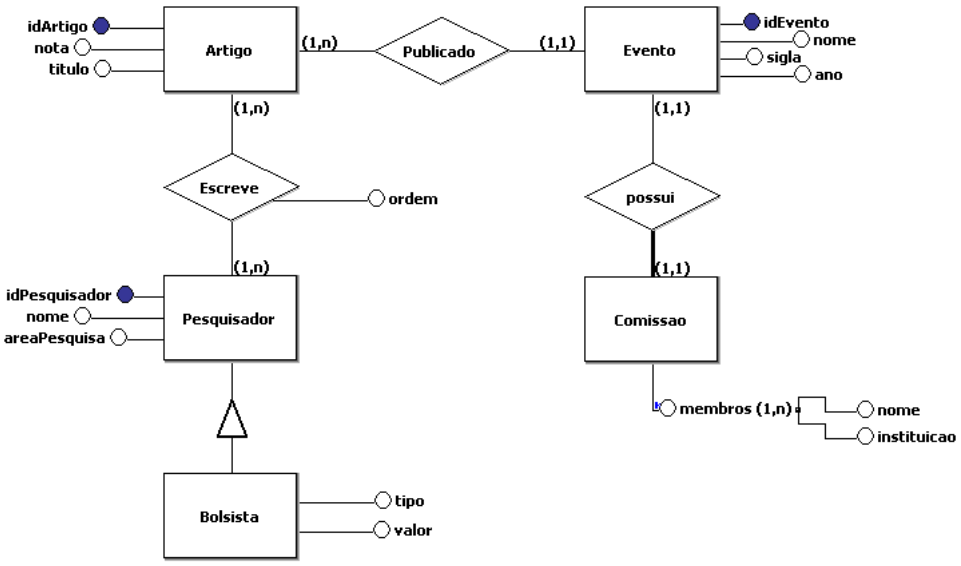


Figura 2.36: Pilha conceitual de web services (IBM, 2009)

Capítulo

3

Web Services RESTful

3.1 Introdução aos Web Services RESTful

Representational State Transfer, ou REST foi criado em 2000 pela tese de doutorado de Roy Fielding (Fielding, 2000), um dos principais autores da especificação HTTP versões 1.0 e 1.1.

O conceito mais importante em REST são os recursos[1], que são identificados por IDs globais usando URIs. Os aplicativos cliente usam métodos HTTP (GET/POST/PUT/-DELETE) para manipular o recurso ou uma coleção de recursos. A implementação de um web service RESTful utiliza o protocolo HTTP e os princípios de REST. Tipicamente, um web service RESTful deve definir os seguintes aspectos:

- A URI base/raiz para o serviço web, tais como o contexto da aplicação e o seus recursos.
- O tipo MIME dos dados de resposta suportados são: JSON, XML, texto puro entre outros.
- O conjunto de operações suportadas pelo serviço. (por exemplo, POST, GET, PUT ou DELETE).

[1]Um recurso pode ser uma página web, figura, imagem ou vídeo.

Além disso, os serviços web RESTful podem ser construídos com ferramentas mínimas, é de baixo custo para a aquisição e, portanto, tem pouca resistência para a sua adoção. O esforço necessário para construir um cliente para um serviço RESTful é muito pequeno, os desenvolvedores podem começar a testar esses serviços a partir de um navegador da web comum, sem ter que desenvolver o software de cliente personalizado. Implantar um serviço web RESTful é muito similar à construção de um site dinâmico (Pautasso et al., 2008).

A parte mais importante de um web service RESTful se refere à suas URIs. Em geral, temos uma URI base para identificar um recurso, e teremos operações que variam conforme o método HTTP utilizado. Essas operações seriam equivalentes às mesmas operações de CRUD[2].

O que é importante ter em mente é que o principal em um restful web service são as URLs do sistema (geralmente são referidas como restful url's), e os resources. Um resource é um recurso, uma entidade, ou seja, é um objeto com informação que será representado por meio de um XML. Em geral, a URL para acessar esse recurso será sempre a mesma, porém caso mudemos o método HTTP (GET, POST, PUT, DELETE) o resultado da requisição será diferente. Veja os exemplos tabela abaixo:

Operações sobre uma coleção:

http://exemplo.com/livros/

- GET obtém a coleção
- POST adiciona um item à coleção
- PUT substitue a coleção
- DELETE remove a coleção

Operações sobre um item:

http://exemplo.com/livros/45

- GET obtém o item
- POST —
- PUT atualiza o item

[2]CRUD é acrônimo de Create, Retrieve, Update e Delete em língua Inglesa, para as quatro operações básicas utilizadas em bancos de dados relacionais ou em interface para usuários para criação, consulta, atualização e destruição de dados.

- DELETE remove o item

Porém saiba que no REST não existe um padrão definido, as operações listadas acima são apenas uma sugestão. Ao desenvolver um web service você não é obrigado a implementar todas as operações para seguir um determinado padrão, você só precisa implementar o que é necessário no seu contexto. Por exemplo, eu posso decidir que os recursos não podem ser apagados e não implementar nenhuma operação para o método DELETE. O que deve ser levado em consideração é a convenção das ações. GET deve ser usado para "leitura", por exemplo listar os detalhes de um recurso. POST deve ser usado para adicionar novos recursos. PUT deve ser utilizado para alterar recursos já existentes. E DELETE para apagar recursos.

Assim como a API JAX-WS, no Java EE, existe uma API chamada JAX-RS que padroniza anotações para criar web services seguindo os princípios REST. Da mesma forma como o Metro[3] é a Implementação de Referência (RI) para a especificação JAX-WS. Existe o Jersey[4] que é a Implementação de Referência (RI) para a especificação JAX-RS.

3.1.1 Anotações

Pelo nosso exemplo anterior, já vimos o Jersey em ação e a única coisa que nós fizemos foi adicionar algumas anotações à nossa classe. O Jersey é todo baseado em anotações, e existem várias delas. Vamos listar as mais importantes.

@Path – essa anotação recebe uma string como parâmentro e indica qual é o path da URL. No exemplo anterior, tivemos a classe anotada com o valor "/helloworld", e por isso que acessamos a URL como http://localhost:8080/jersey-tutorial/helloworld.

@GET – anotação que indica qual o método correspondente do HTTP. Como dito anteriormente, podemos ter a mesma URL para ações diferentes desde que o método HTTP também seja diferente. Da mesma forma, temos as anotações @POST, @PUT e @DELETE.

@Produces – anotação que indica qual o mime-type do conteúdo da resposta que será enviada para o cliente. No exemplo acima, foi "text/plain" para indicar que é texto puro. Em um web service isso é pouco usual, em geral vamos utilizar valores como "text/xml" para devolver XML.

@Consumes – anotação que indica qual o mime-type do conteúdo da requisição. Em geral é utilizado principalmente em requisições do tipo POST ou PUT, em que o cliente precisa enviar a informação do que ele deseja adicionar/alterar. Do mesmo jeito que o web service "devolve" XML, ele pode "consumir" (receber) conteúdo XML.

[3] `http://metro.java.net/getting-started/`
[4] `http://jersey.java.net/nonav/documentation/latest/user-guide.html`

3.1.2 Provedor de serviço Hello World

Primeiro vamos fazer o "Hello World" apenas para testar se o ambiente está configurado corretamente. Veja a classe abaixo:

```
package com.k19.restful.resources;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;

@Path("/helloworld")
public class HelloWorldResource {
        @GET
        @Produces("text/xml")
        public String helloWorldXML() {
                return "<?xml version=\"1.0\" encoding=\"UTF-8\"?>"
                        + "<teste>Olá mundo!</teste>";
        }
}
```

Figura 3.1: Hello World RESTful

Antes de rodar o exemplo, devemos configurar o web.xml e cadastrar a servlet do Jersey. Preste atenção que devemos passar um parâmetro na inicialização. Esse parâmetro corresponde ao pacote onde estão os nossos resources. Basta seguir o exemplo conforme abaixo:

HelloWorldResource.java | web.xml

Geral | Servlets | Filtros | Páginas | Referências | Segurança | XML

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <web-app xmlns="http://java.sun.com/xml/ns/javaee"
4           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5           xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.x
6           version="3.0">
7      <display-name>jersey-tutorial</display-name>
8      <servlet>
9          <servlet-name>Jersey REST Service</servlet-name>
10         <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
11         <init-param>
12             <param-name>com.sun.jersey.config.property.packages</param-name>
13             <param-value>com.k19.restful.resources</param-value>
14         </init-param>
15         <load-on-startup>1</load-on-startup>
16     </servlet>
17     <servlet-mapping>
18         <servlet-name>Jersey REST Service</servlet-name>
19         <url-pattern>/rest/*</url-pattern>
20     </servlet-mapping>
21     <session-config>
22         <session-timeout>
23             30
24         </session-timeout>
25     </session-config>
26 </web-app>
27
```

Figura 3.2: Arquivo Web.xml

Ao tentar rodar o exemplo pela primeira vez, possivelmente o Eclipse mostrará uma caixa de diálogo para escolher qual servidor utilizar. Lembre-se que você já deve ter o Tomcat instalado e configurado no Eclipse para que ele apareça como uma das opções. Bem, após clicar no botão de executar, caso não tenha aparecido nenhuma exception, podemos testar acessando http://localhost:8080/jersey-tutorial/helloworld na barra de endereços do navegador. Altere a URL caso você tenha dado um nome diferente para a aplicação, ou tenha configurado o Tomcat em outra porta. Se tudo estiver certo, você deve ver a mensagem "Olá mundo!". Nada muito complicado, mas já é o suficiente para saber que está funcionando.

3.1.3 Cliente de serviço Hello World

O cliente Hello World em Java pode ser visualizado abaixo:

```
/**
 *
 * @author Ricardo
 */
public class ClienteHelloWorld {

    public static void main(String[] args) {
        try {
            Client client = Client.create();
            //URL do WebService
            WebResource wr = client.resource
                    ("http://localhost:8080/tutorial-jersey/rest/helloworld");

            ClientResponse response =
                    wr.accept("text/xml").get(ClientResponse.class);


            if (response.getStatus() != 200) {
                throw new RuntimeException("Failed : HTTP error code : "
                        + response.getStatus());
            }

            String output = response.getEntity(String.class);

            System.out.println("Output from Server .... \n");
            System.out.println(output);
        } catch (Exception e) {

            e.printStackTrace();

        }
    }
}
```

Figura 3.3: Cliente Hello World em Java

O Cliente pode pode ser visualizado também por meio do browser:

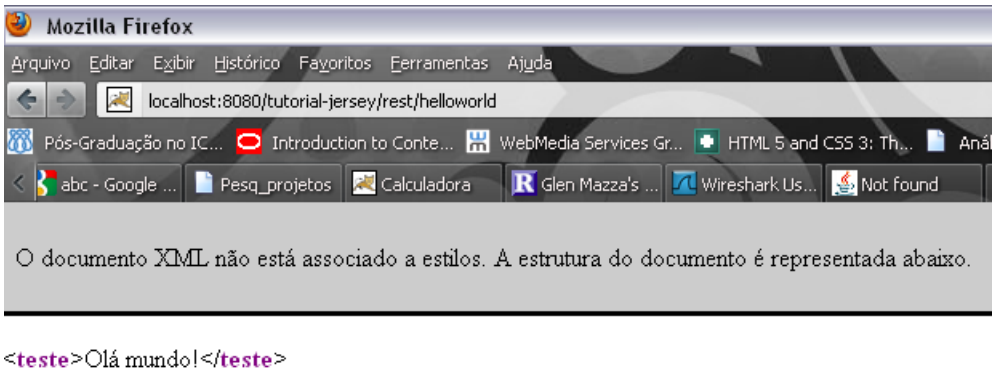


Figura 3.4: Visualização pelo Browser

3.1.4 Provedor de serviço Calculadora

Crie um projeto web com o nome CalculadoraREST com o pacote br.icmc.usp.webservices.rest, dentro do pacote vamos criar a nossa classe que será responsavel pelo nosso serviço web RESTful da calculadora, mas antes de iniciarmos o desenvolvimento do provedor da calculadora precisamos configurar o nosso arquivo web.xml da seguinte maneira como mostrado na Figura xxx.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
         http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
         version="3.0">
    <display-name>CalculadoraREST</display-name>
    <servlet>
        <servlet-name>WebServiceCalculadoraREST</servlet-name>
        <servlet-class>
            com.sun.jersey.spi.container.servlet.ServletContainer
        </servlet-class>
        <init-param>
            <param-name>com.sun.jersey.config.property.packages</param-name>
            <param-value>br.icmc.usp.webservices.rest</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>WebServiceCalculadoraREST</servlet-name>
        <url-pattern>/rest/*</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
</web-app>
```

Figura 3.5: Visualização pelo Browser

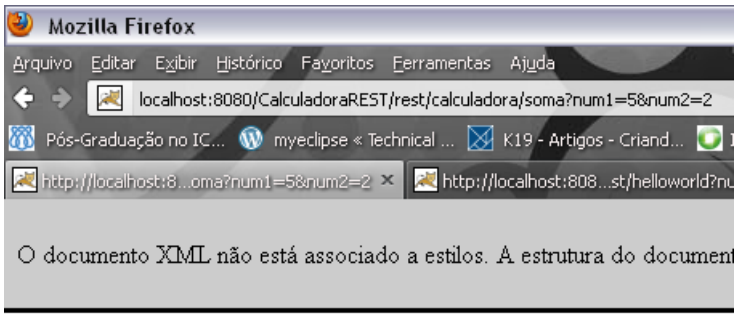
Com o arquivo web.xml configurado podemos começar a desenvolver o nosso serviço web para a nossa Calculadora. Primeiramente vamos onstruir a operação de soma de acordo com a Figura xxx.

Agora precisamos testar o nosso serviço para verificar se está funcionando corretamente.

```
package br.icmc.usp.webservices.rest;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.MediaType;

@Path("/calculadora")
public class ServicoCalculadora {

    @GET
    @Path("/soma")
    @Produces(MediaType.APPLICATION_XML)
    public String soma(@QueryParam("num1") Integer num1,
            @QueryParam("num2") Integer num2) {
        return "<?xml version=\"1.0\" encoding=\"UTF-8\"?>"
                + "<resultado>" + (num1 + num2) + "</resultado>";
    }

}
```

Figura 3.6: Visualização pelo Browser

Acesse pelo seu browser ou navegador o seguinte endereço ou URL: `http://localhost:8080/CalculadoraREST/rest/calculadora/soma?num1=5&num2=2`



<resultado>7</resultado>

Figura 3.7: Visualização pelo Browser

Implemente as outras três operações da calculadora conforme mostrado na Figura xxx.

```
@GET
@Path("/subtracao")
@Produces(MediaType.APPLICATION_XML)
public String subtracao(@QueryParam("num1") Integer num1,
        @QueryParam("num2") Integer num2) {
    return "<?xml version=\"1.0\" encoding=\"UTF-8\"?>"
            + "<resultado>" + (num1 - num2) + "</resultado>";
}

@GET
@Path("/multiplicacao")
@Produces(MediaType.APPLICATION_XML)
public String multiplicacao(@QueryParam("num1") Integer num1,
        @QueryParam("num2") Integer num2) {
    return "<?xml version=\"1.0\" encoding=\"UTF-8\"?>"
            + "<resultado>" + (num1 * num2) + "</resultado>";
}
@GET
@Path("/divisao")
@Produces(MediaType.APPLICATION_XML)
public String divisao(@QueryParam("num1") Integer num1,
        @QueryParam("num2") Integer num2) {

    if (num2 != 0) {
        return "<?xml version=\"1.0\" encoding=\"UTF-8\"?>"
                + "<resultado>" + (num1 / num2) + "</resultado>";
    }
    return "<?xml version=\"1.0\" encoding=\"UTF-8\"?>"
            + "<resultado>" + 0.0 + "</resultado>";
}
```

Figura 3.8: Visualização pelo Browser

Exercicio

Faça os seguintes teste no seu browser ou navegador para a nossa calculadora:

1)72789+5678

2)723567 - 768

3)86013 * 7234

4)90/2

5)700/0

3.1.5 Cliente de serviço Calculadora

3.1.6 Provedor de serviço Biblioteca

Agora vamos fazer um exemplo bem simples de CRUD. Primeiro vamos precisar de uma entidade, um bean para conter as informações que nós queremos manipular. Veja o exemplo abaixo:

```
package br.com.k19.models;

import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class Livro {

    private int id;
    private String titulo;
    private String autor;

    //getters e setters

}
```

Figura 3.9: Classe Livro

Note a presença da anotação **@XmlRootElement** na definição da nossa classe. Ela é fundamental para que o Jersey conheça a representação do objeto livro em XML. Desse modo os atributos irão se tornar nós no momento em que o XML for enviado ao cliente. E do mesmo jeito, quando chegar um XML no corpo de uma requisição, o Jersey vai saber converter esse XML em um objeto do tipo Livro.

O próximo passo é criar o nosso WebResource. No mundo real, geralmente você vai ter uma tabela "Livro" em alguma base de dados e vai desejar buscar a informação quando for requisitado. Aqui para que o exemplo fique mais simples e a gente mantenha o foco apenas na funcionalidade do Jersey, vamos utilizar um Map para simular a nossa base de dados. Veja o exemplo abaixo:

Para testar, basta acessar a `http://localhost:8080/jersey-tutorial/livros` em qualquer navegador, que o resultado será um XML parecido com esse:

E caso queiramos listar apenas um livro em vez de todos? Também podemos incluir a anotação @Path nos métodos igual ao exemplo abaixo:

Nesse caso, a url para acessar o recurso passa a ser http://localhost:8080/jersey-tutorial/livros/id. As chaves indicam que id é um parâmetro, assim podemos passar diferentes valores para acessar diferentes recursos. Note também o uso da anotação @PathParam no parâmetro id, que serve para indicar que o parâmetro do método se refere ao parâmetro no path. Como teste, podemos acessar http://localhost:8080/jersey-tutorial/livros/1 e ver algo parecido com o XML abaixo:

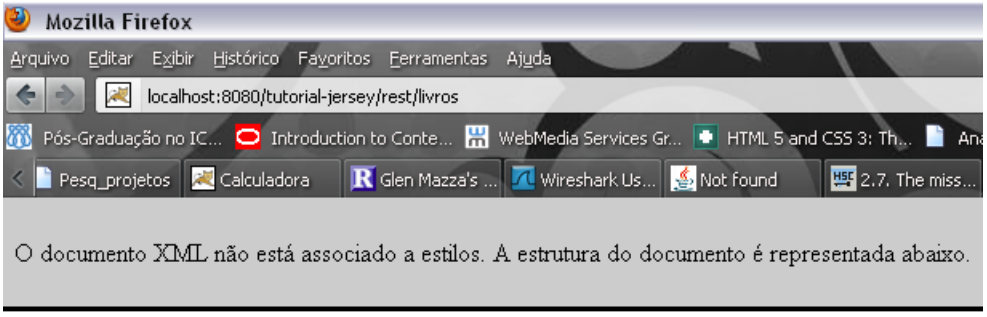
```
@Path("/livros")
public class BibliotecaResource {

    // vamos utilizar um Map estático para
    // "simular" uma base de dados
    static private Map<Integer, Livro> livrosMap;

    static {
        livrosMap = new HashMap<Integer, Livro>();

        Livro livro1 = new Livro();
        livro1.setId(1);
        livro1.setTitulo("Livro de Java");
        livro1.setAutor("Deitel");

        livrosMap.put(livro1.getId(), livro1);

        Livro livro2 = new Livro();
        livro2.setId(2);
        livro2.setTitulo("Livro de C");
        livro2.setAutor("Dennis Ritch");

        livrosMap.put(livro2.getId(), livro2);
    }

    @GET
    @Produces("text/xml")
    public List<Livro> getLivros() {
        return new ArrayList<Livro>(livrosMap.values());
    }

}
//Continua Depois
```

Figura 3.10: Classe Biblioteca Resources

Agora que conseguimos listar os livros, precisamos também de uma maneira de adicioná-los. Veja o exemplo abaixo:

Temos uma mudança de método do HTTP, de GET passou para POST. Fora essa mudança, a URL é a mesma de quando listávamos todas os livros, ou seja, http://localhost:8080-/jersey-tutorial/livros. E agora também recebemos (ou consumimos) um XML que vai representar o livro. E se você reparar bem, também vê que o nosso método recebe um livro como parâmetro. É isso mesmo que você está pensando, o objeto livro é instanciado pelo Jersey e será preenchido com os valores do XML que vierem no conteúdo da requisição. O problema é que agora não temos como testar usando simplesmente o navegador, porque o padrão é que ele execute as requisições como GET, e não teríamos como enviar o XML contendo as informações do novo livro. Para quem usa Firefox, uma opção é utilizar o add-on Poster[5], que permite customizar a requisição de várias maneiras, alterando headers, adicionando conteúdo, etc. Outra opção para quem usa linux, é usar o comando curl. Veja um exemplo para adicionar um livro:

[5] `https://addons.mozilla.org/en-US/firefox/addon/poster/`




```
- <livroes>
  - <livro>
      <autor>Deitel</autor>
      <id>1</id>
      <titulo>Livro de Java</titulo>
    </livro>
  - <livro>
      <autor>Dennis Ritch</autor>
      <id>2</id>
      <titulo>Livro de C</titulo>
    </livro>
  </livroes>
```

Figura 3.11: Resultado do Arquivo XML

```
@Path("{id}")
@GET
@Produces("text/xml")
public Livro getlivro(@PathParam("id") int id) {
    return livrosMap.get(id);
}
```

Figura 3.12: todo

O parâmetro -X serve para alterar o método HTTP. O -H serve para alterar o cabeçalho da requisição. Nesse caso precisamos alterar para informar que estamos enviando um XML. O parâmetro -d são os dados que serão enviados no conteúdo da requisição. E por fim passamos a URL.

Abaixo temos o restante do exemplo de CRUD, com as operações para atualizar e remover livros. O exemplo com PUT é bem semelhante ao de adicionar uma novo livro, a diferença é que devemos atualizar um livro já existente. Já o exemplo do DELETE é ainda mais simples, pois não precisamos consumir nenhum XML, apenas identificamos o id pela URL e excluímos o livro correspondente.

```
<livro>
  <autor>Deitel</autor>
  <id>1</id>
  <titulo>Livro de Java</titulo>
</livro>
```

Figura 3.13: todo

```
@POST
@Consumes("text/xml")
@Produces("text/plain")
public String adicionalivro(Livro livro) {
    livro.setId(livrosMap.size() + 1);
    livrosMap.put(livro.getId(), livro);
    return livro.getTitulo() + " adicionado.";
}
```

Figura 3.14: todo

3.1.7 Cliente de serviço Biblioteca

Para desenvolver o cliente do serviço web da biblioteca vamos utilizar o *Java Architecture for XML Binding* conhecido por JAX-B[6] a sua implementação de referência (RI) atual é a JAXB RI - 2.2.4. Lembrando que o principal objetivo de uma RI é apoiar o desenvolvimento da especificação para validá-lo. Específicas RI podem ter objetivos adicionais, o JAXB RI é uma implementação de qualidade de produção que é usado diretamente em uma série de produtos da Oracle e outros fornecedores.

Usando o JAXB somos capazes de criar facilmente classes a partir de um schema (XSD[7]). Isso mesmo. Com ela somos capazes de transformar um arquivo XSD em um conjunto de classes de uma forma prática e rápida. Mas no momento vamos nos focar nos processos de Marshall e Unmarshall.

JAX-B Marshall e Unmarshall

Marshall é o processo de transformar objetos Java em XML e o Unmarshall faz o caminho inverso, usa os dados de um XML para popular objetos Java.

Vamos usar o processo de Marshall para transformar objetos Java em XML com o uso de anotações. O primeiro passo é criar um novo projeto Java para a Web com o nome ClienteRESTBiblioteca. Crie os pacotes br.icmc.usp.modelo.xml, br.icmc.usp.modelo.xml.colecoes e br.icmc.usp.webservices.restful.

Como sabemos a estrutura e o formato do arquivo XML retornado pelo serviço REST Biblioteca representado na Figura 3.17.

Precisamos criar uma classe que represente exatamente todos os atributos contidos no arquivo XML, assim devemos criar a classe Livro.java no pacote br.icmc.usp.modelo.xml

[6] http://jaxb.java.net/

[7] http://www.liquid-technologies.com/Tutorials/XmlSchemas/XsdTutorial_01.aspx

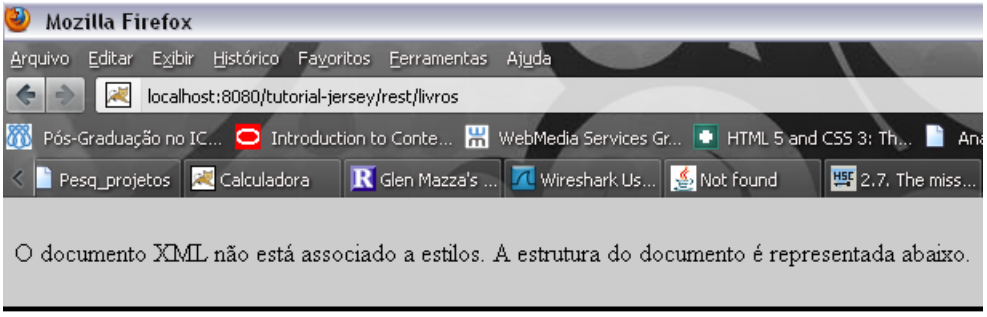
```
$ curl -X POST -H "Content-type:text/xml" \
    -d "<livro><titulo>Teste</titulo><autor>Ricardo</autor></livro>" \

http://localhost:8080/jersey-tutorial/livros
```

Figura 3.15: todo

```
@Path("{id}")
@PUT
@Consumes("text/xml")
@Produces("text/plain")
public String atualizarlivro(Livro livro, @PathParam("id") int id) {
    Livro livroAtual = livrosMap.get(id);
    livroAtual.setTitulo(livro.getTitulo());
    livroAtual.setAutor(livro.getAutor());
    return livro.getTitulo()+ " atualizado.";
}

@Path("{id}")
@DELETE
@Produces("text/plain")
public String removelivro(@PathParam("id") int id) {
    livrosMap.remove(id);
    return "livro removida.";
}
```

Figura 3.16: todo

com os mesmo atributos do arquivo XML do serviço web: id, autor e titulo. Não se esqueça de adicionar a anotação **@XmlRootElement** informando para o JAX-B que essa classe será usando para fazer o Unmarshall do arquivo XML do servidor para objetos java da minha aplicação cliente.




```
- <livroes>
  - <livro>
      <autor>Deitel</autor>
      <id>1</id>
      <titulo>Livro de Java</titulo>
    </livro>
  - <livro>
      <autor>Dennis Ritch</autor>
      <id>2</id>
      <titulo>Livro de C</titulo>
    </livro>
  </livroes>
```

Figura 3.17: Resultado do Arquivo XML

Após criar o bean que representa o recurso da entidade Livro vamos criar uma classe no pacote br.icmc.usp.webservices.restful chamada Constantes com o construtor privado, pois não faz sentido existir uma instância Constantes na minha aplicação. Alguns alunos podem pensar que estou utilizando o padrão de projeto singleton[8] por causa do contrutor privado, mas a classe Constantes não é um Singleton.

Em seguida é nescessário criar uma classe que contenha a minha lista de livros, para isso vamos criar a classe Livros.java no pacote br.icmc.usp.modelo.xml.colecoes, essa classe será usada pelo Jersey a RI do JAX-RS e por meio do JAX-B para adicionar os livros em uma lista, assim como, recuperar a lista de livros para utiliza-la na aplicação. Repare que NUNCA vamos usar a palavra chave NEW do Java para instanciar as classes Livro.java e Livros.java, pois o próprio JAX-B se encarrega criar e instânciar cada classe por Reflection. Usando Java Reflection você pode invocar os métodos de uma classe em tempo de execução. Isto pode ser usado para detectar os métodos getters e setters de uma determinada classe. Você não pode chamar os métodos getters e setters explicitamente, então você terá que percorrer todos os métodos de uma classe e verificar se cada método é um getter ou setter. O próprio JAX-B além de criar e instânciar, vai também gerenciar as nossas instâncias da classe Livro.java e Livros.java [9]. A Figura 3.20 ilustra o código da classe Livros.java.

[8] `http://c2.com/cgi/wiki?JavaSingleton`
[9] `http://tutorials.jenkov.com/java-reflection/index.html`

```java
package br.icmc.usp.modelo.xml;

import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class Livro {

    private Integer id;
    private String titulo;
    private String autor;

    //getters e setters

}
```

Figura 3.18: Classe Livro

```java
package br.icmc.usp.webservices.restful;

public class Constantes {

    private Constantes(){

    }

    public static final String PACOTE_MODELO_XML =
            "br.icmc.usp.modelo.xml:br.icmc.usp.modelo.xml.colecoes";

    public static final String RESTFUL_URL_LIVROS =
            "http://localhost:8080/tutorial-jersey/rest/livros";

}
```

Figura 3.19: Classe Livro

A anotação @XmlElement(name=”livro”) poderia ser usada diretamente no atributo da lista de livros, mas detalhes e informações sobre assunto acesse o endereço: `http://blog.caelum.com.br/jpa-anotacoes-nos-getters-ou-atributos/`.

Agora terminamos de desenvolver todas as nossas entidades agora podemos começar a desenvolver o nosso cliente RESTful para a nossa bblioteca da seguinte forma:

```java
package br.icmc.usp.modelo.xml.colecoes;

import br.icmc.usp.modelo.xml.Livro;
import java.util.ArrayList;
import java.util.List;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement(name="livroes")
public class Livros {

    private List<Livro> livros = new ArrayList<Livro>();

    @XmlElement(name="livro")
    public List<Livro> getLivros() {
        return this.livros;
    }

    public void setLivros(List<Livro> livros){
        this.livros = livros;
    }

}
```

Figura 3.20: Classe Livro

Repare que o construtor da nossa aplicação cliente da biblioteca possue um JAXBContext para criar o XML. Iremos criar um objeto context passando o nome do nosso pacote ao JAXBContext. Esse context nos fornecerá um objeto Marshaller, e é o Marshaller que tem a capacidade de transformar objetos Java em XML. Na terceira linha do construtor usamos o mesmo context para criar um Unmarshaller. Depois o Unmarshaller será usado para converter os arquivos XML do servidor em objetos Java. Lembrando que a nossa aplicação utiliza uma String para armazenar os arquivos XML que são retornado pelo serviço web da biblioteca.

A classe Client é o ponto principal de configuração para a construção de um cliente de serviço web RESTful. Você pode utiliza-la para configurar as propriedades do cliente e vários recursos e indicar quais os prestadores de recurso você deseja usar. Criar uma instância de um cliente é uma operação cara e muito custosa, de modo a tentar evitar a criação de um número desnecessário de instâncias de cliente. Uma boa abordagem é reutilizar uma instância existente, quando possível.

Depois de criar uma instância do cliente, você pode começar a usá-lo. No entanto, para efetuar as requisições propriamente dita, você precisa criar um objeto WebResource, que encapsula um recurso web para o cliente. O código da Figura 3.21 cria um objeto WebResource para um recurso web cujo URI é RESTFUL_URL_LIVROS ou `http://localhost:8080/BibliotecaREST/rest/livros`.

Você pode também usar o objeto WebResource para construir solicitações para enviar e processar a resposta a partir do recurso web. Por exemplo, você pode usar o objeto WebResource para requisições HTTP GET, PUT, POST e DELETE.

Use o método get() na classe WebResource para enviar uma solicitação HTTP GET para o recurso de web, na Figura 3.21 o método get() é executado e o arquivo XML retornado pelo servidor é convertido em uma String de resposta.

O método unmarshal() do JAXB recebe por parametro um Reader[10]. Ele não aceita uma String diretamente e como nós sabemos que queremos ler o XML de uma String, nós usamos a implementação StringReader (*implements Reader*), que recebe a String no construtor. Repare o "new" antes do StringReader: (Livros) unmarshaller.unmarshal(new StringReader(response));

Passamos uma instância de StringReader para o método unmarshal do JAXB. Essa instância encapsula a nossa String response, que contem o nosso XML. Dessa forma, o unmarshal consegue ler a string e converter ela no nosso tipo Livros.

Um ponto importante é a necessidade de um arquivo com o nome jaxb.index, contendo o nome das classes que queremos transformar em XML, e deve estar no mesmo pacote das classes como mostrado na Figura 3.23. Ele é requerido pelo JAXB para gerar as classes em tempo de execução. Essa não é a única forma de fazer isso, mas é a que atende ao nosso caso específico.

[10] `http://download.oracle.com/javase/6/docs/api/java/io/Reader.html`

Agora precisamos testar o nosso cliente RESTful da biblioteca para isso é importante que o serviço da biblioteca RESTful esteja implantado no servidor para o cliente da nossa biblioteca possa consumir o serviço. Em seguida crie a classe ClienteTeste no pacote br.icmc.usp.webservices.teste.

Execute o teste, após executar o teste recebemos uma linda exceção do JAX-B como mostrado na Figura 3.24, mas essa exceção é muito simples de resolver.

Esta exceção é causada pela falta de um outro arquivo jaxb.index no pacote br.icmc.usp.-modelo.xml.colecoes, justamente para o JAX-B poder indexar a nossa classe Livros.java como mostrado na Figura 3.25.

Em seguida após executar o seu projeto a saída pelo terminal será da seguinte forma como mostrado na Figura 3.26.

Agora vamos continuar desenvolvendo a nossa classe do cliente RESTful da biblioteca, crie o método getLivro como ilustrado na Figura xxx.

Repare que a primeira linha do método getLivro() constroi a URL do recurso RESTful e na segunda linha é feita uma chamada ao método path() que recebe uma id, esse método insere o caracter "/" e depois da barra insere o id para que a URL do recurso fique da seguinte forma, por exemplo:

`http://localhost:8080/tutorial-jersey/rest/livros/1`

Exercicio

Desenvolver a classe de Teste para o método getLivro().

Apos construi o método getLivro(), vamos desenvolver agora o métodos para inserir um livro como mostrado na Figura 3.32.

Observe que o método insereLivro() utiliza o método converteLivroParaXML() que recebe um objeto livro em Java e converte em uma String com o arquivo XML, ou seja, é nescessário fazer o Marshal do objeto Java para o arquivo XML.

Além disso, podemos especificar o tipo de MIME aceitáveis para a requisição. Por exemplo, no código do método para inserir especificamos o tipo MIME aceitável para XML.

O método converteLivroParaXML() pode ser visualizado na Figura xxx a seguir. Além disso, o método converteLivroParaXML() instancia um objeto StringWriter que será responsável por armazenar o XML gerado. Após a execução deste método, o objeto do tipo StringWriter contém o XML gerado, bastando somente então converte-lo para String chamando o método toString().

Podemos fazer a analogia com o plugin Poster[11] do firefox como mostrado na Figura xxx a seguir.

Após clicar o botão POST do plugin o método POST do HTTP será executado e a resposta da requisição HTTP será da seguite forma como ilustrado na Figura xxx. O método da Figura 3.32 faz exatamente o que o plugin Poster do Firefox faz, mas utilizando o código Java da minha aplicação.

A Figura xxx mostra os método para editar e remover um livro por meio dos métodos HTTP put e delete respectivamente.

[11] `https://addons.mozilla.org/en-US/firefox/addon/poster/`

Para finalizar o nosso cliente e completar todos os método da nossa classe ClienteRESTBiblioteca.java vamos construir o método close() para fechar o nosso cliente, lembrando que a criação do cliente é uma operção muito cara e custosa que exige e gasta recurso. Por isso devemos fechar o cliente para liberar memória na máquina do cliente. O método close() é bem simples e consiste na chamada ao método destroy() da classe Client. O método close pode ser visualizado na Figura 3.33.

Exercicio

Faça os testes para os método getLivro(), inserir(), editar() e remover() um livro da classe ClienteRESTBiblioteca incluindo o método close().

3.1.8 Exercicios

Todos os serviços Web devem ser implementados usando um banco de dados de sua preferência.

Exercício Fácil

1 - Vamos imaginar o seguinte cenário: um jornal quer disponibilizar um serviço para distribuir notícias pela internet. Sendo assim, as pessoas podem colocar informações confiáveis no seu site, e toda vez que o pessoal da redação adicionar uma nova notícia, ela aparece em todos os sites que consomem o serviço. Com base nesse cenário, crie um serviço web para listar noticias. Obs: A classe Notícia é composta por dois atributos título e texto da notícia.

Exercício Desafio

2 - Implemente um web service RESTful que ofereça o serviço de CRUD (Create, Read, Update e Delete) para cada uma das tabelas do banco de dados abaixo.

Exercício Extra

3 - Crie um web service RESTful para listar os post do Twitter de um usuário na sua aplicação Web.

4 - Crie um web service RESTful para listar os favoritos do Delicious na sua aplicação Web.

```
package br.icmc.usp.webservices.restful;

import br.icmc.usp.modelo.xml.Livro;
import br.icmc.usp.modelo.xml.colecoes.Livros;
import static br.icmc.usp.webservices.restful.Constantes.*;

import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.WebResource;
import com.sun.jersey.api.client.config.ClientConfig;
import com.sun.jersey.api.client.config.DefaultClientConfig;
import java.io.StringReader;
import java.io.StringWriter;
import java.io.Writer;
import java.net.MalformedURLException;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;
import javax.xml.bind.Unmarshaller;

public class ClienteRESTBiblioteca {

    private final Unmarshaller unmarshaller;
    private final Marshaller marshaller;
    private final Client client;
    public ClienteRESTBiblioteca() throws JAXBException {
        JAXBContext jaxbContext = JAXBContext.newInstance(PACOTE_MODELO_XML);
        this.marshaller = jaxbContext.createMarshaller();
        this.unmarshaller = jaxbContext.createUnmarshaller();
        ClientConfig config = new DefaultClientConfig();
        this.client = Client.create(config);
    }

    public Livros getTodosLivros() throws JAXBException, MalformedURLException {
        WebResource webResource = client.resource(RESTFUL_URL_LIVROS);
        String response = webResource.get(String.class);
        return (Livros) unmarshaller.unmarshal(new StringReader(response));
    }

}
```

Figura 3.21: Cliente REST Biblioteca

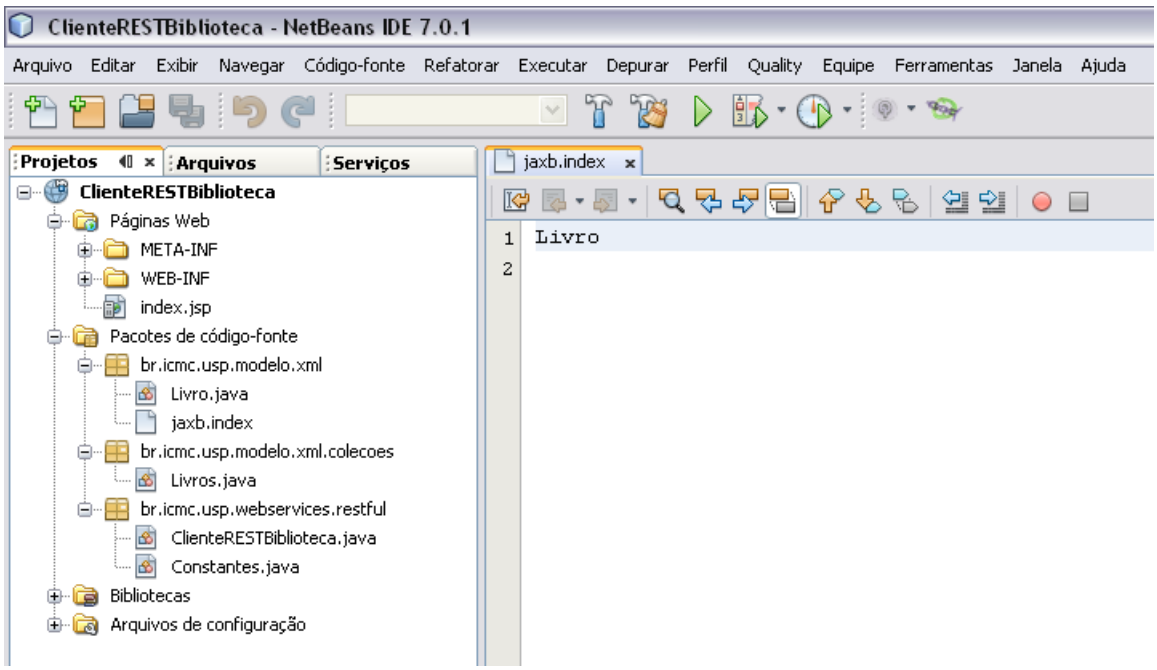


Figura 3.22: Conteúdo do Arquivo jaxb.index

```
package br.icmc.usp.webservices.teste;

import br.icmc.usp.modelo.xml.Livro;
import br.icmc.usp.modelo.xml.colecoes.Livros;
import br.icmc.usp.webservices.restful.ClienteRESTBiblioteca;
import java.net.MalformedURLException;
import javax.xml.bind.JAXBException;

public class ClienteTeste {

    public static void main(String[] args){
        try {
            ClienteRESTBiblioteca cliente = new ClienteRESTBiblioteca();
            Livros livros = cliente.getTodosLivros();
            for(Livro listaLivros: livros.getLivros()){
                System.out.println("#-------------------------------------#");
                System.out.println(listaLivros.getId());
                System.out.println(listaLivros.getAutor());
                System.out.println(listaLivros.getTitulo());
                System.out.println("#-------------------------------------#");
            }
        }
        catch (JAXBException ex) {
            ex.printStackTrace();
        }
        catch(MalformedURLException ex){
            ex.printStackTrace();
        }
    }
}
```

Figura 3.23: Classe Cliente RESTful biblioteca para teste

```
run:
javax.xml.bind.JAXBException: Provider com.sun.xml.internal.bind.v2.ContextFactory could not be instantiated: javax.xml.binc
 - with linked exception:
[javax.xml.bind.JAXBException: "br.icmc.usp.modelo.xml.colecoes" doesnt contain ObjectFactory.class or jaxb.index]
        at javax.xml.bind.ContextFinder.newInstance(ContextFinder.java:146)
        at javax.xml.bind.ContextFinder.find(ContextFinder.java:347)
        at javax.xml.bind.JAXBContext.newInstance(JAXBContext.java:431)
        at javax.xml.bind.JAXBContext.newInstance(JAXBContext.java:394)
        at javax.xml.bind.JAXBContext.newInstance(JAXBContext.java:298)
        at br.icmc.usp.webservices.restful.ClienteRESTBiblioteca.<init>(ClienteRESTBiblioteca.java:27)
        at br.icmc.usp.webservices.teste.ClienteTeste.main(ClienteTeste.java:13)
Caused by: javax.xml.bind.JAXBException: "br.icmc.usp.modelo.xml.colecoes" doesnt contain ObjectFactory.class or jaxb.index
        at com.sun.xml.internal.bind.v2.ContextFactory.createContext(ContextFactory.java:216)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:601)
        at javax.xml.bind.ContextFinder.newInstance(ContextFinder.java:172)
        at javax.xml.bind.ContextFinder.newInstance(ContextFinder.java:132)
        ... 6 more
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

Figura 3.24: Exceção do JAXB JAXBException

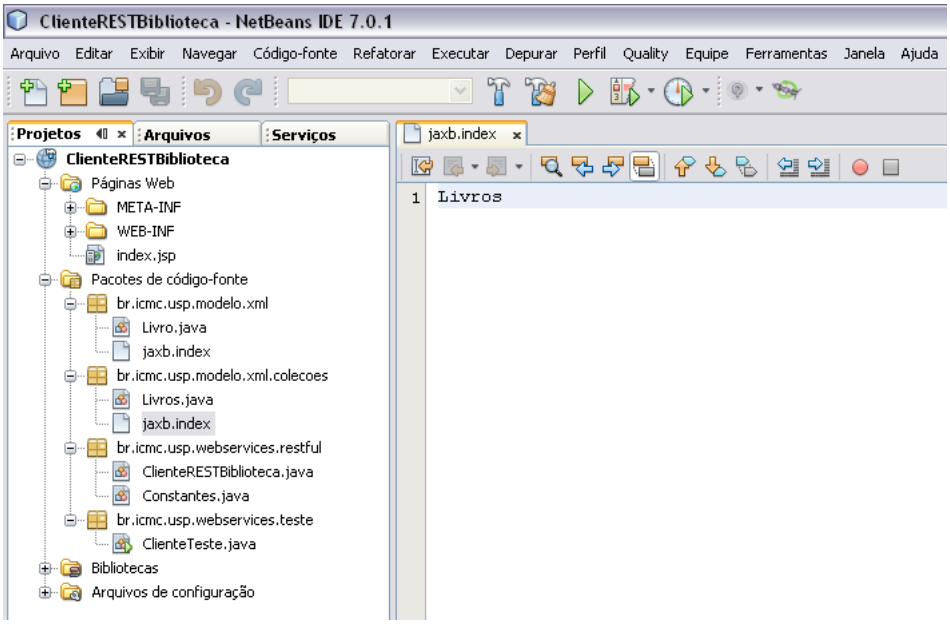


Figura 3.25: Conteúdo do Arquivo jaxb.index

```
run:
#--------------------------------------#
1
Deitel
Livro de Java
#--------------------------------------#
#--------------------------------------#
2
Dennis Ritch
Livro de C
#--------------------------------------#
CONSTRUÍDO COM SUCESSO (tempo total: 1 segundo)
```

Figura 3.26: Saida do Cliente RESTful Biblioteca

```
public Livro getLivro(Integer id) throws JAXBException {
    WebResource webResource = client.resource(RESTFUL_URL_LIVROS);
    webResource.path(id.toString());
    String response = webResource.get(String.class);
    return (Livro) unmarshaller.unmarshal(new StringReader(response));
}
```

Figura 3.27: Cliente RESTful Completo do serviço web Biblioteca

```
public void insereLivro(Livro livro) throws JAXBException {
    String xmlLivro = converteLivroParaXML(livro);
    WebResource webResource = client.resource(RESTFUL_URL_LIVROS);
    webResource.header("Content-Type", "text/xml;charset=utf-8");
    webResource.post(xmlLivro);
}
```

Figura 3.28: Método para Inserir, Editar e Remover um Livro

```
private String converteLivroParaXML(Livro livro) throws JAXBException {
    Writer writer = new StringWriter();
    this.marshaller.marshal(livro, writer);
    return writer.toString();
}
```

Figura 3.29: Método para Inserir, Editar e Remover um Livro

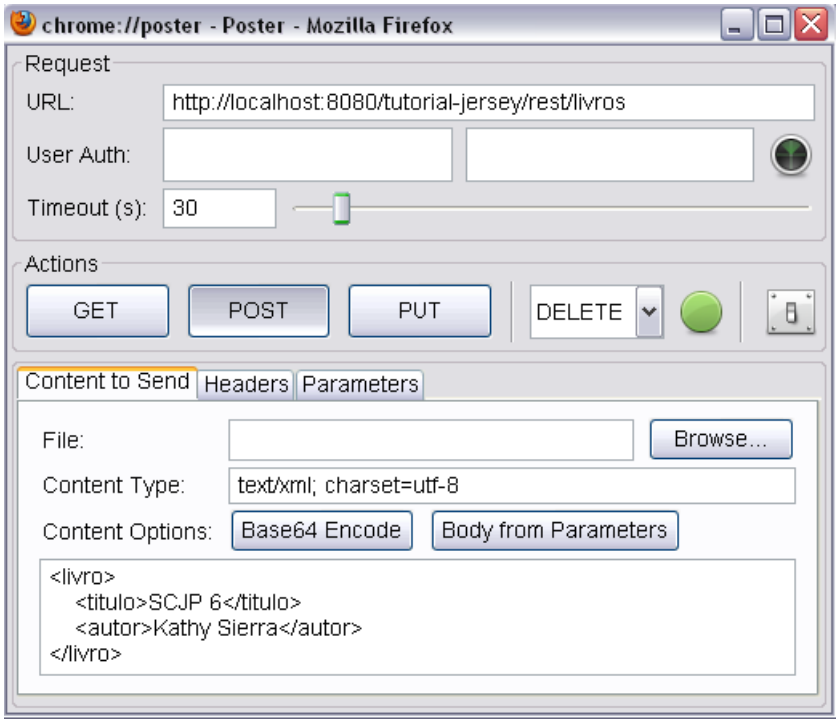


Figura 3.30: Método para Inserir, Editar e Remover um Livro

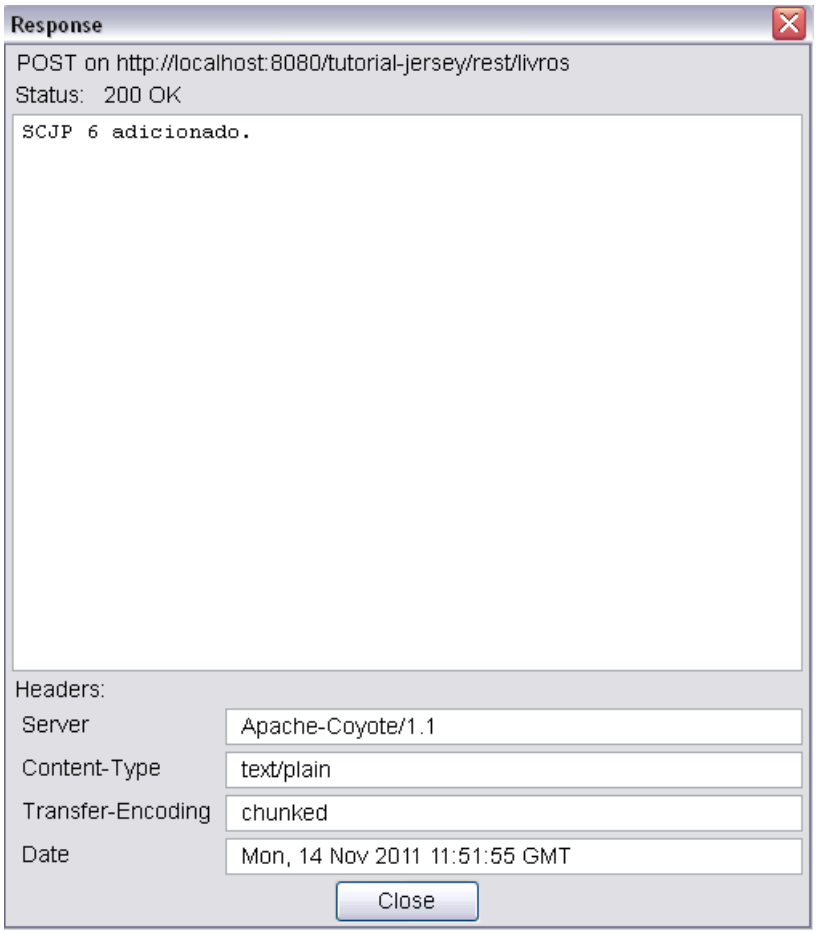


Figura 3.31: Método para Inserir, Editar e Remover um Livro

```
public void editarLivro(Livro livro) throws JAXBException {
    String xmlLivro = converteLivroParaXML(livro);
    WebResource webResource = client.resource(RESTFUL_URL_LIVROS);
    webResource.path(livro.getId().toString());
    webResource.header("Content-Type", "text/xml;charset=utf-8");
    webResource.put(xmlLivro);
}

public void removerLivro(Integer id) throws JAXBException {
    WebResource webResource = client.resource(RESTFUL_URL_LIVROS);
    webResource.path(id.toString());
    webResource.delete();
}
```

Figura 3.32: Método para Inserir, Editar e Remover um Livro

```
public void close() {
    client.destroy();
}
```

Figura 3.33: Método Close

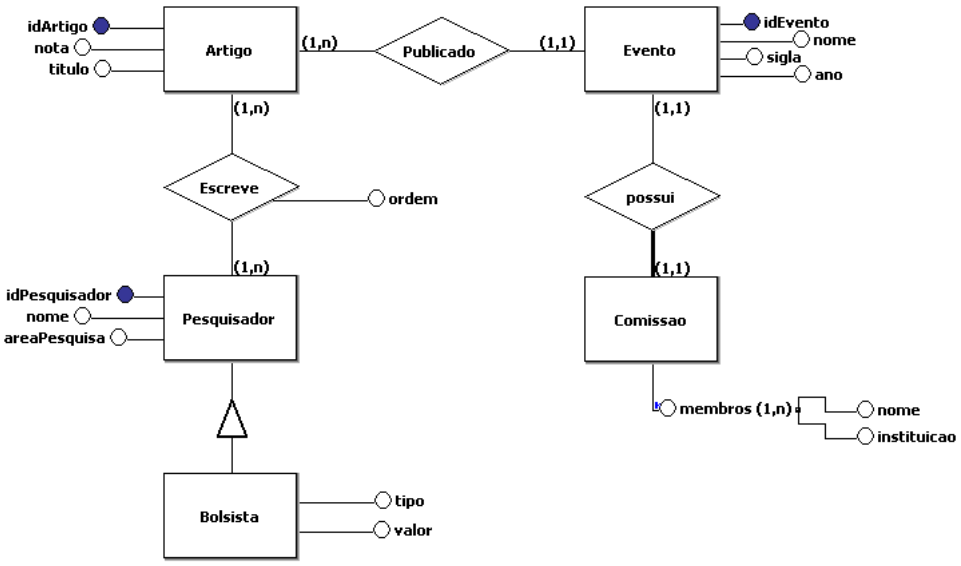


Figura 3.34: Pilha conceitual de web services (IBM, 2009)

Referências

BASCI, D., M. S. Data complexity metrics for xml web services. *Advances in Electrical and Computer Engineering*, v. 9, p. 9 – 15, 2009.

Brooks, C. A. An introduction to web services. 2002.

Campbell, C. E.; Eisenberg, A.; Melton, J. Xml schema. *SIGMOD Rec.*, v. 32, p. 96–101, 2003.
Disponível em `http://doi.acm.org/10.1145/776985.777002`

Erl, T. *Service-oriented architecture - a field guide to integrating xml and web services.* 1st. ed. Prentice Hall, 2004.

Fielding, R. T. *Architectural styles and the design of network-based software architectures.* Tese de Doutoramento, UC Irvine, 2000.

IBM Standards and web services. Último acesso: 01/03/2011, 2009.
Disponível em `http://www-128.ibm.com/developerworks/webservices/standards/`

Jun, Y.; Zhishu, L.; Yanyan, M. Json based decentralized sso security architecture in e-commerce. In: *Proceedings of the 2008 International Symposium on Electronic Commerce and Security*, ISECS '08, Washington, DC, USA: IEEE Computer Society, 2008, p. 471–475 (*ISECS '08*,).
Disponível em `http://dx.doi.org/10.1109/ISECS.2008.171`

Kurose, J. F.; Ross, K. W. *Redes de computadores e a internet: uma nova abordagem. tradução de arlete simille marques.* Addison Wesley, p. 377-440 p., 2003.

Meng, J.; Mei, S.; Yan, Z. Restful web services: A solution for distributed data integration. In: *Computational Intelligence and Software Engineering, 2009. CiSE 2009. International Conference on*, 2009, p. 1 –4.

Pautasso, C.; Zimmermann, O.; Leymann, F. Restful web services vs. "big"' web services: making the right architectural decision. In: *Proceeding of the 17th international conference on World Wide Web*, WWW '08, New York, NY, USA: ACM, 2008, p. 805–814 (*WWW '08*,).
Disponível em `http://doi.acm.org/10.1145/1367497.1367606`

Richardson, L.; Ruby, S. *Restful web services*. First ed. O'Reilly, 2007.

Stal, M. Web services: beyond component-based computing. *Commun. ACM*, v. 45, p. 71–76, 2002.
Disponível em `http://doi.acm.org/10.1145/570907.570934`

W3C Web services architecture. Último acesso: 10/02/2011, 2004.
Disponível em `http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/`

W3C Web services description language (wsdl). Último acesso: 15/02/2011, 2007.
Disponível em `http://www.w3.org/TR/2004/WD-wsdl20-patterns-20040326/`

Wang, H.; Huang, J. Z.; Qu, Y.; Xie, J. Web services: Problems and future directions. *Journal of Web Semantics*, v. 1, n. 3, 2004.
Disponível em `http://www.websemanticsjournal.org/ps/pub/2004-19`

Wei-Chung Hu, Chia Hung Kao, F. P. Y.; Jiau, H. C. Vesta: A view-based software quality assessmentmodel for software evolution management. *Proceedings of the 22nd International Conference on Software Engineering and Knowledge Engineering*, p. p.345–348, 2010.

Apêndice

A

Introdução a JAXB 2

A.1 Introdução a JAXB 2

A JAXB é uma API da plataforma Java EE inclusa na JDK 6, além disso, fornece suporte à manipulação de objetos Java e XML. Sua principal característica é a capacidade de vincular XML a objetos Java e vice-versa.

Conforme mencionado no Capítulo 3, Marshall é o processo de transformar objetos Java em XML e o Unmarshall faz o processo inverso, usa os dados de um XML para objetos Java.

A.2 Marshall: XML para objetos Java

Vamos usar o processo de Marshall para transformar objetos Java em XML com o uso de anotações fornecidas pelo JAXB. O primeiro passo é criar um novo projeto Java com o nome JAXBFuncionario.

Criando os POJOs[1]

Crie a classe principal Funcionario como exibido na Listagem 01. Note que ela possui

[1]O nome é usado para enfatizar que um determinado objeto é um objeto Java comum, não um objeto especial. O termo "POJO" é principalmente usado para denotar um objeto Java que não segue qualquer um dos principais modelos de objetos Java, convenções, ou frameworks. O termo foi cunhado por Martin Fowler, Rebecca Parsons e Josh MacKenzie em Setembro de 2000.

a uma anotação @XmlRootElement. Essa anotação indica que o valor da classe será representado como um elemento XML principal.

```
package jaxb;

import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class Funcionario {

    private String nome;
    private String cpf;
    private int idade;
    private Endereco endereco;

    public Funcionario() {

    }

    public Funcionario(String nome, String cpf, int idade,

            Endereco endereco) {

        this.nome = nome;
        this.cpf = cpf;
        this.idade = idade;
        this.endereco = endereco;

    }

    /* Métodos getters e setters omitidos para não ocupar espaço */
```

Listagem 01 Código do Funcionário.

Devemos agora criar a classe Endereço representada na Listagem 02. A anotação @XmlType indica que essa classe na verdade mapeia um tipo de informação específica (XML schema). Como a classe Funcionario possui o atributo endereco do tipo Endereço, então as informações do endereço estarão dentro do objeto funcionario que será criado.

```
        package jaxb;

        import javax.xml.bind.annotation.XmlType;

        @XmlType
        public class Endereco {

                private String logradouro;
    private int numero;
    private String bairro;
    private String cidade;
    private String cep;
```

```
    public Endereco() {

    }

    public Endereco(String logradouro, int numero, String bairro,

            String cidade, String cep) {

        this.logradouro = logradouro;
        this.numero = numero;
        this.bairro = bairro;
        this.cidade = cidade;
        this.cep = cep;

    }

    /* Métodos getters e setters omitidos para não ocupar espaço */

}
```

Listagem 02 Código do Endereço.

Para testar iremos criar uma classe que chamaremos TesteJAXB como na Listagem 03.

Criamos os objetos funcionario e endereço, e preenchemos todos os seus atributos. Para criar o XML precisamos de um JAXBContext. Iremos criar um objeto context passando o nome do nosso pacote para o JAXBContext. Esse context nos fornecerá um objeto Marshaller, e o Marshaller transformará os objetos Java em XML.

O JAXB nescessita que criemos um arquivo com o nome jaxb.index para indexar todas as classes que queremos transformar em XML, por meio do nome da classe, e deve estar no mesmo pacote das classes (Figura 01). Este arquivo é requerido pelo JAXB para gerar as classes em tempo de execução.

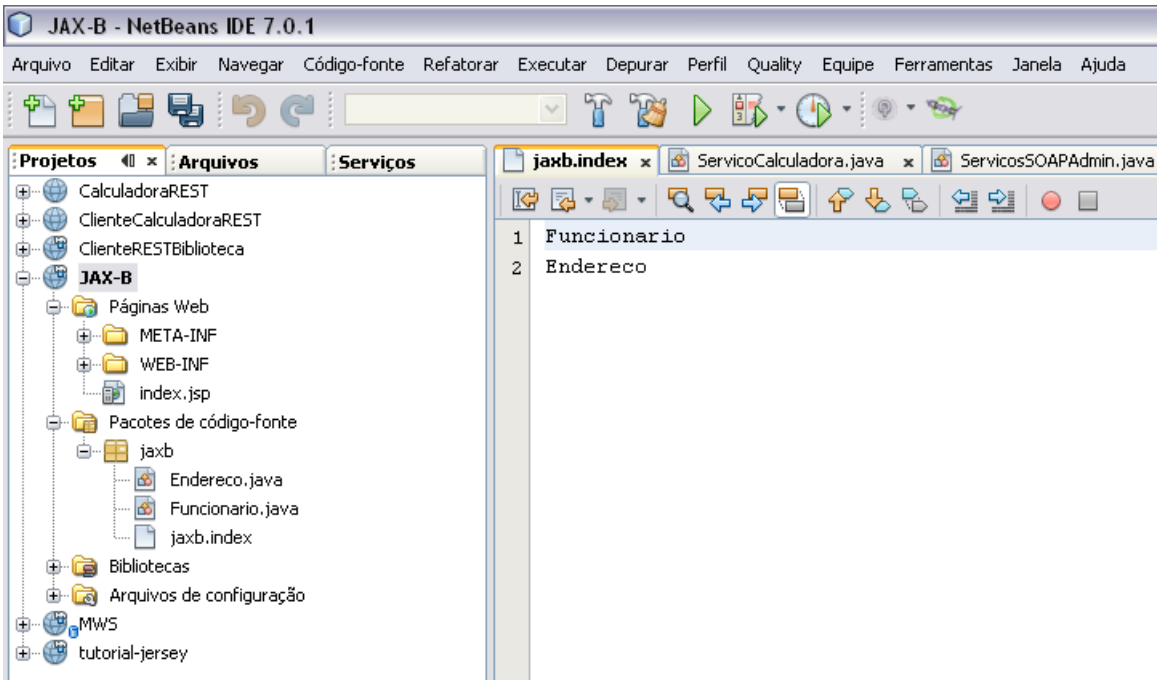


Figura A.1: Arquivo JAXB Index

```
 package jaxb;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;

public class TesteJaxb {

    public static void main(String args[]) {

        try {

            Endereco endereco = new Endereco("Rua Jose Claudino", 7, "Centro",
                    "Sao Paulo", "70000-000");

            Funcionario funcionario = new Funcionario("Ricardo Ramos de Oliveira",
            "777.777.777-77", 20, endereco);

            JAXBContext context = JAXBContext.newInstance("jaxb");

            //saída 1 - console
            Marshaller m = context.createMarshaller();
            m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
            m.marshal(funcionario, System.out);

            //saída 2 - arquivo
            File f = new File("src/funcionario.xml");
```

```
            Marshaller m2 = context.createMarshaller();
            m2.marshal(funcionario, new FileOutputStream(f));

        }catch (JAXBException ex) {
            ex.printStackTrace();
        } catch (FileNotFoundException ex) {
            ex.printStackTrace();
        }

    }
}
```

Listagem 03 Código de teste.

Usamos duas formas de apresentar o XML. A primeira é através do console e a segunda é para arquivo.

A saída na tela e o arquivo gerado devem estar parecidos com a listagem abaixo (Listagem 04). Note que a tag endereço foi criada dentro da tag funcionario. Lembrando que a classe Funcionario possui um atributo do tipo Endereço.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<funcionario>
    <cpf>777.777.777-77</cpf>
    <endereco>
        <bairro>Centro</bairro>
        <cep>70000-000</cep>
        <cidade>Sao Paulo</cidade>
        <logradouro>Rua Jose Claudino</logradouro>
        <numero>7</numero>
    </endereco>
    <idade>20</idade>
    <nome>Ricardo Ramos de Oliveira</nome>
</funcionario>
```

Listagem 04 XML gerado.

A.3 Unmarshall: Objetos Java para XML

Vamos usar o processo de Unmarshall para vincular as informações contidas em um arquivo XML a objetos Java.

Para isso iremos usar o mesmo arquivo criado anteriormente. Vamos copiá-lo e mudar seu nome para funcionario2.xml. Nele iremos alterar algumas informações só para verificar se o processo irá funcionar.

Agora iremos incluir as linhas abaixo (Listagem 05) ao final da nossa classe de teste. Na primeira linha usamos o mesmo context para criar um Unmarshaller. Depois o Unmarshaller cria um objeto usando as informações do arquivo alterado funcionario2.xml. Só para testar iremos mostrar o conteúdo desse objeto no console usando o formato XML, e também através dos getters do novo funcionario criado.

```
Unmarshaller um = context.createUnmarshaller();
Object obj = um.unmarshal(new File("src/funcionario2.xml"));
m.marshal(obj, System.out);
Funcionario funcionario2 = (Funcionario) obj;
System.out.println(funcionario2.getNome() + " " + funcionario2.getCpf());
```

Listagem 05 Fazendo Unmarsheller.

Para Finalizar o JAXB 2 facilita o processo de Marshall e Unmarshall. Consequemente aumentando a produtividade dos programadores.