

Disciplina: Programação Cliente-Servidor

Aula 10: Web Services

Apresentação

Um grande problema nos sistemas antigos era o compartilhamento de dados e serviços, tanto em termos de hardware quanto software.

O hardware evoluiu para um modelo em que os barramentos e conectores são padronizados, possibilitando a utilização de peças de diferentes fornecedores, o que é conhecido em engenharia como COTS. Isso também ocorreu no software, inicialmente com o compartilhamento de dados através de bases apropriadas, e posteriormente com a definição de tipos de serviços padronizados, fazendo com que plataformas distintas possam se comunicar, desde que entendam o vocabulário comum.

Na área de software, os Web Services se tornaram comuns no provimento de serviços independentes de plataforma, ou seja, interoperáveis, sendo divididos em dois tipos: SOAP e RESTful. Para o desenvolvedor Web é fundamental saber trabalhar com Web Services.

Objetivos

- Explicar o conceito de serviços interoperáveis;
- Aplicar Web Services dos tipos SOAP e REST.



(Fonte: TheDigitalArtist / Pixabay)

Serviços interoperáveis

Podemos considerar os Web Services como serviços atuantes no nível da Web que buscam garantir a plena interoperabilidade, pois trata-se de uma tecnologia independente de plataforma e que tramita dados em formato texto, além de atuar de forma transparente aos firewalls.

A busca pela interoperabilidade dos serviços não é algo novo, levando ao aparecimento de diversas tecnologias ao longo do tempo, como CORBA (Common Object Request Broker Architecture) e RPC (Remote Procedure Call); todas essas tecnologias trazem diversos traços em comum.

Sempre ocorre a busca por um **protocolo aberto**, de forma que o mesmo possa ser adotado por ferramentas diversas, como o IIOP (Internet Inter-ORB Protocol) para o CORBA, além de um meio de **descrever os serviços**, como o IDL (Interface Definition Language) para o RPC.

1. 1

O descritor de serviços é normalmente fornecido no formato texto, e as plataformas que suportam a tecnologia associada a ele costumam ser capazes de gerar o módulo de comunicação cliente (**stub**) e o módulo servidor (**skeleton**) de forma automatizada.

2. 2

Sabendo qual o protocolo e o formato de utilização de cada serviço, incluindo parâmetros e tipos de retorno, podemos utilizar as mais diversas ferramentas, como C++, Delphi, C# ou Java, para efetuar chamadas a estes serviços.

3. 3

Além de um protocolo conhecido e aberto e de um descritor de serviços, precisamos também de um registro que nos permita **localizar o serviço** ao nível da rede, como o COS Naming do CORBA.

4. 4

Com isso nós definimos os três componentes fundamentais de uma arquitetura de serviços interoperáveis: serviço de localização, protocolo aberto e descritor de serviços.

Todos estes elementos são comuns em um ambiente de processamento distribuído baseado em serviços, e podemos observar alguns deles na figura seguinte.

XML-RPC

Quando utilizamos tecnologias como CORBA ou RPC, conseguimos uma comunicação simples, mas ainda estamos presos a formatos de transferência de dados binários, o que pode trazer dificuldades em termos de leitura e transmissão destes dados.

Uma primeira tentativa de modificação para o formato texto na transferência de dados, e que teve impacto comercial relevante, foi a tecnologia **XML-RPC**, a qual trata do mesmo modelo já adotado pelo RPC, mas com a utilização de pacotes de dados em formato XML.

Características do XML-RPC:

1

Busca a simplicidade em termos de comunicação, definindo as interfaces de chamadas remotas, mas sem a implementação de métodos ouvintes nos servidores.

2

Utiliza uma gramática baseada em XML.

3

Utiliza poucos comandos (tags), descrevendo funções e tipos de parâmetros e retorno.

4

Utiliza o protocolo HTTP e é voltado para comunicação entre empresas.

5

É transparente para os firewalls, pelo uso de XML, existindo várias implementações disponíveis no mercado.

Podemos encontrar um exemplo muito simples de chamada e resposta XML-RPC no endereço <https://en.wikipedia.org/wiki/XML-RPC> <https://en.wikipedia.org/wiki/XML-RPC>, o qual é replicado a seguir.

• • •

```
<?xml version="1.0"?>

<methodCall>

<methodName>examples.getStateName</methodName>

<params>

<param>

<value><i4>40</i4></value>

</param>

</params>

</methodCall>

<?xml version="1.0"?>

<methodResponse>

<params>

<param>

<value><string>South Dakota</string></value>

</param>

</params>

</methodResponse>
```

Na primeira parte deste exemplo temos uma chamada ao método examples.getStateName, passando como parâmetro o valor inteiro 40; já na segunda parte temos a resposta no formato texto com o estado correspondente (South Dakota).

Entre os elementos da sintaxe XML-RPC, encontram-se:

1

methodCall

Chamada de um método remoto a partir do cliente.

2

methodResponse

Define a resposta enviada pelo servidor ao cliente.

3

methodName

Nome do método chamado no servidor.

4

params

Setor de parâmetros da chamada ou resposta.

5

param

Define um parâmetro, devendo conter o valor e tipo do mesmo.

O modelo de comunicação com uso de XML foi muito bem aceito, mesmo com poucos tipos nativos definidos, pois garantiu a interoperabilidade de uma forma simples, o que justifica o fato de existirem tantas implementações disponíveis, tanto para cliente quanto para servidor, nas mais diversas plataformas de desenvolvimento.

Podemos observar os tipos nativos do XML-RPC na tabela seguinte.

Tag	Significado	Exemplo
<i4> ou <int>	Inteiro sinalizado de 32 bits	-12
<boolean>	Lógicos: falso vale 0 e verdadeiro vale 1.	1
<string>	Texto	hello world
<double>	Ponto flutuante de precisão dupla	-12.214
<dateTime.iso8601>	Data e hora	19980717T14:08:55
<base64>	Binário codificado em Base 64	eW91IGNhbid0IHJlYWQgdGhpcyE=

Atenção

Além desses tipos nativos, podemos trabalhar com estruturas compostas através de **struct**, onde cada elemento da estrutura deve conter nome e valor, onde o valor pode ser outro elemento struct, aplicado de forma recursiva.

Também podem ser criadas coleções com o uso de **array**, podendo conter elementos do tipo struct, os quais podem também utilizar array em seus valores.

Logo, com a combinação de estruturas complexas e coleções (ou vetores), de forma dinâmica e recursiva, podemos expressar tipos de dados diversos com grande facilidade, o que viabiliza a implementação de clientes e servidores nas mais diversas plataformas.



SOAP

O conceito de serviços não é novidade em termos de programação, tratando de componentes independentes de software que podem ser acionados para realizar determinadas tarefas, permitindo através de seu conjunto realizar ações maiores segundo uma ordem de chamada definida pelo aplicativo solicitante; em termos de Internet, os tipos de serviço mais difundidos são **Web Services**.

Os primeiros Web Services tiveram sua funcionalidade baseada no modelo XML-RPC, adotando o formato XML na transferência de dados. Eles foram considerados um marco na evolução das plataformas de processamento distribuído interoperáveis e formaram a base da comunicação **B2B** (Business to Business), servindo também como meio padrão de interfaceamento das arquiteturas orientadas a serviço.

Em termos gerais, o SOAP (Simple Object Access Protocol) define uma comunicação remota estilo RPC com uso de XML, extensível e amplamente utilizada em Web Services.

As características principais do SOAP são:

1

Voltado para a comunicação B2B.

2

Padroniza o formato para envio e recepção de mensagens.

3

Comunicação transparente aos firewalls.

4

Neutralidade de plataforma e ambiente de desenvolvimento.

5

Baseado em sintaxe XML, trazendo simplicidade e expansibilidade.

6

É uma recomendação da W3C desde 2003.

A sintaxe do SOAP é extremamente formal e segue algumas regras importantes, como a obrigatoriedade dos namespaces soap-envelope e soap-encoding, e não permite o uso de DTD nem de instruções de processamento XML.

Podemos observar o esqueleto de uma mensagem SOAP a seguir.

```
...
<?xml version="1.0"?>

<soap:Envelope

xmlns:soap="//www.w3.org/2001/12/soap-envelope"

soap:encodingStyle="//www.w3.org/2001/12/soap-encoding">

<soap:Header> ... </soap:Header>


<soap:Body>

...

<soap:Fault> ... </soap:Fault>

</soap:Body>

</soap:Envelope>
```

 Clique nos botões para ver as informações.

Seção Header



A seção **Header** é opcional e permite a inclusão de informações específicas do aplicativo, como autenticação e pagamento, por exemplo. Se esta seção estiver presente deverá constar como o primeiro elemento filho do envelope SOAP.

Seção Fault



A seção **Fault** também é opcional e serve para indicar mensagens de erro. Quando presente na mensagem, Fault deve se apresentar como um elemento filho de Body, e permite apenas uma ocorrência nessa mensagem.

Body



O elemento mais importante é o **Body**, pois apresenta a mensagem em si, definindo chamada ou resposta de um serviço solicitado, e seus filhos devem ter o namespace qualificado, como podemos observar no exemplo seguinte com a chamada e a resposta SOAP.

Este exemplo encontra-se disponível em [//w3schools.sinsixx.com/ soap/soap_example.asp.htm](http://w3schools.sinsixx.com/soap/soap_example.asp.htm)
[<w3schools.sinsixx.com/soap/soap_example.asp.htm>](http://w3schools.sinsixx.com/soap/soap_example.asp.htm)

• • •

```
<?xml version="1.0"?>

<soap:Envelope

xmlns:soap="//www.w3.org/2001/12/soap-envelope"

soap:encodingStyle="//www.w3.org/2001/12/soap-encoding">

<soap:Body xmlns:m="//www.example.org/stock">

<m:GetStockPrice>

<m:StockName>IBM</m:StockName>

</m:GetStockPrice>

</soap:Body>

</soap:Envelope>
```

```
<?xml version="1.0"?>

<soap:Envelope

xmlns:soap="//www.w3.org/2001/12/soap-envelope"

soap:encodingStyle="//www.w3.org/2001/12/soap-encoding">

<soap:Body xmlns:m="//www.example.org/stock">

<m:GetStockPriceResponse>

<m:Price>34.5</m:Price>

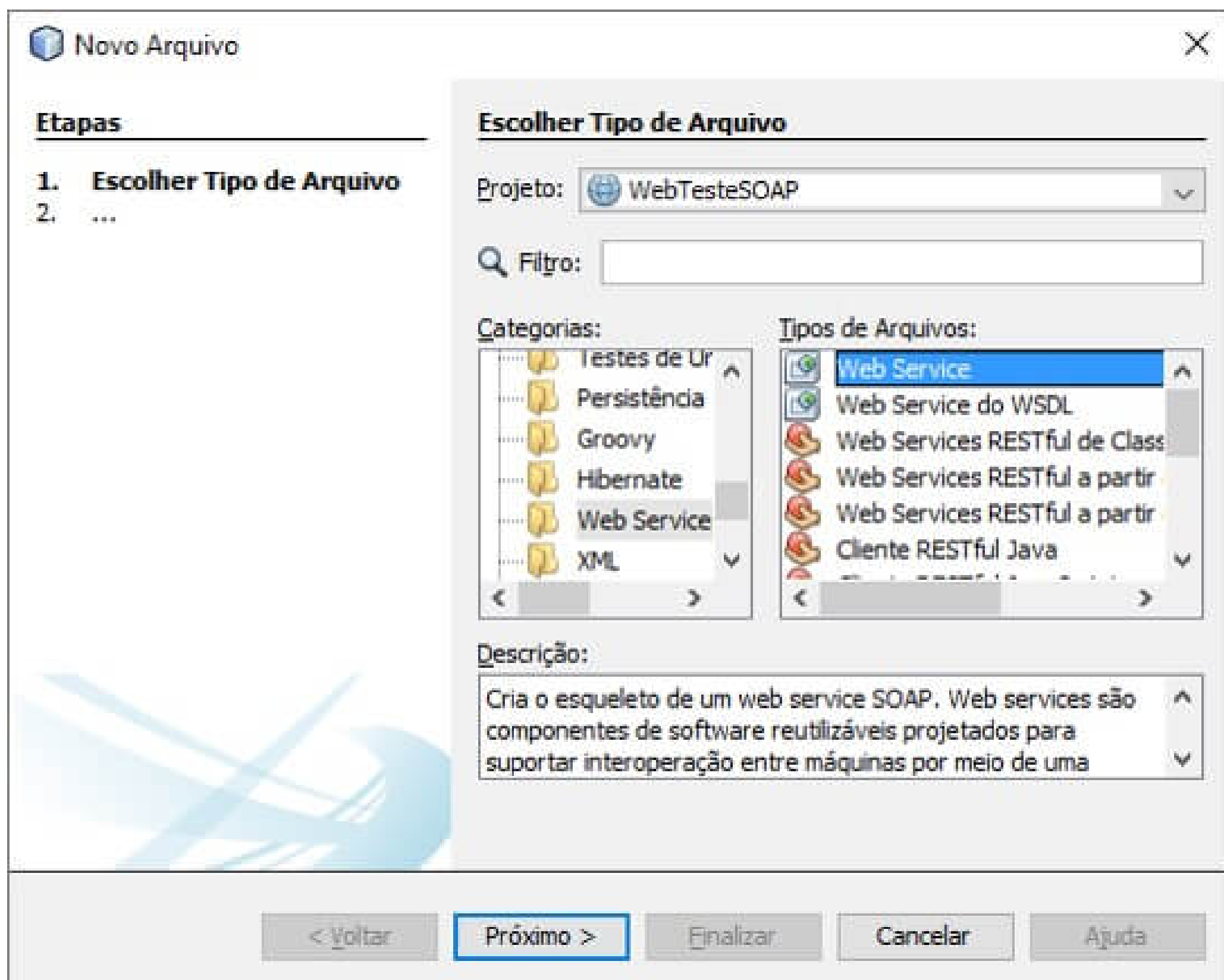
</m:GetStockPriceResponse>

</soap:Body>

</soap:Envelope>
```

Neste exemplo é feita uma chamada com a passagem do nome da empresa (**IBM**), sendo recebida a resposta com o valor de suas ações (**34.5**). Podemos notar alguma complexidade de escrita devido ao formalismo impetrado pelo SOAP, mas o uso do sistema de anotações do Java irá automatizar a criação destes pacotes, facilitando muito o nosso trabalho.

Inicialmente devemos criar um aplicativo Web comum, como fizemos em aulas anteriores, ao qual daremos o nome de **WebTesteSOAP**. Neste aplicativo Web iremos adicionar um novo arquivo e executar os seguintes passos:



1. Selecionar o tipo de componente **Web Services..Web Service** e clicar em **Próximo**.

Etapas

1. Escolher Tipo de Arquivo
2. **Nome e Localização**

Nome e LocalizaçãoNome do Web Service: Projeto: Localização: Pacote: ☒ Criar Web Service do início☐ Criar Web Service a partir do Bean de Sessão ExistenteEnterprise Bean: ☐ Implementar Web Services como Bean de Sessão Sem Estad

< Voltar

Próximo >

Finalizar

Cancelar

Ajuda

2. Definir o nome como **Calculadora** e o pacote como webs, e clicar em **Finalizar**.

Etapas

1. Escolher Tipo de Arquivo
2. ...

Escolher Tipo de Arquivo

Projeto: WebTesteSOAP

Filtro:

Categorias:

- Testes de Ur
- Persistência
- Groovy
- Hibernate
- Web Service
- XML

Tipos de Arquivos:

- Web Service
- Web Service do WSDL
- Web Services RESTful de Class
- Web Services RESTful a partir
- Web Services RESTful a partir
- Cliente RESTful Java

Descrição:

Cria o esqueleto de um web service SOAP. Web services são componentes de software reutilizáveis projetados para suportar interoperação entre máquinas por meio de uma

< Voltar

Próximo >

Finalizar

Cancelar

Ajuda

Etapas

1. Escolher Tipo de Arquivo
2. Nome e Localização

Nome e Localização

Nome do Web Service: Calculadora

Projeto: WebTesteSOAP

Localização: Pacotes de Códigos-fonte

Pacote: webs

☒ Criar Web Service do início☐ Criar Web Service a partir do Bean de Sessão Existente

Enterprise Bean:

Procurar...

☐ Implementar Web Services como Bean de Sessão Sem Estad

< Voltar

Próximo >

Finalizar

Cancelar

Ajuda

O código gerado é apresentado a seguir.

...

```
package webs;
```

```
import javax.jws.WebService;
```

```
import javax.jws.WebMethod;
```

```
import javax.jws.WebParam;
```

```
@WebService(serviceName = "Calculadora")
```

```
public class Calculadora {
```

```
    @WebMethod(operationName = "hello")
```

```
    public String hello(@WebParam(name = "name") String txt) {
```

```
        return "Hello " + txt + " !";
```

```
    }
```

```
}
```

Neste código podemos observar uma classe comum, mas que se transforma em Web Service com o uso de anotações. Estas anotações são:

1

WebService

Aplicada à classe que funcionará como um Serviço Web, e tendo como parâmetro o nome do serviço (**serviceName**).

2

WebMethod

Utilizada no método que será exposto pelo serviço, devendo ser indicado o nome da operação (**operationName**).

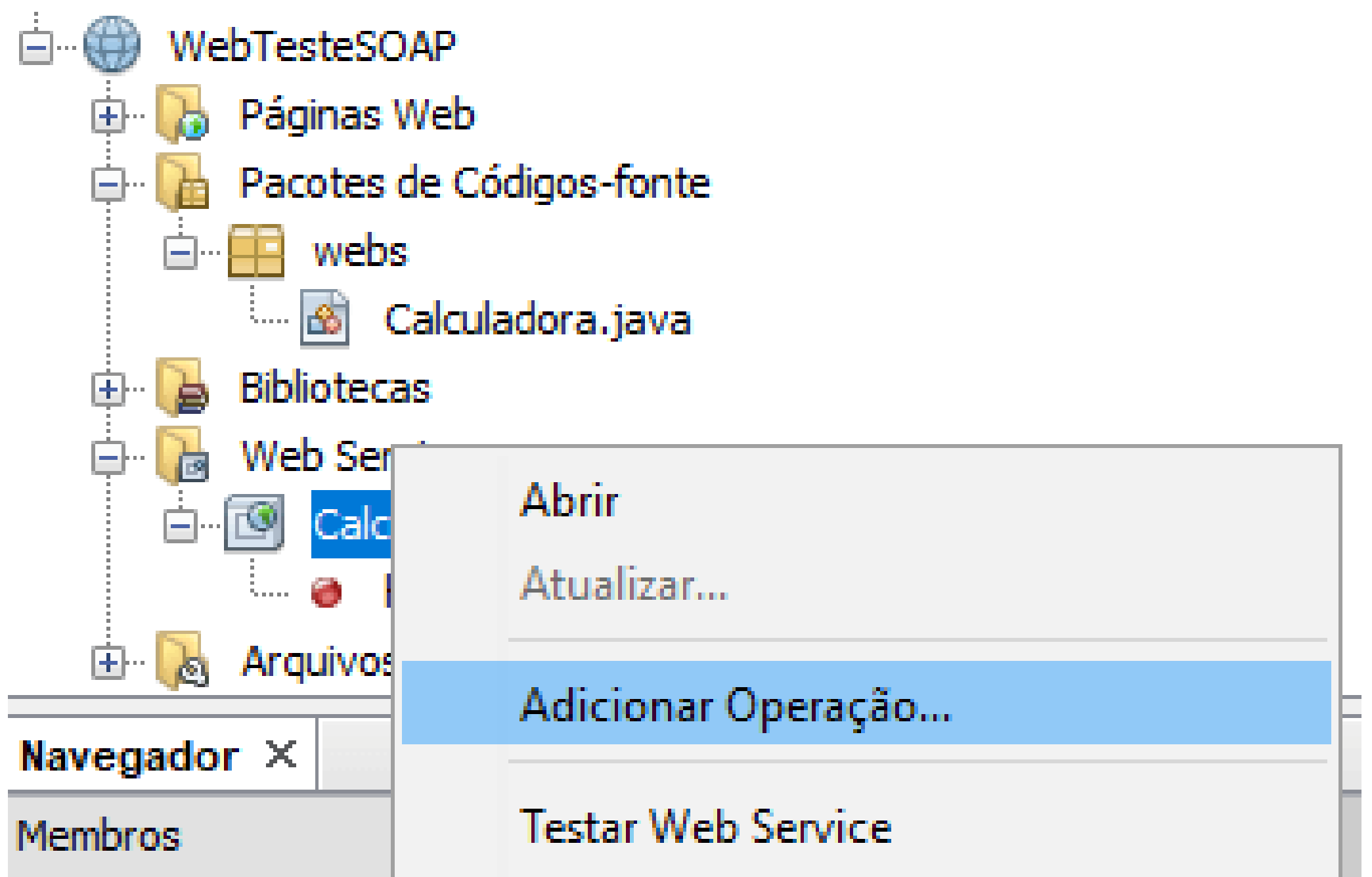
3

WebParam

Aplicada a cada parâmetro do método exposto, necessitando do nome que será utilizado em cada um (**name**).

Inicialmente temos apenas um método exposto, o qual é criado como exemplo pelo NetBeans; este método recebe um parâmetro texto reconhecido como “name” e retorna um texto com mensagem de boas-vindas. Obviamente, tanto a chamada quanto a recepção serão em pacotes SOAP.

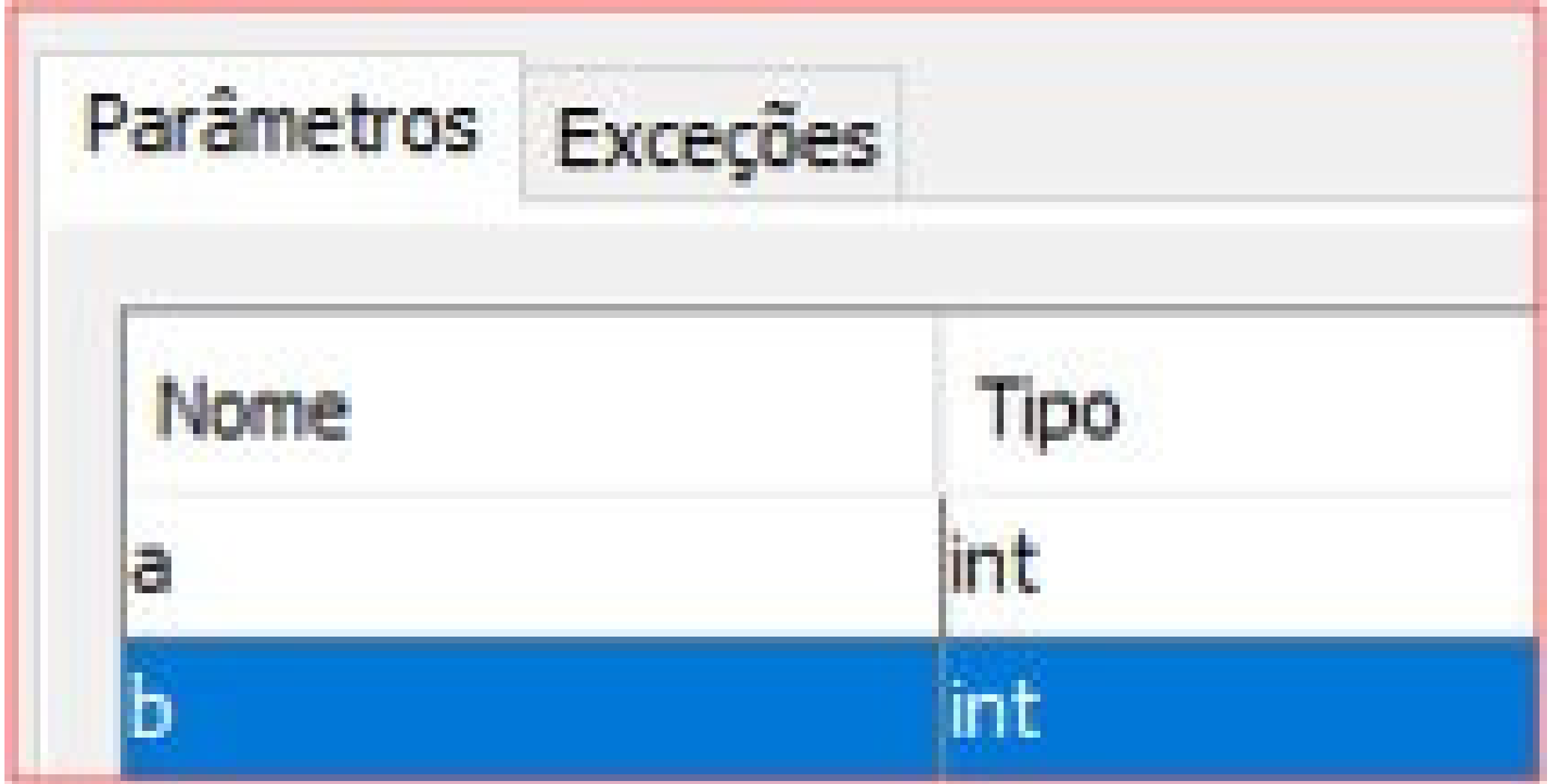
Poderíamos acrescentar novos métodos e simplesmente utilizar as anotações, mas o NetBeans traz um caminho visual para este tipo de tarefa, o qual envolve poucos passos:



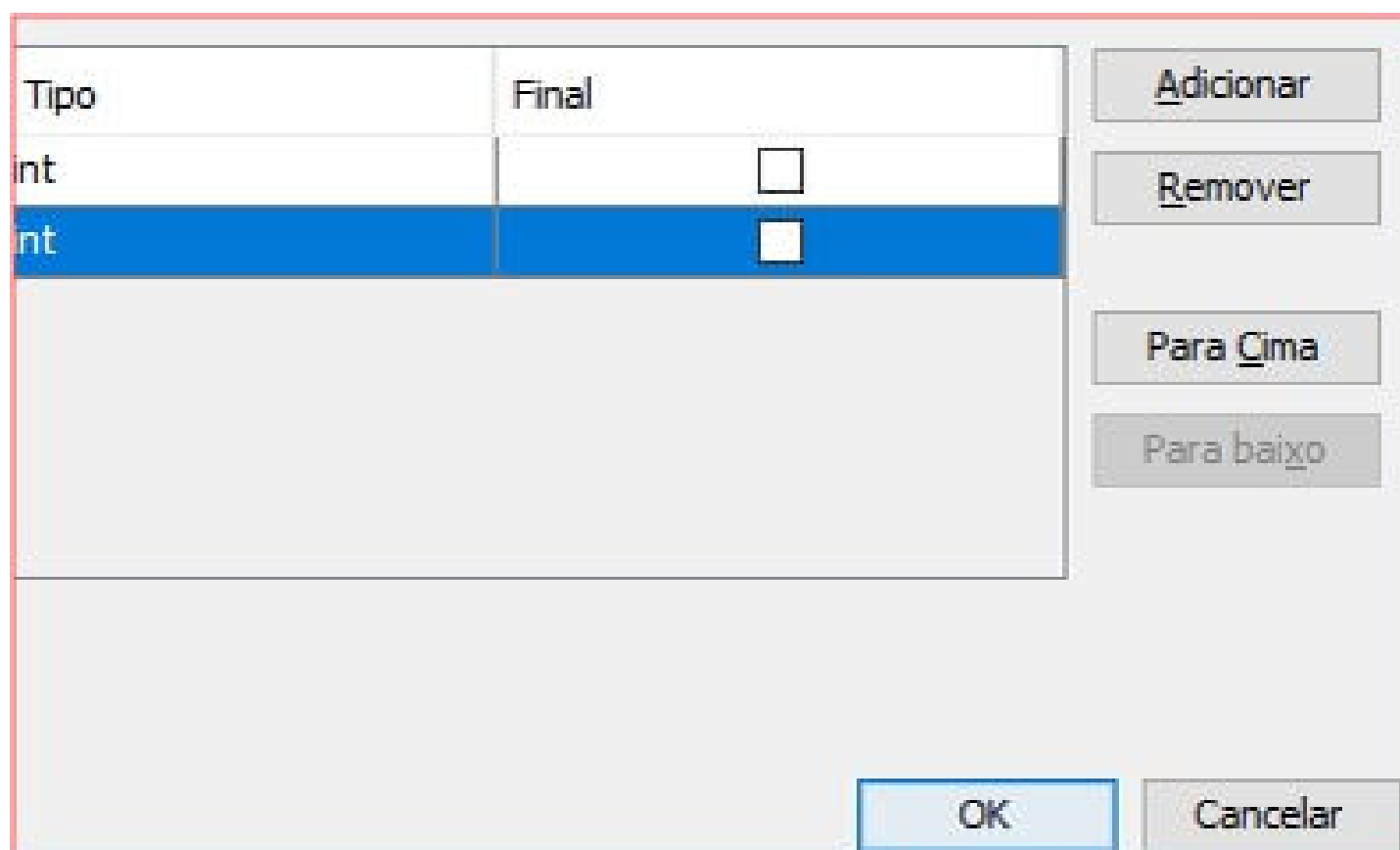
1. Clicar com o botão direito sobre o nome **Calculadora**, na guia **Web Services** do projeto, e selecionar a opção “**Adicionar Operação...**”.

<u>N</u> ome:	somar
<u>T</u> ipo de Retorno:	int

2. Defina o nome da operação como **somar** e o retorno do tipo **int**.



3. Na divisão de **Parâmetros**, Clique em Adicionar duas vezes e defina os parâmetros “**a**” e “**b**”, ambos do tipo **int**.



4. Clique em **OK** para gerar o método.

WebTesteSOAP

- + Páginas Web
- Pacotes de Códigos-fonte
 - webs
 - Calculadora.java
- + Bibliotecas
- Web Services
 - Calculadora
- + Arquivos

Navegador X

Membros

- Abrir
- Atualizar...
- Adicionar Operação...
- Testar Web Service

Nome: somar

Tipo de Retorno: int

Parâmetros

Exceções

Nome

Tipo

a

int

b

int

Tipo

Final

int

☐

int

☒

Adicionar

Remover

Para Cima

Para baixo

OK

Cancelar

Ao final, teremos a geração do método somar, já com as anotações corretas; precisaremos apenas modificar o valor retornado pelo mesmo, como podemos observar a seguir.

...

```
@WebMethod(operationName = "somar")
```

```
public int somar(@WebParam(name = "a") int a,
```

```
@WebParam(name = "b") int b) {
```

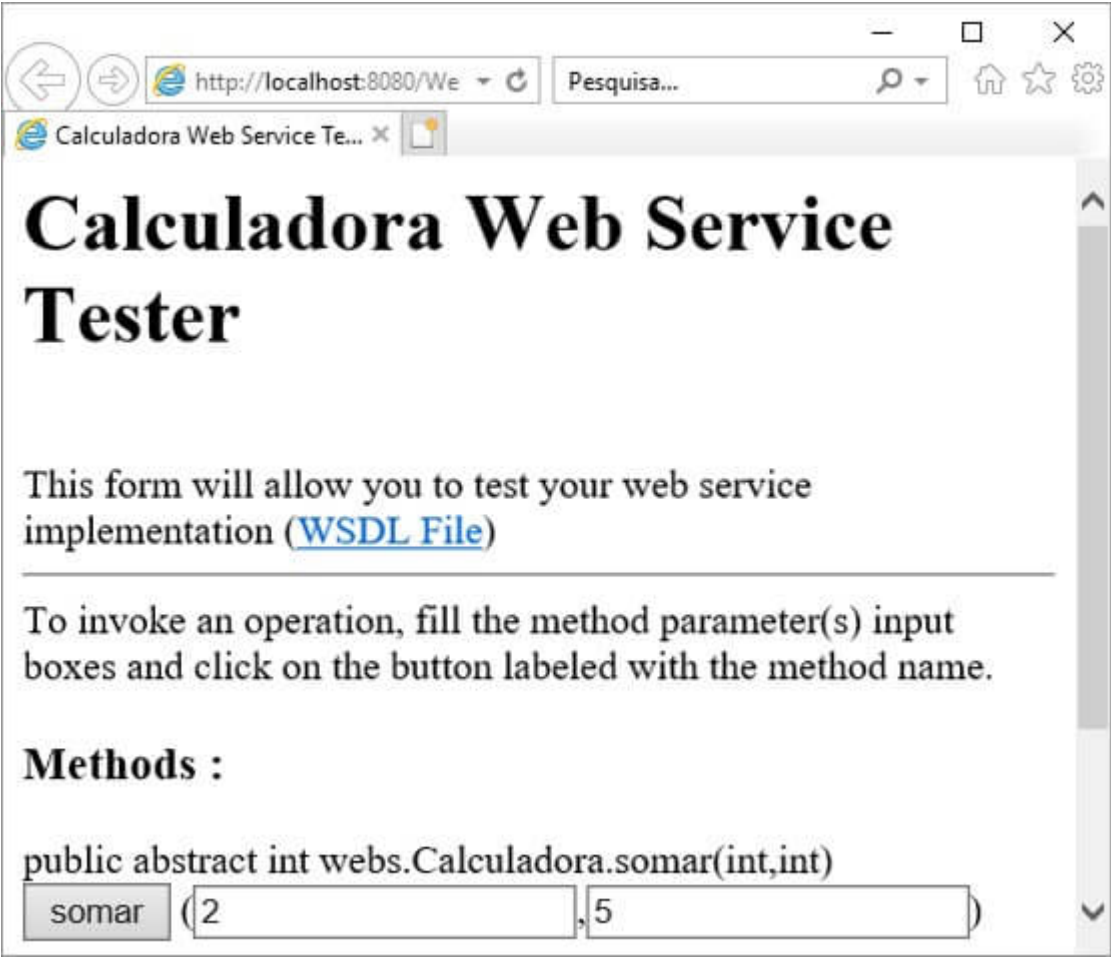
```
return a + b;
```

```
}
```

Com a escolha do servidor **GlassFish**, é possível testar de forma simples o novo Web Service. Basta clicar com o botão direito em **Calculadora**, guia **Web Services**, e escolher a opção de menu “**Testar Web Service**”, o que fará abrir uma página Web de teste, com a possibilidade de preencher os parâmetros de chamada e verificar o retorno no próprio navegador.

Para evitar alguns erros nessa chamada, é aconselhável implantar o projeto antes de invocar a opção de teste.

A janela de teste pode ser observada a seguir, inclusive com o link para o endereço do arquivo **WSDL**.



Nesta janela podemos preencher os valores e clicar em somar para ativar a referida operação no Web Service e verificar os pacotes de envio da solicitação e recepção do resultado na tela seguinte que se abre.

Outro elemento importante é o link para o endereço do **WSDL** (Web Services Description Language), um arquivo XML de grande importância para a geração de código nas diversas plataformas, pois trata da descrição do Web Service, indicando as operações disponíveis e formatos de parâmetros das chamadas, bem como o formato do retorno de cada operação.

Além do SOAP e do WSDL, existe um terceiro arquivo para o registro do Web Service, permitindo a pesquisa em ferramentas próprias de busca, segundo um formato denominado UDDI, ou Universal Description, Discovery and Integration.

Podemos ver na tabela seguinte uma comparação entre algumas das plataformas de processamento distribuído, incluindo Web Services.

Tecnologia	Protocolo	Descrição	Registro
CORBA	IIOP	OMG IDL	Cos Naming
RMI	RMI	Java	RMI Registry
EJB (RMI-IIOP)	IIOP	Java	JNDI
WS (SOAP)	SOAP	WSDL	UDDI

Chamadas AJAX para SOAP

A maior dificuldade em efetuar uma chamada **AJAX** para o **SOAP** não está na recepção dos dados, mas na configuração do pacote XML de solicitação do serviço. Isto ocorre porque precisamos montar um pacote XML bastante complexo para efetuar a chamada, porém a recepção é tratada muito facilmente com o uso de DOM.

Por questões de segurança, uma chamada para Web Service deve ser feita apenas a partir de uma página hospedada em servidor, razão pela qual iremos utilizar o arquivo **index.html** do aplicativo Web em que foi criado o Web Service e alterá-lo para o código seguinte.

```
...
<html><body>

<input type="text" id="valorA"/>

<input type="text" id="valorB"/>

<button onclick="atualizar()">Somar</button>

<div id="resposta">Resposta Aqui</div>

<script>

var objA = document.getElementById("valorA");

var objB = document.getElementById("valorB");

function atualizar(){

var url = "http://localhost:8080/WebTesteSOAP/Calculadora";

var params = '<?xml version="1.0" encoding="UTF-8"?>' +

' <S:Envelope ' +

' xmlns:S="//schemas.xmlsoap.org/soap/envelope/"> ' +

' <S:Header/><S:Body>' +

' <ns2:somar xmlns:ns2="//webs/">' +

' <a>' + objA.value + '</a> <b>' + objB.value + '</b> ' +

' </ns2:somar> ' +

' </S:Body> </S:Envelope>';

xhttp = new XMLHttpRequest();

xhttp.open("POST", url, true);

xhttp.onreadystatechange = AJAX_Callback;

xhttp.setRequestHeader('Content-Type', 'text/xml');

xhttp.send(params);

}
```



```
function AJAX_Callback(){

if (xhttp.readyState == 4 && xhttp.status == 200) {

var parser = new DOMParser();

var xmlDoc = parser.parseFromString(

xhttp.responseText,"text/xml");

var soma = xmlDoc.getElementsByTagName("return");

document.getElementById("resposta").innerHTML =

"Soma: "+soma[0].childNodes[0].nodeValue;

}

}

</script>

</body></html>
```

Neste código são definidas duas caixas de texto para a entrada dos valores, identificadas como **valorA** e **valorB**, e um botão que chamará a função **atualizar**. De forma geral, não difere de outras chamadas AJAX, mas além de utilizar obrigatoriamente o método **POST**, tem como parâmetro um **XML** no formato **SOAP**, onde ocorre a concatenação dos valores digitados nas caixas de texto.

```
• • •
var params = '<?xml version="1.0" encoding="UTF-8"?>' +

' <S:Envelope '+

' xmlns:S="//schemas.xmlsoap.org/soap/envelope/"> '+

' <S:Header/><S:Body>'+

' <ns2:somar xmlns:ns2="//webs/">'+

' <a>'+objA.value+'</a> <b>'+objB.value+'</b> '+

' </ns2:somar> '+

' </S:Body> </S:Envelope>;
```

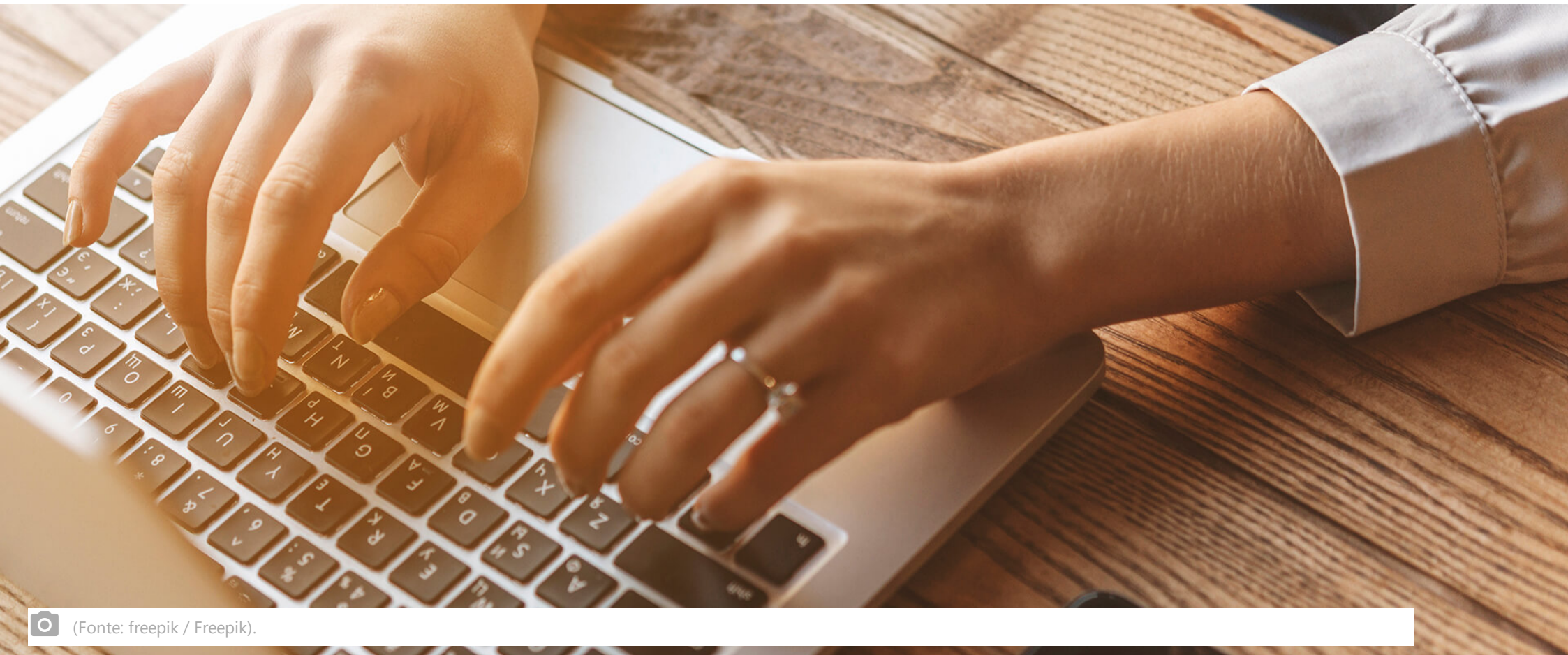
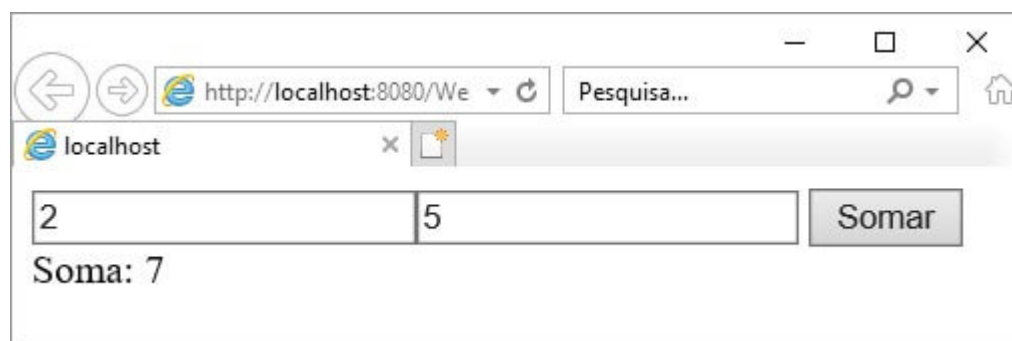
Da mesma forma que os dados são enviados em formato XML-SOAP, também são recebidos neste formato, permitindo o tratamento através de **DOMParser**. Com o uso deste parser, basta efetuarmos a busca pelo nó de elemento denominado **return** e pegar o valor de seu filho imediato, que é um nó de texto.

```
• • •
var soma = xmlDoc.getElementsByTagName("return");

document.getElementById("resposta").innerHTML =

"Soma: "+soma[0].childNodes[0].nodeValue;
```

Podemos observar o resultado da execução na tela seguinte, apresentando o resultado após o preenchimento de valores e clique no botão.



(Fonte: freepik / Freepik).

REST

A arquitetura **REST** (Representational State Transfer) permite a descrição de consultas aos objetos e seus estados, definidos pelos valores dos atributos, com o uso de **URLs**. Não apenas a consulta, mas as diversas operações sobre estes objetos, são efetuadas com o uso do protocolo **HTTP** e seus diferentes métodos.

Além do SOAP, outro tipo de Web Service comum no mercado é o **RESTful**, que leva este nome por utilizar uma arquitetura REST. Trata de um modelo menos formal, com possibilidade de utilização de sintaxe **JSON**, entre outras, e que acaba sendo mais voltado para a comunicação **B2C** (Business to Consumer), ou seja, da empresa para os clientes.

Atenção

Embora o REST seja mais simples, e amplamente adotado no ambiente móvel, ainda hoje temos a utilização do SOAP no meio B2B, pois traz maior formalismo no que se refere à declaração e utilização de serviços.

Uma diferença básica do modelo utilizado pelo JEE5 é a necessidade da presença do arquivo **web.xml**, o que configuraria uma desvantagem, mas acaba sendo uma solução versátil para o trabalho com o formato JSON.

O segundo passo é a criação da entidade cujos estados serão representados pelo REST, o que faremos simplesmente adicionando uma **nova classe** com o nome **CalculadoraResult**, dentro do pacote **unesa**.

Devemos codificar a nova classe, conforme apresentado a seguir.

```
• • •
package unesa;

public class CalculadoraResult {

    private int a;

    private int b;

    private String operacao;

    private int resultado;

    public CalculadoraResult() {

    }

    public int getA() { return a; }

    public int getB() { return b; }

    public String getOperacao() { return operacao; }

    public int getResultado() { return resultado; }

    public void setA(int a) { this.a = a; }

    public void setB(int b) { this.b = b; }

    public void setOperacao(String operacao)

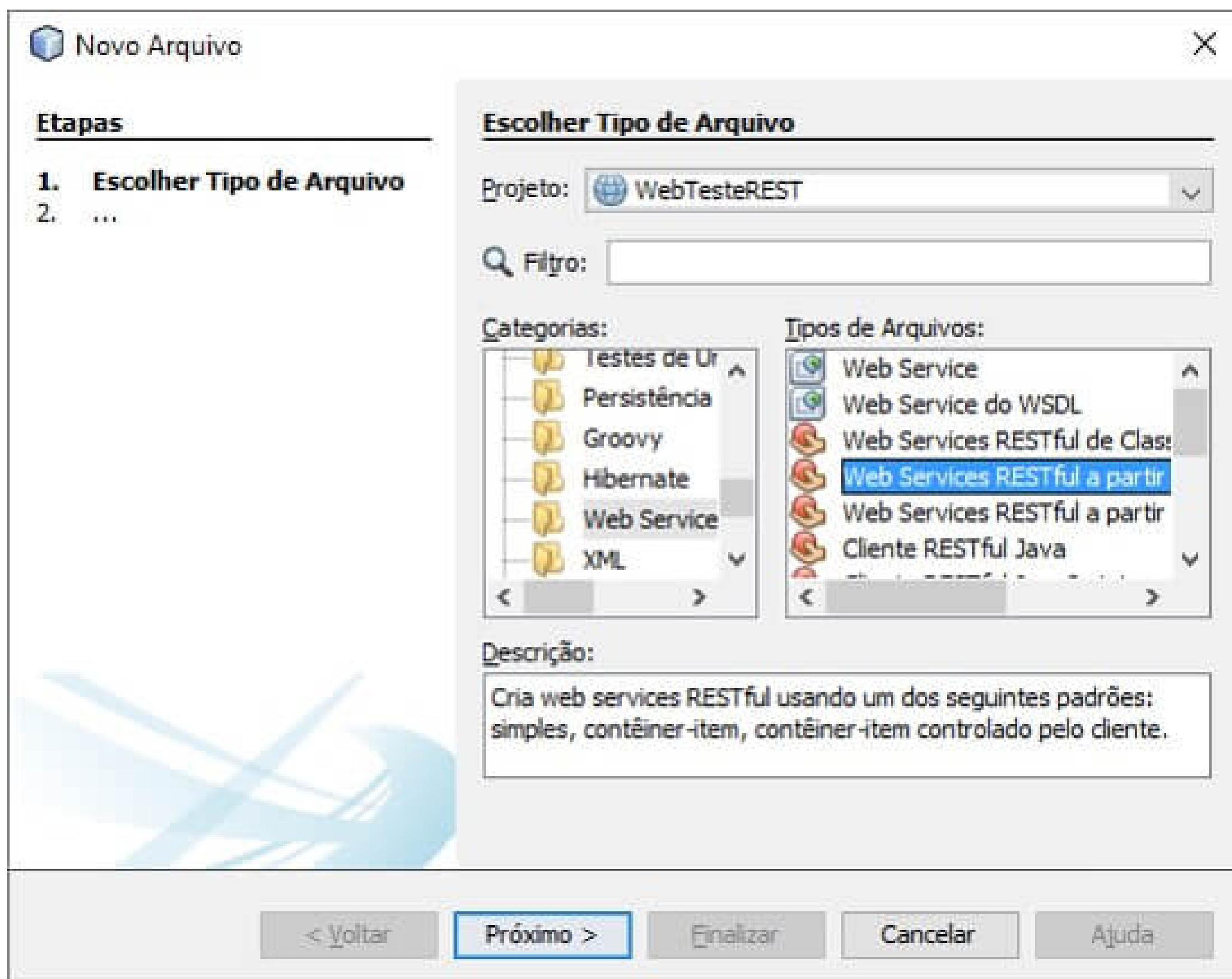
    { this.operacao = operacao; }

    public void setResultado(int resultado)

    { this.resultado = resultado; }

}
```

Esta classe é basicamente um POJO, e servirá de base para a criação do Web Service REST. O que precisaremos fazer é adicionar um **novo arquivo** e seguir os seguintes passos:



1. Selecionar o tipo **Web Services..Web Services RESTful** a partir dos Padrões e clicar em **Próximo**.



Etapas

1. Escolher Tipo de Arquivo
- 2. Selecionar Padrão**
3. Especificar Classes de Recurso

Selecionar Padrão

Selecionar um padrão de projeto do Web service RESTful

- ☒ **Recurso Raiz Simples**
- ☐ Contêiner-Item
- ☐ Contêiner-Item Controlado pelo Cliente

Descrição:

Crie uma classe de recursos raiz RESTful com os métodos GET e PUT usando API Java para o Web Service RESTful (JSR-311). Esse padrão é útil para criar um serviço simples HelloWorld e serviços de encapsulador para invocar os Web Services baseados em WSDL.

< **V**oltar

Próximo >

Finalizar

Cancelar

Ajuda

2. Escolher o tipo “**Recurso Raiz Simples**” e clicar em **Próximo**.

Etapas

1. Escolher Tipo de Arquivo
2. Selecionar Padrão
3. **Especificar Classes de Recurso**

Especificar Classes de Recurso

Projeto:	WebTesteREST
Localização:	Pacotes de Códigos-fonte ▾
Pacote do Recurso\:	unesa ▾
Caminho\:	calculadora
Nome da Classe:	CalculadoraResource
Tipo MIME:	application/json ▾
Classe de Representação:	unesa.Calc Selecionar...

< Voltar

Próximo >

Finalizar

Cancelar

Ajuda

3. Configurar o pacote para **unesa**, caminho como **calculadora** e nome da classe como **CalculadoraResource**, selecionar o formato **application/json** e a classe de representação **CalculadoraResult**, e clicar em **Finalizar**.

Etapas

1. Escolher Tipo de Arquivo
2. ...

Escolher Tipo de ArquivoProjeto:  WebTesteREST

Filtro:

Categorias:

- Testes de Ur
- Persistência
- Groovy
- Hibernate
- Web Service
- XML

Tipos de Arquivos:

- Web Service
- Web Service do WSDL
- Web Services RESTful de Classe
- Web Services RESTful a partir**
- Web Services RESTful a partir
- Cliente RESTful Java

Descrição:

Cria web services RESTful usando um dos seguintes padrões: simples, contêiner-item, contêiner-item controlado pelo cliente.

< Voltar

Próximo >

Finalizar

Cancelar

Ajuda

Etapas

1. Escolher Tipo de Arquivo
2. **Selecionar Padrão**
3. Especificar Classes de Recurso

Selecionar Padrão

Selecionar um padrão de projeto do Web service RESTful

- ☒ **Recurso Raiz Simples**
- ☐ Contêiner-Item
- ☐ Contêiner-Item Controlado pelo Cliente

Descrição:

Crie uma classe de recursos raiz RESTful com os métodos GET e PUT usando API Java para o Web Service RESTful (JSR-311). Esse padrão é útil para criar um serviço simples HelloWorld e serviços de encapsulador para invocar os Web Services baseados em WSDL.

< Voltar

Próximo >

Finalizar

Cancelar

Ajuda

Etapas

1. Escolher Tipo de Arquivo
2. Selecionar Padrão
3. **Especificar Classes de Recurso**

Especificar Classes de Recurso

Projeto:	WebTesteREST
Localização:	Pacotes de Códigos-fonte ▾
Pacote do Recurso\:	unesa ▾
<hr/>	
Caminho\:	calculadora
Nome da Classe:	CalculadoraResource
Tipo MIME:	application/json ▾
Classe de Representação:	unesa.Calc Selecionar...

< Voltar

Próximo >

Finalizar

Cancelar

Ajuda

Além do Web Service REST, é criada uma classe de configuração padrão, cujo código é apresentado a seguir.

```
...
package unesa;

import java.util.Set;

import javax.ws.rs.core.Application;

@javax.ws.rs.ApplicationPath("webresources")

public class ApplicationConfig extends Application {

    @Override

    public Set<Class<?>> getClasses(){

        Set<Class<?>> resources = new java.util.HashSet<>();

        addRestResourceClasses(resources);

        return resources;

    }

    private void addRestResourceClasses(Set<Class<?>> resources){

        resources.add(unesa.CalculadoraResource.class);

    }

}
```

A classe **ApplicationConfig** tem uma anotação com o mapeamento do endereço de base de nossos Web Services, no caso **webresources**.

```
...
@javax.ws.rs.ApplicationPath("webresources")
```

No método **addRestRessourceClasses** são adicionadas as diversas classes anotadas para o modelo REST, como **unesa.CalculadoraResource**.

```
...
private void addRestResourceClasses(Set<Class<?>> resources){

    resources.add(unesa.CalculadoraResource.class);

}
```

Esta classe representará o aplicativo REST, devendo ser mapeada no arquivo **web.xml**, o que foi feito automaticamente neste processo, conforme podemos observar apenas vistoriando o código do arquivo.

```
...
<web-app  version="2.5"  xmlns="//java.sun.com/xml/ns/javaee" xmlns:xsi="//www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="//java.sun.com/xml/ns/javaee //java.sun.com/xml/ns/javaee/web-app_2_5.xsd">

<servlet>
```

```
<servlet-name>ServletAdaptor</servlet-name>

<servlet-class>

org.glassfish.jersey.servlet.ServletContainer

</servlet-class>

<init-param>

<param-name>javax.ws.rs.Application</param-name>

<param-value>unesa.ApplicationConfig</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>ServletAdaptor</servlet-name>

<url-pattern>/webresources/*</url-pattern>

</servlet-mapping>

<session-config>

<session-timeout>

30

</session-timeout>

</session-config>

<welcome-file-list>

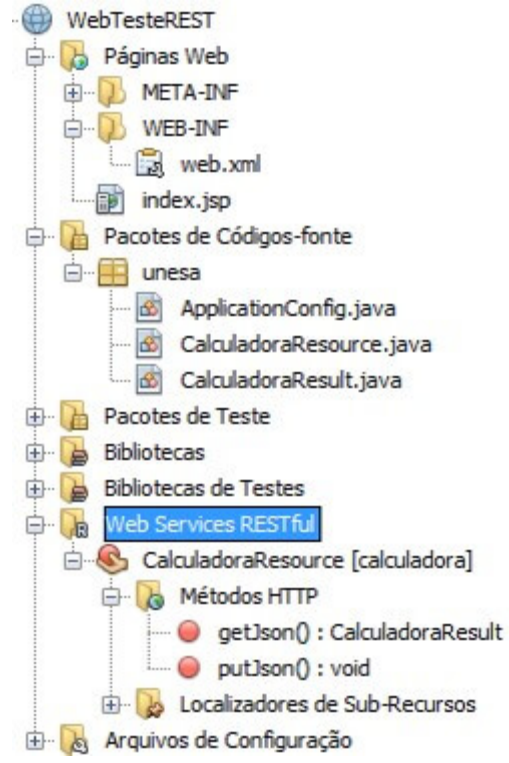
<welcome-file>index.jsp</welcome-file>

</welcome-file-list>

</web-app>
```

A presença desta classe de aplicativo é uma exigência para o uso da biblioteca **Jersey**, responsável pelo ciclo de vida dos Web Services REST anotados do Java.

Podemos observar também o surgimento de uma nova opção na árvore do projeto, referente à gerência de **Web Services RESTful**.



Finalmente, vamos ao código de **CalculadoraResource**. Para tal, devemos compreender a utilização de algumas anotações, conforme é apresentado na tabela seguinte.

Anotação	Utilização
@GET	Resposta ao método GET, normalmente voltado para consultas
@POST	Resposta ao método POST, voltado para criação de entidades
@PUT	Resposta ao método PUT, utilizado na alteração de entidades
@DELETE	Resposta ao método DELETE, efetuando a exclusão a partir da chave
@Path	Definição do caminho para a chamada da operação
@PathParam	Mapeia o parâmetro do método para o parâmetro equivalente no caminho de chamada
@Produces	Formato de dados de retorno para a operação
@Consumes	Formato de dados para a chamada da operação

Iremos codificar **CalculadoraResource** da forma que é apresentada a seguir.

```
...
package unesa;

import javax.ws.rs.core.Context;

import javax.ws.rs.core.UriInfo;

import javax.ws.rs.Produces;

import javax.ws.rs.Consumes;

import javax.ws.rs.GET;

import javax.ws.rs.Path;

import javax.ws.rs.PUT;

import javax.ws.rs.PathParam;

@Path("calculadora")

public class CalculadoraResource {

    @Context

    private UriInfo context;

    public CalculadoraResource() {

    }

    @GET
```

```
@Produces(javax.ws.rs.core.MediaType.APPLICATION_JSON)
```

```
public CalculadoraResult getJson() {
```

```
    CalculadoraResult cr = new CalculadoraResult();
```

```
    cr.setOperacao("nenhuma");
```

```
    return cr;
```

```
}
```

```
@GET
```

```
@Produces(javax.ws.rs.core.MediaType.APPLICATION_JSON)
```

```
@Path("somar/{a}/{b}")
```

```
public CalculadoraResult somar(
```

```
    @PathParam("a")int a,@PathParam("b")int b) {
```

```
    CalculadoraResult cr = new CalculadoraResult();
```

```
    cr.setA(a);
```

```
    cr.setB(b);
```

```
    cr.setOperacao("somar");
```

```
    cr.setResultado(a+b);
```

```
    return cr;
```

```
}
```

```
@PUT
```

```
@Consumes(javax.ws.rs.core.MediaType.APPLICATION_JSON)
```

```
public void putJson(CalculadoraResult content) {
```

```
}
```

```
}
```

O método de maior interesse para nós é o **somar**, que utiliza um mapeamento dinâmico, com a passagem de dois parâmetros.

...

```
@GET
```

```
@Produces(javax.ws.rs.core.MediaType.APPLICATION_JSON)
```

```
@Path("somar/{a}/{b}")
```

```
public CalculadoraResult somar(
```

```
    @PathParam("a")int a,@PathParam("b")int b)
```

Com uma chamada do tipo **somar/12/51**, estaríamos preenchendo o parâmetro **a** com o valor **12** e o parâmetro **b** com o valor **51**.

Internamente é instanciado e configurado um objeto da classe **CalculadoraResult**, onde o resultado recebe a soma dos dois parâmetros; este objeto é retornado pelo método para que seja convertido no formato **JSON** e enviado ao cliente, isto devido à presença da anotação **@Produces**.

...

@GET

@Produces(javax.ws.rs.core.MediaType.**APPLICATION_JSON**)

Agora só precisamos implantar nosso projeto e testar a funcionalidade desejada. Teste mais fácil de efetuar com o uso do comando **curl**, do Windows em linha de comando.

...

curl //localhost:8084/WebTesteREST/webresources/calculadora/somar/2/5

Estando tudo correto, obteremos a saída apresentada a seguir.

...

{"a":2,"b":5,"operacao":"somar","resultado":7}

Como podemos observar, o retorno será a classe no formato JSON, o que pode ser utilizado facilmente por diversas ferramentas, como JQuery.



Chamadas JQuery para REST

Já conhecemos bem as diversas facilidades trazidas pelo JQuery na criação de interfaces gráficas, mas existem diversas outras funcionalidades desta biblioteca que facilitam a comunicação das páginas com servidores REST.

A biblioteca JQuery AJAX poderá ser utilizada neste contexto, e o código de nossa página index deverá ser modificado para o que é apresentado a seguir.

```
• • •
<%@page contentType="text/html" pageEncoding="UTF-8"%>

<!DOCTYPE html>

<html>

<head>

<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.10.2/jquery.min.js">

</script>

</head>

<body>

<input type="text" id="a">

<input type="text" id="b">

<button id="somar">somar</button>

<p id="saida"></p>

<script>

$("#somar").click(function(){

var endereco = "//localhost:8084/WebTesteREST/"+

"webresources/calculadora/somar/"+

$("#a").val() + "/" + $("#b").val();

$.ajax({

url: endereco

}).then(function(data){

$("#saida").html(data.a+ " + "+data.b+ " = "+

data.resultado);

});

});

</script>
```


</body>

</html>

Inicialmente temos a inclusão da biblioteca JQuery, utilizada diretamente pela Web para que não fosse necessário adicionar mais arquivos ao projeto, mas nada impediria que utilizássemos de forma local.

• • •

```
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.10.2/jquery.min.js">
```

```
</script>
```

O código HTML contempla duas caixas de texto, um botão e uma divisão de parágrafo, todos estes elementos identificados via atributo id.

Com o uso de JQuery, temos a definição da resposta ao clicar sobre o botão; os passos iniciais envolvem a montagem da URL correta a partir dos números digitados nas duas caixas de texto.

• • •

```
var endereco = "//localhost:8084/WebTesteREST/"+
```

```
"webresources/calculadora/somar/"+
```

```
$("#a").val() + "/" + $("#b").val();
```

Finalmente, a chamada AJAX em modo GET, com a simples passagem da URL, seguida do tratamento da recepção. Os dados são transferidos para o parâmetro data da função, e como o formato utilizado foi JSON, o acesso aos atributos ocorre de forma direta.

• • •

```
$.ajax({
```

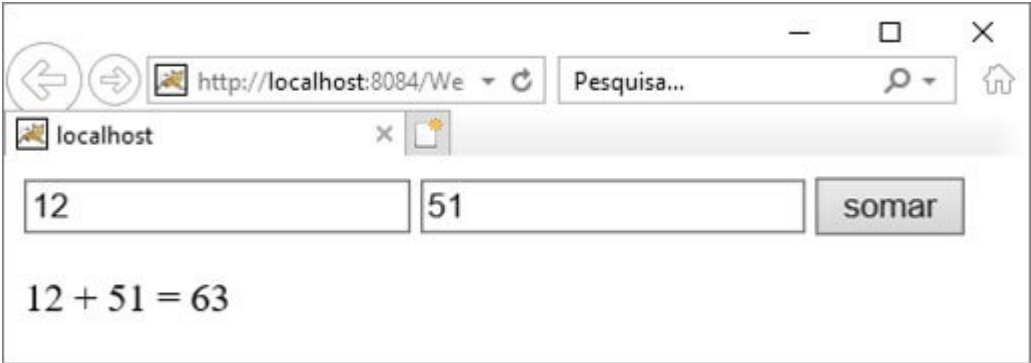
```
url: endereco
```

```
}).then(function(data){
```

```
$("#saida").html(data.a+" + "+data.b+" = "+data.resultado);
```

```
});
```

Agora só precisamos executar o projeto e testar a solução, onde devemos obter um resultado final como o apresentado a seguir.



Atividade

1. A arquitetura interoperável que serviu de referência para a criação do SOAP e, consequente, para o surgimento dos Web Services foi:

- a) CORBA
 - b) RMI
 - c) EJB
 - d) XML-RPC
 - e) REST
-

2. Considerando a classe Java seguinte, acrescente as anotações necessárias para que a mesma seja tratada como um Web Service SOAP, com exposição de todos os métodos.

```
...
public class Roteador {

    public String getURL (int ip1, int ip2, int ip3, int ip4) {

        return "";

    }

    public String getTextoIP (String URL) {

        return "";

    }

}
```

3. Com o uso de REST, qual seria a anotação correta para definir o formato de entrada de dados em uma chamada PUT?
- a) @Produces
 - b) @Path
 - c) @Consumes
 - d) @PathParam
 - e) @PUT
-

Referências

CASSATI, J. P. **Programação servidor em sistemas web**. Rio de Janeiro: Estácio, 2016.

DEITEL, P; DEITEL, H. **Java, como programar**. 8.ed. São Paulo: Pearson, 2010.

LECHETA, R; BURKE, B. **Web services restful**. 1.ed. São Paulo: Novatec, 2015.

MONSON-HAEFEL, R; BURKE, B. **Enterprise java beans 3.0**. 5.ed. São Paulo: Pearson, 2007.

Explore mais

Não deixe de acessar os materiais sugeridos a seguir:

- [The Java EE 5 Tutorial; <https://docs.oracle.com/javaee/5/tutorial/doc/bnayn.html>](https://docs.oracle.com/javaee/5/tutorial/doc/bnayn.html)
- [2 Developing RESTful Web Services; <https://docs.oracle.com/middleware/12212/wls/RESTF/develop-restful-service.htm#RESTF113>](https://docs.oracle.com/middleware/12212/wls/RESTF/develop-restful-service.htm#RESTF113)
- [RESTful services with jQuery and Java using JAX-RS and Jersey. <http://coenraets.org/blog/2011/12/restful-services-with-jquery-and-java-using-jax-rs-and-jersey/>](http://coenraets.org/blog/2011/12/restful-services-with-jquery-and-java-using-jax-rs-and-jersey/)