Sistemas Web Seguros

UNIDADE 03

Autenticação em serviços SOAP

Nesta unidade, veremos o mecanismo de autenticação baseado em usuário e senha para *web services* SOAP, o tratamento de erros utilizando a especificação JAX-WS e o desenvolvimento do sistema SISRH no sentido de disponibilizar seus dados por meio de *web services* SOAP autenticados.

AUTENTICAÇÃO EM SERVIÇOS SOAP

Tratamento de erros

Antes de falarmos da autenticação em *web services* SOAP, é preciso apresentar como os erros, inclusive os de autenticação, devem ser tratados conforme as especificações do SOAP. Erros podem acontecer a qualquer momento, por um problema interno, pelo

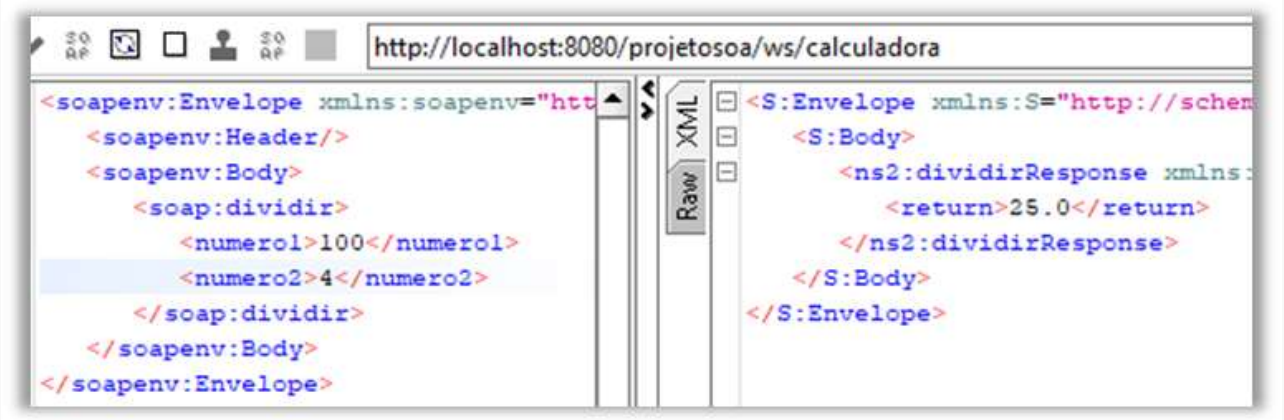
mau uso do serviço ou por motivos inesperados. Não podemos evitar que eles ocorram, mas podemos garantir o que deve ser feito quando o erro surge.

Vamos explicar cada detalhe por meio de um exemplo prático.

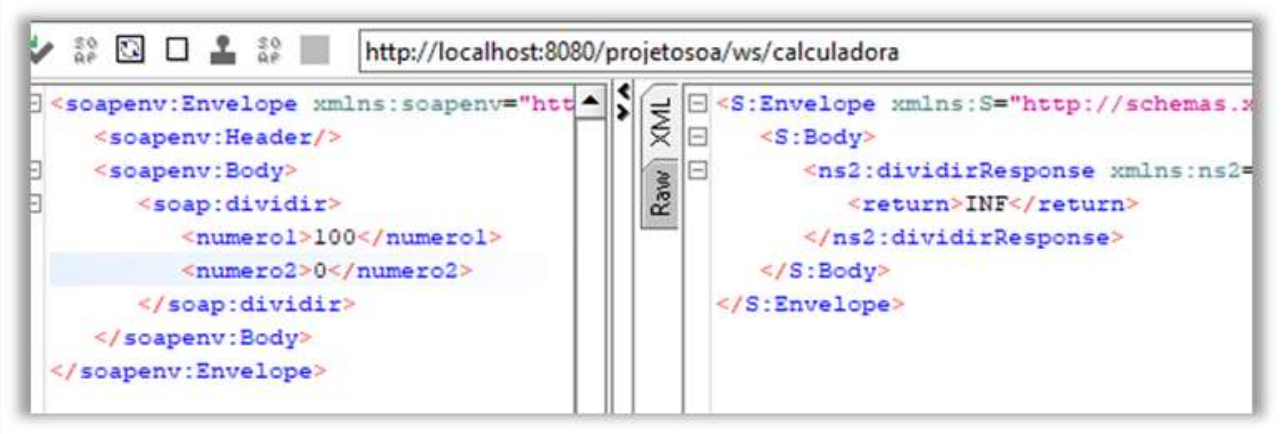
Volte ao nosso serviço **Calculadora** do **projetosoap**. Vamos ao serviço de divisão:

```
package soap;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
import javax.jws.soap.SOAPBinding.Style;

@WebService
@SOAPBinding(style = Style.RPC)
public class Calculadora {

        @WebMethod(action = "dividir")
        public double dividir(
                        @WebParam(name
= "numero1") double numero1,
                        @WebParam(name
= "numero2") double numero2) {
                return numero1 / numero2;
        }
}
```

Se usarmos o serviço corretamente, poderemos ver no SoapUI a divisão acontecendo conforme o esperado.



Mas o que acontecerá se tentarmos dividir 100 por zero?



Bom, neste caso, o próprio Java tentou resolver e retornou o texto **INF**, mas podemos deixar mais claro ao usuário que uma divisão por zero não é permitida e criar uma *fault*, ou seja, uma falha, que será um documento na WSDL; dessa forma, todos saberão o que pode acontecer ao chamar um serviço SOAP.

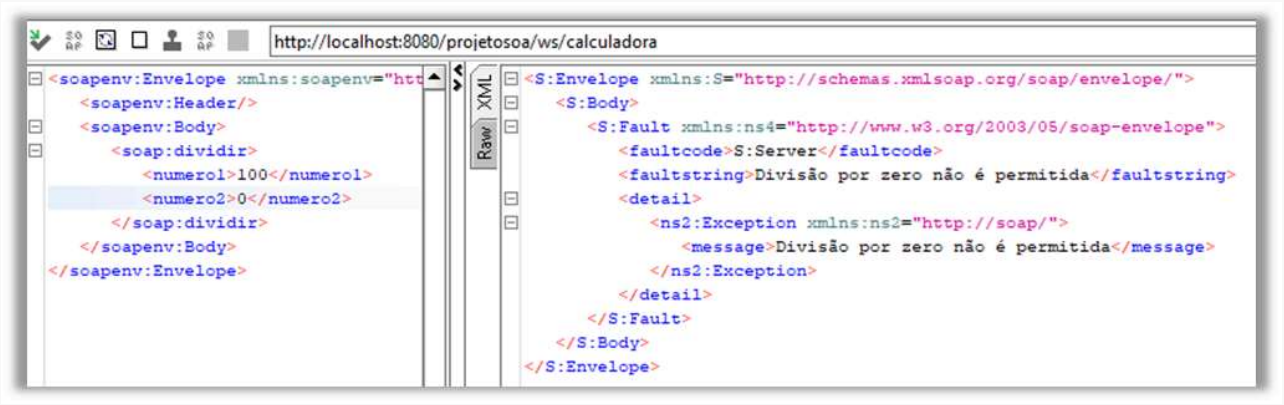
Vamos juntos construir essa solução? Vamos explicando aqui e você pode implementar no seu **projetosoa**.

Acesse o método **dividir**, que foi criado durante os exercícios da unidade anterior. Caso não tenha, tudo bem, você pode criá-lo agora, mas note que já vamos tratar o erro da divisão por zero.

```
@WebMethod(action = "dividir")
public double dividir(
                @WebParam(name = "numero1") double numero1,
                @WebParam(name
= "numero2") double numero2) throws Exception  {

       if (numero2 == 0) {
              throw new Exception("Divisão por zero não é
permitida");
       }

        return numero1 / numero2;
}
```

Notou o que fizemos? Testamos o parâmetro **numero2**; caso seja igual a zero, lançaremos uma exceção Java com a mensagem de erro. Como o método **dividir** lança uma *exception*, precisamos mudar a assinatura do método informando o **throws Exception**.

Veja, agora, o comportamento do teste de divisão por zero no SoapUI:



Observe que o retorno do serviço é uma estrutura XML chamada **Fault**. Nela, temos algumas informações, tais como:

- **faultcode**: define o código da falha.
- **faultstring**: apresenta a descrição da falha.
- **detail**: apresenta mais detalhes sobre a falha.

No **detail**, podemos representar um objeto complexo. Neste exemplo, como retornamos um objeto do tipo **Exception**, ele foi representado apenas com a mensagem de exceção, mas poderíamos retornar outro objeto que herde de **Exception** para apresentar mais informações.

Queremos reforçar a importância de usar a especificação JAX-WS. Note que fizemos um simples ajuste no código Java e tudo passou a se comportar de acordo com o SOAP. Inclusive, nosso documento WSDL foi modificado para representar o novo comportamento da operação **dividir**. Veja:

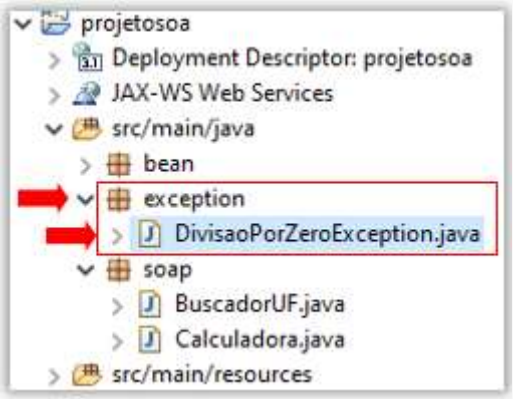
```
<operation name="dividir">
    <soap:operation soapAction="dividir"/>
    <input>
        <soap:body use="literal" namespace="http://soap/"/>
    </input>
    <output>
        <soap:body use="literal" namespace="http://soap/"/>
```

```
        </output>
        <fault name="Exception">
            <soap:fault name="Exception" use="literal"/>
        </fault>
</operation>
```

Agora, a operação **dividir** terá, além das entradas e saída, a falha chamada **Exception**, que terá um atributo chamado **Exception**.

Podemos melhorar o tratamento de erro deixando-o mais claro. Em vez de termos uma falha chamada **Exception**, podemos deixar mais claro o tipo de erro e chamá-lo **DivisaoPorZeroException**. Isso ajudará na leitura do WSDL e ficará claro que a calculadora trata esse erro em específico.

Para isso, vamos organizar nosso **projetosoa**. Crie um pacote chamado **exception** e, nele, uma classe chamada **DivisaoPorZeroException**.



Nossa exceção herdará da classe **Exception** e terá a anotação **WebFault**. Informaremos a mensagem da falha no construtor de classe. Veja:

```
package exception;

import javax.xml.ws.WebFault;

@WebFault(name = "DivisaoPorZero")
public class DivisaoPorZeroException extends Exception {

        private static final long serialVersionUID = 1L;
```

```
        public DivisaoPorZeroException() {
             super("Divisão por zero não é permitida!");
     }
}
```

Na classe **Calculadora**, precisaremos substituir a referência da classe **Exception** para a nossa classe **DivisaoPorZeroException**:

```
@WebMethod(action = "dividir")
public double dividir(
                @WebParam(name = "numero1") double numero1,
                @WebParam(name
= "numero2") double numero2) throws DivisaoPorZeroException  {

        if (numero2 == 0) {
                throw new DivisaoPorZeroException();
        }
        return numero1 / numero2;
}
```

Inclusive, nosso código ficou “limpo”, pois a mensagem de erro está embutida na nossa exceção.

Vamos ver o resultado do teste no SoapUI:

Agora, veja a mudança na operação **dividir** no WSDL:

```
<operation name="dividir" parameterOrder="numero1 numero2">
<input wsam:Action="dividir" message="tns:dividir"/>
```

```
<output wsam:Action="http://soap/Calculadora/dividirResponse
" message="tns:dividirResponse"/>
<fault message="tns:DivisaoPorZeroException" name="DivisaoPo
rZeroException" wsam:Action="http://soap/Calculadora/dividir
/Fault/DivisaoPorZeroException"/>
</operation>
```

Uma boa prática no tratamento de erros é a criação de tipos específicos, pois sua API ficará mais bem documentada e mais fácil de entender. Com esse rico mecanismo para tratar falhas, vamos avançar na autenticação de *web services* SOAP.

Autenticação de *web service* por meio de *headers*

Conforme falamos no início desta unidade, nossos serviços são públicos, de forma que qualquer um que souber a URL dos *endpoints* poderá chamá-los indiscriminadamente. Vamos melhorar a segurança dos nossos serviços incluindo o método de autenticação baseado em usuário e senha.

Como você sabe incluir parâmetros nos serviços, poderia pensar: posso incluir na chamada do serviço os parâmetros **usuário** e **senha** e validá-los? Apesar de não ser uma solução muito elegante, seria possível. Veja como ficaria:

```
@WebMethod(action = "somar")
public int somar(
                @WebParam(name = "numero1") int numero1,
                @WebParam(name = "numero2") int numero2,
                @WebParam(name = "usuario") String usuario,
                @WebParam(name = "senha")
String senha) throws Exception {

        if (usuario.equals("user") && senha.equals("123")){
                return numero1 + numero2;
        }else {
                throw new Exception("Usuário e senha inválido");
        }

}
```

Como comentamos, esta não é uma solução elegante, pois interfere diretamente na definição do serviço. Veja por que não utilizar essa estratégia:

- Conceitualmente, uma soma não deve receber nada além de dois números.
- Se mudarmos o método de segurança de usuário e senha para um *token*, teríamos de mudar a assinatura do serviço, ou seja, mudar o WSDL por uma implementação de segurança.
- Detalhes de segurança devem ficar transparentes para a especificação do serviço.

Não há uma única solução para autenticação de serviços *web*. Neste momento, vamos utilizar um mecanismo simples e muito eficiente: passar o usuário e senha no *header* da requisição.

A chamada de um *web service* é feita por uma requisição HTTP, na qual enviamos os envelopes SOAP no corpo (*body*) da mensagem e podemos transmitir outras informações nos *headers* da requisição HTTP.

Vamos construir essa solução. Siga o passo a passo!

Volte para classe **Calculadora**. Nossa soma não receberá os parâmetros de usuário e senha, ok? Mas note que criaremos um atributo chamado **contexto**, do tipo **WebServiceContext**, com a anotação **@Resource**.

```
import javax.xml.ws.WebServiceContext;

@WebService
@SOAPBinding(style = Style.RPC)
public class Calculadora {

        @Resource
        WebServiceContext context;

         @WebMethod(action = "somar")
         public int somar(
                         @WebParam(name = "numero1") int numero1,
                         @WebParam(name = "numero2") int numero2) {
                 return numero1 + numero2;

         }
     ...
```

Esse objeto **context** terá todas as informações da requisição HTTP que precisaremos para obter o usuário e senha.

Ainda na classe **Calculadora**, crie o método **autenticar**, que receberá o contexto e verificará os *headers* da requisição. Só vamos aceitar requisições do usuário **sisfin**, com senha **sisfin123**.

```
@SuppressWarnings("rawtypes")
private boolean autenticar
(WebServiceContext context) throws Exception {
        MessageContext mc = context.getMessageContext();
        Map httpHeaders =
(Map) mc.get(MessageContext.HTTP_REQUEST_HEADERS);

        if (!httpHeaders.containsKey("usuario"))
                throw new Exception("Informe um usuário");

        if (!httpHeaders.containsKey("senha"))
                throw new Exception("Informe uma senha");

        String usuario =
((List) httpHeaders.get("usuario")).get(0).toString();
        String senha =
((List) httpHeaders.get("senha")).get(0).toString();

        if (usuario.equals("sisfin") && senha.equals("sisfin123"))
{
                return true;
        } else {
                throw new Exception("Usuário e senha inválido");
        }
}
```

Vamos incluir a chamado do método **autenticar** antes da soma no método **somar**.

```
@WebMethod(action = "somar")
public int somar(
                @WebParam(name = "numero1") int numero1,
```

```
                @WebParam(name
= "numero2") int numero2) throws Exception {
        autenticar(context);
        return numero1 + numero2;
}
```

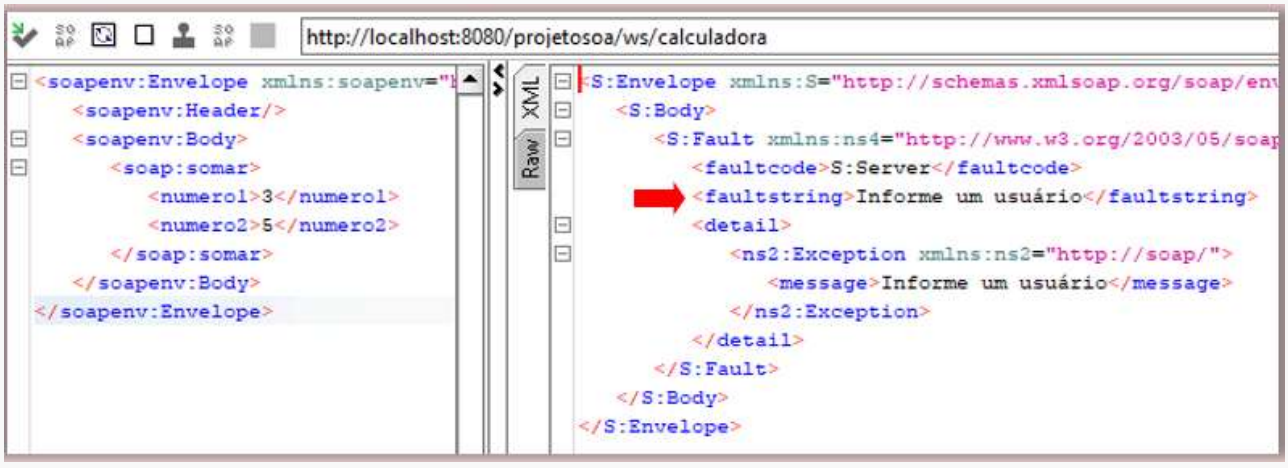
Lembre-se de que, na assinatura do método **soma**, precisamos informar que ele retornará **Exception** (**throws Exception**).

Observação: se por algum motivo tiver dúvidas nos *imports* da classe **Calculadora**, vamos deixar aqui todos os *imports* que utilizamos até o momento.

```
import java.util.List;
import java.util.Map;

import javax.annotation.Resource;
import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
import javax.jws.soap.SOAPBinding.Style;
import javax.xml.ws.WebServiceContext;
import javax.xml.ws.handler.MessageContext;

import exception.DivisaoPorZeroException;
```

Vamos testar?

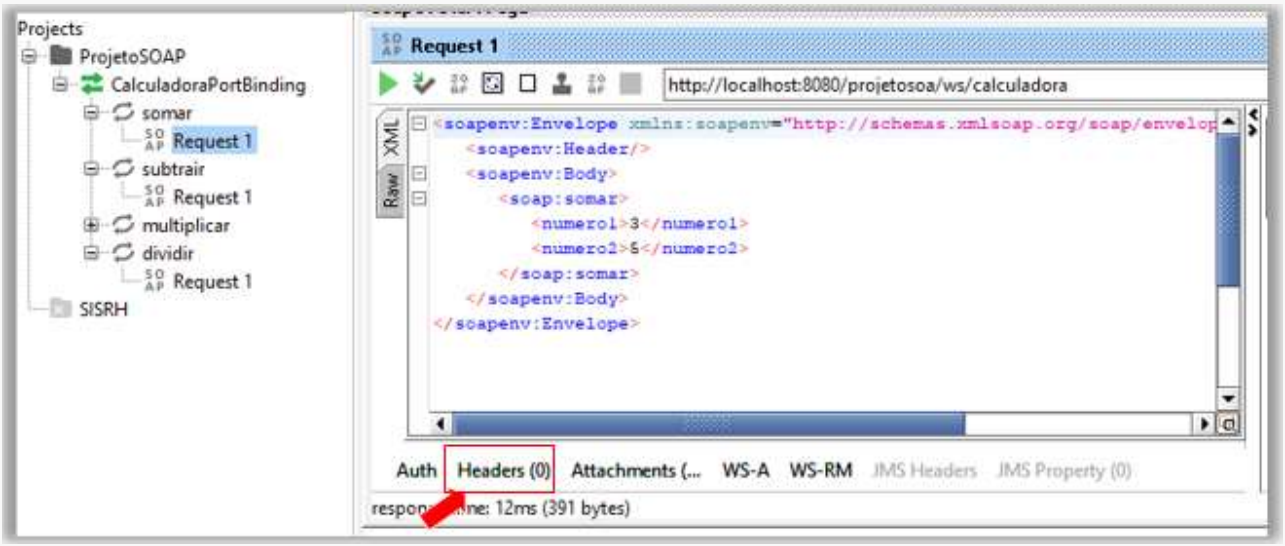
Nosso primeiro teste será a chamada do serviço sem passar os *headers*:



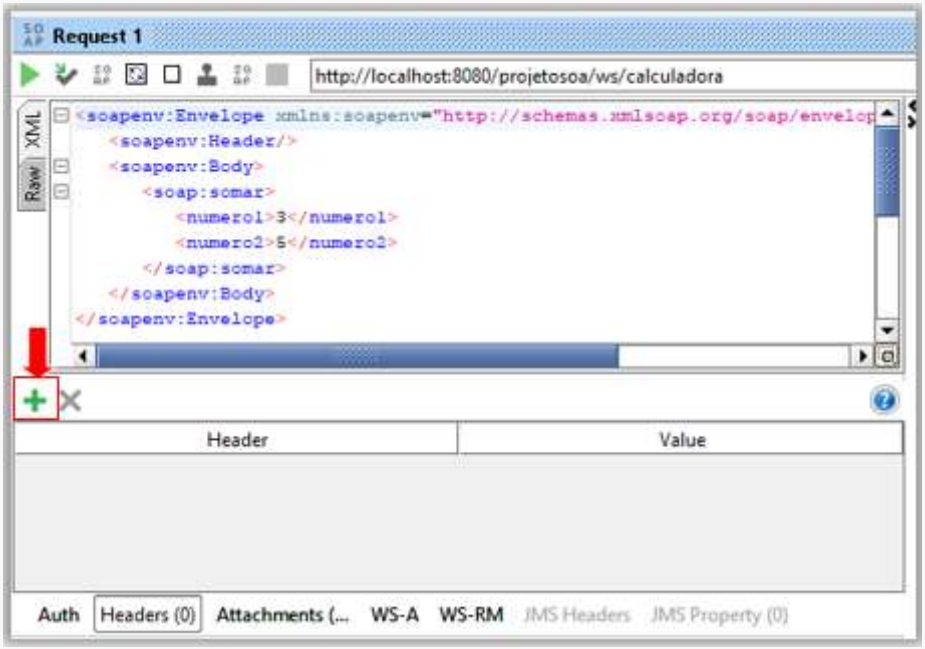
Aqui, utilizamos diretamente uma *exception*, mas você já entendeu que poderíamos criar uma falha personalizada, como, por exemplo, **FalhaDeAutenticacaoException**.

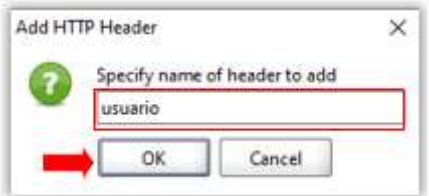
Agora, vamos passar o usuário e senha.

Na própria requisição de teste, selecione a aba **Headers**.



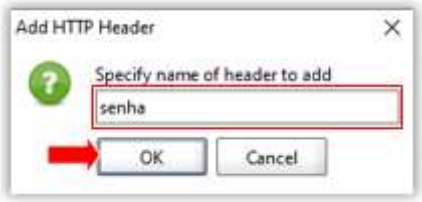
Adicione o *header* **usuario**.



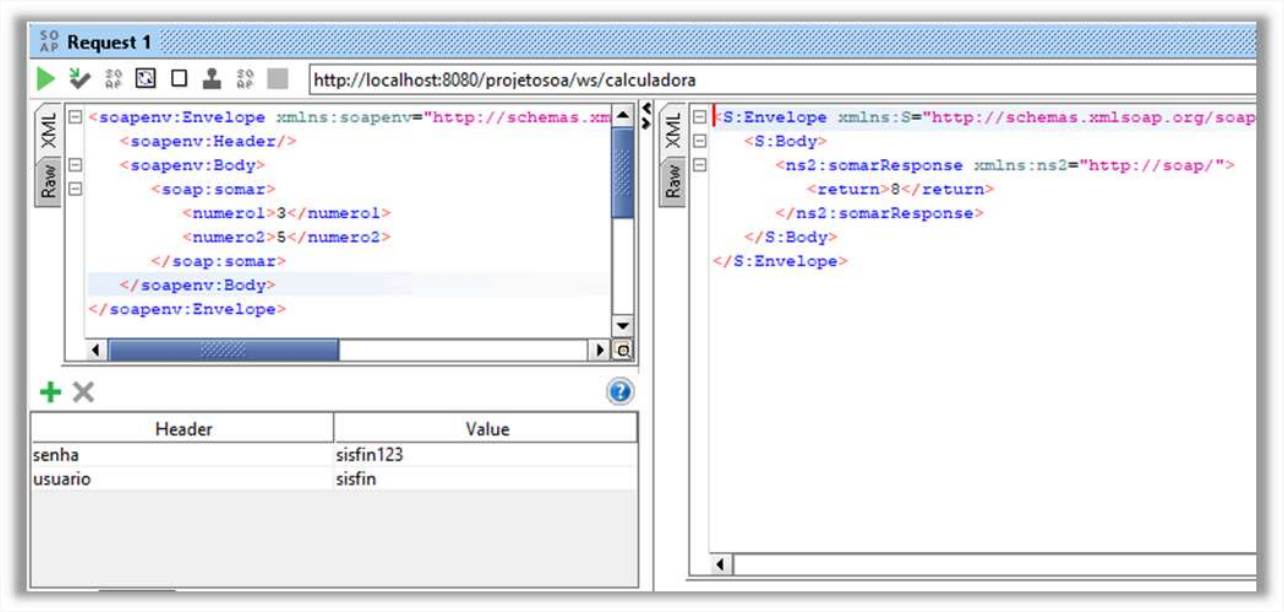


Informe o valor **sisfin**.

Adicione o parâmetro **senha**.



Execute o teste. Observe que o serviço voltou a funcionar corretamente.



Conforme mencionado, há outras estratégias para autenticação de *web services* SOAP. Pessoalmente, gostamos do envio de usuário e senha por *headers* da requisição HTTP; além de ser um dos meios mais utilizados, podemos incluir ou remover *headers* e a assinatura do serviço não mudará, ou seja, o documento WSDL não sofrerá interferência por conta dos mecanismos de autenticação.

Já temos o conhecimento necessário para iniciar a transformação do sistema SISRH, pois agora podemos expor seus dados por meio de *web services* SOAP e criar um mecanismo de autenticação para garantir que os dados dos funcionários não serão chamados por terceiros não autorizados.

Vamos conhecer o sistema SISRH.

SISRH

Faça o download do projeto **SISRH** pelo link abaixo:

sisrh.zip

Nesta videoaula, vamos explicar o projeto SISRH e criar o primeiro web service.

-

CONCLUSÃO

- Falhas em *web services* SOAP são representadas no documento WSDL e possuem uma estrutura própria.
- A especificação JAX-WS automatiza o tratamento de erros por meio da anotação **@WebFault.**
- Uma alternativa para autenticação em *web services* SOAP é o envio das credenciais por meio de *headers* da requisição HTTP.
- Conhecemos o sistema SISRH e criamos os primeiros *web services*.