argonavis
tecnologia e arte

serviços web
SOAP WSDL
JAX-WS

Helder da Rocha

JAVA EE 7

O texto foi elaborado como material de apoio para treinamentos especializados em linguagem Java e explora assuntos detalhados nas especificações e documentações oficiais sobre o tema, utilizadas como principais fontes. A autoria deste texto é de inteira responsabilidade do seu autor, que o escreveu independentemente com finalidade educativa e não tem qualquer relação com a Oracle.

O código-fonte relacionado aos tópicos abordados neste material estão em:

`github.com/helderdarocha/javaee7-course`
`github.com/helderdarocha/CursoJavaEE_Exercicios`
`github.com/helderdarocha/ExercicioMinicursoJMS`
`github.com/helderdarocha/JavaEE7SecurityExamples`

www.argonavis.com.br

R672p Rocha, Helder Lima Santos da, 1968-

Programação de aplicações Java EE usando Glassfish e WildFly.

360p. 21cm x 29.7cm. PDF.

Documento criado em 16 de outubro de 2013.

Atualizado e ampliado entre setembro e dezembro de 2016.

Volumes (independentes): *1: Introdução, 2: Servlets, 3: CDI, 4: JPA, 5: EJB, 6: SOAP, 7: REST, 8: JSF, 9: JMS, 10: Segurança, 11: Exercícios*.

1. Java (*Linguagem de programação de computadores*). 2. Java EE (*Linguagem de programação de computadores*). 3. Computação distribuída (*Ciência da Computação*). I. Título.

CDD 005.13'3

Capítulo 6: Web Services SOAP

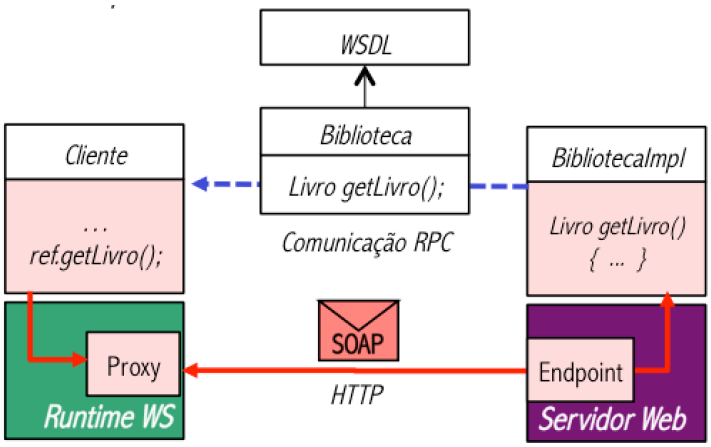
1 SOAP Web Services

Web Services é uma arquitetura de objetos distribuídos que geralmente usa como camada de transporte o protocolo HTTP. SOAP Web Services representa serviços que utiliza protocolos em XML para praticamente todos os serviços, inclusive transporte (SOAP), registro (UDDI) e descrição de interfaces comuns (WSDL).

A comunicação depende de uma interface comum compartilhada e implementada entre cliente e servidor. O objeto remoto implementa interface comum através de um proxy. Essa interface é exportada em WSDL, que é independente de linguagem. Qualquer cliente de qualquer linguagem pode usar um WSDL para gerar código de acesso ao serviço que exportou a interface.

A principal alternativa a SOAP para implementar Web Services é usar a arquitetura REST, que usa a infraestrutura do HTTP para oferecer uma interface de serviços remotos.

O diagrama abaixo ilustra a arquitetura de Web Services SOAP.



2 Web Services SOAP em Java com JAX-WS

JAX-WS e uma API do Java EE que permite a criação de serviços e clientes SOAP, de forma transparente, escondendo todos os detalhes da comunicação. Para criar um serviço SOAP em Java EE há duas alternativas:

1. Através da criação de um **Componente Web** – requer a criação de uma interface de terminal de serviços (*SEI – Service Enpoint Interface*) configurada com anotações, uso de ferramentas para gerar código, e empacotamento das classes compiladas em um WAR para implantação em um container Web.

2. Através de um **Session Bean** (EJB) – É preciso criar um *Stateless Session Bean* com ou sem uma interface para exportar (que será o SEI) configurado com anotações, empacotar como EJB e implantar o EJB-JAR ou EAR em um container EJB.

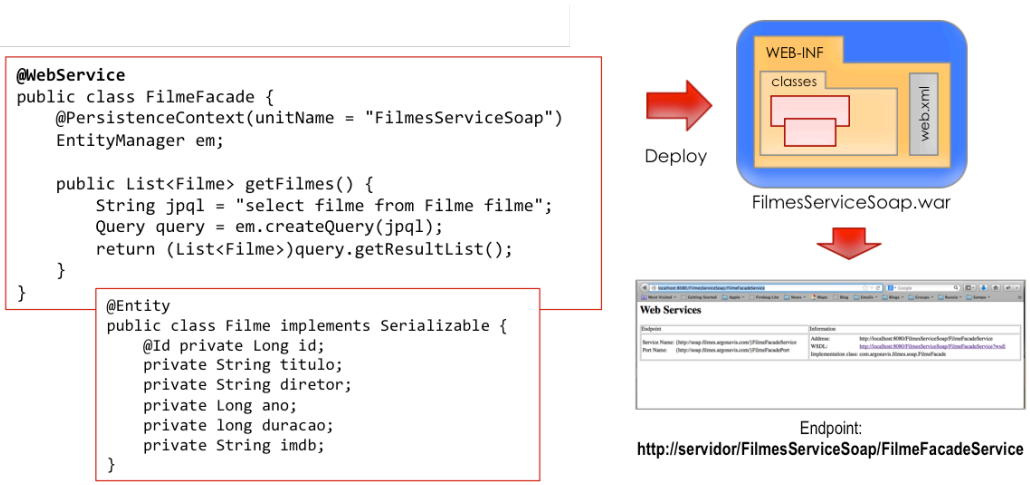
2.1 Componente Web

O serviço é implementado em um JavaBean comum anotado com *@WebService* (pacote *javax.jws*) que representa um *Service Endpoint Interface (SEI)*. Todos os métodos públicos de um SEI são automaticamente incluídos na interface do serviço (Java EE 7).

```
@WebService
public class FilmeFacade {
     @PersistenceContext(unitName = "FilmesServiceSoap")
     EntityManager em;

     public List<Filme> getFilmes() {
         String jpql = "select filme from Filme filme";
         Query query = em.createQuery(jpql);
         return (List<Filme>)query.getResultList();
     }
}
```

Usando a anotação *@WebMethod* em um método, os métodos não anotados serão excluídos da SEI e terão que receber uma anotação *@WebMethod* se devem ser incluídos.

O bean deve ser empacotado em um WAR (seguindo a estrutura comum do WAR, dentro de *WEB-INF/classes*), e depois instalado em um container Web.



2.1.1 WSDL gerado

A partir do deployment, um WSDL será gerado e disponibilizado para os clientes através da URL *http://servidor/nome-do-war/NomeDaClasseService?wsdl*. O WSDL gerado para o exemplo acima está listado abaixo.

```
<definitions targetNamespace="http://soap.filmes.argonavis.com/"
                      name="FilmeFacadeService">
    <types>
            <xs:schema version="1.0" targetNamespace="http://soap.filmes.argonavis.com/">
                <xs:element name="getFilmes"         type="tns:getFilmes"/>
                <xs:element name="getFilmesResponse" type="tns:getFilmesResponse"/>
                <xs:complexType name="filme">
                    <xs:sequence>
                        <xs:element name="ano"     type="xs:long"   minOccurs="0"/>
                        <xs:element name="diretor" type="xs:string" minOccurs="0"/>
                        <xs:element name="duracao" type="xs:long"/>
                        <xs:element name="id"      type="xs:long"   minOccurs="0"/>
                        <xs:element name="imdb"    type="xs:string" minOccurs="0"/>
                        <xs:element name="titulo"  type="xs:string" minOccurs="0"/>
                    </xs:sequence>
                </xs:complexType>

                <xs:complexType name="getFilmes"> <xs:sequence/> </xs:complexType>

                <xs:complexType name="getFilmesResponse">
                    <xs:sequence>
                        <xs:element name="return" type="tns:filme"
                                    minOccurs="0" maxOccurs="unbounded"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:schema>
    </types>
    <message name="getFilmes">
        <part name="parameters" element="tns:getFilmes"/>
    </message>
    <message name="getFilmesResponse">
        <part name="parameters" element="tns:getFilmesResponse"/>
    </message>
    <portType name="FilmeFacade">
        <operation name="getFilmes">…</operation>
    </portType>
    <binding name="FilmeFacadePortBinding" type="tns:FilmeFacade">… </binding>
    <service name="FilmeFacadeService">
        <port name="FilmeFacadePort" binding="tns:FilmeFacadePortBinding">
          <soap:address
            location="http://localhost:8080/FilmesServiceSoap/FilmeFacadeService"/>
        </port>
    </service>
</definitions>
```

2.2 Anotações e Contexto

Além de *@WebService* (única anotação obrigatória), várias outras anotações, dos pacotes javax.jws.* e javax.jws.soap.*, podem ser usadas para configurar detalhes do serviço.

Anotações aplicadas na classe:

- *@SOAPBinding* – especifica mapeamento SOAP;
- *@BindingType* – especifica tipo de mapeamento;
- *@HandlerChain* – associa o Web Service a uma cadeia de handlers;

Anotações usadas em métodos:

- *@WebMethod* – configura métodos da interface SEI, inclui e exclui;
- *@OneWay* – declara método uma operação sem retorno (só mensagem de ida);

Anotações para parâmetros de um método:

- *@WebParam* – configura nomes dos parâmetros;

Anotações para valores de retorno de um método

- *@WebResult* – configura nome e comportamento;

2.2.1 @OneWay

Métodos anotados com @OneWay têm apenas mensagem de requisição (sem resposta). Esta anotação pode ser usada em métodos que retornam void. Exemplos:

```
@WebMethod @OneWay
public void enviarAvisoDesligamento() {
   …
}
@WebMethod @OneWay
public void ping() {
   …
}
```

2.2.2 @WebParam e @WebResult

Permitem configurar o WSDL que será gerado e o mapeamento entre o SEI e o SOAP.

@WebResult serve para configurar o elemento XML de retorno. No exemplo abaixo, a resposta estará dentro de um elemento XML *<filme></filme>*. O default é *<return></return>*.

```
@WebMethod @WebResult(name="filme")
public Filme getFilme(String imdbCode) {
        return getFilmeObject(imdbCode);
}
```

@WebParam permite configurar nomes dos parâmetros. No exemplo abaixo, o parâmetro da operação *getFilme()* no SOAP e WSDL é *imdb*. Seria *imdbCode* (o nome da variável local) se o *@WebParam* não estivesse presente:

```
@WebMethod
public Filme getFilme(@WebParam(name="imdb") String imdbCode) {
        return getFilmeObject(imdbCode);
}
```

2.3 Stateless remote SOAP bean

Usar EJB é a forma mais simples de criar e implantar um serviço. Session Beans podem ter sua interface exportada como um Web Service SOAP. O resultado é idêntico à do Web Service via componente Web, mas a configuração é mais simples e permite acesso aos serviços básicos do EJB, mais o acesso remoto via porta HTTP.

Pode-se criar um session bean remoto que exporta uma interface SEI criando uma interface anotada com @WebService:

```
@WebService
interface LoteriaWeb {
     int[] numerosDaSorte();
}
```

E depois implementando-a em um bean @Stateless:

```
@Stateless
public class LoteriaWebBean implements LoteriaWeb {
    @Override
    public int[] numerosDaSorte() {
        int[] numeros = new int[6];
        for(int i = 0; i < numeros.length; i++) {
            int numero = (int)Math.ceil(Math.random() * 60);
            numeros[i] = numero;
        }
        return numeros;
    }
}
```

Se a interface não puder ser alterada (não puder receber a anotação), ela ainda pode ser configurada no próprio bean usando @WebService com o atributo endpointInterface:

```
@Stateless
@WebService(endpointInterface="nome.da.Interface")
```

Com o deploy, o servidor gera as classes TIE do servidor, WSDL, e estabelece um endpoint. Por exemplo, o bean anterior pode ser configurado da seguinte forma:

```
@Stateless @WebService(endpointInterface="LoteriaWeb")
public class LoteriaWebBean implements LoteriaWeb {
```

```
    @Override
    public int[] numerosDaSorte() {
        int[] numeros = new int[6];
        for(int i = 0; i < numeros.length; i++) {
            int numero = (int)Math.ceil(Math.random() * 60);
            numeros[i] = numero;
        }
        return numeros;
    }
}
```

3 Clientes SOAP

Clientes SOAP podem ser criados de várias formas, em Java ou mesmo em outras linguagens. A maneira mais simples consiste em usar ferramentas para gerar código estaticamente compilando o WSDL, ou usar um container do fabricante. Outras estratégias permite gerar stubs, proxies e classes dinamicamente, ou ainda usar *reflection* para chamar a interface dinamicamente.

O exemplo abaixo mostra um cliente (estático) rodando como aplicação standalone:

```
public class WSClient {
    public static void main(String[] args) throws Exception {
        LoteriaWebService service = new LoteriaWebService();
        LoteriaWeb port = service.getLoteriaWebPort();
        List<String> numeros = port.numerosDaSorte();

        System.out.println("Numeros a jogar na SENA:");
        for(String numero : numeros) {
            System.out.println(numero);
        }
    }
}
```

As classes em destaque acima (*LoteriaWebService* e *LoteriaWeb*) são classes que foram geradas a partir do WSDL por ferramentas.

3.1 Compilação de WSDL

Ferramentas de compilação do WSDL existem em vários IDEs e fazem parte do Java Development Kit. No Java SDK existe a ferramenta *wsimport* (Java). O CXF (JBoss/WildFly) usa uma ferramenta semelhante chamada *wsconsume*. Ambas geram artefatos Java necessários para clientes através da compilação de WSDL.

Exemplo de geração de código com *wsimport* (Java SE SDK)

```
wsimport -keep -s gensrc -d genbin
         -p com.argonavis.filmes.client.soap.generated
         http://localhost:8080/FilmesServiceSoap/FilmeFacadeService?wsdl
```

Exemplo de geraçao de código com *wsconsume* (WildFly / JBoss)

```
wsconsume.sh -k -s gensrc -o genbin
             -p com.argonavis.filmes.client.soap.generated
             http://localhost:8080/FilmesServiceSoap/FilmeFacadeService?wsdl
```

Abaixo está uma lista das classes geradas para os exemplos mostrados anteriormente (filmes). Estas classes são as que o cliente precisará usar para utilizar o serviço remote. Elas devem ser incluídas no classpath do cliente:

- *Filme.class*
- *FilmeFacade.class*
- *FilmeFacadeService.class*
- *GetFilmes.class*
- *GetFilmesResponse.class*
- *ObjectFactory.class*
- *package-info.class*

As ferramentas de linha de comando podem ser executadas durante a construção da aplicação em plugins Maven.

3.1.1 Cliente SOAP usando classes geradas

O exemplo abaixo mostra um cliente SOAP que usa as classes geradas:

```
public class FilmeClient {

    public static void main(String[] args) {
        FilmeFacadeService service = new FilmeFacadeService();
        FilmeFacade proxy = service.getFilmeFacadePort();

        listarFilmes(proxy.getFilmes());
    }
    public static void listarFilmes(List<Filme> filmes) {
        for(Filme f : filmes) {
            System.out.println(f.getImdb()+": " + f.getTitulo()
                                        + "(" + f.getAno() + ")");
            System.out.println("           " + f.getDiretor());
            System.out.println("           " + f.getDuracao() + " minutos\n");
        }
    }
}
```

Executando a classe acima, o proxy conecta-se ao servidor e obtém os objetos remotos:

```
$ java -jar FilmeClient.jar
tt0081505: The Shining(1980)
           Stanley Kubrick
           144 minutos
tt1937390: Nymphomaniac(2013)
           Lars von Trier
           330 minutos
tt0069293: Solyaris(1972)
           Andrei Tarkovsky
           167 minutos
tt1445520: Hearat Shulayim(2011)
           Joseph Cedar
```

3.2 Tipos de clientes

Clientes podem ser mais dinâmicos e menos acoplados, depender menos de implementações e de interfaces, e configurar-se em tempo de execução. Há duas estratégias:

- *Proxy dinâmico:* tem cópia local da interface do serviço, mas gera código em tempo de execução através do WSDL remoto
- *Cliente totalmente dinâmico:* não depende de interface, WSDL ou quaisquer artefatos gerados para enviar requisições e obter resposta, mas é necessário trabalhar no nível mais baixo das mensagens XML (SOAP)

Cliente com *proxy dinâmico:*

```
URL wsdl = new URL("http://servidor/app/AppInterfaceService?wsdl");
QName nomeServ = new QName("http://app.ns/", "AppInterfaceService");
Service service = Service.create(wsdl, nomeServ);
AppInterface proxy = service.getPort(AppInterface.class);
```

Cliente 100% dinâmico (Dispatch client – trecho):

```
Dispatch<Source> dispatch =
    service.createDispatch(portName, Source.class, Service.Mode.PAYLOAD);
String reqPayload  = "<ans1:getFilmes xmlns:ans1=\"http://soap.filmes.argonavis.com/\">"
                   + "</ans1:getFilmes>";
Source resPayload = dispatch.invoke(new StreamSource(new StringReader(reqPayload)));

DOMResult domTree = new DOMResult();
TransformerFactory.newInstance().newTransformer().transform(resPayload, domTree);
Document document = (Document)domTree.getNode();
Element root = document.getDocumentElement();
Element filmeElement =
    (Element)root.getElementsByTagName("return").item(0); // <return>…</return>
String tituloDoFilme =
    filmeElement.getElementsByTagName("titulo").item(0).getFirstChild()
                .getTextContent(); // <titulo>CONTEUDO</titulo>…
```

A classe abaixo ilustra uma implementação de cliente para os exemplos de WebServices listados anteriormente, usando a técnica de cliente dinâmico (Proxy dinâmico):

```
public class FilmeDynamicClient {
    public static void main(String[] args) throws MalformedURLException {
        URL wsdlLocation =
           new URL("http://localhost:8080/FilmesServiceSoap/FilmeFacadeService?wsdl");
        QName serviceName =
           new QName("http://soap.filmes.argonavis.com/", "FilmeFacadeService");
        Service service = Service.create(wsdlLocation, serviceName);
        FilmeFacade proxy = service.getPort(FilmeFacade.class);
        listarFilmes(proxy.getFilmes());
    }

    public static void listarFilmes(List<Filme> filmes) {
        for(Filme f : filmes) {
            System.out.println(f.getImdb()+": " +f.getTitulo()
                                        + "(" +f.getAno()+ ")");
            System.out.println("           " +f.getDiretor());
            System.out.println("           " +f.getDuracao()+ " minutos\n");
        }
    }
}
```

3.3 Cliente de WebService rodando em container

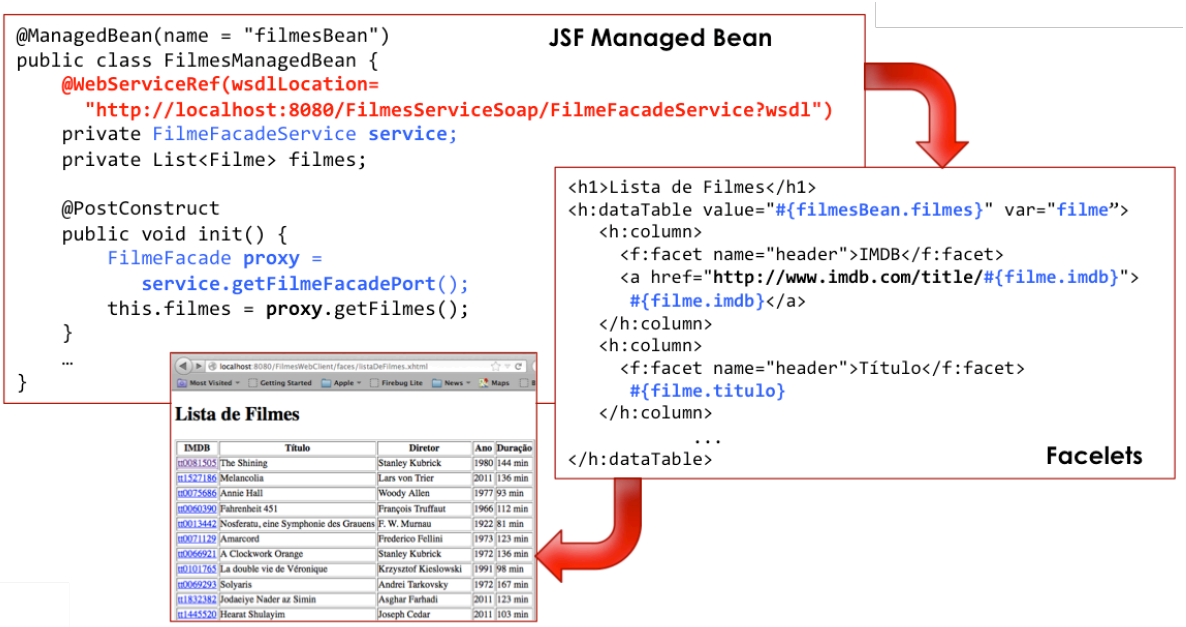
Clientes localizados em um container Java EE (ex: servlet ou managed bean) podem injetar o serviço através da anotação @WebServiceRef:

```
@Named("filmesBean")
public class FilmesManagedBean {
    @WebServiceRef(wsdlLocation=
       "http://localhost:8080/FilmesServiceSoap/FilmeFacadeService?wsdl")
    private FilmeFacadeService service;
    private List<Filme> filmes;

    @PostConstruct
    public void init() {
        FilmeFacade proxy = service.getFilmeFacadePort();
        this.filmes = proxy.getFilmes();
    }
    …
}
```

O JSF abaixo usa o bean acima:

```
<h1>Lista de Filmes</h1>
<h:dataTable value="#{filmesBean.filmes}" var="filme">
   <h:column>
     <f:facet name="header">IMDB</f:facet>
     <a href="http://www.imdb.com/title/#{filme.imdb}">
      #{filme.imdb}</a>
```

```
    </h:column>
    <h:column>
      <f:facet name="header">Título</f:facet>
       #{filme.titulo}
    </h:column>
            ...
</h:dataTable>
```



4 Metadados de uma mensagem SOAP

4.1 WebServiceContext

WebServiceContext dá acesso a um objeto de contexto que permite acesso a informações sobre a mensagem e autenticação/autorização, se houver. Os métodos são:

- *MessageContext* **getMessageContext()**: retorna objeto *MessageContext* que permite acesso a metadados da mensagem (cabeçalhos, porta, serviço, info do servlet, etc.)
- *Principal* **getUserPrincipal()**: permite acesso ao *javax.security.Principal* do usuário autenticado. Principal contém informações de autenticação.
- *boolean* **isUserInRole***(String role)*: retorna *true* se usuário autenticado faz parte de um grupo de autorizações (role).

Roles de autorização precisam ser configurados em web.xml para o contexto do Web Service (Web Resource Collection). Veja detalhes na documentação sobre segurança de aplicações Web.

Exemplo de uso:

```
@Resource
```

```
private WebServiceContext ctx;

@WebMethod()
public String metodoSeguro(String msg) {
     String userid = ctx.getUserPrincipal().getName();
     if (userid.equals("czar")) {
        ...
     } else if (ctx.isUserInRole("admin")) {
        ...
     }
}
```

4.2 MessageContext

MessageContext é um dos objetos obtidos de um WebServiceContext. É um dos objetos que permite acesso a metadados de uma mensagem (ex: cabeçalhos SOAP, se mensagem é inbound ou outbound, dados do WSDL, servidor, etc.)

O acesso às propriedades ocorre através do método get(), passando-se uma constante correspondente à propriedade desejada. As propriedades são armazenadas em um *Map*. Algumas propriedades incluem:

- MESSAGE_OUTBOUND_PROPERTY (boolean)
- INBOUND_MESSAGE_ATTACHMENTS (Map)
- HTTP_REQUEST_METHOD (String)
- WSDL_OPERATION (Qname)

Exemplo de uso (para obter os cabeçalhos HTTP da mensagem):

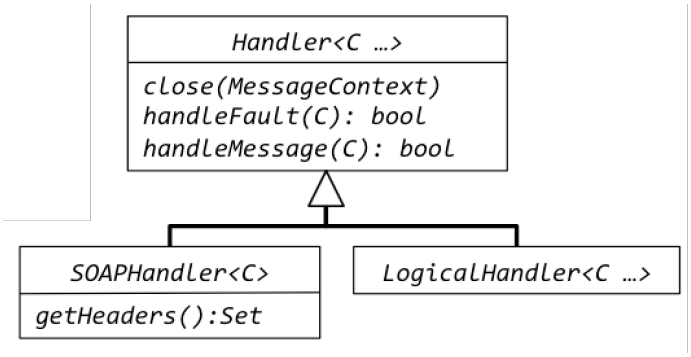
```
@Resource
WebServiceContext wsctx;

@WebMethod
public void metodo() {
     MessageContext ctx = wsContext.getMessageContext();
    Map headers = (Map)ctx.get(MessageContext.HTTP_REQUEST_HEADERS);
…
}
```

5 JAX-WS Handlers

Handlers são interceptadores (filtros) de mensagens. Há dois tipos: protocol (*MESSAGE*) e logical (*PAYLOAD*):

Eles podem ser usados para fazer alterações na mensagem antes que ela seja processada no servidor ou cliente. Geralmente são configurados em cascata.

Para criar um handler deve-se implementar *SOAPHandler* ou *LogicalHandler*:



Depois e necessário criar um XML para configurar a corrente:

```
<javaee:handler-chains xmlns:javaee="http://java.sun.com/xml/ns/javaee"
                       xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <javaee:handler-chain>
     <javaee:handler>
       <javaee:handler-class>pacote.ClasseDoHandler<javaee:handler-class>
     </javaee:handler>
   </javaee:handler-chain>
</javaee:handler-chains>
```

Para usar anota-se a classe do Web Service com *@HandlerChain*, passando como parâmetro o arquivo XML de configuração:

```
@HandlerChain(file="handlers.xml")
```

5.1 SOAP Handler

Implementa javax.xml.ws.handler.soap.SOAPHandler. E um interceptador para a *mensagem inteira* (permite acesso a cabeçalhos da mensagem):

```
public class MyProtocolHandler implements SOAPHandler {

    public Set getHeaders() { return null; }

    public boolean handleMessage(SOAPMessageContext ctx) {
        SOAPMessage message = ctx.getMessage();
        // fazer alguma coisa com a mensagem
        return true;
```

```
    }

    public boolean handleFault(SOAPMessageContext ctx) {
        String operacao = (String)
            ctx.get (MessageContext.WSDL_OPERATION);
        // logar nome da operacao que causou erro
        return true;
    }
    public void close(MessageContext messageContext) {}
}
```

5.2 Logical Handler

Implementa a interface *javax.xml.ws.handler.LogicalHandler*. Dá acesso apenas ao payload (corpo) da mensagem:

```
public class MyLogicalHandler implements LogicalHandler {
    public boolean handleMessage(LogicalMessageContext ctx) {
        LogicalMessage msg = context.getMessage();
        Source payload = msg.getPayload(); // XML Source
        Transformer transformer =
            TransformerFactory.newInstance().newTransformer();
        Result result = new StreamResult(System.out);
        transformer.transform(payload, result); // Imprime XML na saida
    }
    public boolean handleFault(LogicalMessageContext ctx) {
        String operacao = (String)
            ctx.get (MessageContext.WSDL_OPERATION);
        // logar nome da operacao que causou erro
        return true;
    }
    public void close(MessageContext context) {}
}
```

5.3 Como usar handlers

Arquivo de configuração (handlers-chains.xml)

```
<javaee:handler-chains xmlns:javaee="http://java.sun.com/xml/ns/javaee"
                       xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <javaee:handler-chain>
     <javaee:handler>
       <javaee:handler-class>com.acme.MyProtocolHandler</javaee:handler-class>
     </javaee:handler>
     <javaee:handler>
       <javaee:handler-class>com.acme.MyLogicalHandler</javaee:handler-class>
     </javaee:handler>
   </javaee:handler-chain>
</javaee:handler-chains>
```

Uso no servidor (no SEI)

```
@WebService
@HandlerChain(file="handler-chains.xml")
public class FilmesFacade { … }
```

Uso no cliente (edite classe gerada que implementa o Service)

```
@HandlerChain(file="handler-chains.xml")
public class FilmesFacadeService extends Service { … }
```

6 Referências

6.1 Especificações

[1] *WSDL* http://www.w3.org/TR/wsdl

[2] *SOAP* http://www.w3.org/TR/soap/

[3] *MTOM* http://www.w3.org/TR/soap12-mtom/

[4] *XOP* http://www.w3.org/TR/xop10/

[5] *WS-Addressing* http://www.w3.org/TR/ws-addr-core/

[6] *WS-Security* http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf

[7] *WS-I Basic Profile* http://ws-i.org/Profiles/BasicProfile-1.2-2010-11-09.html

[8] *JAX-WS* https://jax-ws.java.net/

[9] *SOAP sobre JMS* http://www.w3.org/TR/soapjms/

6.2 Artigos e tutoriais

[10] Russell Butek. "*Which style of WSDL should I use?*" IBM Developerworks, 2005. http://www.ibm.com/developerworks/webservices/library/ws-whichwsdl/

[11] Jaromir Hamala. "*SOAP over JMS between WebLogic 12 and Apache CXF*" C2B2, 2013. http://blog.c2b2.co.uk/2013/09/soap-over-jms-between-weblogic-12-and.html

[12] *Como implementar a interface Provider* (Apache CXF documentation). http://cxf.apache.org/docs/provider-services.html

[13] Rama Pulavarthi. "*Introduction to handlers in JAX-WS*". Java.net articles. https://jax-ws.java.net/articles/handlers_introduction.html

[14] *JAX-WS WS-Addressing* https://jax-ws.java.net/nonav/jax-ws-21-ea2/docs/wsaddressing.html